SCHOOL OF SCIENCE & ENGINEERING

FALL 2023

CSC 4301 Introduction to Artificial Intelligence

Term Project: 4

Image Classification using CNN

December 5th, 2023

Realized by:

Khadija Salih Alj

Noura Ogbi

Younes Bouziane

Supervised by:

Prof. Tajjedine Rachidi

Table of Contents

I. Introduction 3

II. First Experimentation (78% Accuracy) 4

III. Second Experimentation (79.5% Accuracy) 7

IV. Third Experimentation (87% Accuracy) 10

V. YouTube Demo 17

VI. Hyperparameter Exploration 18

VII. Conclusion 27

I. Introduction

In the domain of Computer Vision, where machines endeavor to mirror human visual capabilities, image classification plays a pivotal role. This project delves into the captivating intersection of Computer Vision and Machine Learning, where we aim to develop a Convolutional Neural Network (CNN) designed for image object classification. Guided by Keras, a robust and open-source high-level API, our primary objective is to craft a resilient CNN model.

In recent years, advancements in the field of Computer Vision and Machine Learning have led to remarkable developments in image recognition and classification. The ability to accurately identify and categorize objects within images has widespread implications across numerous industries, including healthcare, automotive, retail, and security. This project seeks to contribute to this trend by exploring the construction and optimization of CNN, ultimately aiming to enhance the accuracy and efficiency of image classification.

Our focus centers on the practical implementation and the accuracy of the model. The primary aim is to construct a CNN model that excels in classifying images, with particular emphasis on refining network layers, establishing connections, and training the model. The dataset employed for this endeavor is CIFAR-10, a collection of 8 million images spanning ten distinct classes.

This report provides a comprehensive account of the steps taken to adapt source code for image classification. The model’s performance is meticulously assessed, with a keen eye on accuracy calculations during the training phase and a thorough evaluation of its final performance, measured through the detection rate. Additionally, the report delves into the exploration of diverse hyperparameters, presenting a nuanced analysis accompanied by visual representations to elucidate their impact on the model’s accuracy and efficiency.

II. First Experimentation (78% Accuracy)

1) Data Loading & Preprocessing:

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

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

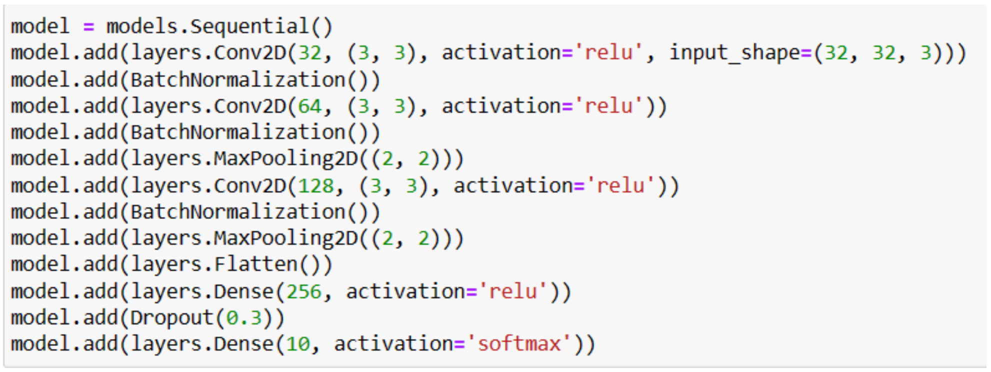
This part **loads** the CIFAR-10 dataset using TensorFlow’s built-in dataset loader. The pixel values of the images are then **normalized** to the range [0, 1] by dividing them by 255.0.

2) Data Visualization:

```
# visualize data by plotting images
fig, ax = plt.subplots(5, 5)
k = 0

for i in range(5):
    for j in range(5):
        ax[i][j].imshow(train_images[k], aspect='auto')
        k += 1

plt.show()
```

The code visualizes a set of images, presumably from the training dataset (train_images). The 5x5 grid allows for the simultaneous display of 25 images, making it a compact and efficient way to inspect a subset of the dataset. The imshow function is commonly used for displaying images, and the loop ensures that each subplot is populated with a unique image.

3) Model Definition:

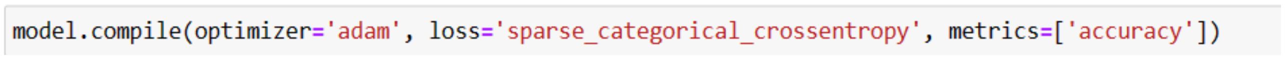


```
model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)))
model.add(BatchNormalization())
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(BatchNormalization())
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(128, (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'))
```

This section defines a sequential model using Keras. The model consists of convolutional layers with Batch Normalization, MaxPooling layers, a Flatten layer, a Dense layer with ReLU activation, a Dropout layer, and a final Dense layer with softmax activation for classification.

- The initial **convolutional layers (Conv2D)** with *ReLU* activation are designed to extract hierarchical features from the input images. The increasing number of filters (32, 64, 128) in subsequent layers allows the model to capture more complex and abstract representations.
- **Batch Normalization** is applied after each convolutional layer. This normalizes the activations, mitigating internal covariate shift and potentially accelerating training.
- **MaxPooling layers** are incorporated to down sample spatial dimensions, reducing computational complexity, and promoting *translation invariance*.
- The **Flatten layer** transforms the multi-dimensional output from the convolutional layers into a flat vector, preparing it for input to the dense layers.
- A **Dense layer** with 256 units and *ReLU* activation follows. This layer functions as a feature aggregator, capturing high-level patterns in the flattened representation.
- **Dropout** with a rate of 0.3 is introduced to mitigate overfitting by randomly dropping out 30% of the neurons during training.
- The final **Dense layer** with 10 units and *softmax* activation is suitable for multi-class classification. It outputs probability scores for each of the 10 classes in the CIFAR-10 dataset.

4) Model Compilation:



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

The model is compiled using the **Adam** optimizer, **sparse categorical crossentropy** loss (suitable for integer-encoded class labels), and **accuracy** as the evaluation metric.

5) Model Training:

```
model.fit(train_images, train_labels, epochs=20, validation_data=(test_images, test_labels))
```

The model is trained for 20 epochs with the training data, and validation data is used for monitoring performance during training.

```
Epoch 13/20
1563/1563 [==============================] - 601s 385ms/step - loss: 0.2288 - accuracy: 0.9212 - val_loss: 0.9144 - val_accu
racy: 0.7667
Epoch 14/20
1563/1563 [==============================] - 603s 386ms/step - loss: 0.2031 - accuracy: 0.9300 - val_loss: 1.0396 - val_accu
racy: 0.7538
Epoch 15/20
1563/1563 [==============================] - 591s 378ms/step - loss: 0.1921 - accuracy: 0.9342 - val_loss: 1.0108 - val_accu
racy: 0.7657
Epoch 16/20
1563/1563 [==============================] - 603s 386ms/step - loss: 0.1894 - accuracy: 0.9351 - val_loss: 0.9756 - val_accu
racy: 0.7716
Epoch 17/20
1563/1563 [==============================] - 4827s 3s/step - loss: 0.1680 - accuracy: 0.9419 - val_loss: 0.9901 - val_accura
cy: 0.7739
Epoch 18/20
1563/1563 [==============================] - 395s 253ms/step - loss: 0.1663 - accuracy: 0.9438 - val_loss: 1.0955 - val_accu
racy: 0.7644
Epoch 19/20
1563/1563 [==============================] - 375s 240ms/step - loss: 0.1590 - accuracy: 0.9454 - val_loss: 1.0217 - val_accu
racy: 0.7686
Epoch 20/20
1563/1563 [==============================] - 335s 215ms/step - loss: 0.1434 - accuracy: 0.9520 - val_loss: 1.0207 - val_accu
racy: 0.7809
```

6) Model Evaluation:

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

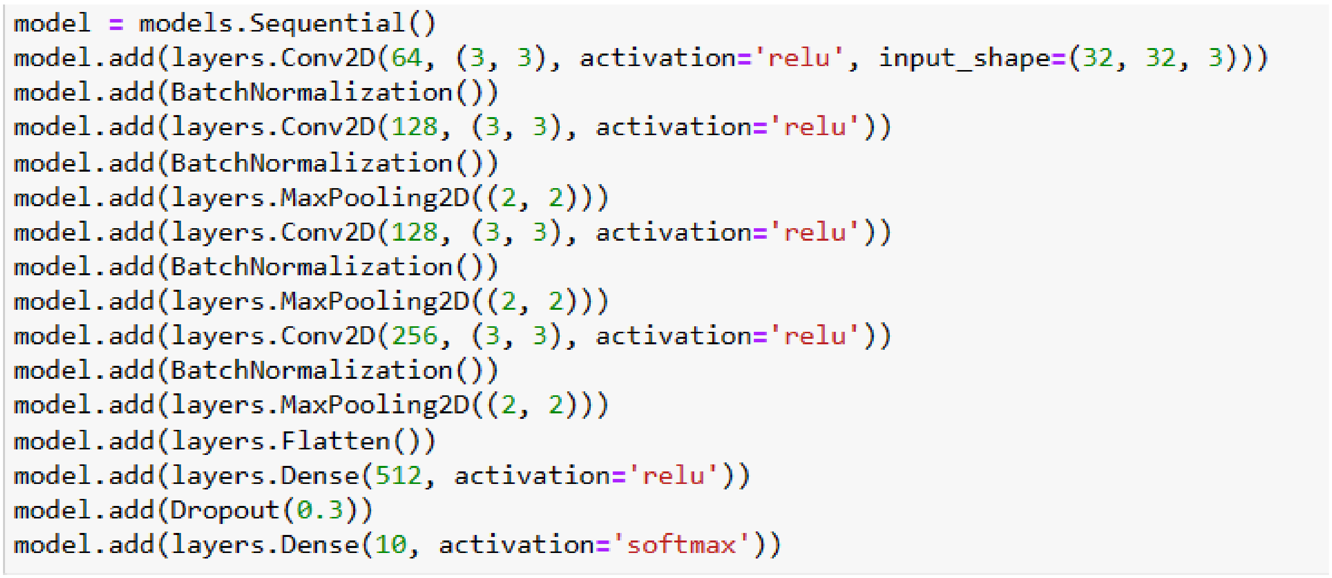
The trained model is evaluated on the test dataset, and the test accuracy is printed.

```
313/313 [==============================] - 20s 62ms/step - loss: 1.0207 - accuracy: 0.7809
Test accuracy: 0.7809000015258789
```

In brief, the model architecture consists of convolutional layers with Batch Normalization to extract features, MaxPooling layers for down sampling, and Dense layers for classification. The use of Dropout helps prevent overfitting during training. The Adam optimizer, sparse categorical crossentropy loss, and accuracy metric are employed for model compilation.

III. Second Experimentation (79.5% Accuracy)

1) Model Definition:



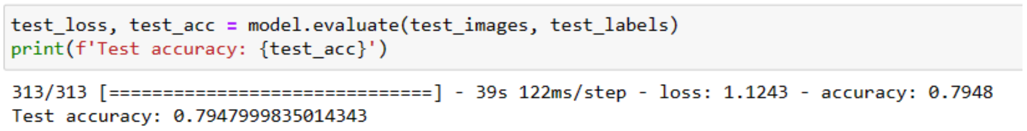
```
model = models.Sequential()
model.add(layers.Conv2D(64, (3, 3), activation='relu', input_shape=(32, 32, 3)))
model.add(BatchNormalization())
model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(BatchNormalization())
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(128, (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'))
```

The new model has an additional **convolutional layer** with 256 filters (doubled compared to previous one) followed by **Batch Normalization and MaxPooling**. This potentially allows the model to capture more intricate hierarchical features compared to the previous model.

Additionally, the model has a denser architecture with a **Dense layer of 512 units** compared to the previous model's 256 units. This increases the complexity of the feature aggregation step, potentially enabling the model to learn more nuanced patterns.

In brief, this second model is designed to be more expressive with increased depth in convolutional layers and a denser architecture in the fully connected layers. These changes introduce more parameters to the model, potentially enabling it to learn more intricate features. However, this complexity comes at the cost of <u>increased computational requirements</u> (13 hours of training instead of 6) and the possibility of **overfitting**, which we mitigated by the inclusion of **Dropout** layers.

2) Model Evaluation:



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

```
313/313 [==============================] - 39s 122ms/step - loss: 1.1243 - accuracy: 0.7948
Test accuracy: 0.7947999835014343
```

We observe that the modifications we made to the previous model increased the accuracy with 1.5%.

3) Model Prediction and Evaluation:

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

```
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 Report:\n{class_report}')
```

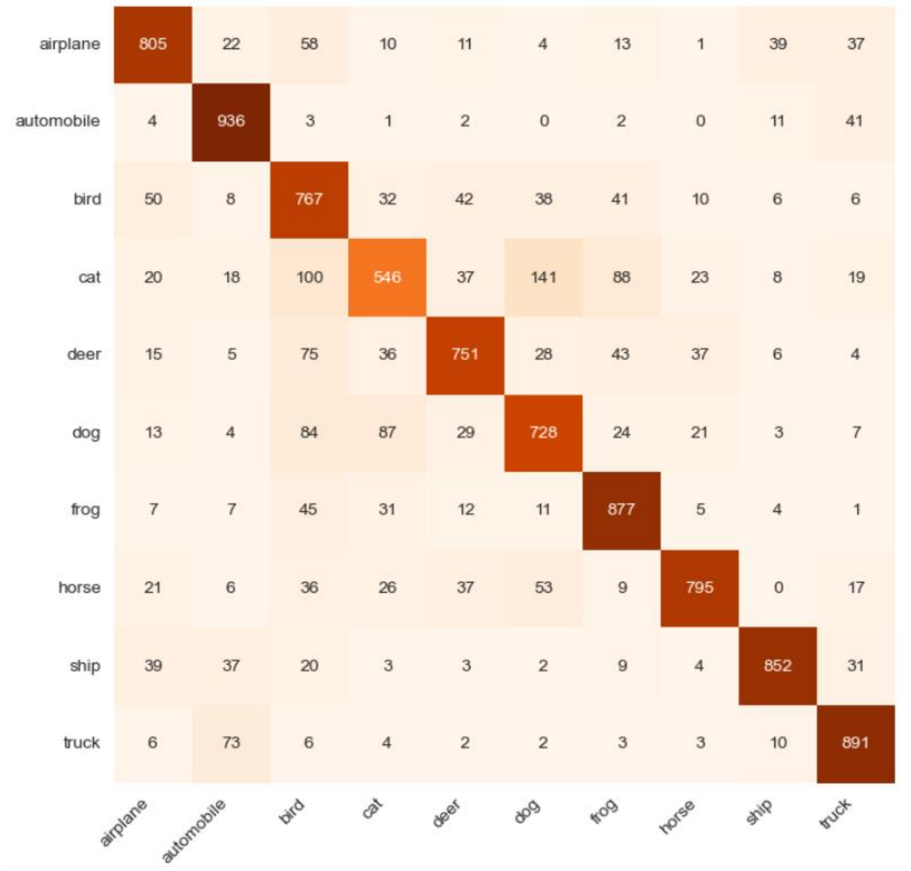
After training the model, predictions are made on the test dataset. The model’s predictions are converted into class labels, and a classification report is generated to provide detailed insights into the model’s performance on each class.

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

    airplane       0.82      0.81      0.81      1000
  automobile       0.84      0.94      0.88      1000
        bird       0.64      0.77      0.70      1000
         cat       0.70      0.55      0.61      1000
        deer       0.81      0.75      0.78      1000
         dog       0.72      0.73      0.73      1000
        frog       0.79      0.88      0.83      1000
       horse       0.88      0.80      0.84      1000
        ship       0.91      0.85      0.88      1000
       truck       0.85      0.89      0.87      1000

    accuracy                           0.79     10000
   macro avg       0.80      0.79      0.79     10000
weighted avg       0.80      0.79      0.79     10000
```

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

- **Precision:** represents the accuracy of the positive predictions made by the model. It is calculated as the ratio of true positives to the sum of true positives and false positives. The precision values range from 0.64 to 0.91, indicating that the model performs well in making accurate positive predictions across different classes.
- **Recall:** (also known as sensitivity or true positive rate) measures the ability of the model to correctly identify all relevant instances. It is calculated as the ratio of true positives to the sum of true positives and false negatives. The recall values range from 0.55 to 0.94, indicating variability in the model's ability to capture all relevant instances.
- **F1-Score:** it is the harmonic mean of precision and recall, providing a balanced measure of a model's performance. The F1-score values range from 0.61 to 0.88, reflecting a balance between precision and recall.
- **Overall Accuracy:** it is reported at 0.79, representing the ratio of correct predictions to the total number of predictions.

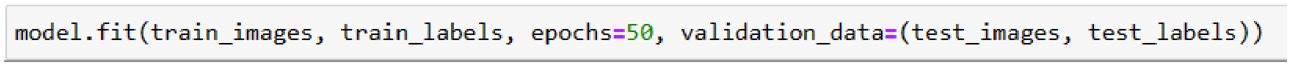
In summary, this second model shows good performance overall, with a weighted average F1-score of 79%. However, there is variability in performance across individual classes, with some classes achieving higher precision and recall than others…



	airplane	automobile	bird	cat	deer	dog	frog	horse	ship	truck
airplane	805	22	58	10	11	4	13	1	39	37
automobile	4	936	3	1	2	0	2	0	11	41
bird	50	8	767	32	42	38	41	10	6	6
cat	20	18	100	546	37	141	88	23	8	19
deer	15	5	75	36	751	28	43	37	6	4
dog	13	4	84	87	29	728	24	21	3	7
frog	7	7	45	31	12	11	877	5	4	1
horse	21	6	36	26	37	53	9	795	0	17
ship	39	37	20	3	3	2	9	4	852	31
truck	6	73	6	4	2	2	3	3	10	891

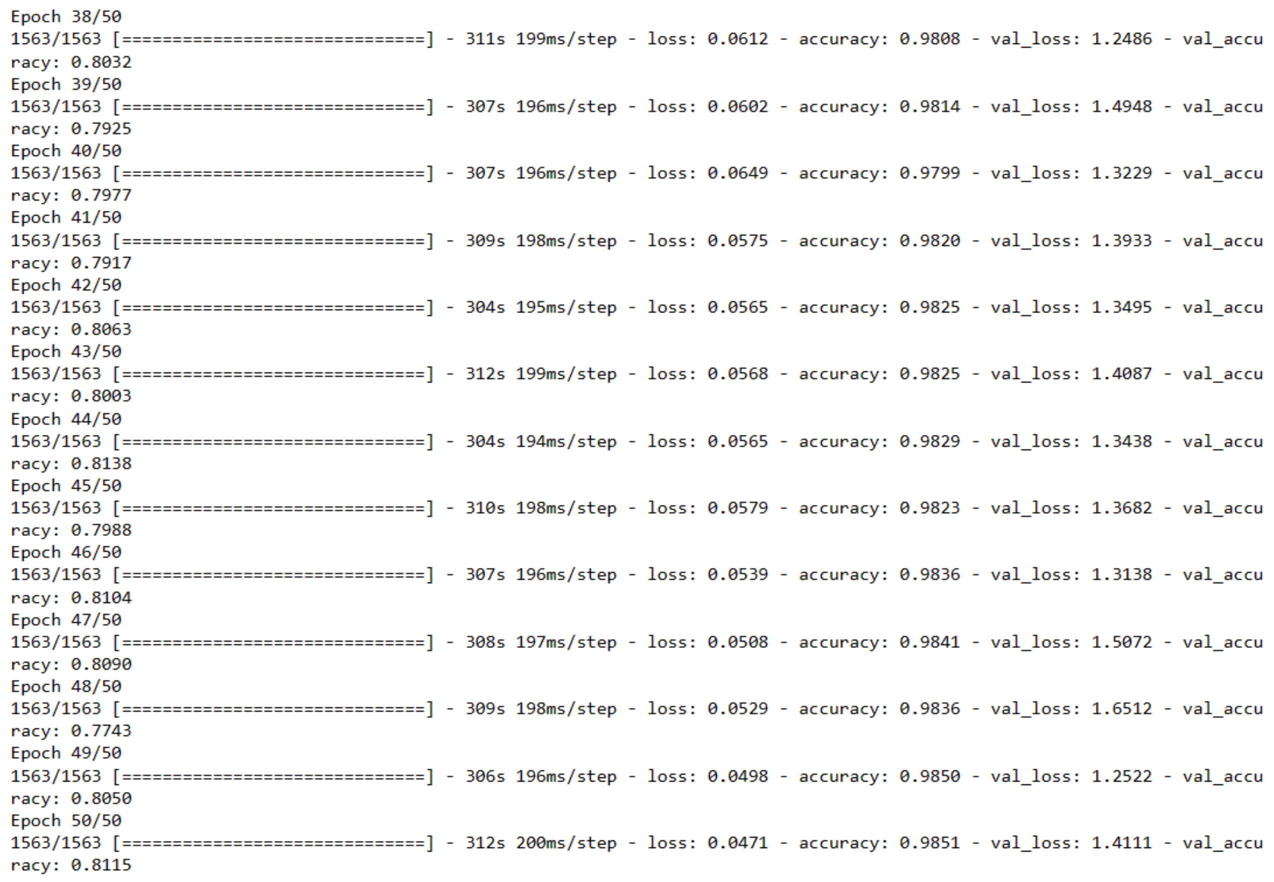
IV. Third Experimentation (87% Accuracy)

In this third experimentation with the second model, the notable modification involved an increase in the number of training **epochs from 20 to 50**.



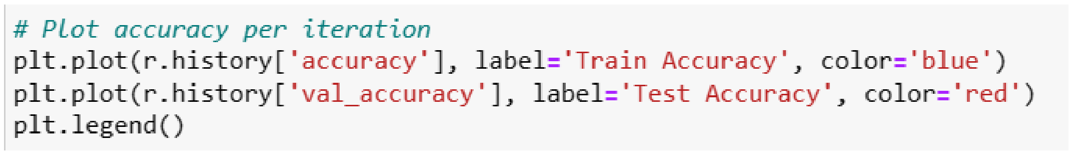
```
model.fit(train_images, train_labels, epochs=50, validation_data=(test_images, test_labels))
```

This adjustment aimed to allow the model more opportunities to learn intricate patterns and representations within the dataset.



```
Epoch 38/50
1563/1563 [==============================] - 311s 199ms/step - loss: 0.0612 - accuracy: 0.9808 - val_loss: 1.2486 - val_accu
racy: 0.8032
Epoch 39/50
1563/1563 [==============================] - 307s 196ms/step - loss: 0.0602 - accuracy: 0.9814 - val_loss: 1.4948 - val_accu
racy: 0.7925
Epoch 40/50
1563/1563 [==============================] - 307s 196ms/step - loss: 0.0649 - accuracy: 0.9799 - val_loss: 1.3229 - val_accu
racy: 0.7977
Epoch 41/50
1563/1563 [==============================] - 309s 198ms/step - loss: 0.0575 - accuracy: 0.9820 - val_loss: 1.3933 - val_accu
racy: 0.7917
Epoch 42/50
1563/1563 [==============================] - 304s 195ms/step - loss: 0.0565 - accuracy: 0.9825 - val_loss: 1.3495 - val_accu
racy: 0.8063
Epoch 43/50
1563/1563 [==============================] - 312s 199ms/step - loss: 0.0568 - accuracy: 0.9825 - val_loss: 1.4087 - val_accu
racy: 0.8003
Epoch 44/50
1563/1563 [==============================] - 304s 194ms/step - loss: 0.0565 - accuracy: 0.9829 - val_loss: 1.3438 - val_accu
racy: 0.8138
Epoch 45/50
1563/1563 [==============================] - 310s 198ms/step - loss: 0.0579 - accuracy: 0.9823 - val_loss: 1.3682 - val_accu
racy: 0.7988
Epoch 46/50
1563/1563 [==============================] - 307s 196ms/step - loss: 0.0539 - accuracy: 0.9836 - val_loss: 1.3138 - val_accu
racy: 0.8104
Epoch 47/50
1563/1563 [==============================] - 308s 197ms/step - loss: 0.0508 - accuracy: 0.9841 - val_loss: 1.5072 - val_accu
racy: 0.8090
Epoch 48/50
1563/1563 [==============================] - 309s 198ms/step - loss: 0.0529 - accuracy: 0.9836 - val_loss: 1.6512 - val_accu
racy: 0.7743
Epoch 49/50
1563/1563 [==============================] - 306s 196ms/step - loss: 0.0498 - accuracy: 0.9850 - val_loss: 1.2522 - val_accu
racy: 0.8050
Epoch 50/50
1563/1563 [==============================] - 312s 200ms/step - loss: 0.0471 - accuracy: 0.9851 - val_loss: 1.4111 - val_accu
racy: 0.8115
```

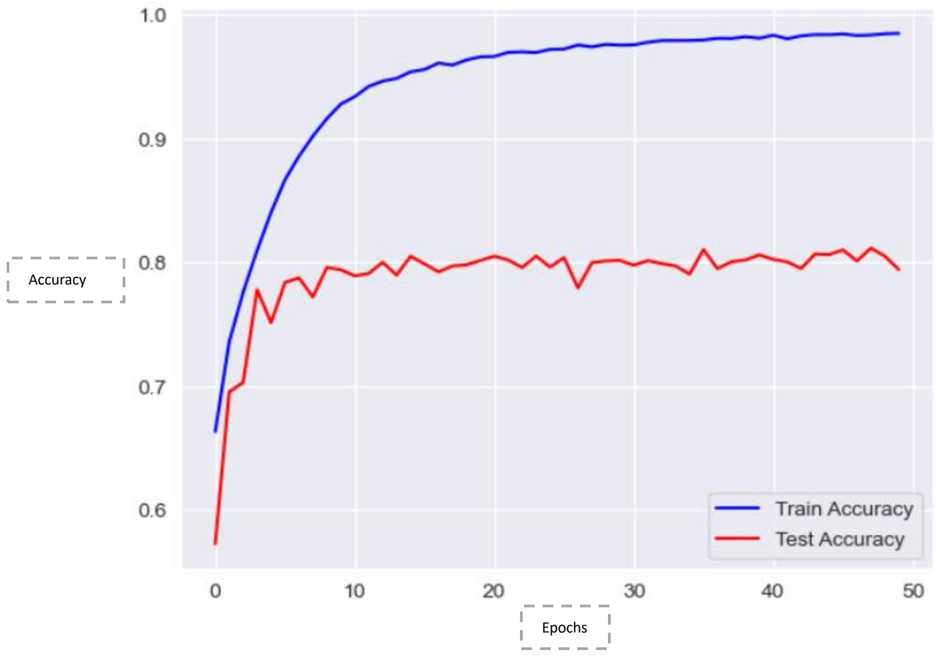
1) Model Evaluation:



```
# 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()
```

The above code provides a quick and effective way to assess the performance of our model during training by visualizing accuracy metrics.

- A rising trend in both training and validation accuracy is generally positive, indicating that the model is learning and generalizing well.
- If the training accuracy continues to rise while the validation accuracy plateaus or decreases, it may suggest overfitting.
- If both training and validation accuracies are low or plateau, it may indicate underfitting, and model adjustments may be necessary.



The accuracy plot reveals an initial rise (before 20 epochs), but a subsequent plateau suggests potential overfitting. This phenomenon occurs when the model excels on training data but struggles with new, diverse inputs. To address this, fine-tuning and regularization techniques (data augmentation) are essential for maintaining a balance between fitting the training data and generalizing effectively.

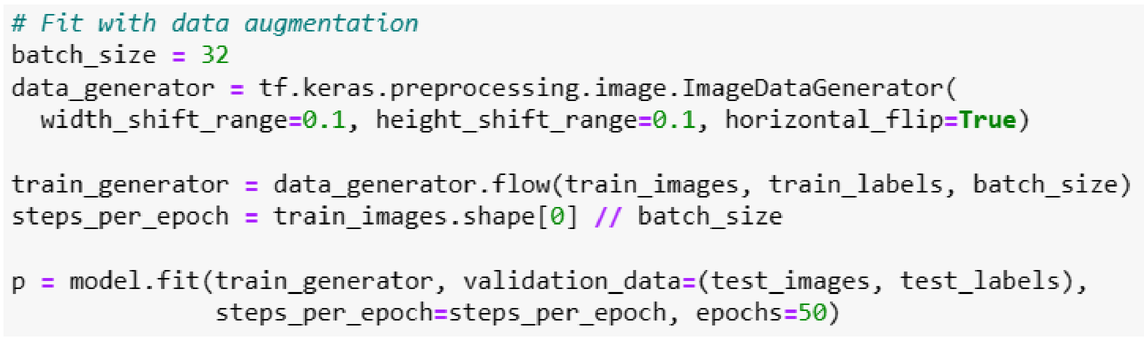
Furthermore, as a result of the extended training duration, the model demonstrated improved performance, with the accuracy experiencing an evident increase from **79% to 81%**.

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

```
313/313 [==============================] - 18s 56ms/step - loss: 1.4111 - accuracy: 0.8115
Test accuracy: 0.8115000128746033
```

The justification for this enhancement lies in the nature of deep learning models, where prolonged training periods can enable the network to refine its internal representations, capture nuanced features, and better generalize to the intricacies present in the data.

2) Data Augmentation + Training:

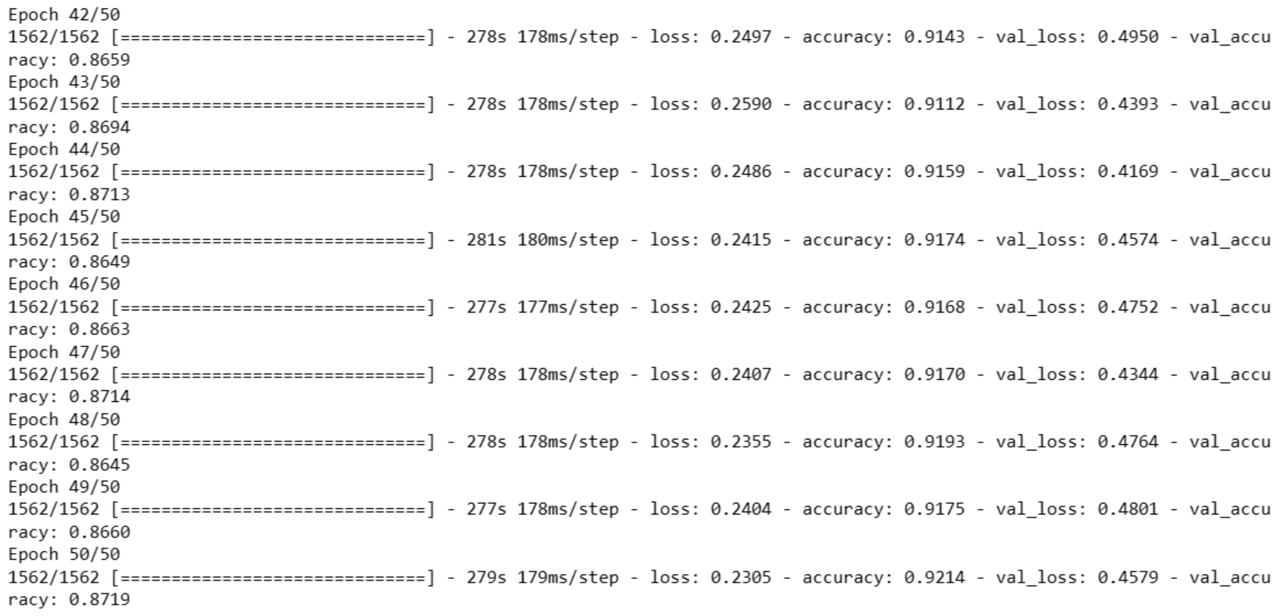


```
# Fit with data augmentation
batch_size = 32
data_generator = tf.keras.preprocessing.image.ImageDataGenerator(
  width_shift_range=0.1, height_shift_range=0.1, horizontal_flip=True)

train_generator = data_generator.flow(train_images, train_labels, batch_size)
steps_per_epoch = train_images.shape[0] // batch_size

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

The provided code implements data augmentation during the training. Data augmentation is a crucial technique in machine learning, particularly for image classification tasks, as it artificially expands the diversity of the training dataset. This is achieved through random transformations applied to the input images, enhancing the model's ability to generalize well to unseen data.

The '**ImageDataGenerator**' class from Keras is employed to define the data augmentation techniques. Specifically, random shifts in width and height (up to 10% of the total width and height) and horizontal flips are applied. These transformations help expose the model to various perspectives of the input images, reducing the risk of overfitting and enhancing the model's robustness.

The choice of a batch size of 32 indicates that the training data is processed in batches during each iteration, which can lead to more stable training, especially when computational resources are limited.



```
Epoch 42/50
1562/1562 [==============================] - 278s 178ms/step - loss: 0.2497 - accuracy: 0.9143 - val_loss: 0.4950 - val_accu
racy: 0.8659
Epoch 43/50
1562/1562 [==============================] - 278s 178ms/step - loss: 0.2590 - accuracy: 0.9112 - val_loss: 0.4393 - val_accu
racy: 0.8694
Epoch 44/50
1562/1562 [==============================] - 278s 178ms/step - loss: 0.2486 - accuracy: 0.9159 - val_loss: 0.4169 - val_accu
racy: 0.8713
Epoch 45/50
1562/1562 [==============================] - 281s 180ms/step - loss: 0.2415 - accuracy: 0.9174 - val_loss: 0.4574 - val_accu
racy: 0.8649
Epoch 46/50
1562/1562 [==============================] - 277s 177ms/step - loss: 0.2425 - accuracy: 0.9168 - val_loss: 0.4752 - val_accu
racy: 0.8663
Epoch 47/50
1562/1562 [==============================] - 278s 178ms/step - loss: 0.2407 - accuracy: 0.9170 - val_loss: 0.4344 - val_accu
racy: 0.8714
Epoch 48/50
1562/1562 [==============================] - 278s 178ms/step - loss: 0.2355 - accuracy: 0.9193 - val_loss: 0.4764 - val_accu
racy: 0.8645
Epoch 49/50
1562/1562 [==============================] - 277s 178ms/step - loss: 0.2404 - accuracy: 0.9175 - val_loss: 0.4801 - val_accu
racy: 0.8660
Epoch 50/50
1562/1562 [==============================] - 279s 179ms/step - loss: 0.2305 - accuracy: 0.9214 - val_loss: 0.4579 - val_accu
racy: 0.8719
```

3) Model Prediction and Evaluation:

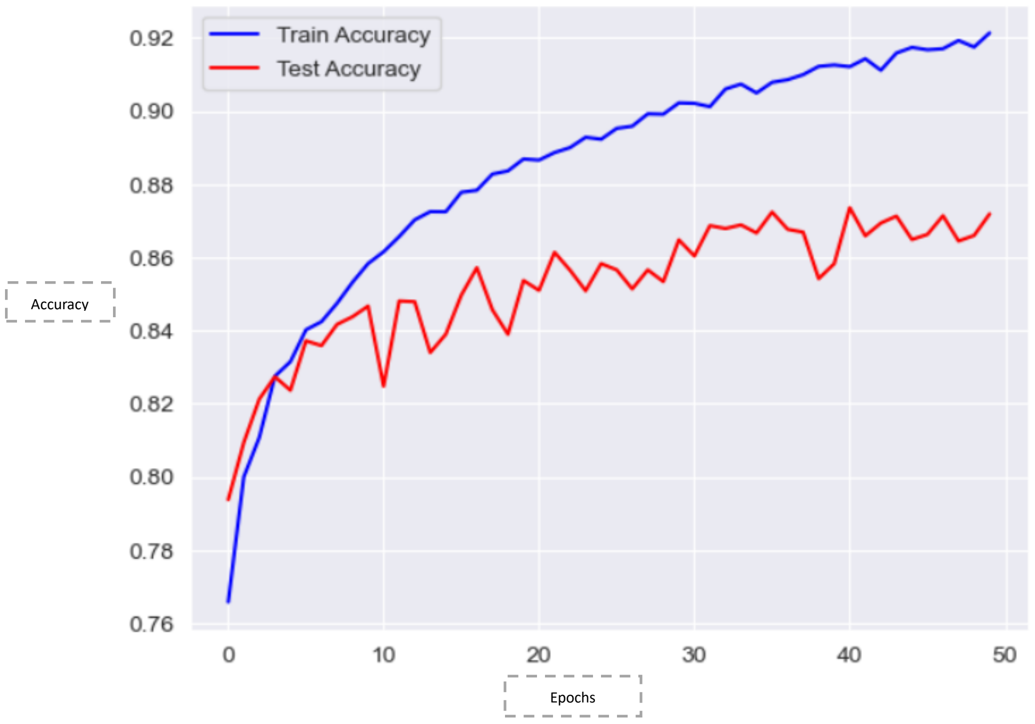
```
test_loss, test_acc = model.evaluate(test_images, test_labels)
print(f'Test accuracy: {test_acc}')
```

```
313/313 [==============================] - 19s 59ms/step - loss: 0.4579 - accuracy: 0.8719
Test accuracy: 0.8719000220298767
```

We can see that data augmentation increased the test accuracy to 87%.

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

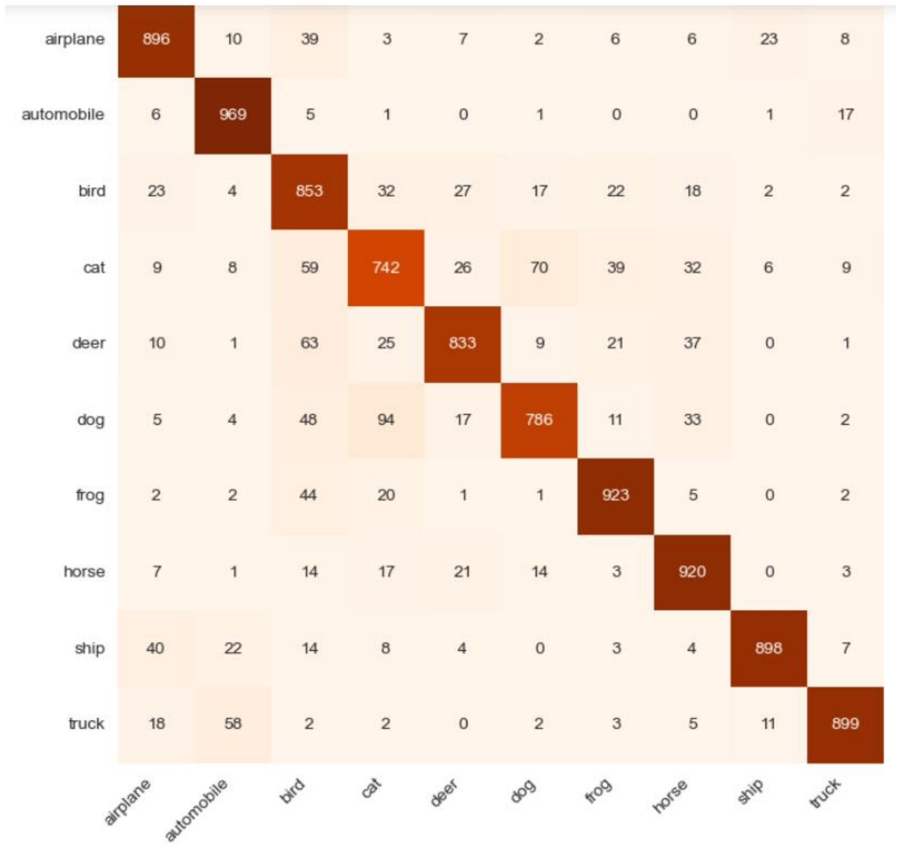
```
<matplotlib.legend.Legend at 0x276ef578f10>
```



After implementing data augmentation, a notable improvement is observed in the accuracy plot, showing a significant increase to 87%. This enhancement suggests that the model's performance benefits from the augmented dataset, which aids in better generalization. Data augmentation introduces variations in the training set, helping the model to learn diverse patterns and reducing the risk of overfitting. The rise in accuracy post-augmentation signifies a positive impact on the model's ability to handle previously unseen data, showcasing the efficacy of this technique in refining the model's performance.

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

    airplane       0.88      0.90      0.89      1000
  automobile       0.90      0.97      0.93      1000
        bird       0.75      0.85      0.80      1000
         cat       0.79      0.74      0.76      1000
        deer       0.89      0.83      0.86      1000
         dog       0.87      0.79      0.83      1000
        frog       0.90      0.92      0.91      1000
       horse       0.87      0.92      0.89      1000
        ship       0.95      0.90      0.93      1000
       truck       0.95      0.90      0.92      1000

    accuracy                           0.87     10000
   macro avg       0.87      0.87      0.87     10000
weighted avg       0.87      0.87      0.87     10000
```

The varying precision, recall, and F1 scores observed across classes, along with an increase in overall accuracy, can be attributed to the model's nuanced adjustments for each class. Classes displaying improvements likely benefited from enhanced precision in positive predictions, contributing to their overall scores. Notably, there are no classes with reduced scores, highlighting the model's consistent performance across all categories.

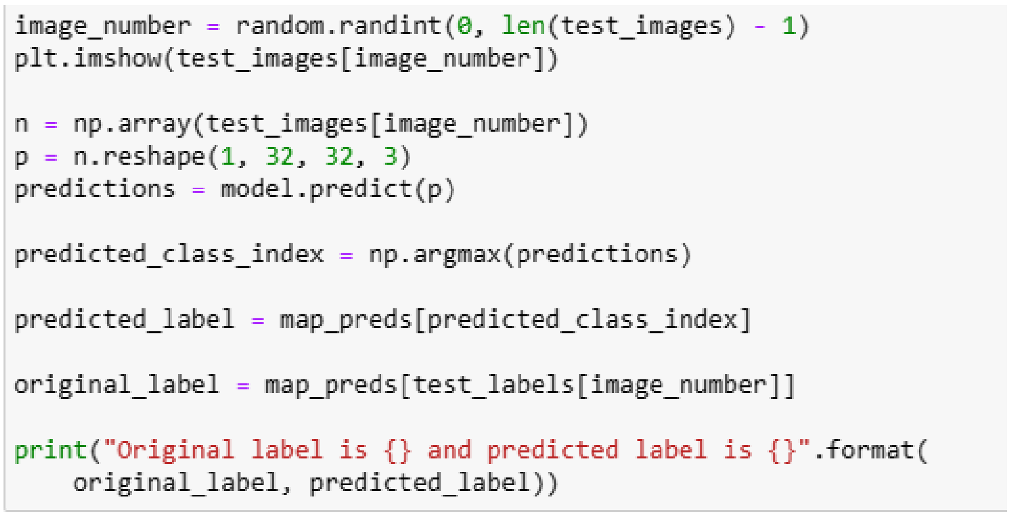


	airplane	automobile	bird	cat	deer	dog	frog	horse	ship	truck
airplane	896	10	39	3	7	2	6	6	23	8
automobile	6	969	5	1	0	1	0	0	1	17
bird	23	4	853	32	27	17	22	18	2	2
cat	9	8	59	742	26	70	39	32	6	9
deer	10	1	63	25	833	9	21	37	0	1
dog	5	4	48	94	17	786	11	33	0	2
frog	2	2	44	20	1	1	923	5	0	2
horse	7	1	14	17	21	14	3	920	0	3
ship	40	22	14	8	4	0	3	4	898	7
truck	18	58	2	2	0	2	3	5	11	899

Following meticulous optimization and fine-tuning processes, a **considerable enhancement in accuracy and efficiency** has been discerned. The encouraging findings are prominently manifested in the confusion matrix plot, *revealing many zero entries*. This observed pattern serves as an indicator of the model's heightened capacity to effectively categorize instances, with zeros representing true negatives. Such advancements denote a positive progression towards heightened precision and dependability in the predictive capabilities of our model.

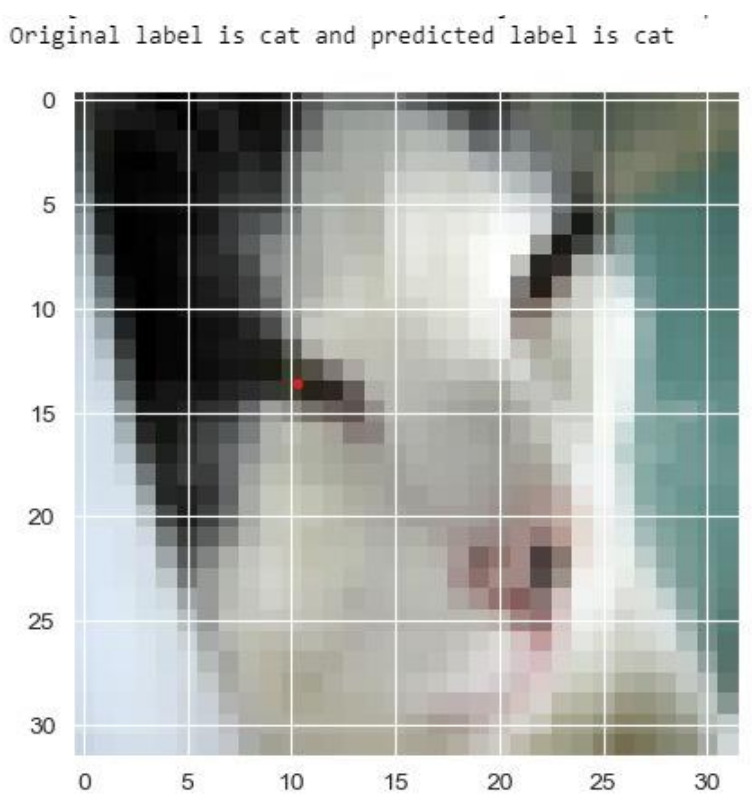
4) Model Testing:

In this part, we will showcase the performance of our model on the test dataset, delving into the outcomes of rigorous evaluations.



```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("Original label is {} and predicted label is {}".format(
    original_label, predicted_label))
```

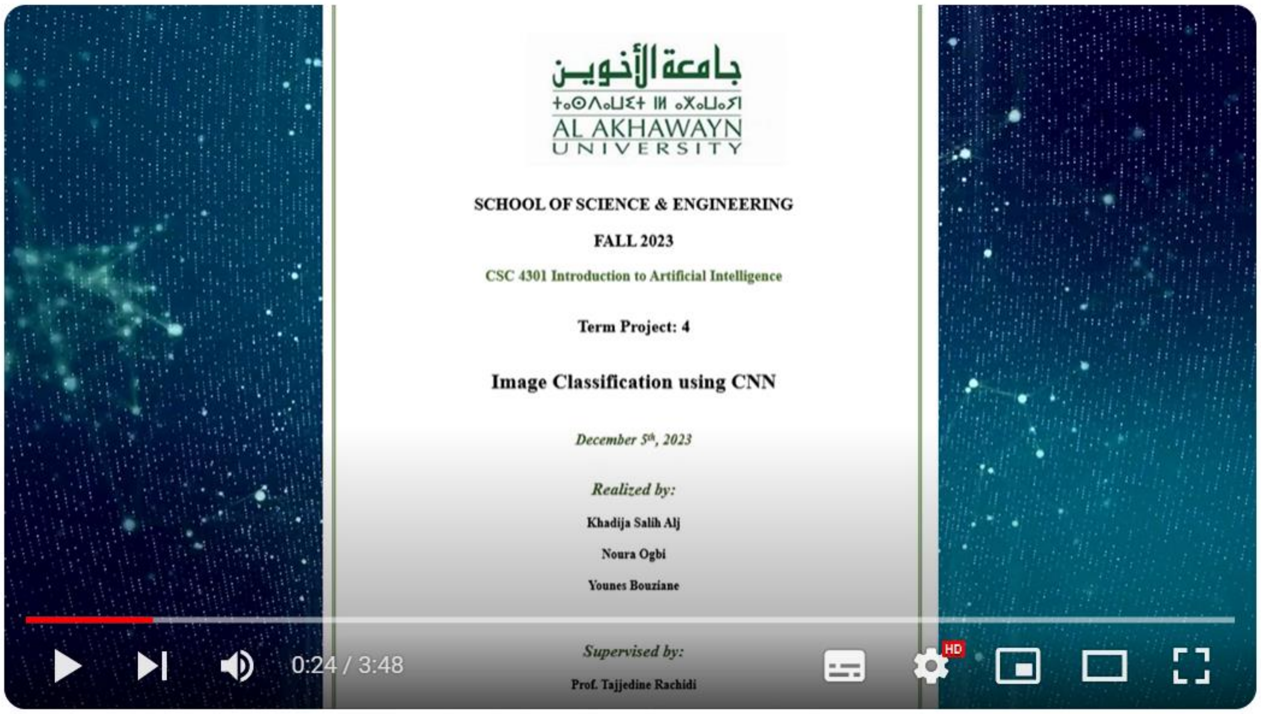
This code snippet conducts image classification using the CIFAR-10 dataset. It randomly selects an image index from the CIFAR-10 test dataset, utilizes Matplotlib to visualize the chosen image, and preprocesses it to align with the input requirements of the neural network model. The image is then fed into the model, and the resulting predictions determine the predicted class index. By mapping this index to a human-readable label based on CIFAR-10 categories, the code establishes the predicted label. Additionally, it retrieves the original label of the randomly selected CIFAR-10 test image. The final output is a print statement displaying both the original and predicted labels, offering a straightforward presentation of the model's classification outcome within the context of the CIFAR-10 dataset. Overall, this code encapsulates a fundamental image classification workflow, specifically tailored to CIFAR-10.


```
Original label is cat and predicted label is cat
```

As evidenced in the Youtube demo, our model excels in predictive capabilities, demonstrating proficiency in accurately anticipating outcomes. The showcased performance underscores its reliability and effectiveness, with observed predictive accuracy of 87% 😊.

V. YouTube Demo

[CNN Image Classification on CIFAR-10 (youtube.com)]


جامعة الأخوين

ⵜⴰⵙⴷⴰⵡⵉⵜ ⵏ ⵍⴰⵅⴰⵡⴰⵢⵏ

AL AKHAWAYN UNIVERSITY

SCHOOL OF SCIENCE & ENGINEERING

FALL 2023

CSC 4301 Introduction to Artificial Intelligence

Term Project: 4

Image Classification using CNN

December 5th, 2023

Realized by:

Khadija Salih Alj

Noura Ogbi

Younes Bouziane

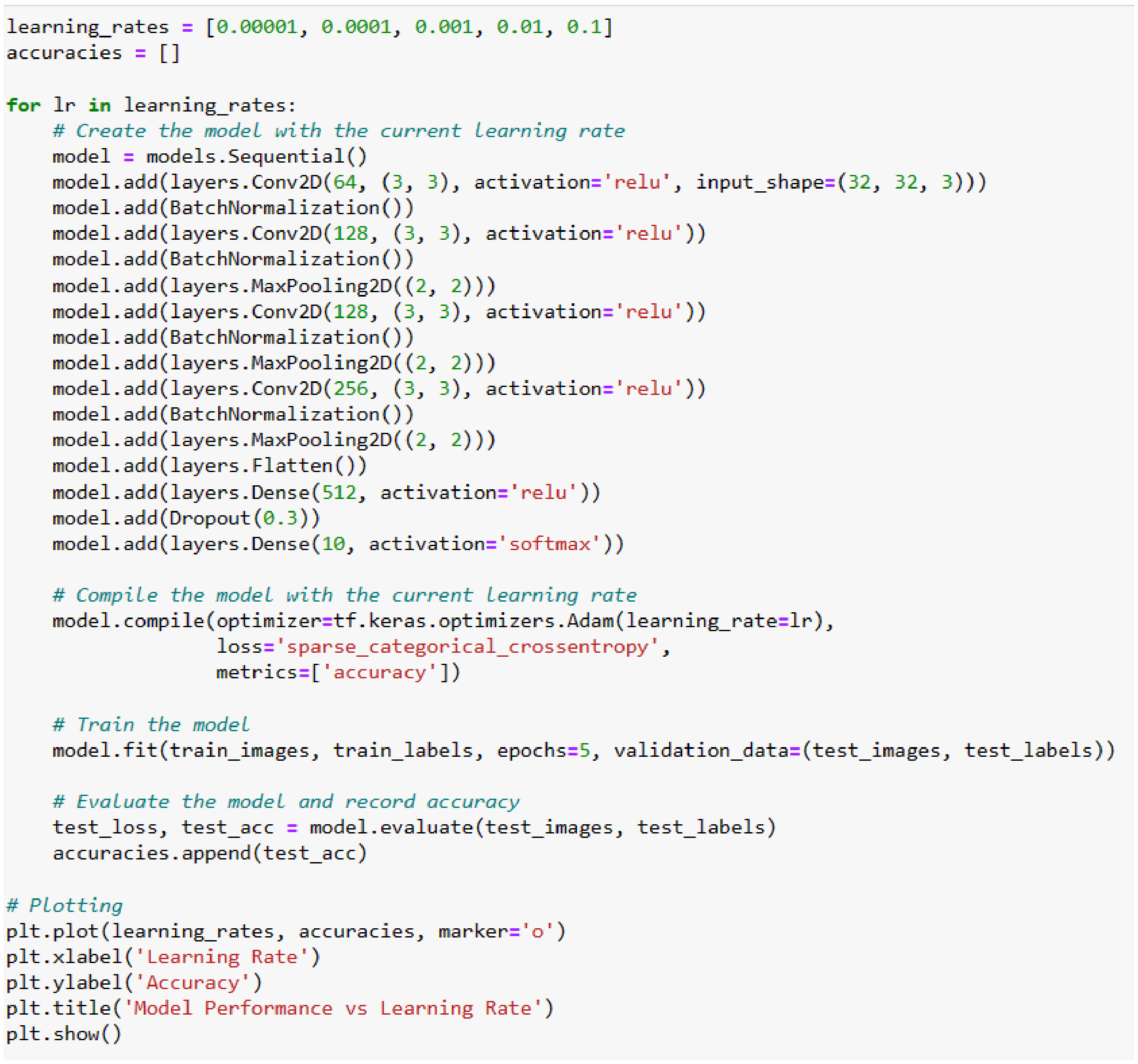
Supervised by:

Prof. Tajjedine Rachidi

CNN Image Classification on CIFAR-10

VI. Hyperparameter Exploration

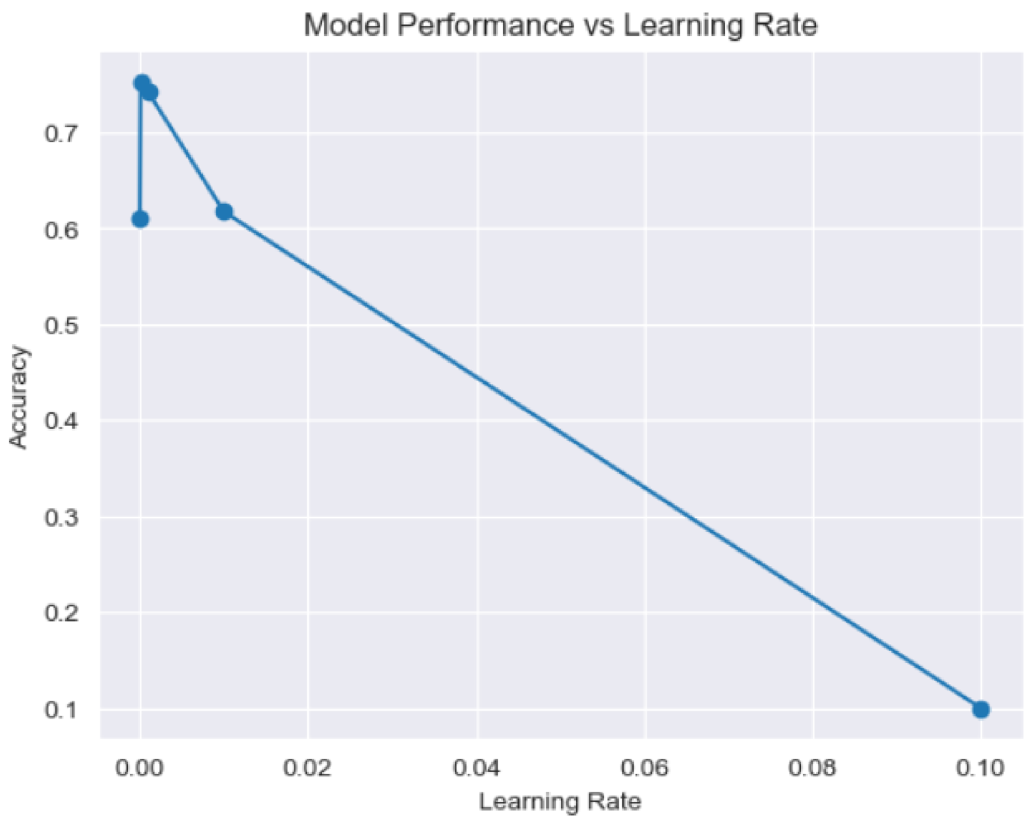
1) Learning Rate



```python
learning_rates = [0.00001, 0.0001, 0.001, 0.01, 0.1]
accuracies = []

for lr in learning_rates:
    # 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 and record accuracy
    test_loss, test_acc = model.evaluate(test_images, test_labels)
    accuracies.append(test_acc)

# Plotting
plt.plot(learning_rates, accuracies, marker='o')
plt.xlabel('Learning Rate')
plt.ylabel('Accuracy')
plt.title('Model Performance vs Learning Rate')
plt.show()
```

The above code performs a systematic analysis of the impact of different learning rates on the performance of a convolutional neural network (CNN) model. For each specified learning rate in the ‘learning_rates’ array, a new instance of the CNN model is created. This model consists of convolutional layers, batch normalization, max-pooling, and dense layers (same one from second ewperimentation model). The model is compiled with the current learning rate using the Adam optimizer, and then trained on the training data for a fixed number of epochs (5). After training, the model’s accuracy on the test set is evaluated and recorded. The process is repeated for each learning rate in the array.

The learning rate is a crucial parameter in gradient descent optimization for CNNs. A **high learning rate can lead to rapid convergence**, but it carries the risk of **overshooting the optimal minimum**, potentially leading to suboptimal training results. Conversely, a low learning rate, while ensuring more **stable convergence**, might result in slow progress and the risk of the model getting trapped in **local minima**. To address these challenges, **adaptive learning rates** are employed in optimizers like Adam and Adagrad. These adaptive methods dynamically adjust the learning rate during training, which can lead to more efficient convergence by meeting the varying needs of the model at different training stages.



Finally, the script plots a graph illustrating the relationship between learning rates and model accuracy, providing a visual representation of how changes in the learning rate affect the overall performance of the model.

- **Plot Analysis:**

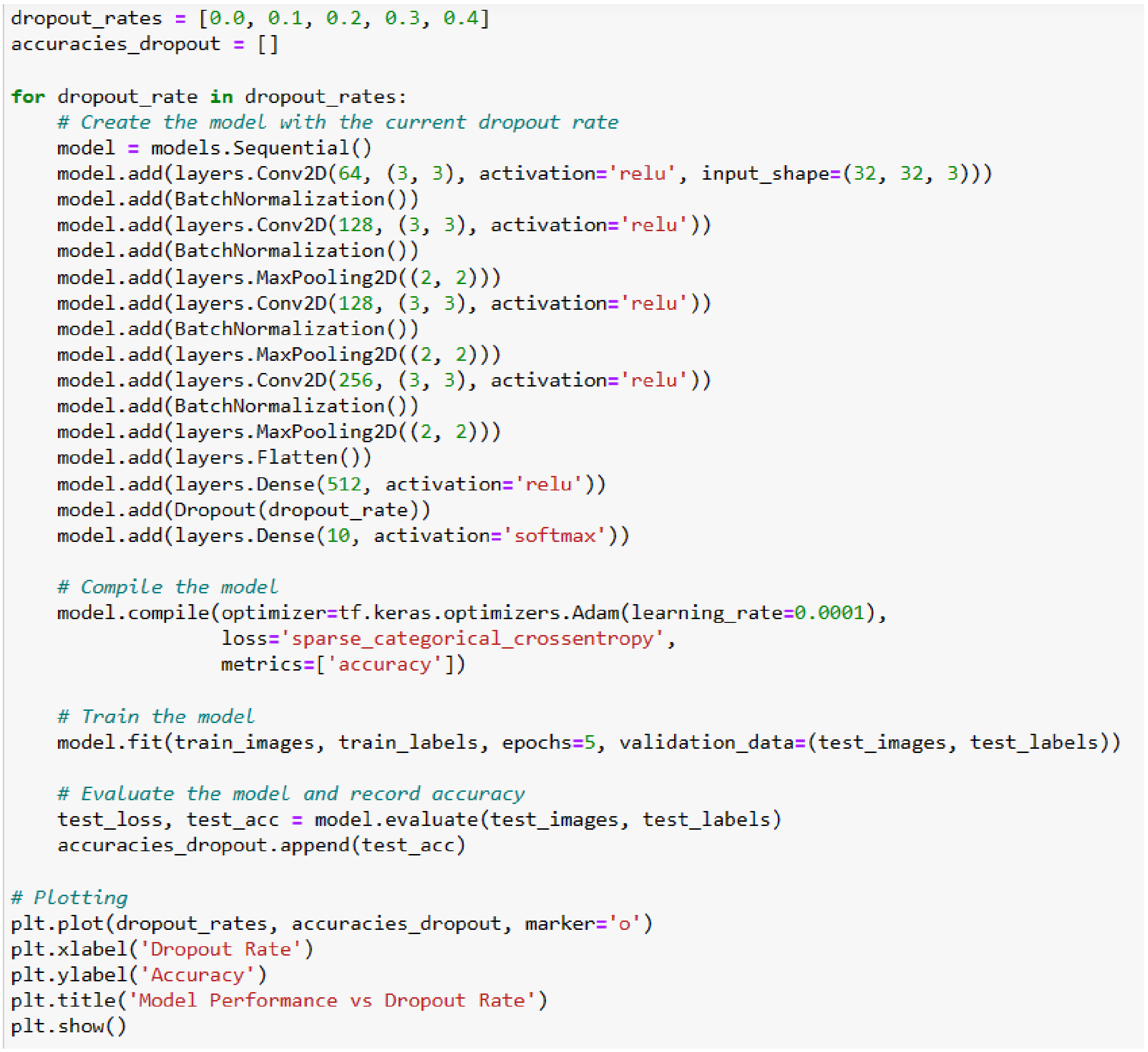
The analysis of the plot reveals interesting insights into the impact of different learning rates on the model’s accuracy.

- For a learning rate of 0.00001, the accuracy is 0.6097. This very low learning rate appears to hinder the model’s ability to converge effectively during training, resulting in a comparatively lower accuracy.

- At a learning rate of 0.0001, the accuracy sees a significant improvement, reaching 0.7532. This suggests that a moderate learning rate allows the model to strike a balance between rapid convergence and stability during training, leading to enhanced accuracy.
- For a learning rate of 0.001, the accuracy slightly decreases to 0.7418. This suggests that, in this context, a higher learning rate might lead to overshooting optimal parameter values during training, impacting accuracy negatively.
- The model's accuracy decreases further at a learning rate of 0.01, reaching 0.6174. This indicates that a higher learning rate may cause the model to oscillate around optimal parameter values, hindering convergence and resulting in decreased accuracy.
- Finally, at a learning rate of 0.1, the accuracy sharply drops to 0.1. This substantial decrease suggests that an excessively high learning rate can lead to divergent behavior during training, preventing the model from effectively learning and achieving accurate predictions.

In summary, the analysis underscores the critical role of selecting an appropriate learning rate to facilitate effective model training and achieve optimal accuracy. The model's sensitivity to learning rate variations emphasizes the need for careful hyperparameter tuning to strike the right balance between convergence speed and stability.

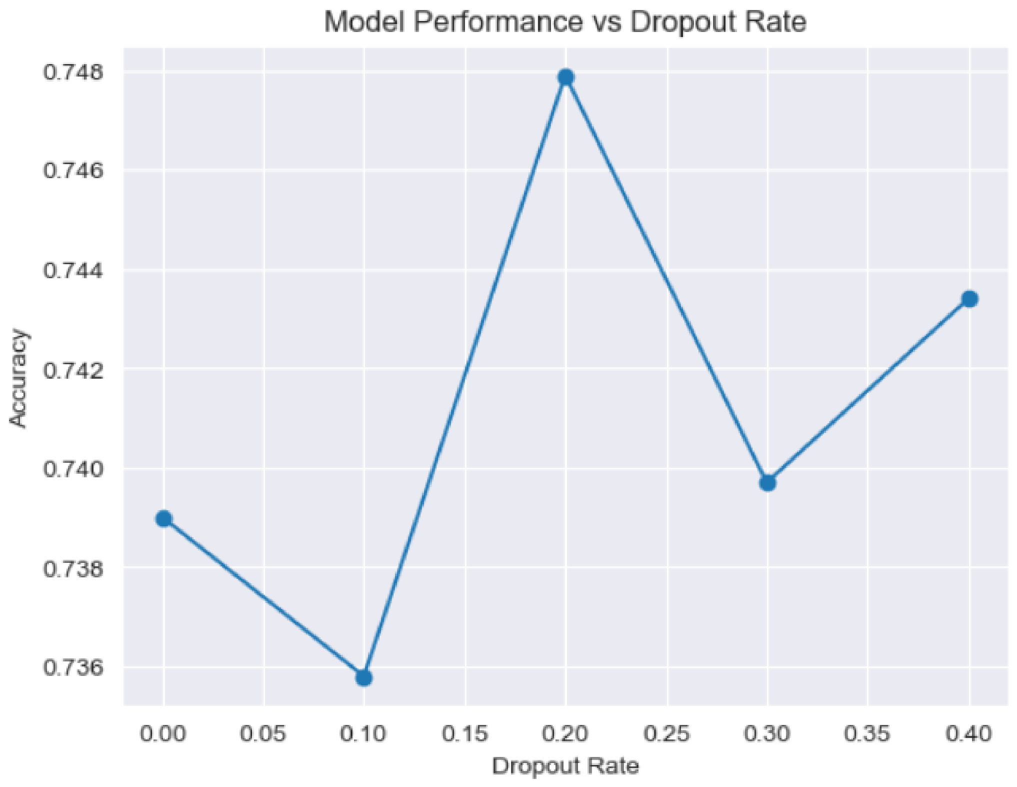
2) Drop Rate



```python
dropout_rates = [0.0, 0.1, 0.2, 0.3, 0.4]
accuracies_dropout = []

for dropout_rate in dropout_rates:
    # Create the model with the current dropout 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(dropout_rate))
    model.add(layers.Dense(10, activation='softmax'))

    # Compile the model
    model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.0001),
                  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 and record accuracy
    test_loss, test_acc = model.evaluate(test_images, test_labels)
    accuracies_dropout.append(test_acc)

# Plotting
plt.plot(dropout_rates, accuracies_dropout, marker='o')
plt.xlabel('Dropout Rate')
plt.ylabel('Accuracy')
plt.title('Model Performance vs Dropout Rate')
plt.show()
```

This code examines the influence of varying dropout rates on a Convolutional Neural Network (CNN) designed for image classification. It creates multiple CNN models with different dropout rates, ranging from 0.0 to 0.4, and evaluates their performance on the CIFAR-10 dataset. The models consist of convolutional layers, batch normalization, max-pooling, and dense layers, with dropout applied to enhance regularization. The Adam optimizer, fixed learning rate, and sparse categorical crossentropy loss are employed for compilation. Training each model for 5 epochs, the test accuracy is recorded and plotted against dropout rates.

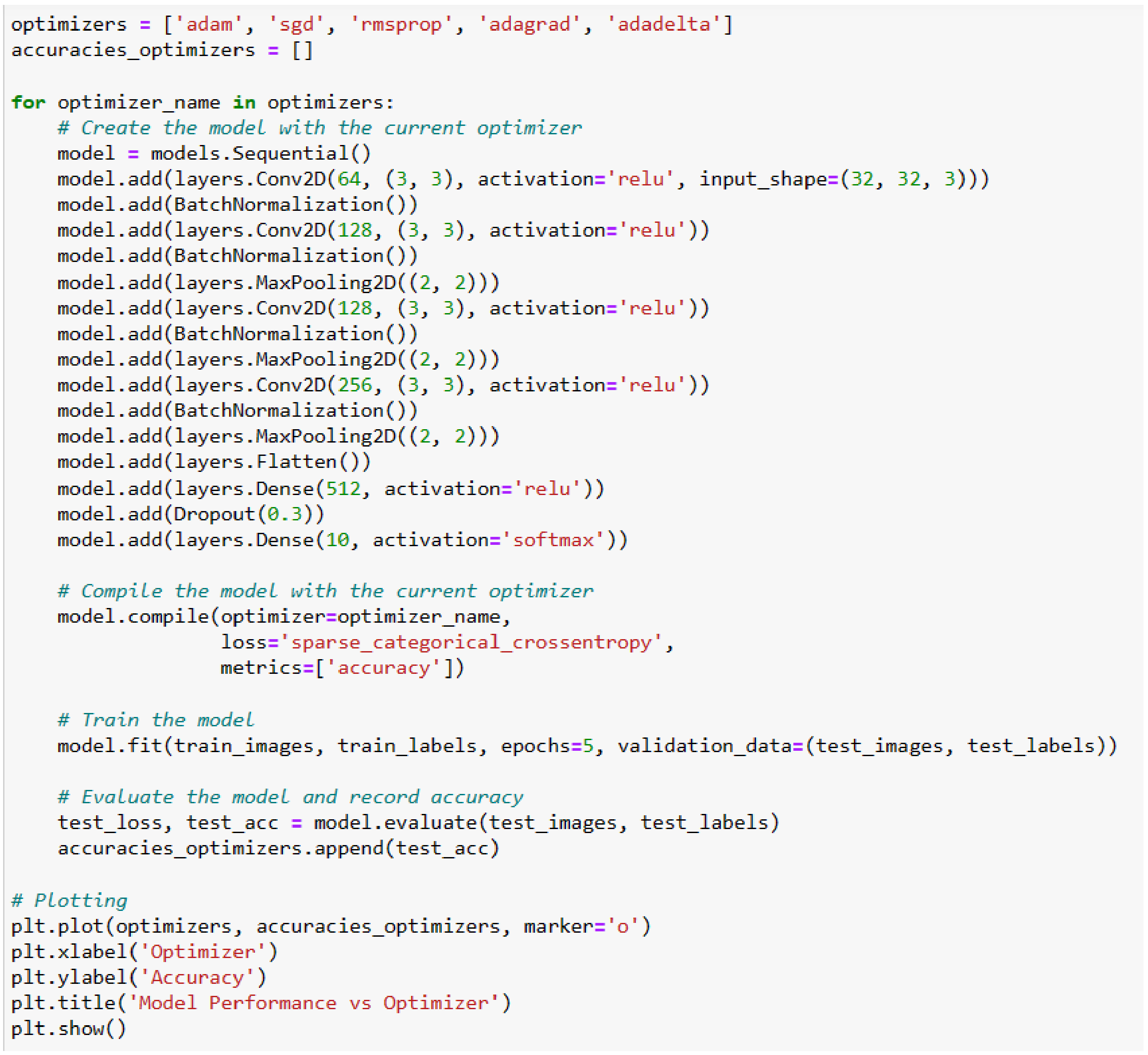
The dropout rate is a regularization technique used to improve the **robustness and generalization of CNNs**. By varying the dropout rate, the model can prevent complex co-adaptations on training data, thereby **reducing the risk of overfitting**. Implementing different dropout rates at various layers of the network allows for a nuanced approach to regularization, which can optimize the learning and generalization capabilities of the network. Layer-specific dropout rates provide the flexibility to tailor the dropout's impact according to the complexity and requirements of different layers within the network.



➤ <u>**Plot Analysis:**</u>

In analyzing the above plot, it appears that a dropout rate of 0.2 yields the highest accuracy of 74.79%, showcasing the model's optimal performance on the CIFAR-10 dataset. However, it is important to note that dropout rates of 0, 0.1, 0.3, and 0.4 also demonstrate competitive accuracies, ranging from 73.90% to 74.34%. This suggests that the model is relatively robust to variations in dropout rates within this range, with subtle fluctuations in accuracy. It's advisable to consider the trade-off between model complexity and regularization when selecting the dropout rate, aiming for a balance that enhances generalization while maintaining performance.

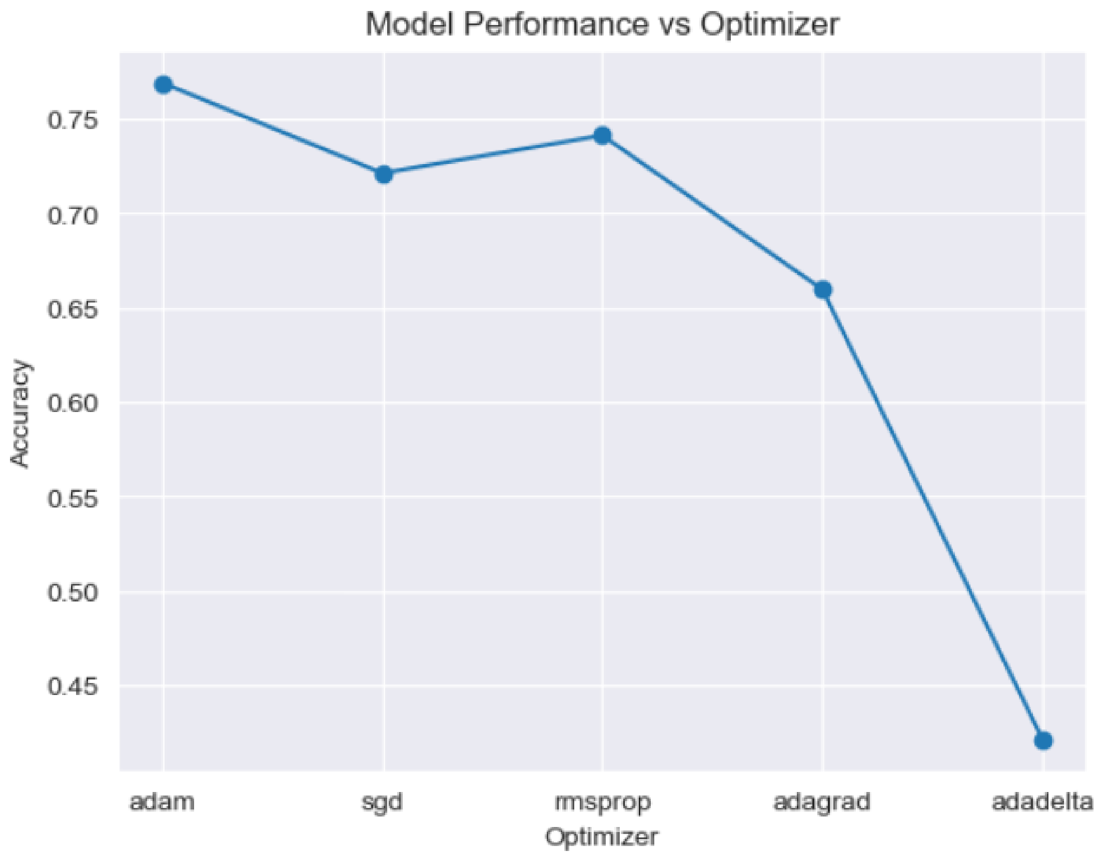
3) Optimizer Choice



```python
optimizers = ['adam', 'sgd', 'rmsprop', 'adagrad', 'adadelta']
accuracies_optimizers = []

for optimizer_name in optimizers:
    # Create the model with the current optimizer
    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 optimizer
    model.compile(optimizer=optimizer_name,
                  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 and record accuracy
    test_loss, test_acc = model.evaluate(test_images, test_labels)
    accuracies_optimizers.append(test_acc)

# Plotting
plt.plot(optimizers, accuracies_optimizers, marker='o')
plt.xlabel('Optimizer')
plt.ylabel('Accuracy')
plt.title('Model Performance vs Optimizer')
plt.show()
```

The above code is designed to assess the influence of various optimization algorithms. It systematically tests the ‘Adam’ optimizer, which stands for Adaptive Moment Estimation, along with ‘SGD’ (Stochastic Gradient Descent), ‘RMSprop’ (Root Mean Square Propagation), ‘Adagrad’ (Adaptive Gradient Algorithm), and ‘Adadelta’ (Adaptive Delta).

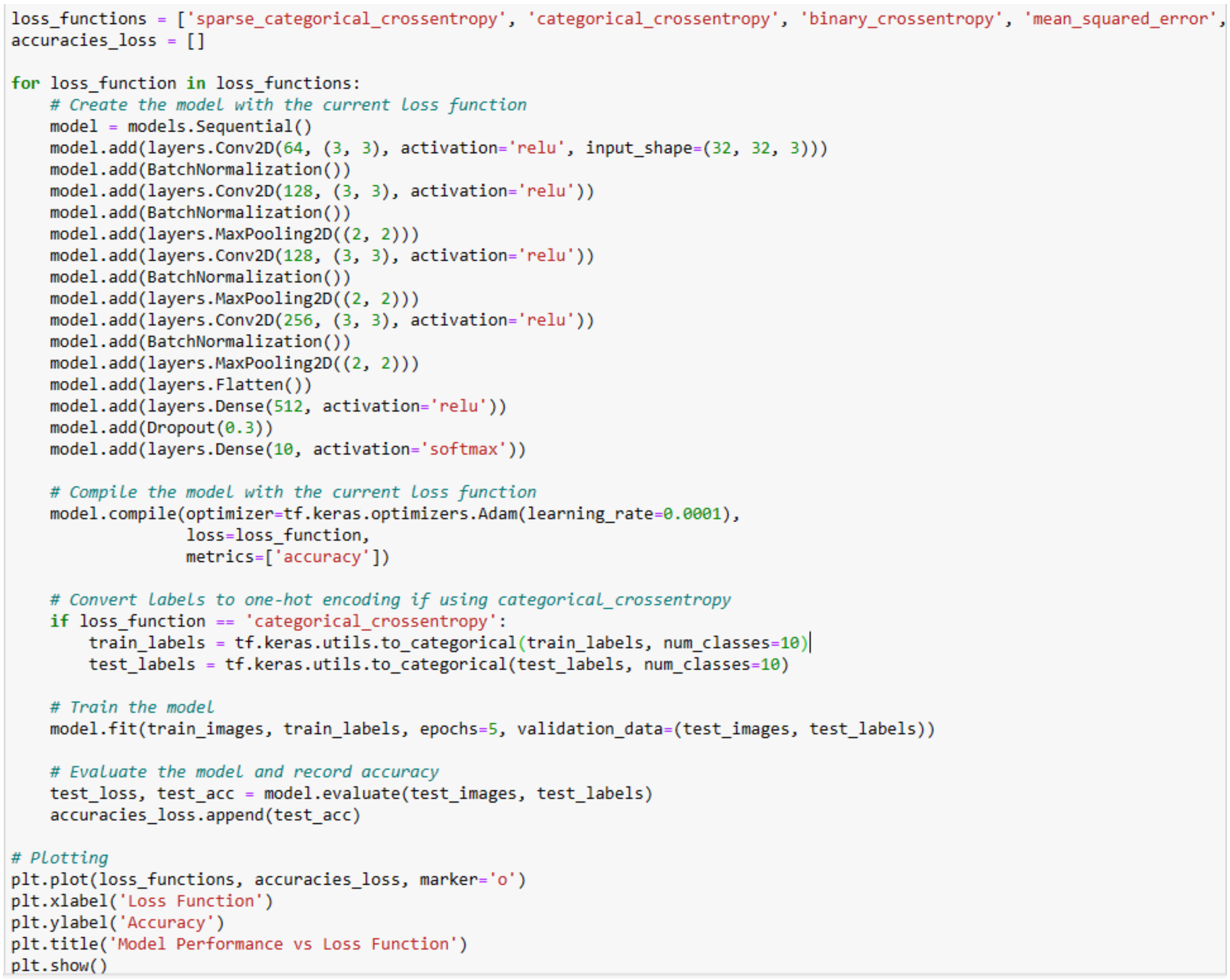
The choice of optimizer is key in determining how effectively a CNN learns. Different optimizers, such as Adam, SGD, and RMSprop, each use unique mathematical algorithms to minimize loss functions. These differences can significantly impact the training process, with some optimizers better suited for specific types of tasks. For instance, Adam is often preferred for its **adaptive learning rate capabilities,** which can lead to quicker convergence in complex scenarios.



- **<u>Plot Analysis:</u>**

In the analysis of various optimizers, the plot illustrates distinct impacts on the model's performance. The **'adam' optimizer outperforms others**, achieving the highest accuracy (76.91%), indicating its effectiveness in optimizing the model parameters. On the other hand, the 'sgd' optimizer, representing **Stochastic Gradient Descent**, and **'rmsprop' (Root Mean Square Propagation)** exhibit lower accuracies (72.13% and 74.13% respectively), suggesting limitations in convergence. Notably, **'adagrad' and 'adadelta'** optimizers perform less, having lower accuracies of 65.97% and 42.18% respectively. From this graph, we conclude that the choice of the optimizer significantly influences the learning process.

4) Loss Function Selection



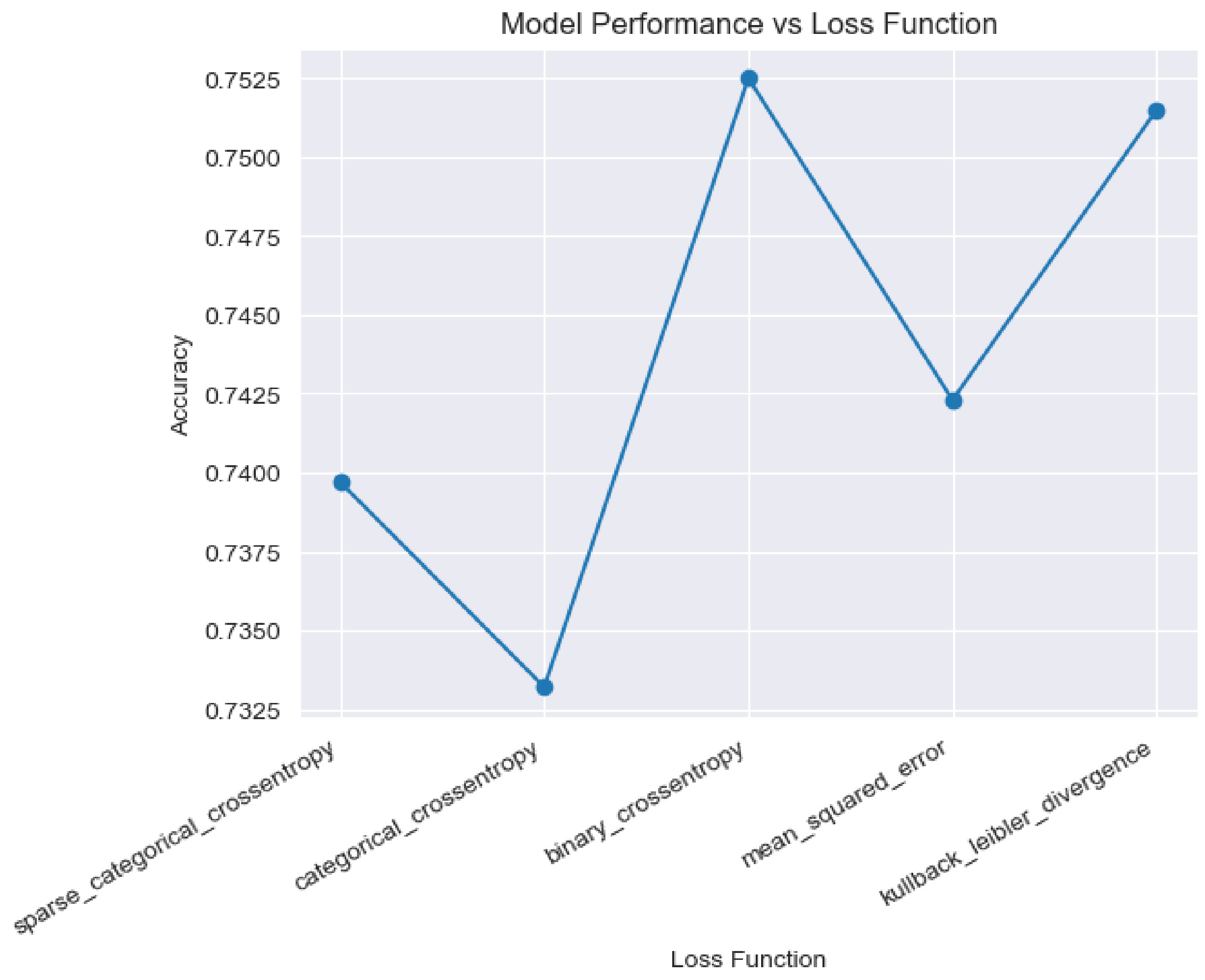
```
loss_functions = ['sparse_categorical_crossentropy', 'categorical_crossentropy', 'binary_crossentropy', 'mean_squared_error',
accuracies_loss = []

for loss_function in loss_functions:
    # Create the model with the current loss function
    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 loss function
    model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.0001),
                  loss=loss_function,
                  metrics=['accuracy'])

    # Convert labels to one-hot encoding if using categorical_crossentropy
    if loss_function == 'categorical_crossentropy':
        train_labels = tf.keras.utils.to_categorical(train_labels, num_classes=10)
        test_labels = tf.keras.utils.to_categorical(test_labels, num_classes=10)

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

    # Evaluate the model and record accuracy
    test_loss, test_acc = model.evaluate(test_images, test_labels)
    accuracies_loss.append(test_acc)

# Plotting
plt.plot(loss_functions, accuracies_loss, marker='o')
plt.xlabel('Loss Function')
plt.ylabel('Accuracy')
plt.title('Model Performance vs Loss Function')
plt.show()
```

The above code tests the performance of our convolutional neural network using various loss functions. It goes through a list of different loss functions, such as categorical crossentropy, mean squared error, and kullback leibler divergence.

After encountering a Value Error, we realized that sparse categorical crossentropy expects integer labels, while categorical crossentropy expects one-hot encoded labels. Therefore, we adjusted the labels in the case of categorical crossentropy loss function, and then evaluated the accuracy on a test dataset.

Selecting an appropriate loss function is critical in guiding CNN towards accurate predictions. Different loss functions, like categorical crossentropy and mean squared error, respond variably to errors based on the model's task. Understanding these differences and their mathematical formulations is essential for fine-tuning the model's learning process. This choice directly

influences how the model weights are updated during training, affecting both accuracy and efficiency.



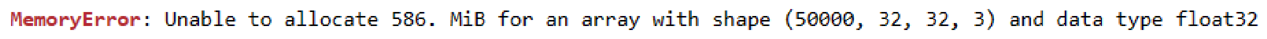
- **<u>Plot Analysis:</u>**

Analyzing the plot, it appears that the model's performance varies slightly depending on the choice of loss function. The highest accuracy is achieved with the 'binary_crossentropy' loss function, with an accuracy of 75.25%. It is interesting to note that the differences in accuracy among the loss functions are relatively small. This suggests that, in this specific scenario, the choice of loss function may not have a significant impact on the overall model performance.

VII. Conclusion

In this Computer Vision project, our focal point was the development of a high-performing image classifier leveraging Convolutional Neural Networks (CNNs) within the Keras framework. This report encapsulates a triad of experiments, each building upon the preceding one, ultimately attaining an impressive accuracy of 87%. These iterations involve strategic enhancements, encompassing model deepening, prolonged training durations, and meticulous adjustments to enhance accuracy.

Nevertheless, our path was not without challenges. Overheating issues, spontaneous shutdowns, and out-of-memory errors highlighted the intense computational demands of training these models. This report serves as a transparent narrative of our CNN journey, shedding light on the complexities of model training in this domain.



```
MemoryError: Unable to allocate 586. MiB for an array with shape (50000, 32, 32, 3) and data type float32
```

Looking forward, the insights gained from our iterative refinement offer a roadmap for future applications. The lessons learned about hyperparameters, model architecture, and computational considerations provide valuable guidance for future projects at the intersection of computer vision and machine learning. As technology progresses, this fusion holds great promise, and the experiences shared here contribute to the foundation of that promising future…