جامعة الأخوين

ⵜⴰⵙⴷⴰⵡⵉⵜ ⵏ ⵍⴰⵅⴰⵡⴰⵢⵏ

AL AKHAWAYN
UNIVERSITY

Projet#4
Introduction to Artificial Intelligence
CSC 3309

By: Firdaous Zinebi 100043 (Section 01)
Maissae Azaroual 100430 (Section 02)
Rihab Zouitni 100552 (Section 01)

Supervised by: Dr. Tajjeeddine Rachidi

Table of Contents

Task Distribution

For this project, we divided the tasks among team members to ensure efficient and timely progress. Each member was responsible for a specific part and contributed a section to the final report, reflecting her assigned work.

Each team member actively participated in the initial step of running the code to familiarize themselves with the project. Following this, each member began experimenting with the hyperparameters to explore potential improvements. To ensure efficient and timely progress, we divided tasks among team members, with each person focusing on a specific aspect of the project. Additionally, each member contributed a section to the final report, documenting her assigned work.

The first enhancement was implemented by Firdaous, the second by Maissae, and the third by Rihab, showcasing a collaborative effort to refine and optimize the project.

I. Introduction

Computer Vision is a crucial field within artificial intelligence, it presents a unique challenge which is creating systems capable of perceiving and interpreting the world as humans do. This field is responsible of various tasks, but at its core lies image classification, it is a process that enables computers to recognize and categorize objects in images. Mastering this capability paves the way for advanced applications like object detection, facial recognition, and scene understanding, with transformative potential across industries. This project focuses on researching and developing a high-performing ***Convolutional Neural Network (CNN)*** for precise image classification using Keras, a widely-used deep learning framework, that makes it possible to seamlessly translate established concepts into practical implementation, ensuring the development of an effective and impactful model for this essential task.

The capacity of Convolutional Neural Networks (CNNs) to learn and extract complex characteristics from visual data has made them the most favorite method in computer vision. starting with fundamental components like edges and progressively identifying increasingly complex textures and forms, these neural networks aim to imitate how the human brain interprets visual information. CNNs are useful tools for applications such as image classification because of their capacity to efficiently learn characteristics inside images. In this project, the famous CIFAR-10 dataset will be used to train and evaluate the models developed by Kolar Maureen and Jane Goodman. With 60,000 photos categorized into 10 groups, such as vehicles, birds, aircraft, and animals, this dataset is ideal for developing and testing CNN-based systems.

The goal of this project is to create, train, and optimize a CNN that performs very well in image classification using the CIFAR-10 dataset. The main goal is to optimize the model's accuracy while making sure it operates consistently on unknown inputs. Iterative testing and meticulous fine-tuning, including hyperparameter optimization, sophisticated training methods, and a thorough examination of network architecture, are required to do this. The project aims to determine how variables like learning rates, dropout rates, and batch sizes interact and affect model performance by carefully examining these aspects.

Techniques like data augmentation and regularization will be used to improve the findings even further, making sure the model is resistant to overfitting and attains high accuracy.

This report explores the complex process of developing and improving CNNs, providing a thorough description of the techniques used and the knowledge acquired. We begin with a basic baseline design and gradually increase the network's functionality and complexity until the classification accuracy significantly improves. A thorough analysis of the outcomes is included at each step of the process, giving readers a clear picture of the advantages and disadvantages of various design options. Accuracy graphs and confusion matrices are examples of visualizations that show the model's performance in greater detail and point out areas that could use improvement.

This project emphasizes how crucial it is to find a balance between creativity and practicality in an industry that is changing quickly. It seeks to improve CNNs in image classification while adding to the larger debate on machine learning best practices by combining modern techniques with a solid understanding of fundamental concepts. The results and techniques discussed here provide a framework for further investigation, encouraging further research and applications in fields like personalized healthcare and autonomous systems. This work is more than just a technical accomplishment; it shows how artificial intelligence might change our view and interactions with the visual environment by crossing the gap between computer vision and human perception.

II. Computer Vision Concepts

Computer Vision concepts are fundamental to how Convolutional Neural Networks (CNNs) operate, through this project we are going to get more familiar with some concepts in the implementation but before that, we need to master the understanding of those concepts.

1. Convolution

It is the foundation of CNNs, a mathematical process that involves sliding a filter, which is a tiny matrix of numbers, over an image to find particular patterns like edges, textures, or shapes it is also the first layer of a CNN model. The filter creates a feature map that highlights significant features in the image by computing a dot product between the filter values and the corresponding pixel values as it passes over local portions of the image. The resolution and efficacy of the feature map are greatly influenced by parameters such as filter size, stride (the filter's step size), and padding (the additional borders surrounding the image).

2. Activation Function

An activation function is used to give the system non-linear characteristics following the convolution operation. The Rectified Linear Unit (ReLU), the most popular activation function in CNNs, is utilized to increase the image's non-linearity. This is the function we employ in our model for this project. ReLUs are used because they decrease the chance that the gradient would disappear, which enables models to learn more quickly and function better.

3. Pooling

It decreases the spatial dimensions of the feature maps, and then enhances their computational effectiveness and adaptability to slight changes in the image. While *average pooling* captures general patterns, *maximum pooling* preserves the most noticeable feature within a region. In addition to making the model simpler, this downsampling technique *reduces overfitting*. CNNs naturally pick up hierarchical features: the deeper layers recognize whole objects, the middle layers recognize shapes or themes, and the first layers collect basic edges and textures. The way the human brain interprets visual information is reflected in this development.

4. **Strides and padding**

A CNN's processing of an image is influenced by strides and padding. While padding adds extra pixels around the image to maintain its dimensions or methodically reduce its size, strides determine the amount of movement of the filter during convolution. By standardizing the outputs of earlier layers, normalization techniques like batch normalization stabilize and speed up training. Faster convergence results from avoiding high activation values and ensuring smoother gradients.

5. **Dropout**

Dropout is a regularization technique used during training where some neurons are randomly turned off. This prevents the network from relying too heavily on specific neurons, encouraging it to learn more robust and generalized features rather than simply memorizing the training data.

6. **Data Augmentation**

Data augmentation is a technique used to increase the variety of training data by applying transformations to existing images. These transformations, such as flipping, rotating, scaling, or changing brightness, introduce subtle variations that mimic real-world scenarios. By exposing the model to these diverse inputs, data augmentation helps it become more adaptable and improves its ability to generalize to new, unseen data, making it more robust and reliable in practical applications.

7. **Optimizer**

Optimizers play a key role in improving a model’s performance by reducing the loss function, which measures the difference between the model’s predictions and the actual results. They adjust the model’s parameters, like weights and biases, during training to make better predictions. By using gradients calculated through backpropagation, optimizers guide the model to learn and improve over time. In this project, we rely on the Adam optimizer, a popular and efficient choice, to fine-tune the model and achieve better accuracy.

8. **Loss Function**

The loss function is like a guide for the model, showing how far its predictions are from the actual answers. It acts as feedback, helping the model understand where it's going wrong. During training, the model works to minimize this loss, gradually learning to make better predictions. By comparing what the model predicts to the true values, the loss function helps fine-tune the model so it gets closer to the correct answers over time.

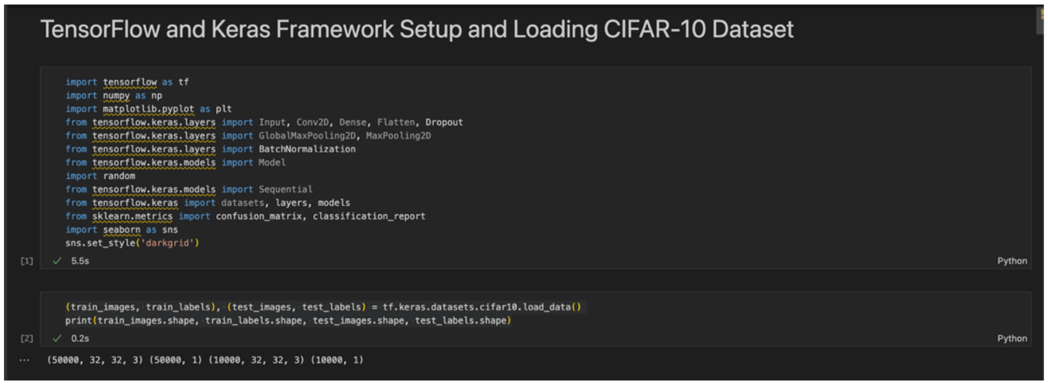
9. **Batch Implementation**

By normalizing layer inputs to have a constant mean and variance, batch normalization is a method for stabilizing and speeding up neural network training. it is especially crucial for our project because the CIFAR-10 dataset requires a deep CNN architecture because of its numerous and complicated image classes. BatchNorm enables the model to be developed more efficiently by addressing problems such as vanishing or exploding gradients and lowering sensitivity to weight initialization. Additionally, it has a regularizing effect, enhancing the model's adaptability to unseen data, and making it vital for achieving improved accuracy and stability in this image classification issue.

III. Setup:

1. Dataset Preparation

This project is based on the CIFAR-10 dataset, which consists of 60,000 images in 10 categories (e.g., cars, airplanes, animals). The dataset is properly loaded and ready for training using frameworks like TensorFlow or Keras.



TensorFlow and Keras Framework Setup and Loading CIFAR-10 Dataset

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.layers import Input, Conv2D, Dense, Flatten, Dropout
from tensorflow.keras.layers import GlobalMaxPooling2D, MaxPooling2D
from tensorflow.keras.layers import BatchNormalization
from tensorflow.keras.models import Model
import random
from tensorflow.keras.models import Sequential
from tensorflow.keras import datasets, layers, models
from sklearn.metrics import confusion_matrix, classification_report
import seaborn as sns
sns.set_style('darkgrid')
```

[1] ✓ 5.5s Python

```
(train_images, train_labels), (test_images, test_labels) = tf.keras.datasets.cifar10.load_data()
print(train_images.shape, train_labels.shape, test_images.shape, test_labels.shape)
```

[2] ✓ 0.2s Python

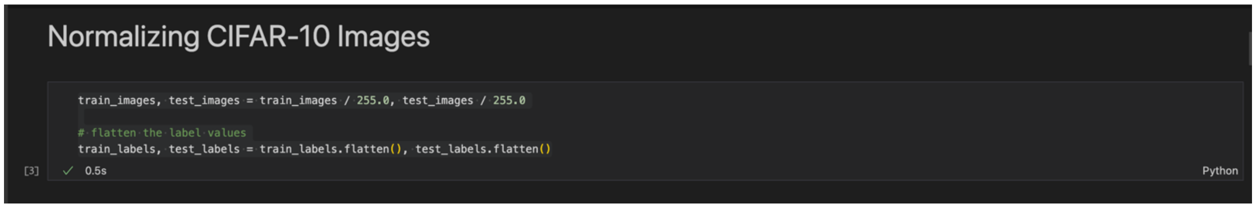
```
(50000, 32, 32, 3) (50000, 1) (10000, 32, 32, 3) (10000, 1)
```

- *Normalization of dataset:*

  The pixel values in the CIFAR-10 dataset initially range from 0 to 255, representing the intensity of each pixel. By dividing these values by 255.0, the data is normalized to fall within the range [0, 1]. This normalization process ensures consistency in the input data, which is crucial for improving the training process. Neural networks perform better and converge faster when the input data is scaled uniformly to smaller ranges, making normalization a vital step in preparing the dataset for training.

- *Flattening the dataset:*

  In the CIFAR-10 dataset, the labels are initially stored as 2D arrays (e.g., [[1], [5], [7], ...]), which are not compatible with many machine learning libraries. To address this, the labels are flattened into a 1D array (e.g., [1, 5, 7, ...]). This conversion ensures the labels are in the correct format, making them suitable for use in the model and enabling seamless training and evaluation processes.



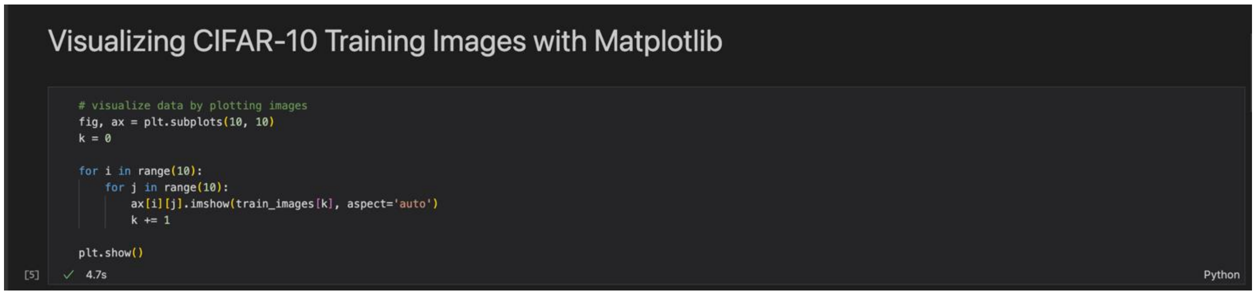
Normalizing CIFAR-10 Images

```
train_images, test_images = train_images / 255.0, test_images / 255.0

# flatten the label values
train_labels, test_labels = train_labels.flatten(), test_labels.flatten()
```

[3] ✓ 0.5s Python

- *Visualizing the dataset*

  This code visualizes a grid of 100 images from the CIFAR-10 training dataset using Matplotlib. It creates a 10x10 grid of subplots and iteratively displays images from the train_images dataset in each subplot. The imshow() function is used to render the images, ensuring they fit properly in each grid cell. This visualization is useful for inspecting the dataset, verifying that the images are loaded correctly, and understanding the diversity of the CIFAR-10 dataset.



Visualizing CIFAR-10 Training Images with Matplotlib

```python
# visualize data by plotting images
fig, ax = plt.subplots(10, 10)
k = 0

for i in range(10):
    for j in range(10):
        ax[i][j].imshow(train_images[k], aspect='auto')
        k += 1

plt.show()
```

[5] ✓ 4.7s Python

- *Output of the dataset*

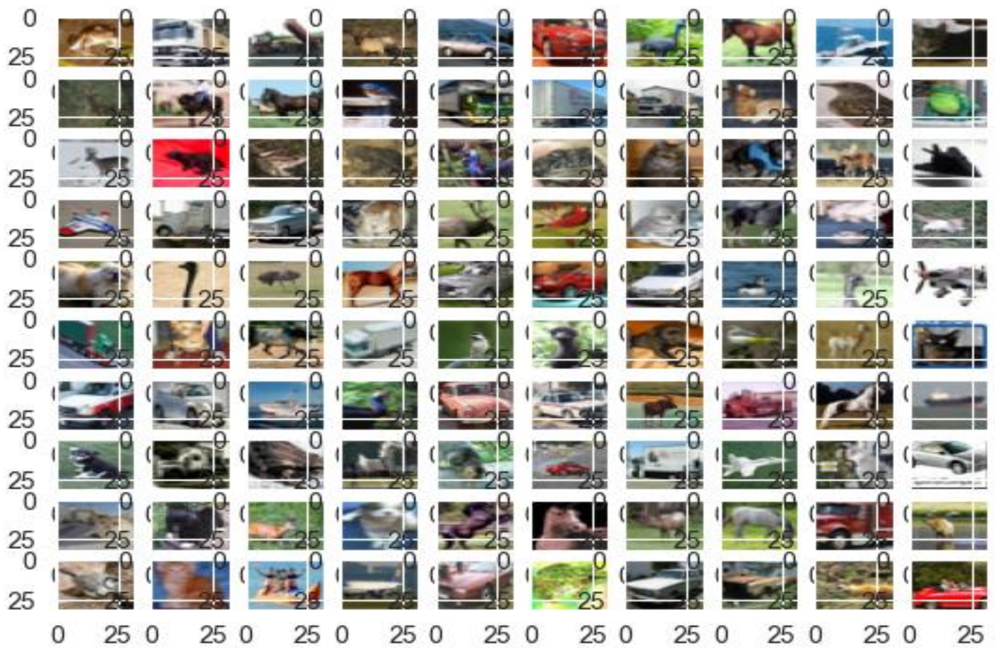
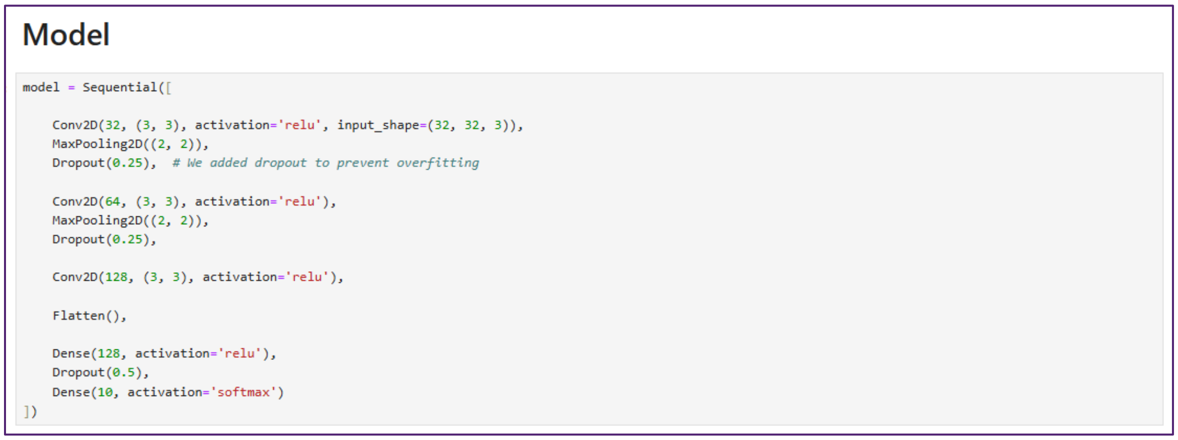


Figure 1: Snippet of the Dataset CIFAR-10

2. Model Implementation(terrible)

- *Building the CNN model*



Model

```
model = Sequential([

    Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)),
    MaxPooling2D((2, 2)),
    Dropout(0.25),  # We added dropout to prevent overfitting

    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    Dropout(0.25),

    Conv2D(128, (3, 3), activation='relu'),

    Flatten(),

    Dense(128, activation='relu'),
    Dropout(0.5),
    Dense(10, activation='softmax')
])
```

- As you can see in this initial CNN model, we tried following the example of the video tutorial, using the *Sequential* API of a deep learning framework and only 3 convolutional blocks:

- *First Convolutional Block:*

  The first convolutional block begins with a **Conv2D layer** that applies 32 filters of size 3x3, using "relu" as the **activation function**. This layer expects input images with a shape of 32x32 pixels and 3 color channels (RGB).

  Following the convolution, a **MaxPooling2D layer** with a 2x2 window is used to reduce the spatial dimensions of the feature maps, helping to downsample the data.

  To prevent overfitting, a **Dropout layer** is included, which randomly drops 25% of the nodes during training.

- *Second Convolutional Block*

  Follows the same pattern, except its 2D convolutional layer applies 64 filters of size 3x3 and Relu activation.

- *Third Convolutional Block*

Is the same except for its 2D convolutional layer, it applies 128 filters of size 3x3 and Relu activation.

- *Flattening Layer*

Flattens the multi-dimensional tensor output from the previous layer into a single vector, making it suitable for dense layers.

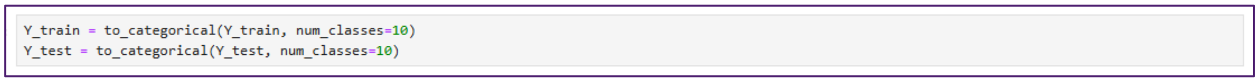
- *Fully Connected Layers*

`Dense(128, activation='relu')`: A dense (fully connected) layer with 128 units and Relu activation.

`Dropout(0.5)`: Drops 50% of the nodes to prevent overfitting.

`Dense(10, activation='softmax')`: The final dense layer with 10 units (representing 10 classes) and a softmax activation function to output class probabilities.

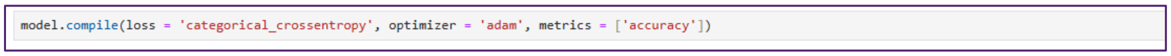
- *Pre-Processing*



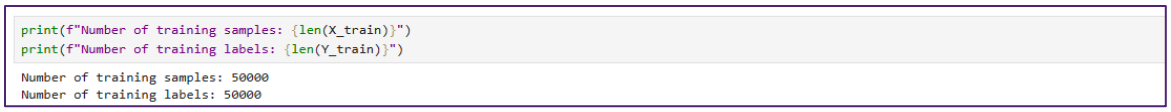
```
Y_train = to_categorical(Y_train, num_classes=10)
Y_test = to_categorical(Y_test, num_classes=10)
```

➔ Converts the labels into a one-hot encoded format for classification into 10 classes.

- *Compiling*

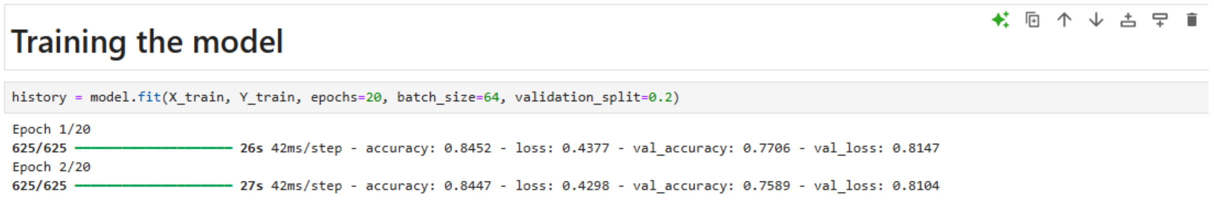


```
model.compile(loss = 'categorical_crossentropy', optimizer = 'adam', metrics = ['accuracy'])
```



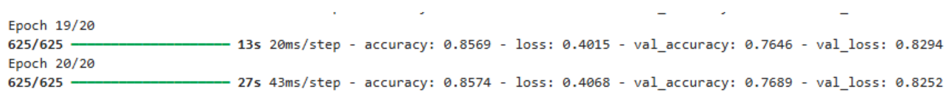
```
print(f"Number of training samples: {len(X_train)}")
print(f"Number of training labels: {len(Y_train)}")

Number of training samples: 50000
Number of training labels: 50000
```

- *Training*



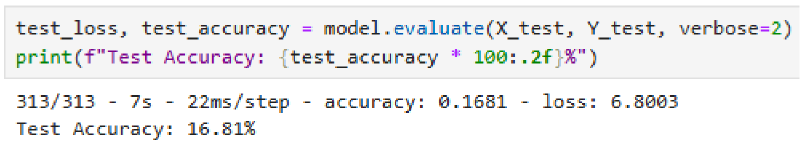
Training the model

```
history = model.fit(X_train, Y_train, epochs=20, batch_size=64, validation_split=0.2)

Epoch 1/20
625/625 ━━━━━━━━━━━━━━━━━━━━ 26s 42ms/step - accuracy: 0.8452 - loss: 0.4377 - val_accuracy: 0.7706 - val_loss: 0.8147
Epoch 2/20
625/625 ━━━━━━━━━━━━━━━━━━━━ 27s 42ms/step - accuracy: 0.8447 - loss: 0.4298 - val_accuracy: 0.7589 - val_loss: 0.8104
```



```
Epoch 19/20
625/625 ━━━━━━━━━━━━━━━━━━━━ 13s 20ms/step - accuracy: 0.8569 - loss: 0.4015 - val_accuracy: 0.7646 - val_loss: 0.8294
Epoch 20/20
625/625 ━━━━━━━━━━━━━━━━━━━━ 27s 43ms/step - accuracy: 0.8574 - loss: 0.4068 - val_accuracy: 0.7689 - val_loss: 0.8252
```

- *Results*



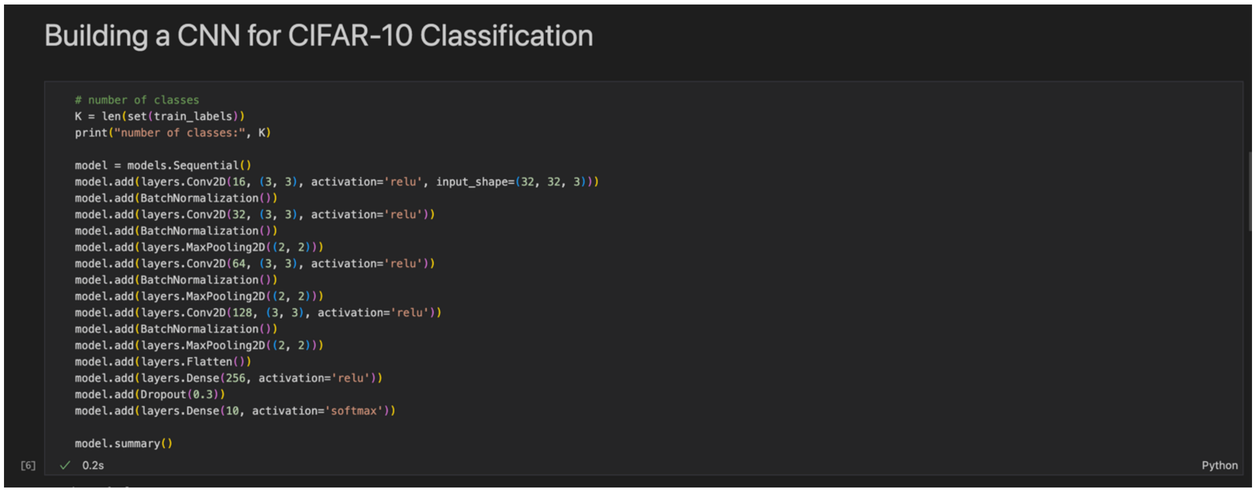
```
test_loss, test_accuracy = model.evaluate(X_test, Y_test, verbose=2)
print(f"Test Accuracy: {test_accuracy * 100:.2f}%")
```

```
313/313 - 7s - 22ms/step - accuracy: 0.1681 - loss: 6.8003
Test Accuracy: 16.81%
```

➔ Obviously, the results were poor, with a test accuracy of just 16.81%, despite achieving a training accuracy of 85.74%. To address these disappointing results, we took a different approach by modifying the model's structure and experimenting with the configuration of layers and neurons.

3. First enhancement (77.56%)

- *Building the CNN Model:*



Building a CNN for CIFAR-10 Classification

```python
# number of classes
K = len(set(train_labels))
print("number of classes:", K)

model = models.Sequential()
model.add(layers.Conv2D(16, (3, 3), activation='relu', input_shape=(32, 32, 3)))
model.add(BatchNormalization())
model.add(layers.Conv2D(32, (3, 3), activation='relu'))
model.add(BatchNormalization())
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(BatchNormalization())
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(BatchNormalization())
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Flatten())
model.add(layers.Dense(256, activation='relu'))
model.add(Dropout(0.3))
model.add(layers.Dense(10, activation='softmax'))

model.summary()
```

```
[6]  ✓ 0.2s                                                              Python
```

After the apparent failure with the first model, we upgraded the model implementation. This CNN model for CIFAR-10 classification is built using a series of layers, each with a specific role in transforming the input images into meaningful patterns for classification. The network is designed to extract features, reduce complexity, and predict probabilities for the 10 classes in the dataset, with a focus on simplicity for basic performance. We also used a different implementation style in the new model, we built it layer by layer using the add() method as shown in the video tutorial, providing more flexibility and clarity in constructing the architecture.

This model is simple, serving as a starting point to establish a performing model. The layer choices and values (e.g., filter sizes and dropout rate) are designed to provide a balance of simplicity and functionality, with plans to adjust them later to improve accuracy as the project progresses.

- *Convolutional Layers*

The convolutional layers are the backbone of the CNN, applying filters to the input images to detect features like edges, shapes, and textures. As the network goes deeper, the number of filters increases (16, 32, 64, 128), allowing the model to capture more complex patterns. These layers are crucial for extracting the visual features necessary for classification tasks.

- *Batch Normalization*

  Batch normalization normalizes the outputs of each convolutional layer to have a consistent mean and variance, stabilizing the learning process. This layer speeds up training, allows higher learning rates, and reduces sensitivity to weight initialization, making the model more robust and efficient.

- *MaxPooling Layers*

  MaxPooling layers reduce the spatial dimensions of the feature maps while retaining the most critical information by selecting the maximum value from each region. This helps lower computational complexity, makes the model less sensitive to small distortions in the images, and prevents overfitting by focusing on prominent features.

- *Dense Layer*

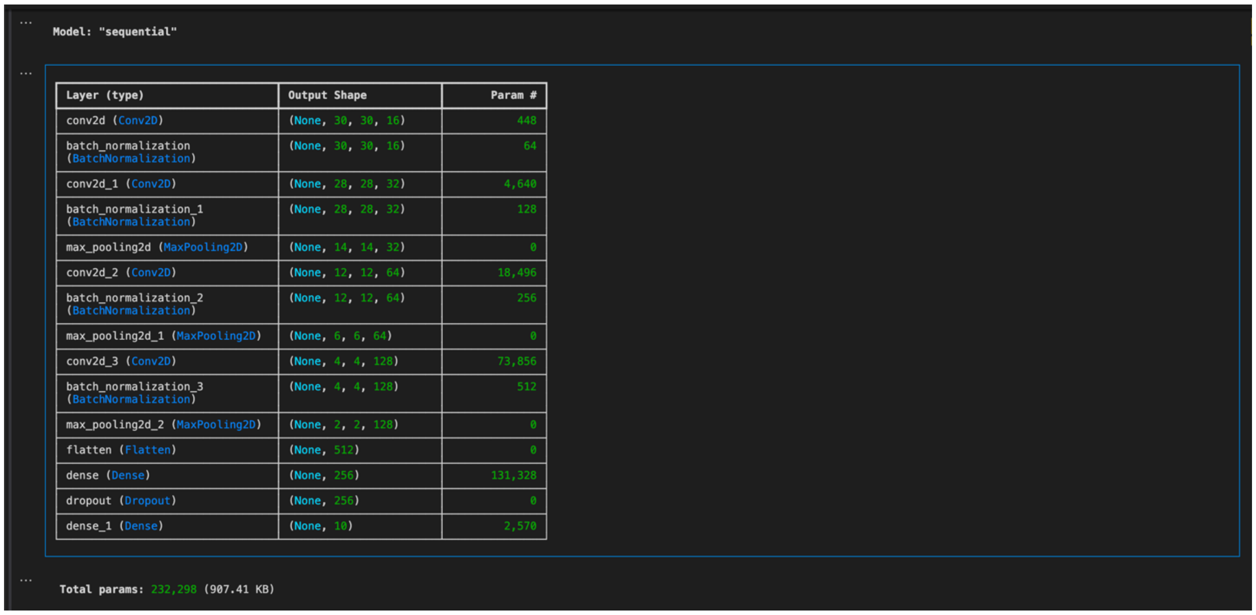
  The dense layer with 256 neurons connects all extracted features and learns high-level patterns to make predictions. This layer consolidates the information from the convolutional layers and maps it to the final decision space, enabling the network to classify the images effectively.

- *Dropout Layer*

  The dropout layer randomly disables 30% of neurons during training to prevent overfitting. By forcing the model to rely on different combinations of neurons, dropout ensures that the network learns robust and generalized features instead of memorizing the training data.

- *Output Layer*

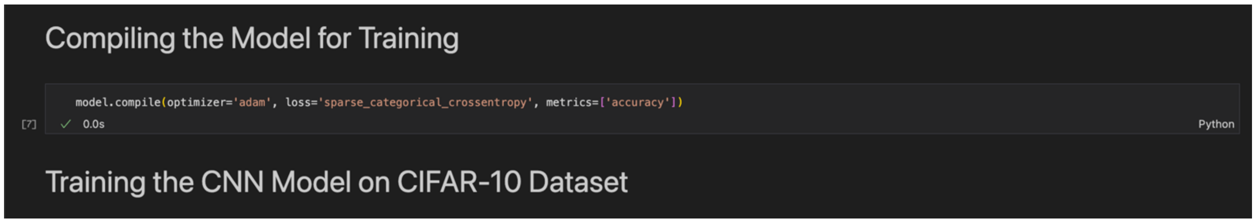
  The output layer contains 10 neurons corresponding to the 10 classes in CIFAR-10 and uses a softmax activation function. This function converts the raw outputs into probabilities, ensuring the network predicts the most likely class for each input image.



```
Model: "sequential"

┌──────────────────────────────────┬────────────────────────┬───────────┐
│ Layer (type)                     │ Output Shape           │   Param # │
├──────────────────────────────────┼────────────────────────┼───────────┤
│ conv2d (Conv2D)                  │ (None, 30, 30, 16)     │       448 │
├──────────────────────────────────┼────────────────────────┼───────────┤
│ batch_normalization              │ (None, 30, 30, 16)     │        64 │
│ (BatchNormalization)             │                        │           │
├──────────────────────────────────┼────────────────────────┼───────────┤
│ conv2d_1 (Conv2D)                │ (None, 28, 28, 32)     │     4,640 │
├──────────────────────────────────┼────────────────────────┼───────────┤
│ batch_normalization_1            │ (None, 28, 28, 32)     │       128 │
│ (BatchNormalization)             │                        │           │
├──────────────────────────────────┼────────────────────────┼───────────┤
│ max_pooling2d (MaxPooling2D)     │ (None, 14, 14, 32)     │         0 │
├──────────────────────────────────┼────────────────────────┼───────────┤
│ conv2d_2 (Conv2D)                │ (None, 12, 12, 64)     │    18,496 │
├──────────────────────────────────┼────────────────────────┼───────────┤
│ batch_normalization_2            │ (None, 12, 12, 64)     │       256 │
│ (BatchNormalization)             │                        │           │
├──────────────────────────────────┼────────────────────────┼───────────┤
│ max_pooling2d_1 (MaxPooling2D)   │ (None, 6, 6, 64)       │         0 │
├──────────────────────────────────┼────────────────────────┼───────────┤
│ conv2d_3 (Conv2D)                │ (None, 4, 4, 128)      │    73,856 │
├──────────────────────────────────┼────────────────────────┼───────────┤
│ batch_normalization_3            │ (None, 4, 4, 128)      │       512 │
│ (BatchNormalization)             │                        │           │
├──────────────────────────────────┼────────────────────────┼───────────┤
│ max_pooling2d_2 (MaxPooling2D)   │ (None, 2, 2, 128)      │         0 │
├──────────────────────────────────┼────────────────────────┼───────────┤
│ flatten (Flatten)                │ (None, 512)            │         0 │
├──────────────────────────────────┼────────────────────────┼───────────┤
│ dense (Dense)                    │ (None, 256)            │   131,328 │
├──────────────────────────────────┼────────────────────────┼───────────┤
│ dropout (Dropout)                │ (None, 256)            │         0 │
├──────────────────────────────────┼────────────────────────┼───────────┤
│ dense_1 (Dense)                  │ (None, 10)             │     2,570 │
└──────────────────────────────────┴────────────────────────┴───────────┘

 Total params: 232,298 (907.41 KB)
```

This is the **model summary** output generated by calling model.summary() in a deep learning framework like TensorFlow or Keras. It provides a detailed overview of the architecture of the CNN built for the CIFAR-10 classification task.

- *Training the Model*

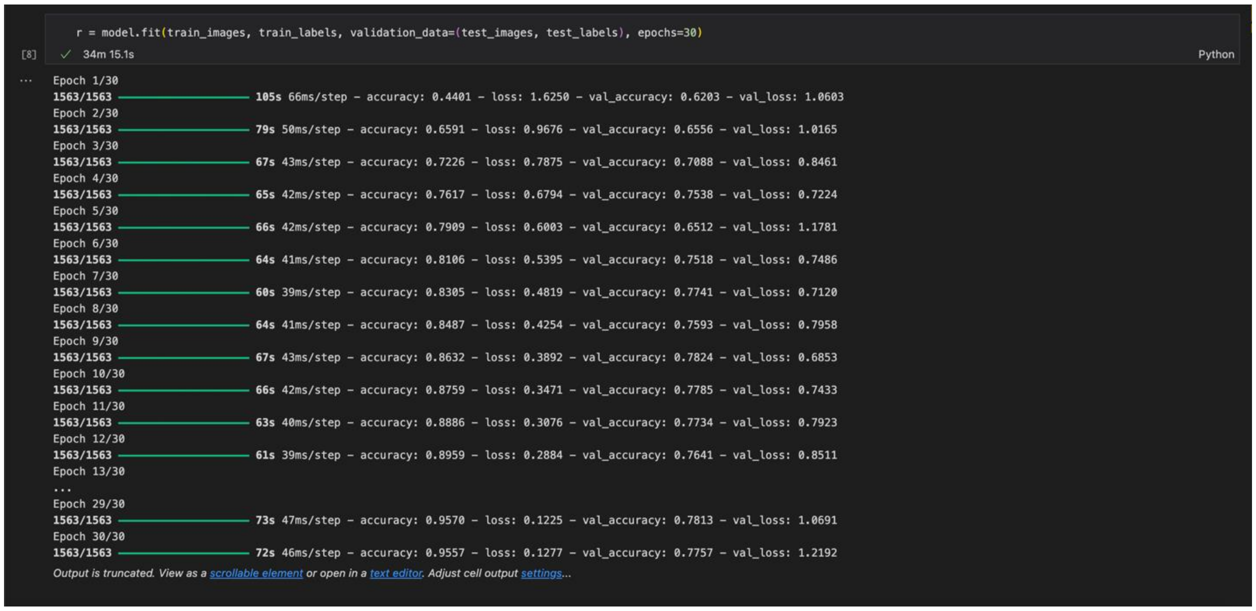


Compiling the Model for Training

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

[7] ✓ 0.0s Python

Training the CNN Model on CIFAR-10 Dataset



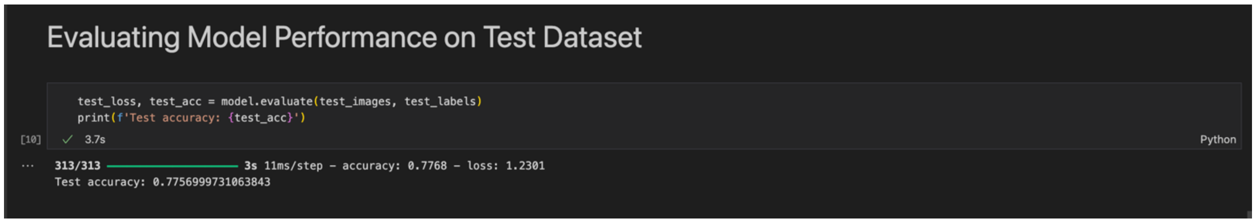
```
r = model.fit(train_images, train_labels, validation_data=(test_images, test_labels), epochs=30)
```

```
[8]  ✓ 34m 15.1s                                                                                Python
```

```
Epoch 1/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 105s 66ms/step - accuracy: 0.4401 - loss: 1.6250 - val_accuracy: 0.6203 - val_loss: 1.0603
Epoch 2/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 79s 50ms/step - accuracy: 0.6591 - loss: 0.9676 - val_accuracy: 0.6556 - val_loss: 1.0165
Epoch 3/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 67s 43ms/step - accuracy: 0.7226 - loss: 0.7875 - val_accuracy: 0.7088 - val_loss: 0.8461
Epoch 4/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 65s 42ms/step - accuracy: 0.7617 - loss: 0.6794 - val_accuracy: 0.7538 - val_loss: 0.7224
Epoch 5/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 66s 42ms/step - accuracy: 0.7909 - loss: 0.6003 - val_accuracy: 0.6512 - val_loss: 1.1781
Epoch 6/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 64s 41ms/step - accuracy: 0.8106 - loss: 0.5395 - val_accuracy: 0.7518 - val_loss: 0.7486
Epoch 7/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 60s 39ms/step - accuracy: 0.8305 - loss: 0.4819 - val_accuracy: 0.7741 - val_loss: 0.7120
Epoch 8/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 64s 41ms/step - accuracy: 0.8487 - loss: 0.4254 - val_accuracy: 0.7593 - val_loss: 0.7958
Epoch 9/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 67s 43ms/step - accuracy: 0.8632 - loss: 0.3892 - val_accuracy: 0.7824 - val_loss: 0.6853
Epoch 10/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 66s 42ms/step - accuracy: 0.8759 - loss: 0.3471 - val_accuracy: 0.7785 - val_loss: 0.7433
Epoch 11/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 63s 40ms/step - accuracy: 0.8886 - loss: 0.3076 - val_accuracy: 0.7734 - val_loss: 0.7923
Epoch 12/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 61s 39ms/step - accuracy: 0.8959 - loss: 0.2884 - val_accuracy: 0.7641 - val_loss: 0.8511
Epoch 13/30
...
Epoch 29/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 73s 47ms/step - accuracy: 0.9570 - loss: 0.1225 - val_accuracy: 0.7813 - val_loss: 1.0691
Epoch 30/30
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 72s 46ms/step - accuracy: 0.9557 - loss: 0.1277 - val_accuracy: 0.7757 - val_loss: 1.2192
Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...
```

- We got an accuracy of 95.57% when training the model.

- *Testing the Model*



Evaluating Model Performance on Test Dataset

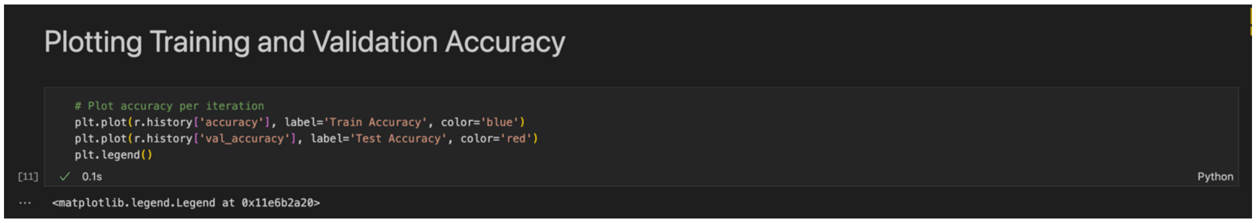
```
test_loss, test_acc = model.evaluate(test_images, test_labels)
print(f'Test accuracy: {test_acc}')
```

```
[10]  ✓ 3.7s                                                                               Python
```

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 3s 11ms/step - accuracy: 0.7768 - loss: 1.2301
Test accuracy: 0.7756999731063843
```

- After testing the model, we got an accuracy of **77.56%** which is better than the first implementation but still **not the best.**

- *Plotting Training and Testing Accuracies*

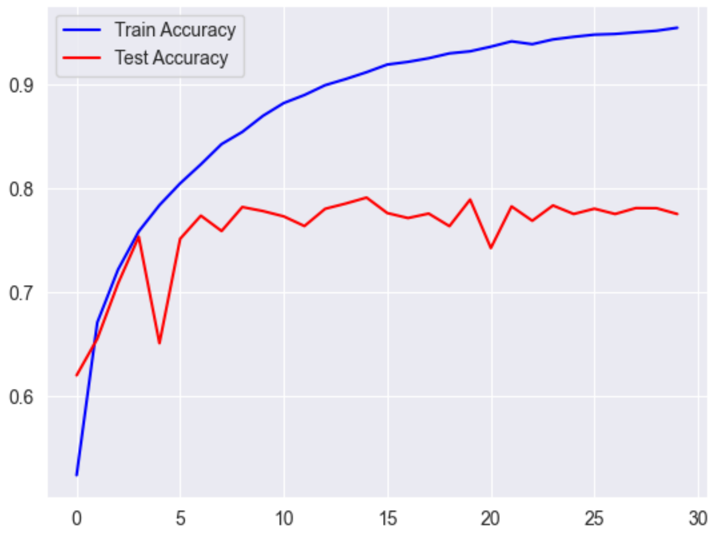


Plotting Training and Validation Accuracy

```
# Plot accuracy per iteration
plt.plot(r.history['accuracy'], label='Train Accuracy', color='blue')
plt.plot(r.history['val_accuracy'], label='Test Accuracy', color='red')
plt.legend()
```

```
[11]  ✓ 0.1s                                                                               Python
```

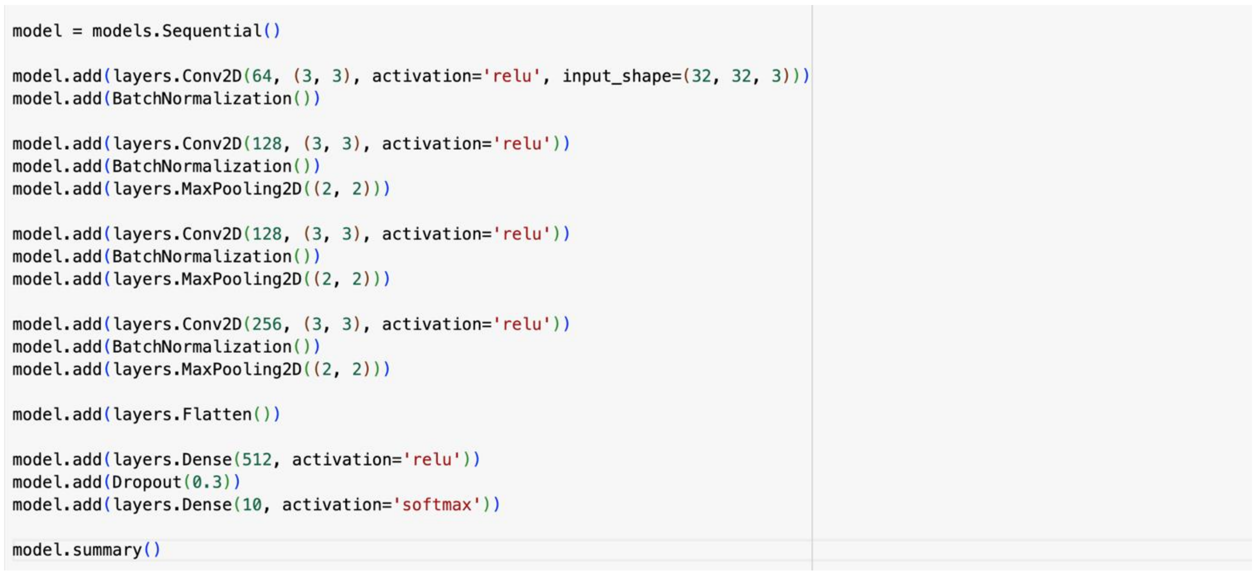
```
<matplotlib.legend.Legend at 0x11e6b2a20>
```



- This plot shows the training (blue) and testing (red) accuracy of the CNN over multiple epochs. The training accuracy steadily increases, eventually nearing 100%, indicating that the model learns the patterns in the training data well. However, the testing accuracy plateaus and fluctuates, showing a gap between the two curves. This suggests **overfitting**, where the model performs well on training data but struggles to generalize to unseen data. To address this, techniques like increasing regularization, applying data augmentation, or using early stopping could help improve generalization and narrow the gap between training and testing performance. This is exactly what we are going to use in the next models which are improved versions of this one.

4. Second Enhancement (80.5%)

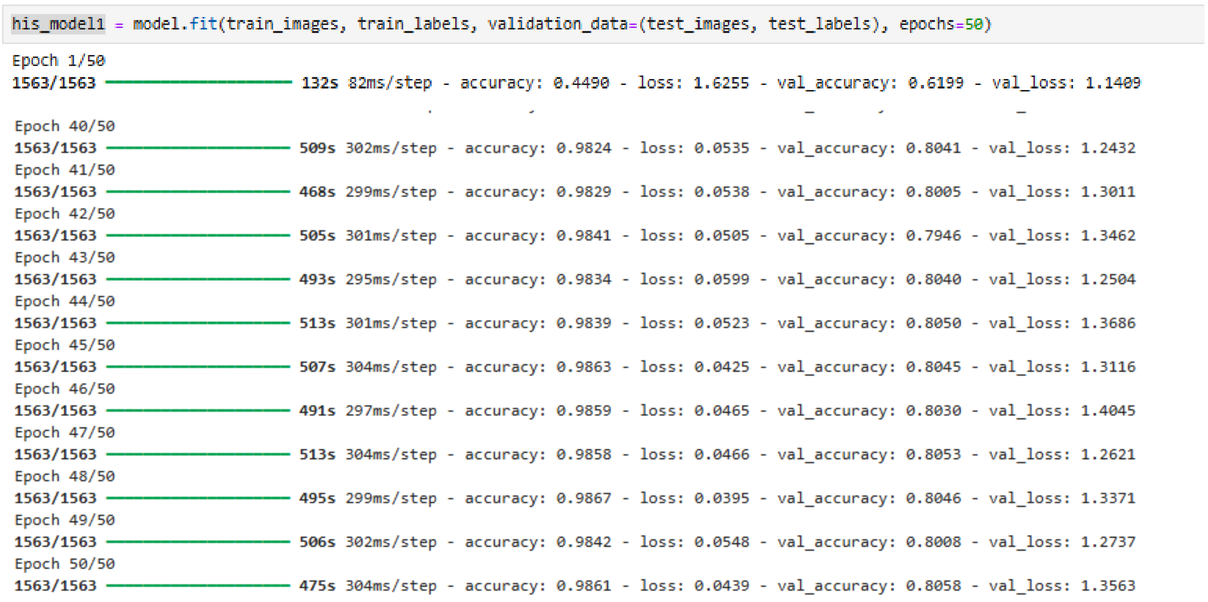
- *Building the CNN Model:*



```
model = models.Sequential()

model.add(layers.Conv2D(64, (3, 3), activation='relu', input_shape=(32, 32, 3)))
model.add(BatchNormalization())

model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(BatchNormalization())
model.add(layers.MaxPooling2D((2, 2)))

model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(BatchNormalization())
model.add(layers.MaxPooling2D((2, 2)))

model.add(layers.Conv2D(256, (3, 3), activation='relu'))
model.add(BatchNormalization())
model.add(layers.MaxPooling2D((2, 2)))

model.add(layers.Flatten())

model.add(layers.Dense(512, activation='relu'))
model.add(Dropout(0.3))
model.add(layers.Dense(10, activation='softmax'))

model.summary()
```

- This new model steps things up by starting with 64 filters, two additional convolutional blocks with 128, and adding an **extra convolutional layer with 256 filters,** giving it the ability to capture even more detailed and complex patterns in the data. Think of it as giving the model sharper tools to zoom in on the finer details.
- The **dense layer with 512 units** is still here, making sure the model can pull all those learned features together and make smarter predictions. To keep things in check and prevent them from getting too attached to the training data, we've added a **Dropout layer** to mix things up during training.
- Overall, this model is designed to be more powerful and insightful, capable of tackling more complex tasks. Sure, it takes more time and resources to train, but we've added just the right amount of **regularization** to strike a balance between performance and reliability.

- *Training the Model*



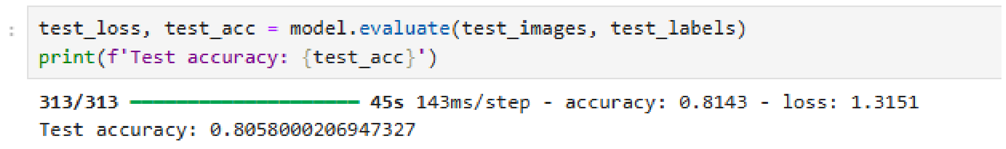
```
his_model1 = model.fit(train_images, train_labels, validation_data=(test_images, test_labels), epochs=50)
```

```
Epoch 1/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 132s 82ms/step - accuracy: 0.4490 - loss: 1.6255 - val_accuracy: 0.6199 - val_loss: 1.1409

Epoch 40/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 509s 302ms/step - accuracy: 0.9824 - loss: 0.0535 - val_accuracy: 0.8041 - val_loss: 1.2432
Epoch 41/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 468s 299ms/step - accuracy: 0.9829 - loss: 0.0538 - val_accuracy: 0.8005 - val_loss: 1.3011
Epoch 42/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 505s 301ms/step - accuracy: 0.9841 - loss: 0.0505 - val_accuracy: 0.7946 - val_loss: 1.3462
Epoch 43/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 493s 295ms/step - accuracy: 0.9834 - loss: 0.0599 - val_accuracy: 0.8040 - val_loss: 1.2504
Epoch 44/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 513s 301ms/step - accuracy: 0.9839 - loss: 0.0523 - val_accuracy: 0.8050 - val_loss: 1.3686
Epoch 45/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 507s 304ms/step - accuracy: 0.9863 - loss: 0.0425 - val_accuracy: 0.8045 - val_loss: 1.3116
Epoch 46/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 491s 297ms/step - accuracy: 0.9859 - loss: 0.0465 - val_accuracy: 0.8030 - val_loss: 1.4045
Epoch 47/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 513s 304ms/step - accuracy: 0.9858 - loss: 0.0466 - val_accuracy: 0.8053 - val_loss: 1.2621
Epoch 48/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 495s 299ms/step - accuracy: 0.9867 - loss: 0.0395 - val_accuracy: 0.8046 - val_loss: 1.3371
Epoch 49/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 506s 302ms/step - accuracy: 0.9842 - loss: 0.0548 - val_accuracy: 0.8008 - val_loss: 1.2737
Epoch 50/50
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 475s 304ms/step - accuracy: 0.9861 - loss: 0.0439 - val_accuracy: 0.8058 - val_loss: 1.3563
```

- We used 50 epochs and the training accuracy went up to 98.61%

- *Testing the Model*

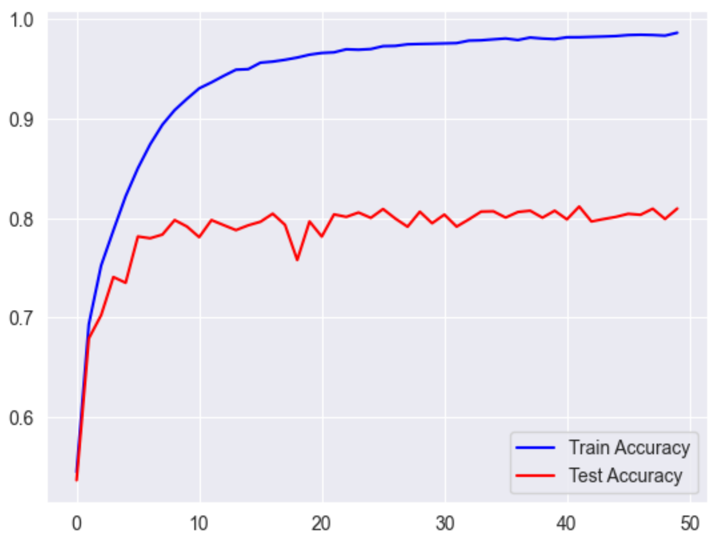
We observe that the modifications we made to the previous model increased the accuracy of our model.



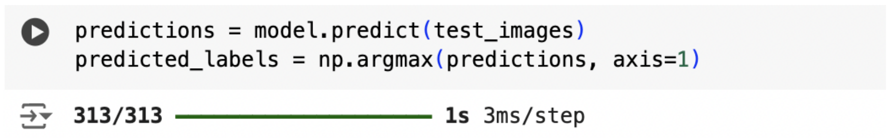
```
test_loss, test_acc = model.evaluate(test_images, test_labels)
print(f'Test accuracy: {test_acc}')
```

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 45s 143ms/step - accuracy: 0.8143 - loss: 1.3151
Test accuracy: 0.8058000206947327
```

- Testing accuracy: **80.5%.** We went from 77.56% to 80.5%, which is a good improvement, **but we know that we can do better**

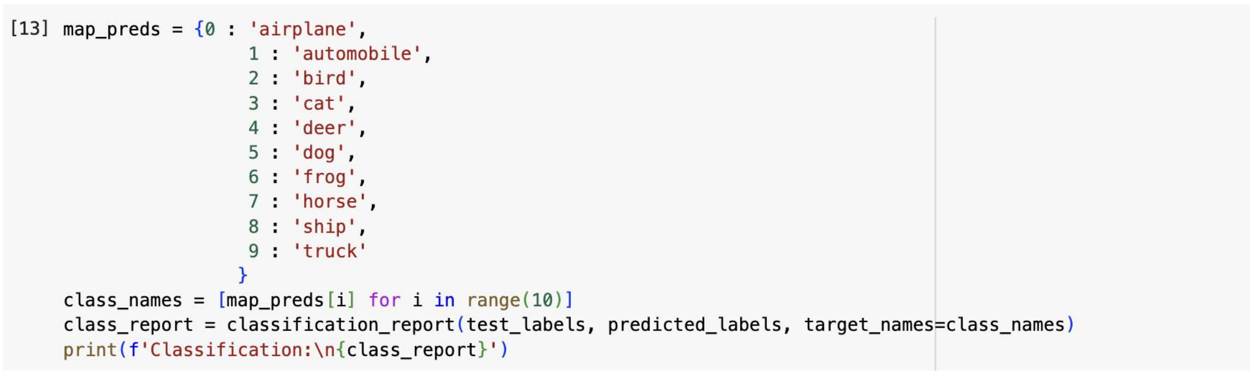


- *Model Prediction and Evaluation:*



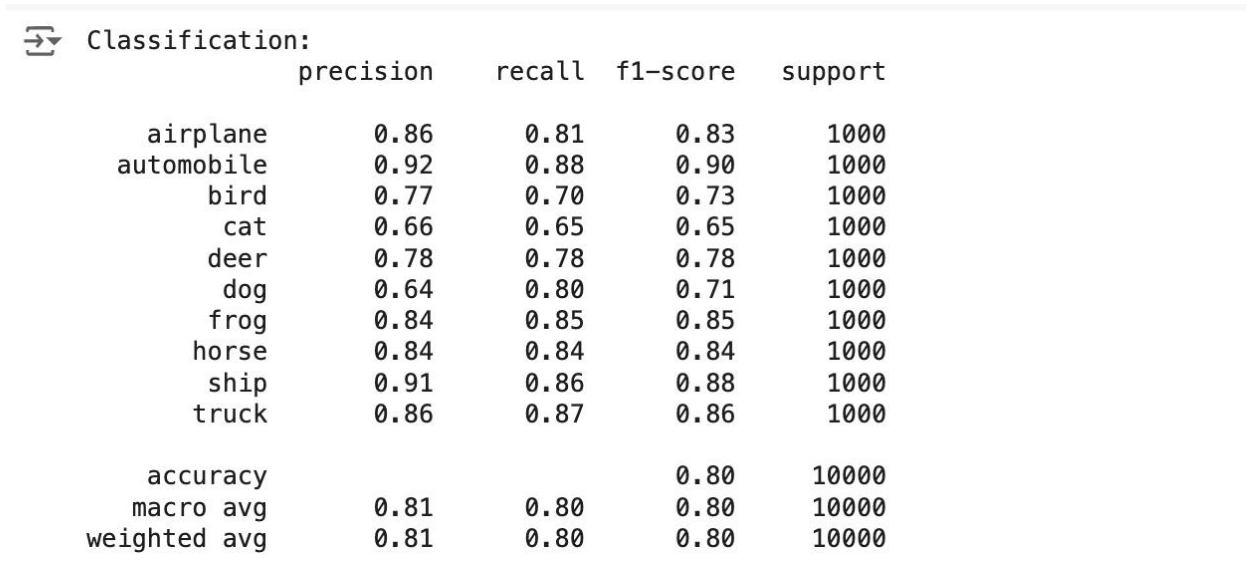
```
predictions = model.predict(test_images)
predicted_labels = np.argmax(predictions, axis=1)
```

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step
```



```
[13] map_preds = {0 : 'airplane',
                  1 : 'automobile',
                  2 : 'bird',
                  3 : 'cat',
                  4 : 'deer',
                  5 : 'dog',
                  6 : 'frog',
                  7 : 'horse',
                  8 : 'ship',
                  9 : 'truck'
                 }
     class_names = [map_preds[i] for i in range(10)]
     class_report = classification_report(test_labels, predicted_labels, target_names=class_names)
     print(f'Classification:\n{class_report}')
```

After training the model, predictions are made on the test dataset. The model's predictions are converted into class labels, and a classification report is generated to provide detailed insights into the model's performance in each class.



```
Classification:
              precision    recall  f1-score   support

    airplane       0.86      0.81      0.83      1000
  automobile       0.92      0.88      0.90      1000
        bird       0.77      0.70      0.73      1000
         cat       0.66      0.65      0.65      1000
        deer       0.78      0.78      0.78      1000
         dog       0.64      0.80      0.71      1000
        frog       0.84      0.85      0.85      1000
       horse       0.84      0.84      0.84      1000
        ship       0.91      0.86      0.88      1000
       truck       0.86      0.87      0.86      1000

    accuracy                           0.80     10000
   macro avg       0.81      0.80      0.80     10000
weighted avg       0.81      0.80      0.80     10000
```

This classification report provides a detailed analysis of the model’s performance across different classes.

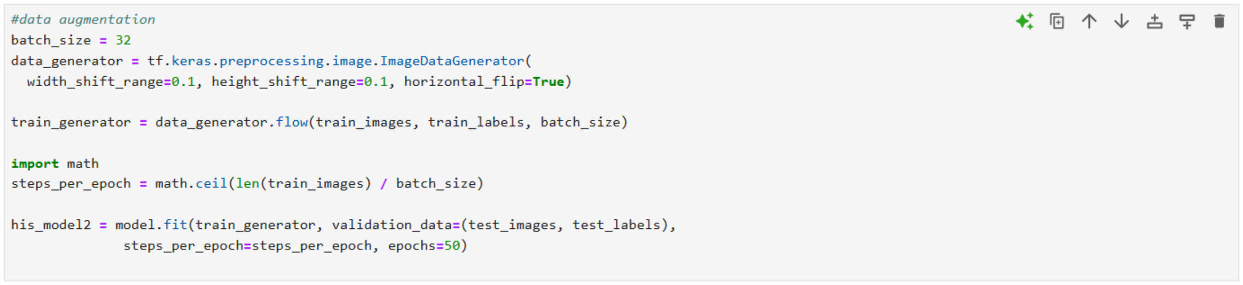
- **Precision**: Precision tells us how accurate the model is when it predicts something positive. It's the percentage of correct positive predictions out of all positive predictions made. The values range from **0.64 for 'dog' to 0.92 for 'automobile'**, meaning the model is better at identifying some classes than others.
- **Recall**: Recall, or sensitivity, measures how good the model is at catching all the correct instances for a class. It ranges from **0.65 for 'cat' to 0.88 for 'automobile'**, showing that while the model does well for some categories, it misses more instances for others.
- **F1-Score**: The F1-score balances precision and recall, giving us an overall picture of the model's performance for each class. It varies from **0.65 for 'cat' to 0.90 for 'automobile'**, highlighting that the model is well-balanced for some classes but struggles with others.
- **Overall Accuracy**: The model has an accuracy of **80%**, meaning 8 out of 10 predictions are correct across all classes.
- **Macro and Weighted Averages**:

- **Macro Average**: With precision, recall, and F1-score at **0.81**, this shows the model's performance averaged equally across all classes.
- **Weighted Average**: Also at **0.81**, this reflects the model's performance while considering the importance of larger classes.

In short, the model performs well overall, with some classes like 'automobile' and 'ship' standing out, while others like 'cat' and 'dog' need improvement. The 80% accuracy and balanced averages indicate the model is solid but could be fine-tuned for better consistency across all categories. This is exactly what we did in the next section.

5. Third Enhancement – Data Augmentation (88.4%)

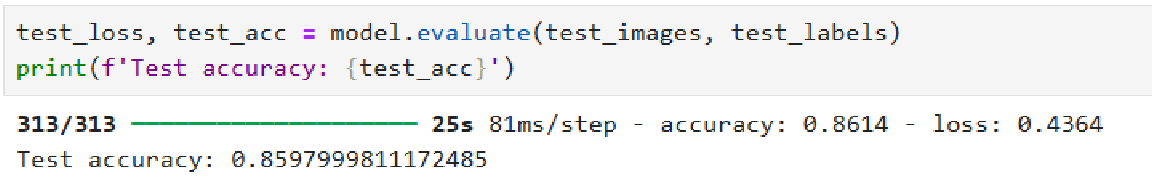
While working on this, we found out about data augmentation which is a technique used in machine learning to artificially expand the size of a training dataset by applying random transformations to the original data. These transformations, such as rotations, flips, shifts, zooms, or changes in brightness, create modified versions of the input data while preserving their labels. This process helps improve model generalization by exposing it to a wider variety of examples, reducing overfitting, and making it more robust to variations in real-world data



```
#data augmentation
batch_size = 32
data_generator = tf.keras.preprocessing.image.ImageDataGenerator(
  width_shift_range=0.1, height_shift_range=0.1, horizontal_flip=True)

train_generator = data_generator.flow(train_images, train_labels, batch_size)

import math
steps_per_epoch = math.ceil(len(train_images) / batch_size)

his_model2 = model.fit(train_generator, validation_data=(test_images, test_labels),
              steps_per_epoch=steps_per_epoch, epochs=50)
```

- With ***"Batch_size"*** The model processes 32 samples per training step.
- The ***data augmentation function*** Creates an ***ImageDataGenerator*** object for data augmentation using these parameters:
    - ***width_shift_range*** = 0.1: Randomly shifts images horizontally by up to 10% of the width.
    - ***height_shift_range*** = 0.1: Randomly shifts images vertically by up to 10% of the height.
    - ***horizontal_flip*** = True: Randomly flips images horizontally.
- Then ***"train_generator"*** generates batches of augmented images and their corresponding labels in real time during training.

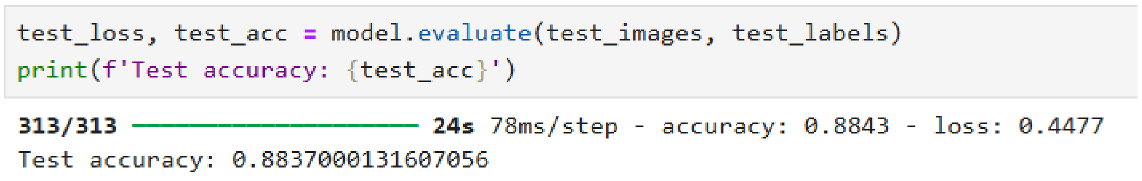
- *Testing Results:*



```
test_loss, test_acc = model.evaluate(test_images, test_labels)
print(f'Test accuracy: {test_acc}')
```

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 25s 81ms/step - accuracy: 0.8614 - loss: 0.4364
Test accuracy: 0.8597999811172485
```

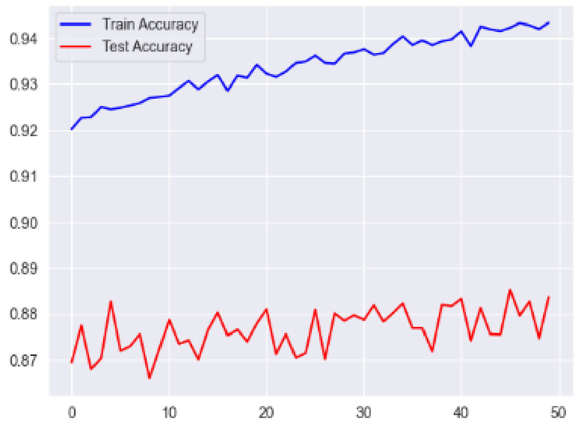
- We went from 80.2% to 85.9%, which is a good improvement, but we knew that we could do even better that is why we ran it again for about 70 epochs more and got a better accuracy:



```
test_loss, test_acc = model.evaluate(test_images, test_labels)
print(f'Test accuracy: {test_acc}')
```

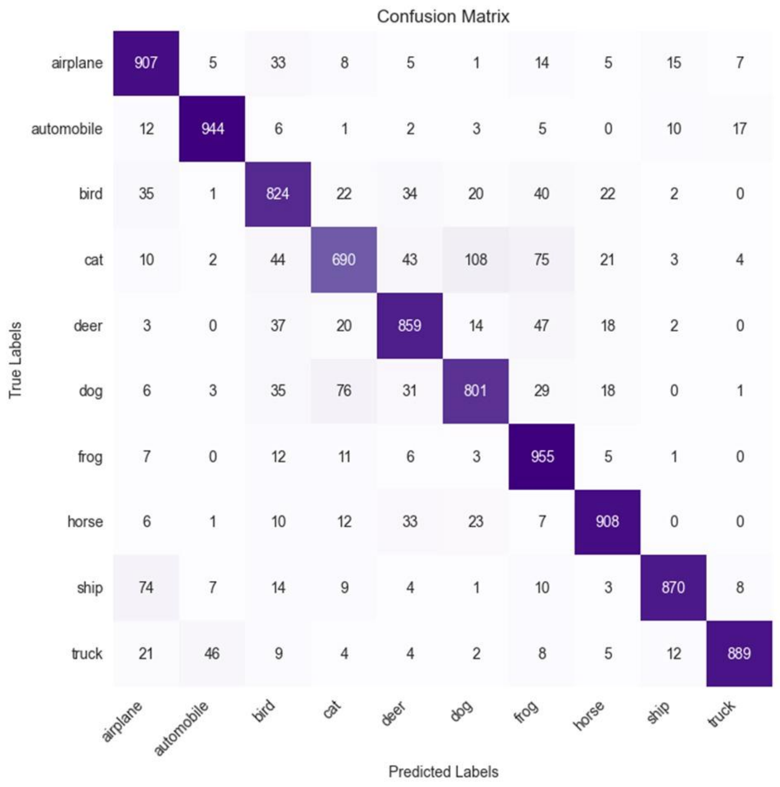
```
313/313 ━━━━━━━━━━━━━━━━━━━━ 24s 78ms/step - accuracy: 0.8843 - loss: 0.4477
Test accuracy: 0.8837000131607056
```

- Testing accuracy: **88.4%,** and just like that, we went from 85.9% to **88.4%**



This is the best accuracy we were able to get! After trying to improve this accuracy, our model started overfitting.

- *Confusion Matrix:*



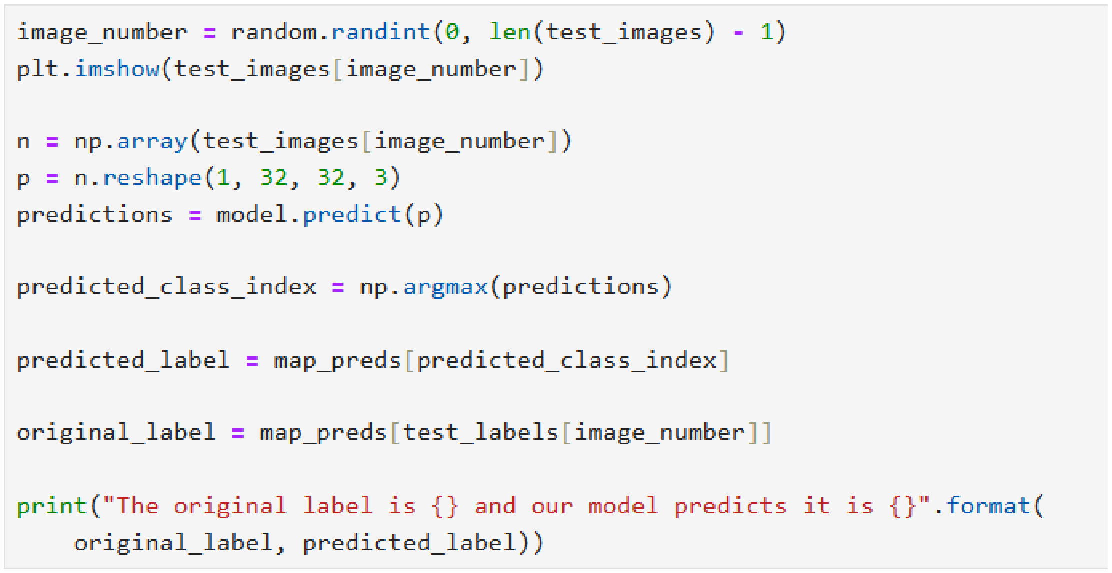
Confusion Matrix

True Labels \ Predicted Labels	airplane	automobile	bird	cat	deer	dog	frog	horse	ship	truck
airplane	907	5	33	8	5	1	14	5	15	7
automobile	12	944	6	1	2	3	5	0	10	17
bird	35	1	824	22	34	20	40	22	2	0
cat	10	2	44	690	43	108	75	21	3	4
deer	3	0	37	20	859	14	47	18	2	0
dog	6	3	35	76	31	801	29	18	0	1
frog	7	0	12	11	6	3	955	5	1	0
horse	6	1	10	12	33	23	7	908	0	0
ship	74	7	14	9	4	1	10	3	870	8
truck	21	46	9	4	4	2	8	5	12	889

Each cell in the confusion matrix indicates the number of times a class was predicted versus its actual class, summarizing the classification model's performance over ten categories. The diagonal elements show accurate predictions, demonstrating excellent performance in classes such as "automobile" (944) and "frog" (955). However, categories like "cat" (690), which is frequently mistaken for "dog" (801 times), and "deer" exhibit lower results.   Notable misclassifications are highlighted by off-diagonal elements; for example, "truck" is predicted as "automobile" 46 times and "airplane" as "bird" 33 times. The model shows decent accuracy overall, but it has trouble with some overlapping classes, which suggests room for development.

6. Showcasing the Model’s Performance & Demo:

- *Showcasing the model’s performance:*



```python
image_number = random.randint(0, len(test_images) - 1)
plt.imshow(test_images[image_number])

n = np.array(test_images[image_number])
p = n.reshape(1, 32, 32, 3)
predictions = model.predict(p)

predicted_class_index = np.argmax(predictions)

predicted_label = map_preds[predicted_class_index]

original_label = map_preds[test_labels[image_number]]

print("The original label is {} and our model predicts it is {}".format(
    original_label, predicted_label))
```

This code selects a random image from the test dataset, displays it using Matplotlib, and reshapes it into a format suitable for the model. The image is passed through the trained model to generate predictions, and the class with the highest confidence is determined as the predicted label. It then retrieves the original label of the image from the dataset and compares it with the predicted label. Finally, the true label and the model's prediction are printed, allowing for a quick evaluation of the model's performance on a specific test image.

- *Demo:*

Link to the Demo Video

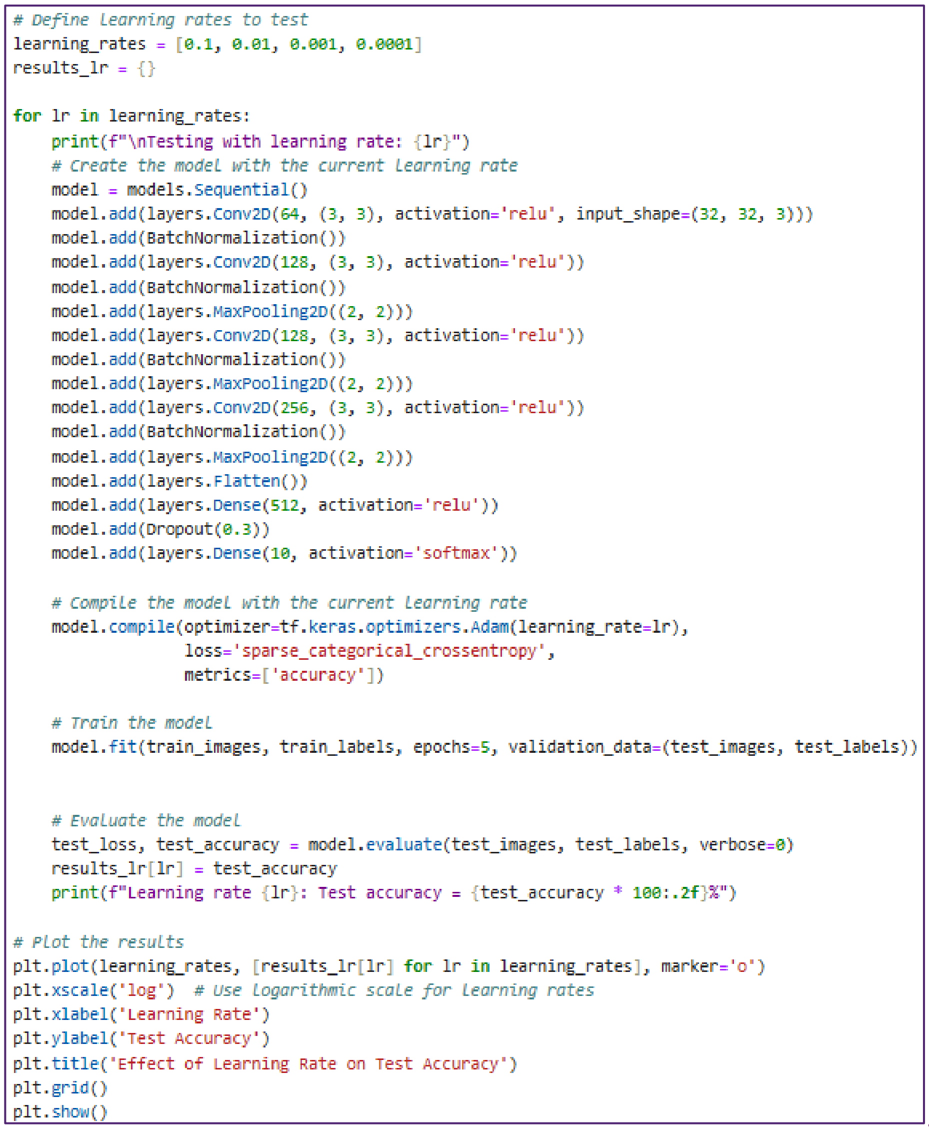
- *Testing the Final Model*

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 50ms/step
The original label is 'airplane' and our model predicts it is 'airplane'
```

IV. Hyperparameters:

- ***Learning rate:***

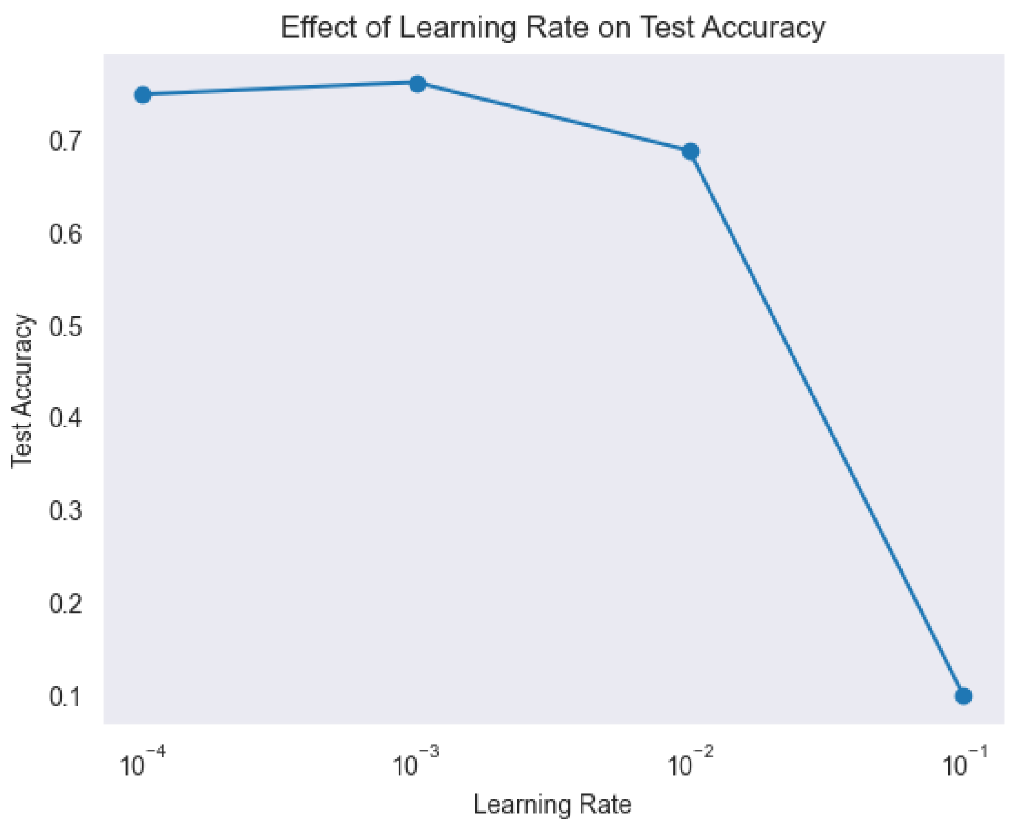
Vary the learning rate while keeping the architecture constant:



```
# Define learning rates to test
learning_rates = [0.1, 0.01, 0.001, 0.0001]
results_lr = {}

for lr in learning_rates:
    print(f"\nTesting with learning rate: {lr}")
    # Create the model with the current learning rate
    model = models.Sequential()
    model.add(layers.Conv2D(64, (3, 3), activation='relu', input_shape=(32, 32, 3)))
    model.add(BatchNormalization())
    model.add(layers.Conv2D(128, (3, 3), activation='relu'))
    model.add(BatchNormalization())
    model.add(layers.MaxPooling2D((2, 2)))
    model.add(layers.Conv2D(128, (3, 3), activation='relu'))
    model.add(BatchNormalization())
    model.add(layers.MaxPooling2D((2, 2)))
    model.add(layers.Conv2D(256, (3, 3), activation='relu'))
    model.add(BatchNormalization())
    model.add(layers.MaxPooling2D((2, 2)))
    model.add(layers.Flatten())
    model.add(layers.Dense(512, activation='relu'))
    model.add(Dropout(0.3))
    model.add(layers.Dense(10, activation='softmax'))

    # Compile the model with the current learning rate
    model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=lr),
                  loss='sparse_categorical_crossentropy',
                  metrics=['accuracy'])

    # Train the model
    model.fit(train_images, train_labels, epochs=5, validation_data=(test_images, test_labels))


    # Evaluate the model
    test_loss, test_accuracy = model.evaluate(test_images, test_labels, verbose=0)
    results_lr[lr] = test_accuracy
    print(f"Learning rate {lr}: Test accuracy = {test_accuracy * 100:.2f}%")

# Plot the results
plt.plot(learning_rates, [results_lr[lr] for lr in learning_rates], marker='o')
plt.xscale('log')  # Use logarithmic scale for learning rates
plt.xlabel('Learning Rate')
plt.ylabel('Test Accuracy')
plt.title('Effect of Learning Rate on Test Accuracy')
plt.grid()
plt.show()
```

:

A key factor in CNN gradient descent optimization is the learning rate. Although a high learning rate has the potential to provide unsatisfactory training outcomes, it also bears the danger of overshooting the ideal minimum, which can result in quick convergence. On the other hand, with a poor learning rate, while making sure of stable convergence, there is a chance that the model will stall and become stuck in local minima.



We can see from the graph the effect of different learning rates on the test accuracy of a model. The x-axis is logarithmic, showing learning rates ranging from 10^{-4} to 10^{-1}, while the y-axis represents test accuracy, ranging from 0 to 1.

➔ *Low Learning Rates 0.0001 to 0.001*

- The model achieves relatively high and stable test accuracy, with values close to 0.75.
- A low learning rate allows the model to make small, incremental updates to its parameters, which can lead to better convergence and more precise optimization of the loss function.
- However, the training process at these learning rates might take longer, as the model requires more iterations to reach optimal parameter values.

➔ *Moderate Learning Rate 0.01*

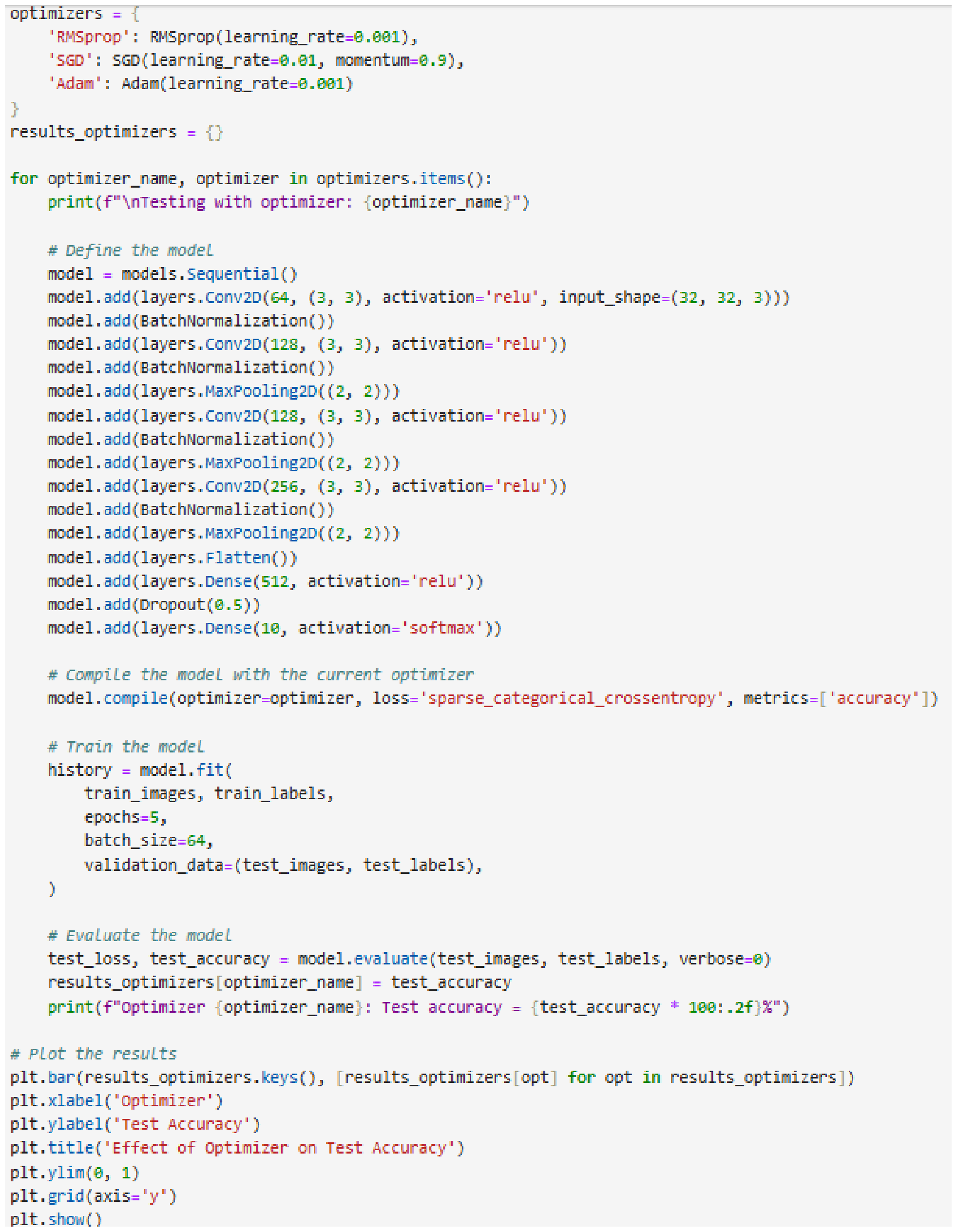
- The test accuracy begins to decrease compared to lower learning rates, dropping to around 0.6.

- A moderate learning rate can speed up training, but it might overshoot the optimal solution, resulting in slightly reduced performance on the test set.
- The decline in accuracy suggests that the learning rate is starting to become too large for effective optimization.

➔ *High Learning Rate 0.1*

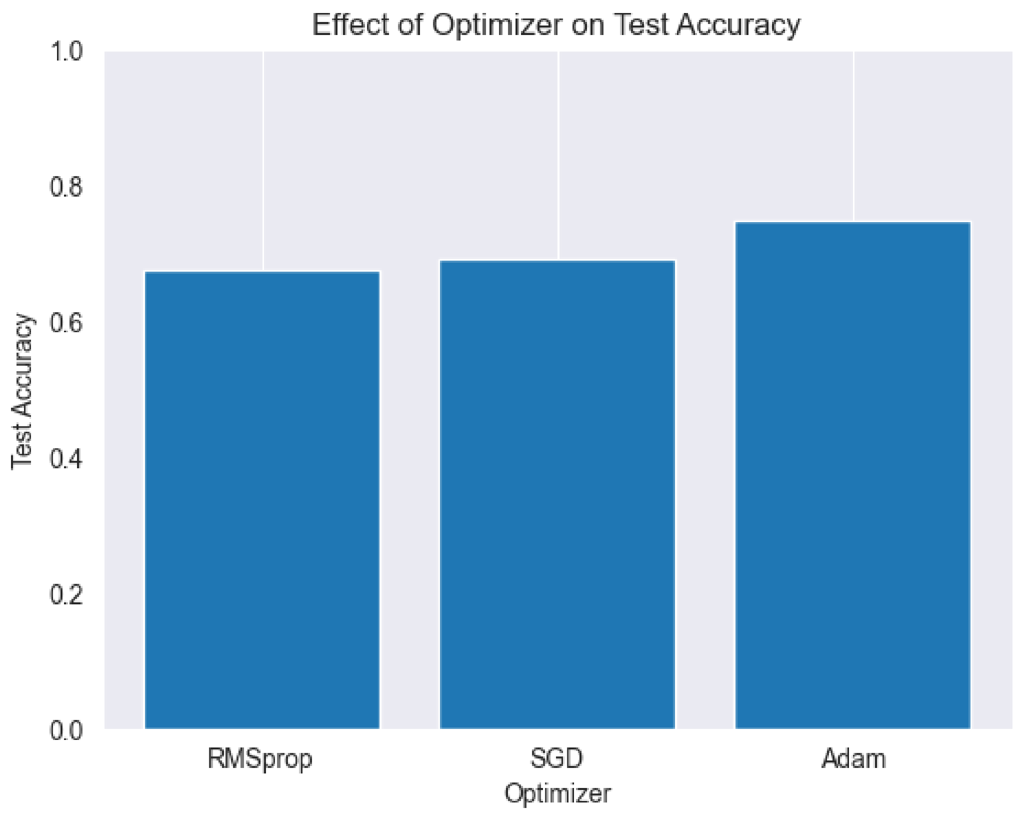
- The test accuracy drops dramatically to nearly 0.1, indicating a near-random performance level.
- A very high learning rate causes the model to make large parameter updates, which can prevent convergence and lead to divergence or oscillation around the optimal solution.
- The model is unable to learn effectively at this rate, as it cannot adequately minimize the loss function.

- ***Optimizer:***



```
optimizers = {
    'RMSprop': RMSprop(learning_rate=0.001),
    'SGD': SGD(learning_rate=0.01, momentum=0.9),
    'Adam': Adam(learning_rate=0.001)
}
results_optimizers = {}

for optimizer_name, optimizer in optimizers.items():
    print(f"\nTesting with optimizer: {optimizer_name}")

    # Define the model
    model = models.Sequential()
    model.add(layers.Conv2D(64, (3, 3), activation='relu', input_shape=(32, 32, 3)))
    model.add(BatchNormalization())
    model.add(layers.Conv2D(128, (3, 3), activation='relu'))
    model.add(BatchNormalization())
    model.add(layers.MaxPooling2D((2, 2)))
    model.add(layers.Conv2D(128, (3, 3), activation='relu'))
    model.add(BatchNormalization())
    model.add(layers.MaxPooling2D((2, 2)))
    model.add(layers.Conv2D(256, (3, 3), activation='relu'))
    model.add(BatchNormalization())
    model.add(layers.MaxPooling2D((2, 2)))
    model.add(layers.Flatten())
    model.add(layers.Dense(512, activation='relu'))
    model.add(Dropout(0.5))
    model.add(layers.Dense(10, activation='softmax'))

    # Compile the model with the current optimizer
    model.compile(optimizer=optimizer, loss='sparse_categorical_crossentropy', metrics=['accuracy'])

    # Train the model
    history = model.fit(
        train_images, train_labels,
        epochs=5,
        batch_size=64,
        validation_data=(test_images, test_labels),
    )

    # Evaluate the model
    test_loss, test_accuracy = model.evaluate(test_images, test_labels, verbose=0)
    results_optimizers[optimizer_name] = test_accuracy
    print(f"Optimizer {optimizer_name}: Test accuracy = {test_accuracy * 100:.2f}%")

# Plot the results
plt.bar(results_optimizers.keys(), [results_optimizers[opt] for opt in results_optimizers])
plt.xlabel('Optimizer')
plt.ylabel('Test Accuracy')
plt.title('Effect of Optimizer on Test Accuracy')
plt.ylim(0, 1)
plt.grid(axis='y')
plt.show()
```

A CNN's learning efficiency is largely dependent on the **optimizer** it uses. Various optimizers, including Adam, SGD, and RMSprop, minimize loss functions using distinct mathematical procedures. Certain optimizers are more appropriate for particular job types, and these variations can have a substantial effect on the training process. For example, Adam is frequently chosen because of its capacity for flexible learning rates, which can result in faster convergence in challenging situations.

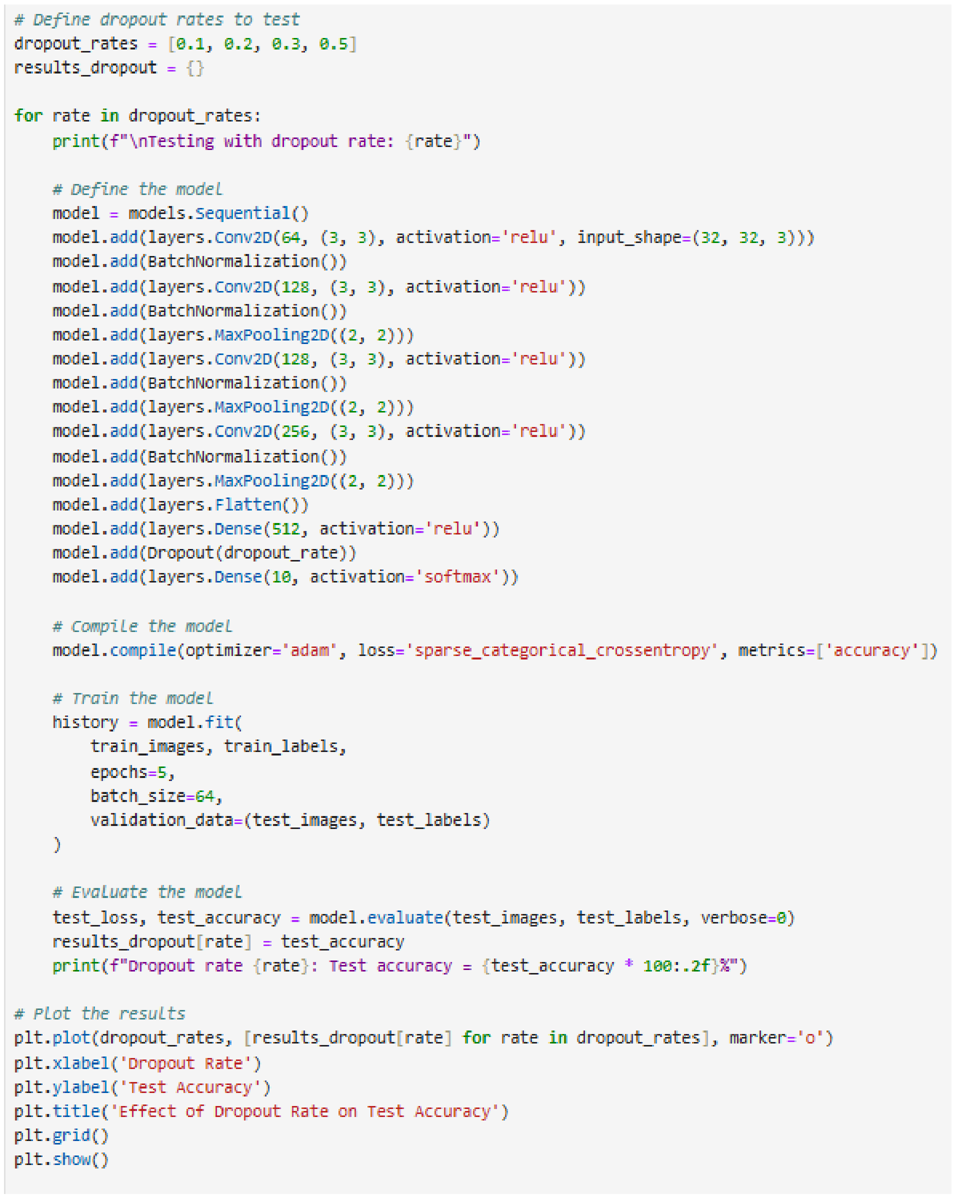


This bar graph compares the test accuracy of a model using three different optimization algorithms: RMSprop, SGD (Stochastic Gradient Descent), and Adam.

- **The Adam optimizer** achieves the highest test accuracy among the three, indicating its effectiveness in optimizing the model's parameters. This suggests that Adam, which combines the benefits of momentum and adaptive learning rates, is particularly suited for this task or dataset.
- **RMSprop and SGD** perform noticeably worse, with RMSprop showing slightly better accuracy than SGD. However, both are less effective compared to Adam.

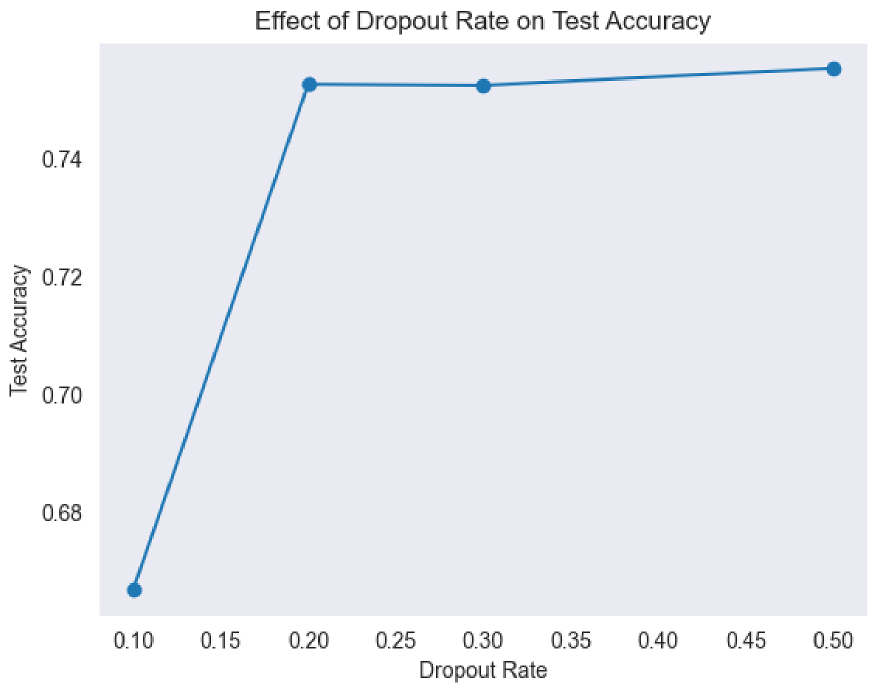
- ***Drop Out:***

We systematically modify the Dropout rate while keeping the rest of the architecture constant



```
# Define dropout rates to test
dropout_rates = [0.1, 0.2, 0.3, 0.5]
results_dropout = {}

for rate in dropout_rates:
    print(f"\nTesting with dropout rate: {rate}")

    # Define the model
    model = models.Sequential()
    model.add(layers.Conv2D(64, (3, 3), activation='relu', input_shape=(32, 32, 3)))
    model.add(BatchNormalization())
    model.add(layers.Conv2D(128, (3, 3), activation='relu'))
    model.add(BatchNormalization())
    model.add(layers.MaxPooling2D((2, 2)))
    model.add(layers.Conv2D(128, (3, 3), activation='relu'))
    model.add(BatchNormalization())
    model.add(layers.MaxPooling2D((2, 2)))
    model.add(layers.Conv2D(256, (3, 3), activation='relu'))
    model.add(BatchNormalization())
    model.add(layers.MaxPooling2D((2, 2)))
    model.add(layers.Flatten())
    model.add(layers.Dense(512, activation='relu'))
    model.add(Dropout(dropout_rate))
    model.add(layers.Dense(10, activation='softmax'))

    # Compile the model
    model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])

    # Train the model
    history = model.fit(
        train_images, train_labels,
        epochs=5,
        batch_size=64,
        validation_data=(test_images, test_labels)
    )

    # Evaluate the model
    test_loss, test_accuracy = model.evaluate(test_images, test_labels, verbose=0)
    results_dropout[rate] = test_accuracy
    print(f"Dropout rate {rate}: Test accuracy = {test_accuracy * 100:.2f}%")

# Plot the results
plt.plot(dropout_rates, [results_dropout[rate] for rate in dropout_rates], marker='o')
plt.xlabel('Dropout Rate')
plt.ylabel('Test Accuracy')
plt.title('Effect of Dropout Rate on Test Accuracy')
plt.grid()
plt.show()
```

The **dropout rate** is a regularization method that enhances CNNs' generalization and resilience. By altering the dropout rate, the model can lower the danger of overfitting by avoiding intricate co-adaptations on training data. Different dropout rates can be implemented at different levels enabling a more sophisticated regularization strategy that can maximize the network's capacity for learning and generalization. With layer-specific dropout rates, the impact of the dropout may be customized to fit the needs and complexity of various network layers.



In examining the plot, a dropout rate of 0.5 achieves the highest accuracy, indicating optimal model performance on the CIFAR-10 dataset. Nevertheless, dropout rates of 0.2 and 0.3 also show comparable accuracies, ranging from 75.27% to 75.25%. This highlights the model's resilience to variations in dropout rates within this range, with only minor differences in performance. When selecting a dropout rate, it is essential to balance model complexity and regularization to improve generalization without compromising accuracy.

V. Conclusion

In this project, we set out to create a high-performing image classifier using Convolutional Neural Networks (CNNs) within the Keras framework. After running three experiments, each building on the last, we successfully achieved an accuracy of **88%**. This progress was made possible through careful fine-tuning of hyperparameters, such as using the Adam optimizer, setting a learning rate of **0.0001**, a batch size of **32**, and gradually increasing dropout. Along the way, we discovered that hyperparameters don’t work in isolation but interact in complex ways, what worked best was finding the right combination rather than focusing on optimizing them individually.

The process wasn’t without its challenges. Overheating, system shutdowns, and insanely long training times reminded us just how demanding training deep neural networks can be. Despite these obstacles, by strategically deepening the model, extending training times, and making thoughtful adjustments, we were able to significantly enhance performance.

Looking to the future, the insights we gained from this journey provide a solid foundation for future projects in computer vision and machine learning. This experience not only highlights the importance of understanding hyperparameters and architecture but also underscores the potential of machine learning to drive advancements in technology. Our efforts here contribute to a growing body of knowledge, paving the way for exciting possibilities in the field.