UECS3413 Digital Image Processing

Jan 2023

Group Assignment

Task: Image Classification

Group Member Details:

ID	Name	Programme	Role
2004564	Olivia Ong Yee Ming	SE	Leader
1801970	Chow Kok Keong	SE	Member
2000515	Cheng Li En	SE	Member

Table of Contents

Abstract

In this report, we evaluated the performance of four different models, namely LeNet 5, CNN, AlexNet, and Inception V4, on a given dataset. The primary objective was to assess their effectiveness in terms of training and validation losses, as well as training and validation accuracies. Additionally, the models' performance was tested on an independent test dataset to determine their generalization capabilities.

Among the models evaluated, the Inception V4 model utilizing the Adam optimizer emerged as the top performer. It achieved the lowest test loss of 1.7417 and the highest test accuracy of 0.6401, indicating its strong predictive power on unseen data. The model also demonstrated impressive results in terms of training and validation losses, with values of 0.1978 and 1.8616, respectively. Furthermore, it achieved high training and validation accuracies, measuring 0.9373 and 0.6294, respectively.

These findings highlight the superiority of the Inception V4 model in terms of its ability to minimize errors and accurately classify the dataset. Its superior performance suggests that the model is capable of effectively capturing and leveraging complex features within the data, leading to improved predictions. The use of the Adam optimizer further enhances the model's optimization process, contributing to its overall success.

Based on these results, it is recommended to consider the Inception V4 model with the Adam optimizer for similar classification tasks. Its strong performance on both the training and test datasets indicates its potential to deliver reliable and accurate predictions in real-world scenarios.

Introduction

The goal of this assignment is to develop a classification model using CNN to classify the different types of textures into their respective categories accurately in the dataset of textural images. The model should also be able to generalize well on new texture images. This problem is a multi-class classification problem.

This report will include the methodology used to classify the images, a literature review on CNN, the dataset used, the proposed method, the results and analysis of the results, some findings and recommendations, and finally the code involved.

Methodology

i. Exploratory Data Analysis (EDA): Conduct a thorough EDA on the texture dataset to gain insights into the data. The EDA should include statistical analysis, visualization, and interpretation of the dataset.

ii. Data Pre-processing: Pre-process the texture images by resizing, normalizing, and augmenting the data to improve the model's performance.

iii. Model Architecture: Design a CNN architecture suitable for the texture classification task. The architecture should include convolutional layers, pooling layers, and fully connected layers. Experiment with different hyperparameters to improve the model's performance.

iv. Model Training: Train the CNN model on the training set using appropriate optimization techniques such as Stochastic Gradient Descent (SGD) or Adam.

v. Model Evaluation: Evaluate the performance of the classification model using different evaluation metrics such as accuracy, precision, recall, and F1-score. Use the validation set to fine-tune the model.

vi. Model Deployment: Once the model is trained and evaluated, deploy the model to the testing set and evaluate the performance on the testing set. Report the final results and discuss the limitations and future work.

Literature Review

Convolutional Neural Network (CNN)

Reference: https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-convolutional-neural-networks

CNNs are a type of neural network widely used in computer vision tasks such as image classification. A typical CNN consists of an input image, convolution layers (CONV), pooling layers (POOL), and one or more fully connected layers (FC).

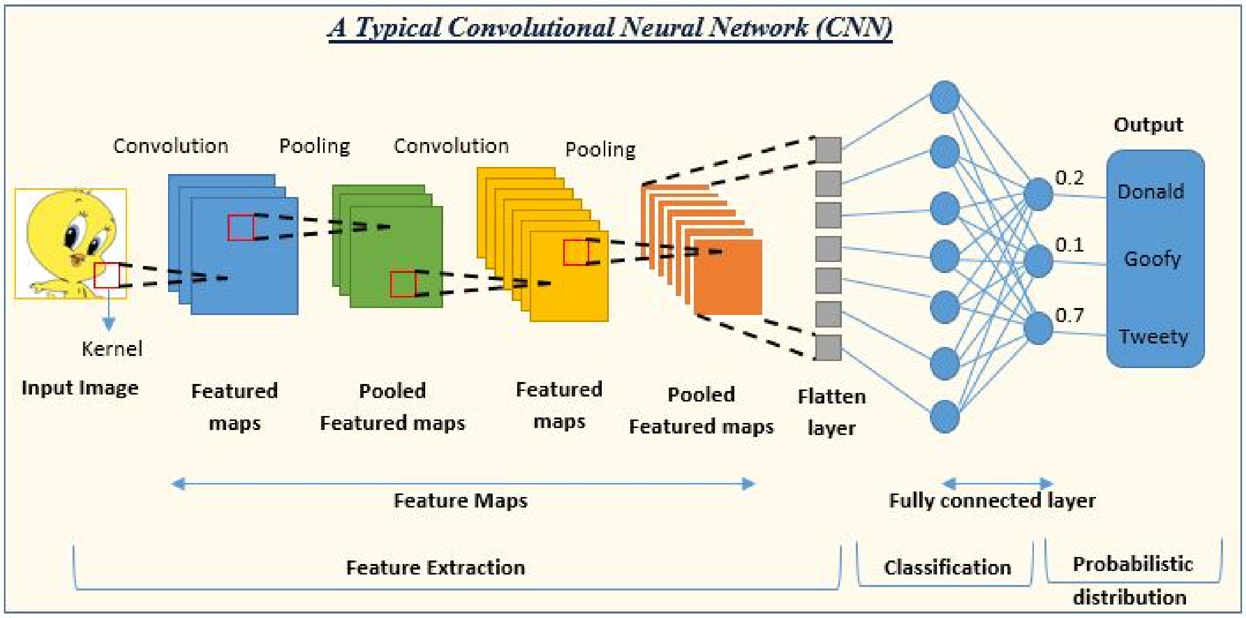
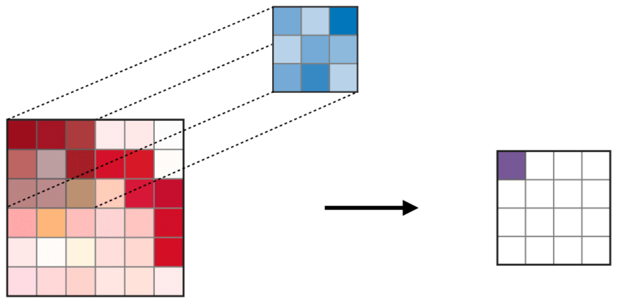


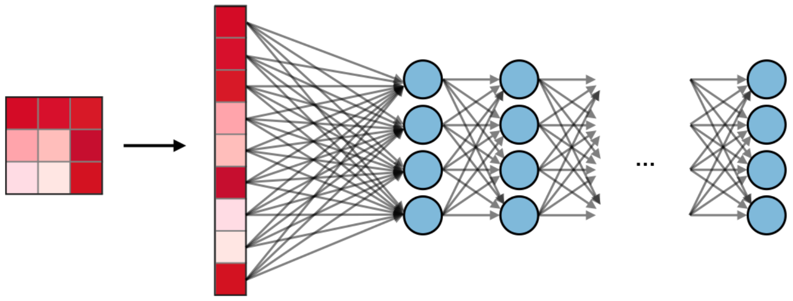
Image is taken from: https://www.analyticsvidhya.com/blog/2022/01/convolutional-neural-network-an-overview/

CONV performs feature extraction by using filters that perform convolution operations as it scans the input image I concerning its dimensions. Its hyperparameters include the filter size F which specifies the spatial extent of the filter and stride S which determines the number of pixels to skip while sliding the filter over the input image. The resulting output O is called a feature map or activation map that captures the presence of features at different spatial locations in the input image.



POOL is a downsampling operation, typically applied after a convolution layer, which does some spatial invariance. By reducing the number of features in the CNN, the model increases its computational efficiency while retaining most of the defining features of the images. Max pooling is the most used pooling operation, where the maximum value in each subregion of the feature map is taken. Average pooling, where the average value in each subregion is taken, is used in LeNet.

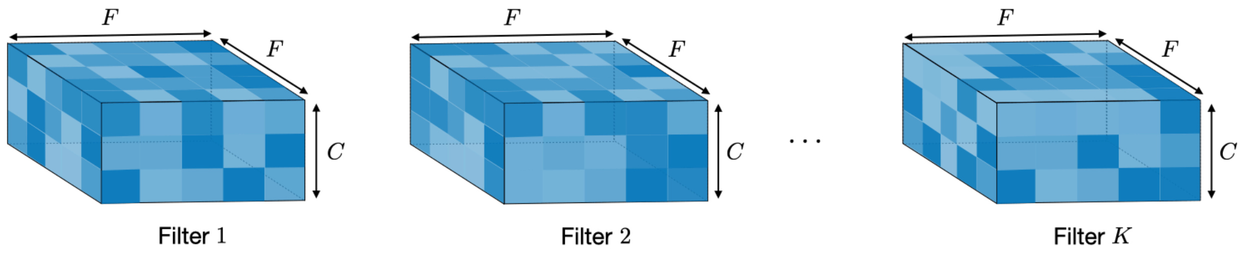
FC operates on a flattened input where each input is connected to all neurons. FC performs the task of classification and is found towards the end of CNN architectures.



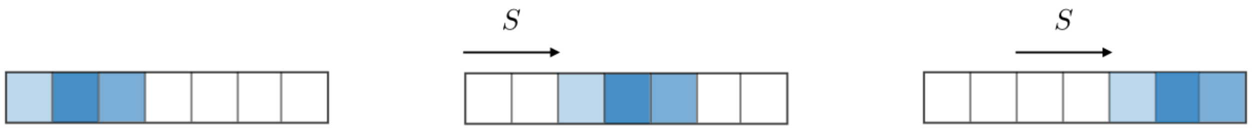
Filter Hyperparameters

Reference: https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-convolutional-neural-networks

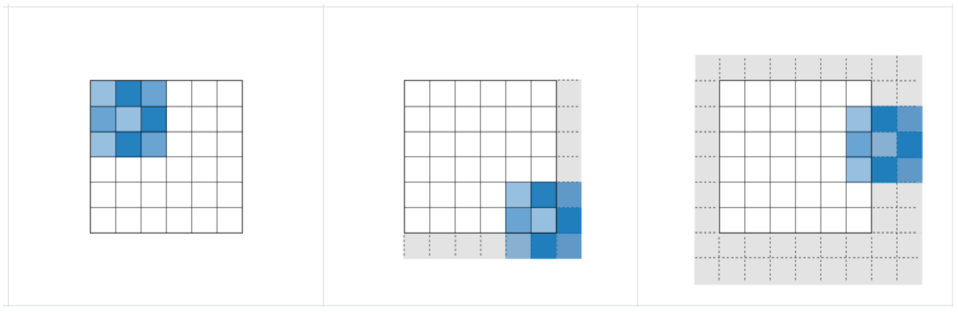
Dimensions: A filter of size F x F applied to an input containing C channels is a F x F x C volume that performs convolutions on an input of size I x I x C to produce an output feature map of size O x O x 1. The application of K filters of size F x F results in an output feature map of size O x O x K.



Stride: The stride S denotes the number of pixels by which the window moves after each operation.



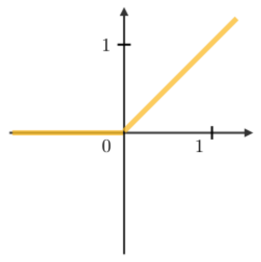
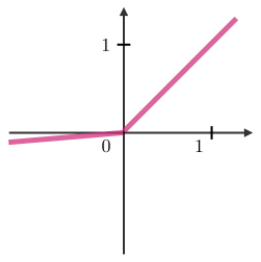
Zero-padding: Zero-padding denotes the process of adding P zeroes to each side of the boundaries of the input.



Activation Functions

Reference: https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-convolutional-neural-networks

Rectified Linear Unit (ReLU): An activation function g that is used on all elements of the volume. It aims to introduce non-linearities to the network and transform the elements of the output of the CONV into the range from 0 to infinity by replacing any negative values with 0. Leaky ReLU can address the dying ReLU issue for negative values.

ReLU	Leaky ReLU
$g(z) = \max(0, z)$	$g(z) = \max(\epsilon z, z)$ with $\epsilon \ll 1$
	

Softmax: The softmax step can be seen as a generalized logistic function that takes as input a vector of scores x and outputs a vector of output probability p through a softmax function at the end of the architecture.

$$\boxed{p = \begin{pmatrix} p_1 \\ \vdots \\ p_n \end{pmatrix}} \quad \text{where} \quad \boxed{p_i = \frac{e^{x_i}}{\displaystyle\sum_{j=1}^{n} e^{x_j}}}$$

Steps to Train a Convolutional Neural Network (CNN) to Classify Images

Reference: https://www.tensorflow.org/tutorials/images/cnn

Because the Keras Sequential API is used, creating and training the model takes just a few lines of code.

Step 1: Import libraries.

```
import tensorflow as tf
from tensorflow.keras import datasets, layers, models
import matplotlib.pyplot as plt
```

Step 2: Prepare the dataset.

The dataset used in this example is the CIFAR-10 dataset consisting of 60000 32x32 colour images in 10 classes with 6000 images per class, divided into 50000 training images and 10000 test images.

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

Step 3: Pre-process dataset.

The pixel values are normalized to be between 0 and 1.

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

Step 4: Create CNN model.

First, create the convolutional base. The base is defined using a stack of Conv2D and MaxPooling2D layers. As input, CNN takes tensors of shape (image_height, image_width, colour_channels). The CNN below is configured to process inputs of shape (32, 32, 3) by passing the argument input_shape to the first layer.

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

The architecture of the model can be displayed.

```
model.summary()
```

The output of every Conv2D and MaxPooling2D layer is a 3D tensor of shape (height, width, channels). The width and height dimensions tend to shrink when going deeper into the network. The number of output channels for each Conv2D layer is controlled by the first argument (e.g., 32 or 64). Typically, as the width and height shrink, the model can afford computationally to add more output channels in each Conv2D layer.

Then, flatten the 3D output to 1D.

```
model.add(layers.Flatten())
```

To complete the model, the last output tensor from the convolutional base is fed into one or more Dense layers to perform classification. Dense layers take vectors as input (which are 1D). CIFAR has 10 output classes, so the final Dense layer has 10 outputs.

```
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(10))
```

Step 5: Compile the model.

```
model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logi
ts=True),
              metrics=['accuracy'])
```

Step 6: Train model.

```
history = model.fit(train_images, train_labels, epochs=10,
                    validation_data=(test_images, test_labels))
```

Step 7: Evaluate the model.

```
plt.plot(history.history['accuracy'], label='accuracy')
plt.plot(history.history['val_accuracy'], label = 'val_accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.ylim([0.5, 1])
plt.legend(loc='lower right')

test_loss, test_acc = model.evaluate(test_images,  test_labels,
verbose=2)

print(test_acc)
```

The above steps shall be taken as a reference for this assignment.

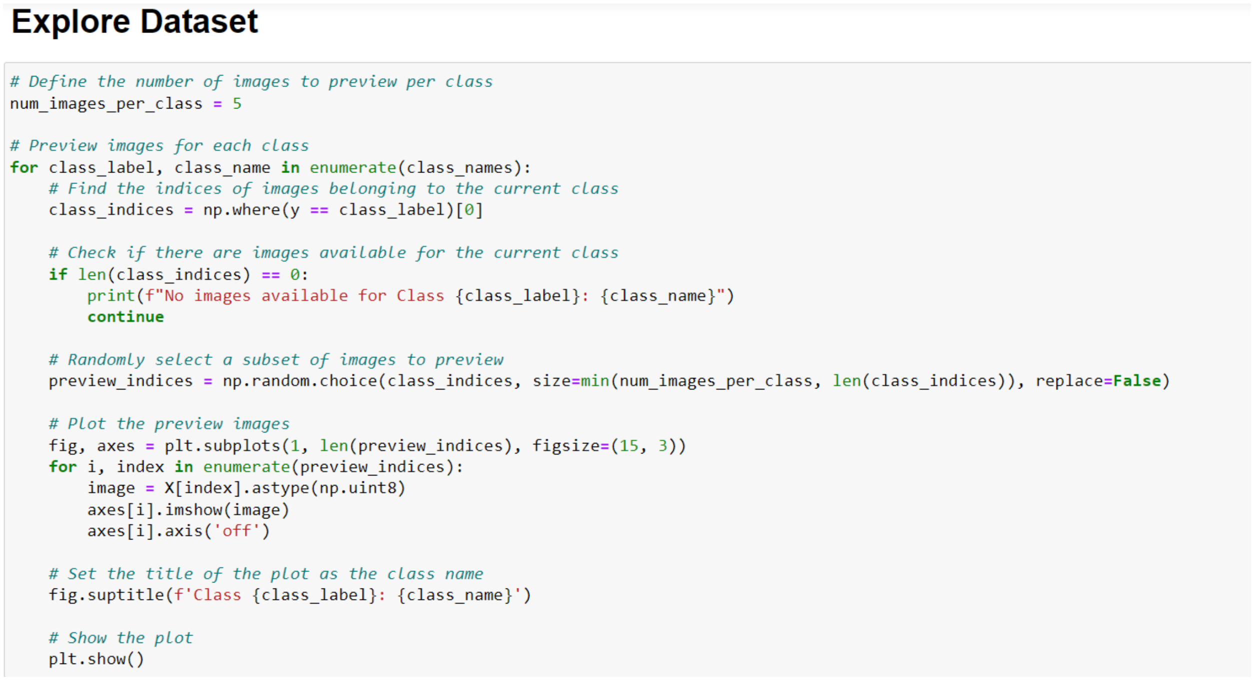
Dataset

The dataset used in this assignment is the Describable Textures Dataset (DTD) which is a texture database consisting of *5640 images* organized into *47 categories* with each category consisting of *120 images*. Image sizes range between 300x300 and 640x640, and the images contain at least 90% of the surface representing the category attribute.

The images were collected from Google and Flickr and made available to the computer vision community for research purposes. The image collection of textural images is inspired by the perceptual properties of textures that elicit a rich array of visual impressions and the power of the human visual system to vividly describe the content of images. The aim is to reproduce this capability in machines and to gain further insight into how textural information may be processed, analyzed, and represented by an intelligent system. Compared to the classic task of textural analysis such as material recognition, perceptual properties are much richer in variety and structure, inviting new challenges.

A link to the image database: *https://www.robots.ox.ac.uk/~vgg/data/dtd/*

Below are some example images of each class:



Explore Dataset

```
# Define the number of images to preview per class
num_images_per_class = 5

# Preview images for each class
for class_label, class_name in enumerate(class_names):
    # Find the indices of images belonging to the current class
    class_indices = np.where(y == class_label)[0]

    # Check if there are images available for the current class
    if len(class_indices) == 0:
        print(f"No images available for Class {class_label}: {class_name}")
        continue

    # Randomly select a subset of images to preview
    preview_indices = np.random.choice(class_indices, size=min(num_images_per_class, len(class_indices)), replace=False)

    # Plot the preview images
    fig, axes = plt.subplots(1, len(preview_indices), figsize=(15, 3))
    for i, index in enumerate(preview_indices):
        image = X[index].astype(np.uint8)
        axes[i].imshow(image)
        axes[i].axis('off')

    # Set the title of the plot as the class name
    fig.suptitle(f'Class {class_label}: {class_name}')

    # Show the plot
    plt.show()
```

Class 0: banded

Class 1: blotchy

Class 2: braided

Class 3: bubbly

Class 4: bumpy

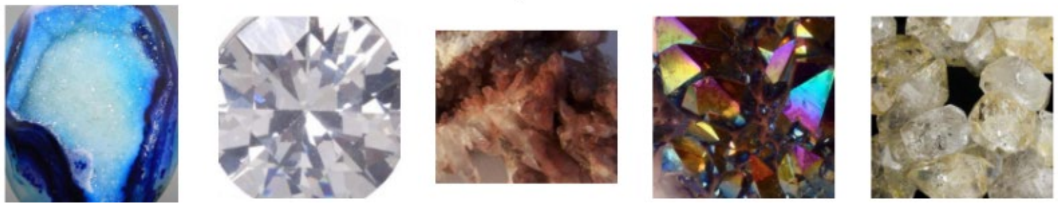
Class 5: chequered

Class 6: cobwebbed

Class 7: cracked

Class 8: crosshatched

Class 9: crystalline

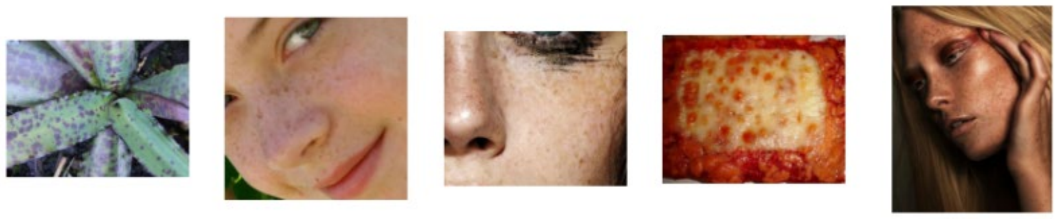


Class 10: dotted

Class 11: fibrous

Class 12: flecked

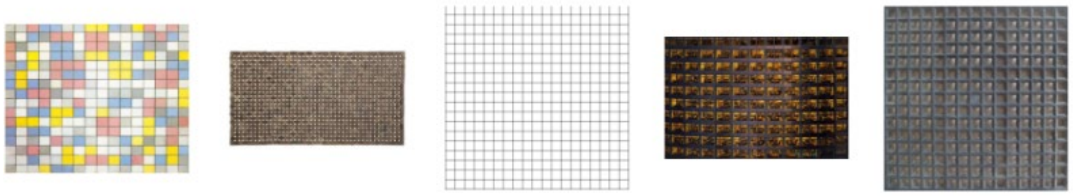
Class 13: freckled



Class 14: frilly

Class 15: gauzy

Class 16: grid



Class 17: grooved

Class 18: honeycombed

Class 19: interlaced

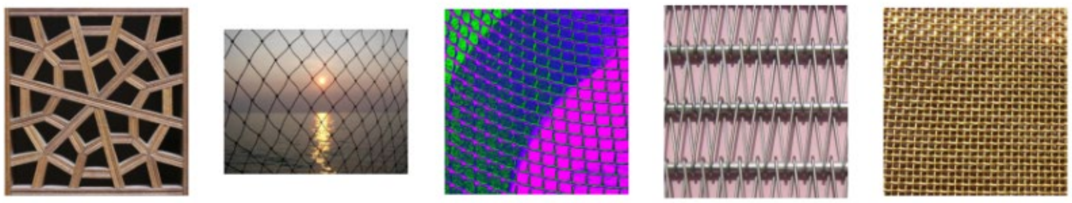
Class 20: knitted

Class 21: lacelike

Class 22: lined

Class 23: marbled

Class 24: matted


Class 25: meshed

Class 26: paisley

Class 27: perforated

Class 28: pitted

Class 29: pleated

Class 30: polka-dotted

Class 31: porous

Class 32: potholed

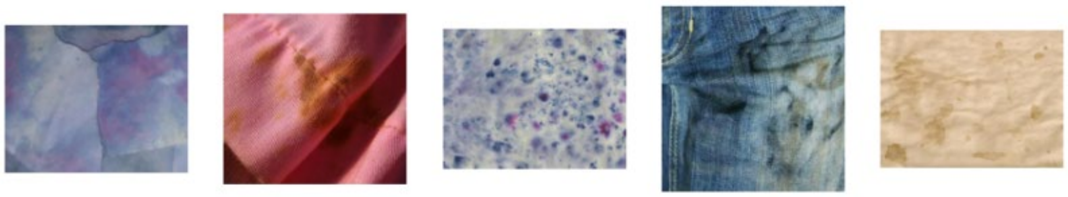
Class 33: scaly

Class 34: smeared

Class 35: spiralled

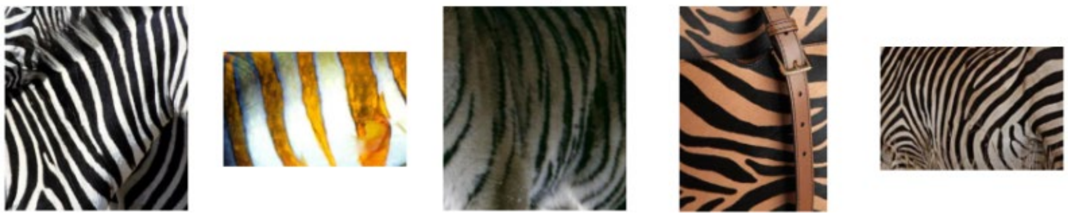
Class 36: sprinkled

Class 37: stained



Class 38: stratified

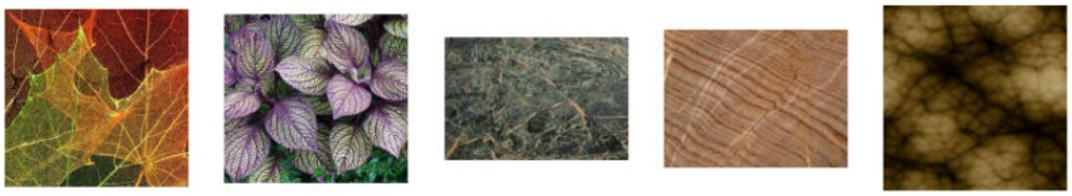
Class 39: striped



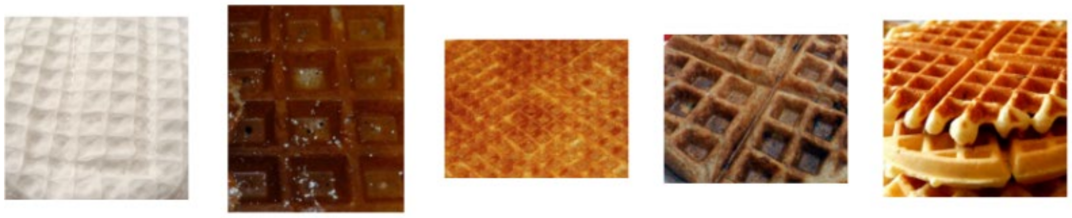
Class 40: studded

Class 41: swirly

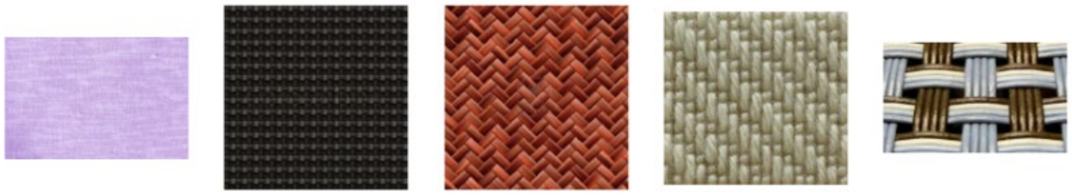
Class 42: veined



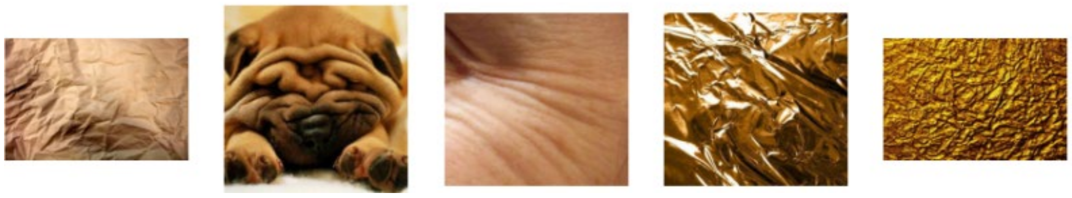
Class 43: waffled



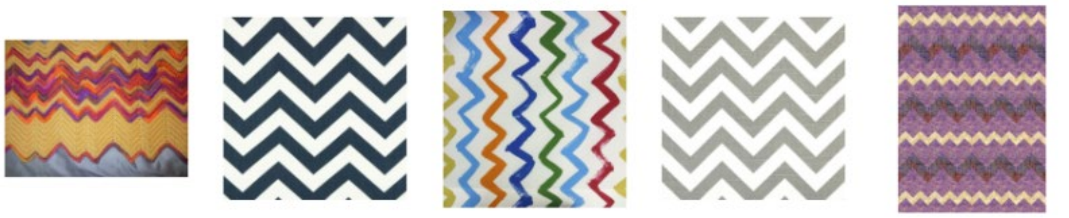
Class 44: woven



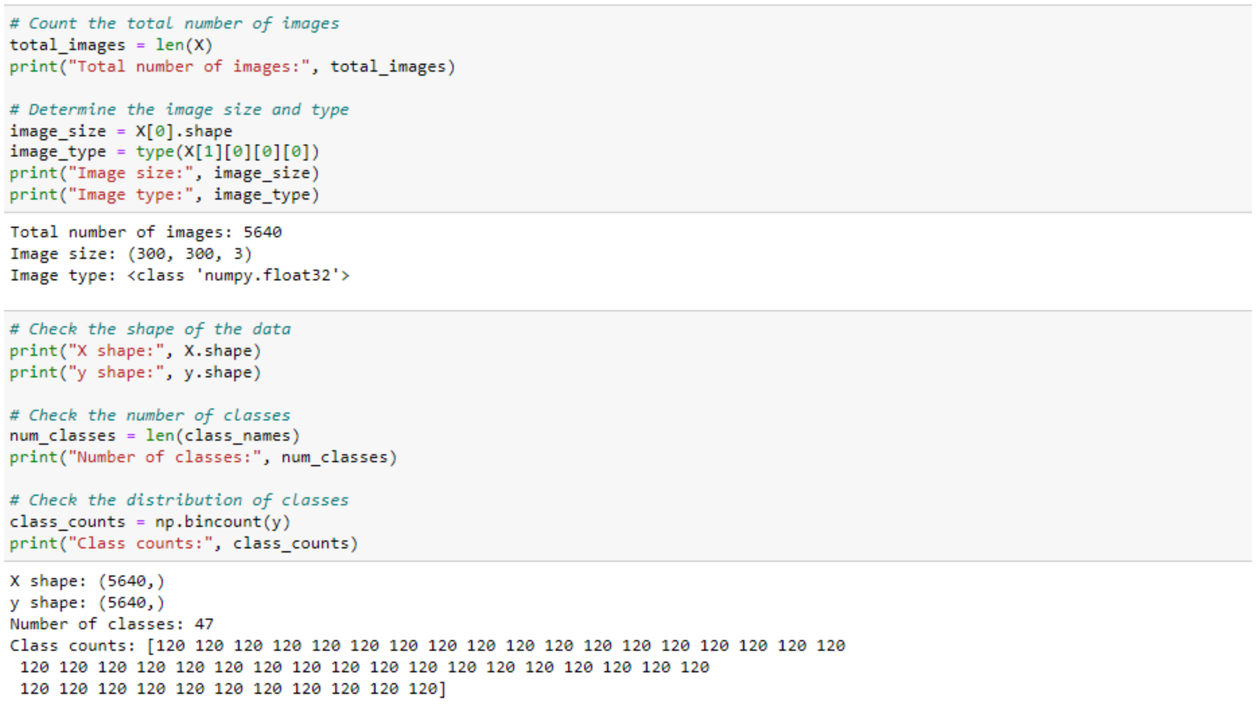
Class 45: wrinkled



Class 46: zigzagged



Below is some information about the dataset:



```python
# Count the total number of images
total_images = len(X)
print("Total number of images:", total_images)

# Determine the image size and type
image_size = X[0].shape
image_type = type(X[1][0][0][0])
print("Image size:", image_size)
print("Image type:", image_type)
```

```
Total number of images: 5640
Image size: (300, 300, 3)
Image type: <class 'numpy.float32'>
```

```python
# Check the shape of the data
print("X shape:", X.shape)
print("y shape:", y.shape)

# Check the number of classes
num_classes = len(class_names)
print("Number of classes:", num_classes)

# Check the distribution of classes
class_counts = np.bincount(y)
print("Class counts:", class_counts)
```

```
X shape: (5640,)
y shape: (5640,)
Number of classes: 47
Class counts: [120 120 120 120 120 120 120 120 120 120 120 120 120 120 120 120 120 120
 120 120 120 120 120 120 120 120 120 120 120 120 120 120 120 120 120 120
 120 120 120 120 120 120 120 120 120 120 120]
```

Proposed Method

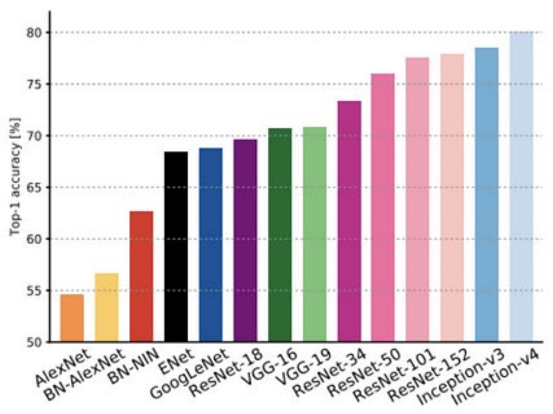
Reference: https://towardsdatascience.com/transfer-learning-using-pre-trained-alexnet-model-and-fashion-mnist-43898c2966fb

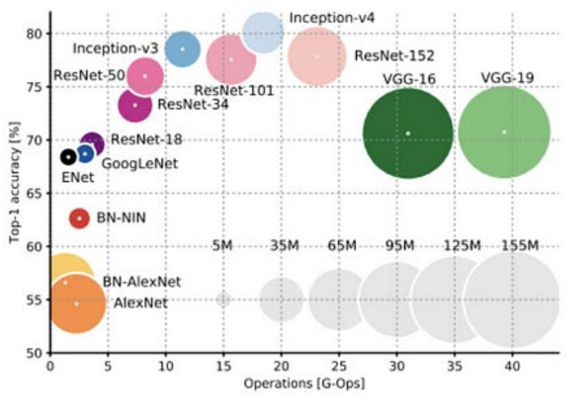
A proposed method for this classification task is transfer learning using pre-trained models for domain adaptation and strengthening of model accuracy.

Limited by our computational capabilities, training a larger and deeper model such as the VGG16 would require more computational power than we had available. Thus, other models such as AlexNet and Inception V4 are considered.

The architecture of AlexNet contains 60,000 total parameters within 8 total layers: 5 CONV, and 3 FC.

To reduce overfitting in the model, dropout is used where neurons that do not contribute to the feedforward pass and do not participate in the backpropagation are dropped.





Reference: https://www.sciencedirect.com/science/article/pii/S1877050918309335

In this work, the CIFAR10 and CIFAR100 datasets were used to examine the performance of three distinct CNNs: AlexNet, GoogleNet, and ResNet50. To assess how consistently each of these CNNs makes predictions, the study focused on 10 classes from each dataset.

The outcomes demonstrated that transfer learning outperformed existing approaches and demonstrated higher accuracy rates. It was also noticed that difficult-to-understand frames frequently made it difficult for networks to identify and recognise the picture. For instance, beds and couches, which are distinct and simple to distinguish objects in the actual world, were confused by the trained networks.

The study also discovered that the training and accuracy rates were directly impacted by the number of layers in a network. A select few things, such as "chair," "train," and "wardrobe," were flawlessly recognised by 147-layered networks, whereas "cars" were flawlessly recognised by 177-layered networks. The effectiveness of 27-layered networks, however, was not as appreciated.

In conclusion, CNNs are proving to be the most effective methods for imbuing robots with intelligence to address a variety of practical object categorization issues. CNNs have a lot of potential for use in a wide range of applications and are simple to integrate into different platforms, even though effectively recognising and categorising complicated frames is still a challenge. Although the network must be trained on hardware that meets particular requirements, once trained, the model produced can be used with minimal constraints.

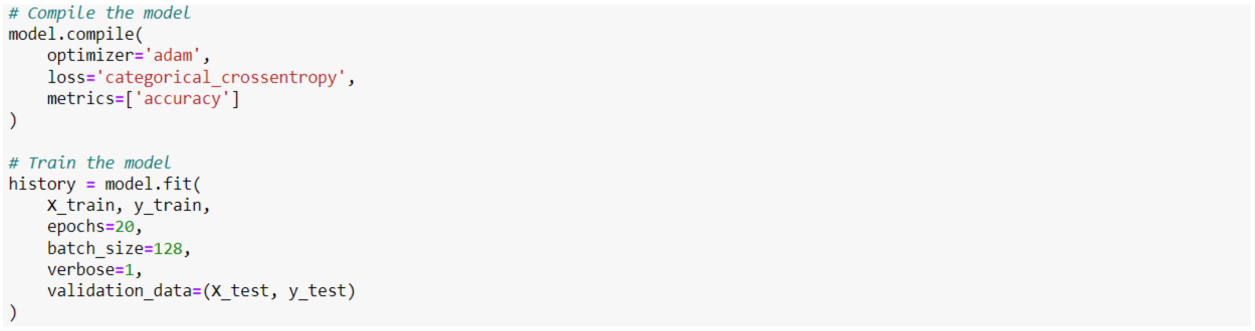
Overall, this study adds to the expanding body of knowledge about CNNs and emphasises how crucial it is to comprehend the subtleties and difficulties of object recognition and categorization tasks.

Results, Findings, and Recommendations

For all models, the input shape is (224, 224, 3) which indicates that the model expects input images of size 224x224 with 3 channels (RGB).

And for all model compilations, the loss function is categorical cross-entropy for multi-class classification, and the metric is accuracy.

The models are trained using the fit() function and the validation data is used to evaluate the model's performance during the training. The training progress and performance metrics (loss and accuracy) for both training and validation sets are stored in the 'history' object.



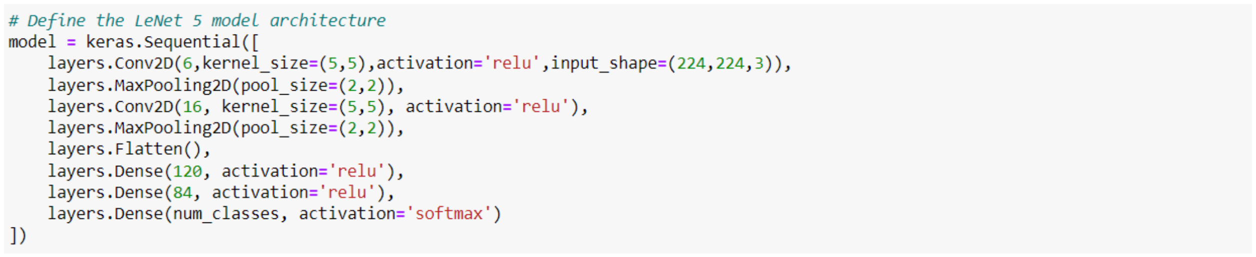
```
# Compile the model
model.compile(
    optimizer='adam',
    loss='categorical_crossentropy',
    metrics=['accuracy']
)

# Train the model
history = model.fit(
    X_train, y_train,
    epochs=20,
    batch_size=128,
    verbose=1,
    validation_data=(X_test, y_test)
)
```

For each epoch, the training and validation loss is reported. The loss value represents the average loss of the model's predictions compared to the true labels.

Also, for each epoch, the training accuracy and validation accuracy are reported. The accuracy value represents the percentage of correctly predicted labels out of all the samples.

LeNet 5 Model



```
# Define the LeNet 5 model architecture
model = keras.Sequential([
    layers.Conv2D(6,kernel_size=(5,5),activation='relu',input_shape=(224,224,3)),
    layers.MaxPooling2D(pool_size=(2,2)),
    layers.Conv2D(16, kernel_size=(5,5), activation='relu'),
    layers.MaxPooling2D(pool_size=(2,2)),
    layers.Flatten(),
    layers.Dense(120, activation='relu'),
    layers.Dense(84, activation='relu'),
    layers.Dense(num_classes, activation='softmax')
])
```

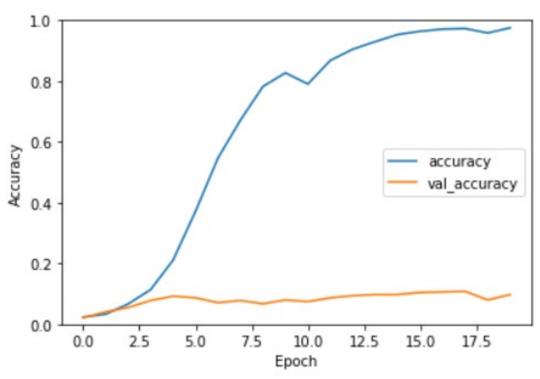
The model contains the following layers:

- *CONV layer with 6 filters of size 5x5 and ReLU activation.*
- *POOL layer with a pool size of 2x2.*
- *CONV layer with 16 filters of size 5x5 and ReLU activation.*
- *POOL layer with a pool size of 2x2.*
- *Dense layer with 120 units and ReLU activation.*
- *Dense layer with 84 units and ReLU activation.*
- *Dense layer with 47 units (output layer) and softmax activation.*

The training is performed for 20 epochs with a batch size of 128. Below are the results:

```
Epoch 1/20
36/36 [==============================] - 61s 2s/step - loss: 4.0446 - accuracy: 0.0235 -
val_loss: 3.8281 - val_accuracy: 0.0213
Epoch 2/20
36/36 [==============================] - 61s 2s/step - loss: 3.8046 - accuracy: 0.0330 -
val_loss: 3.7937 - val_accuracy: 0.0408
Epoch 3/20
36/36 [==============================] - 66s 2s/step - loss: 3.7191 - accuracy: 0.0667 -
val_loss: 3.7697 - val_accuracy: 0.0550
Epoch 4/20
36/36 [==============================] - 69s 2s/step - loss: 3.5514 - accuracy: 0.1137 -
val_loss: 3.8276 - val_accuracy: 0.0780
Epoch 5/20
36/36 [==============================] - 69s 2s/step - loss: 3.1494 - accuracy: 0.2108 -
val_loss: 3.8722 - val_accuracy: 0.0922
Epoch 6/20
36/36 [==============================] - 70s 2s/step - loss: 2.5277 - accuracy: 0.3717 -
val_loss: 4.2708 - val_accuracy: 0.0869
Epoch 7/20
36/36 [==============================] - 70s 2s/step - loss: 1.8446 - accuracy: 0.5470 -
val_loss: 5.3815 - val_accuracy: 0.0709
Epoch 8/20
36/36 [==============================] - 69s 2s/step - loss: 1.3209 - accuracy: 0.6718 -
val_loss: 6.1650 - val_accuracy: 0.0780
Epoch 9/20
36/36 [==============================] - 69s 2s/step - loss: 0.9263 - accuracy: 0.7821 -
val_loss: 7.1192 - val_accuracy: 0.0674
Epoch 10/20
36/36 [==============================] - 74s 2s/step - loss: 0.7416 - accuracy: 0.8271 -
val_loss: 8.5597 - val_accuracy: 0.0798
Epoch 11/20
36/36 [==============================] - 74s 2s/step - loss: 0.9566 - accuracy: 0.7903 -
val_loss: 8.1215 - val_accuracy: 0.0745
Epoch 12/20
36/36 [==============================] - 73s 2s/step - loss: 0.5742 - accuracy: 0.8681 -
val_loss: 9.8621 - val_accuracy: 0.0869
Epoch 13/20
36/36 [==============================] - 72s 2s/step - loss: 0.4384 - accuracy: 0.9049 -
val_loss: 9.6499 - val_accuracy: 0.0940
Epoch 14/20
36/36 [==============================] - 74s 2s/step - loss: 0.3155 - accuracy: 0.9300 -
val_loss: 10.1538 - val_accuracy: 0.0975
Epoch 15/20
36/36 [==============================] - 75s 2s/step - loss: 0.2114 - accuracy: 0.9532 -
val_loss: 10.5784 - val_accuracy: 0.0975
Epoch 16/20
36/36 [==============================] - 70s 2s/step - loss: 0.1713 - accuracy: 0.9639 -
val_loss: 11.3988 - val_accuracy: 0.1046
Epoch 17/20
36/36 [==============================] - 70s 2s/step - loss: 0.1490 - accuracy: 0.9710 -
val_loss: 11.6427 - val_accuracy: 0.1064
Epoch 18/20
36/36 [==============================] - 67s 2s/step - loss: 0.1480 - accuracy: 0.9730 -
val_loss: 12.1059 - val_accuracy: 0.1082
Epoch 19/20
36/36 [==============================] - 66s 2s/step - loss: 0.2020 - accuracy: 0.9583 -
val_loss: 11.6016 - val_accuracy: 0.0798
Epoch 20/20
36/36 [==============================] - 69s 2s/step - loss: 0.1253 - accuracy: 0.9752 -
val_loss: 12.0452 - val_accuracy: 0.0975
```

With each epoch, the training loss decreases and the training accuracy increases, indicating that the model is learning from the training data. However, the training accuracy reaches a very high value (97.52%) while the validation accuracy remains relatively low (9.75%), suggesting overfitting. This means that the model performs well on the training data but fails to generalize well to unseen data.

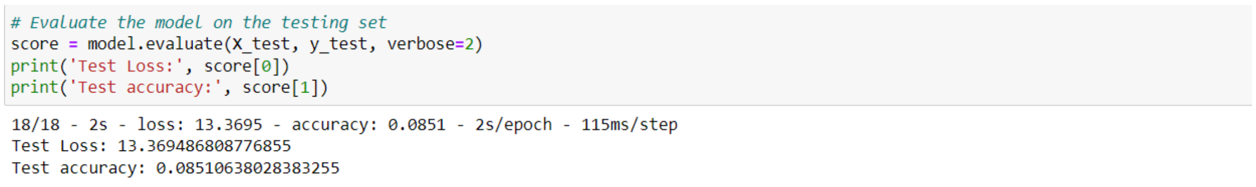


The validation loss increases as the number of epochs increases, indicating that the model is not performing well on the validation data. This further supports the observation of overfitting.

To address this issue, some possible approaches are:

- Apply regularization techniques, such as dropout or weight decay, to reduce overfitting.
- Adjust the model architecture, such as changing the number of filters, adding more convolutional layers, or increasing the complexity of the network.
- Experiment with different optimization algorithms or learning rates to improve the convergence of the model.

Below are the results when tested on new unseen data:



```
# Evaluate the model on the testing set
score = model.evaluate(X_test, y_test, verbose=2)
print('Test Loss:', score[0])
print('Test accuracy:', score[1])
```

```
18/18 - 2s - loss: 13.3695 - accuracy: 0.0851 - 2s/epoch - 115ms/step
Test Loss: 13.369486808776855
Test accuracy: 0.08510638028383255
```

A classification report that provides an overview of the performance of the model on the testing set is also generated:

***Precision**: Measures the proportion of correctly predicted instances among the predicted instances for each class.*

***Recall**: Measures the proportion of correctly predicted instances among the actual instances for each class.*

***F1-score**: The harmonic mean of precision and recall and provides a balanced measure of the model's performance.*

***Support**: The number of instances of each class in the testing set.*

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

           0       0.00      0.00      0.00        15
           1       0.00      0.00      0.00        16
           2       0.10      0.14      0.11        14
           3       0.13      0.13      0.13        15
           4       0.00      0.00      0.00        12
           5       0.11      0.23      0.15        13
           6       0.10      0.07      0.08        14
           7       0.00      0.00      0.00        10
           8       0.00      0.00      0.00        14
           9       0.33      0.11      0.16        19
          10       0.19      0.30      0.23        10
          11       0.00      0.00      0.00         9
          12       0.00      0.00      0.00        13
          13       0.11      0.06      0.08        17
          14       0.11      0.09      0.10        11
          15       0.00      0.00      0.00         7
          16       0.18      0.17      0.17        12
          17       0.03      0.11      0.05         9
          18       0.07      0.12      0.09         8
          19       0.16      0.25      0.20        16
          20       0.00      0.00      0.00        12
          21       0.00      0.00      0.00        10
          22       0.19      0.27      0.22        15
          23       0.00      0.00      0.00        12
          24       0.00      0.00      0.00        15
          25       0.11      0.06      0.08        16
          26       0.00      0.00      0.00        10
          27       0.33      0.06      0.10        17
          28       0.09      0.09      0.09        11
          29       0.29      0.12      0.17        16
          30       0.05      0.08      0.06        12
          31       0.00      0.00      0.00         9
          32       0.00      0.00      0.00        12
          33       0.00      0.00      0.00         3
          34       0.05      0.11      0.07         9
          35       0.14      0.12      0.13         8
          36       0.05      0.06      0.05        17
          37       0.06      0.22      0.10         9
          38       0.08      0.12      0.10         8
          39       0.16      0.23      0.19        13
          40       0.14      0.11      0.12         9
          41       0.00      0.00      0.00         9
          42       0.00      0.00      0.00         6
          43       0.00      0.00      0.00        12
          44       0.19      0.19      0.19        16
          45       0.07      0.09      0.08        11
          46       0.11      0.08      0.09        13

    accuracy                           0.09       564
   macro avg       0.08      0.08      0.07       564
weighted avg       0.09      0.09      0.08       564
```

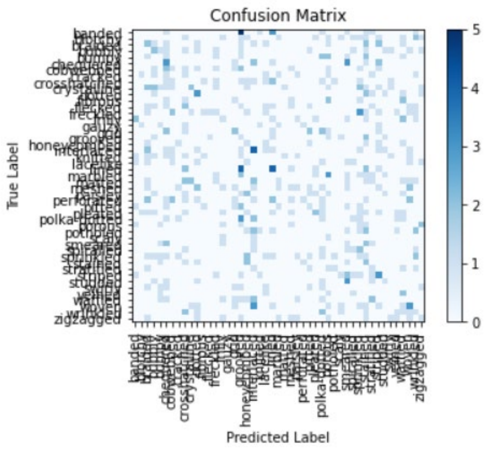
The precision scores for most classes are very low, ranging from 0.00 to 0.33. This indicates that the model has a high number of false positives, meaning it often incorrectly predicts an instance to belong to a certain class.

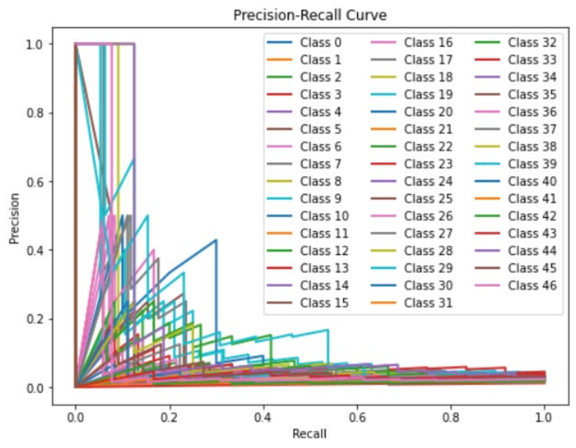
The recall scores vary across classes, with values ranging from 0.00 to 0.30. Low recall scores suggest that the model has a high number of false negatives, meaning it fails to correctly predict instances of certain classes.

The F1-scores for most classes are low, ranging from 0.00 to 0.23. This indicates that the model struggles to achieve both high precision and high recall simultaneously for most classes.

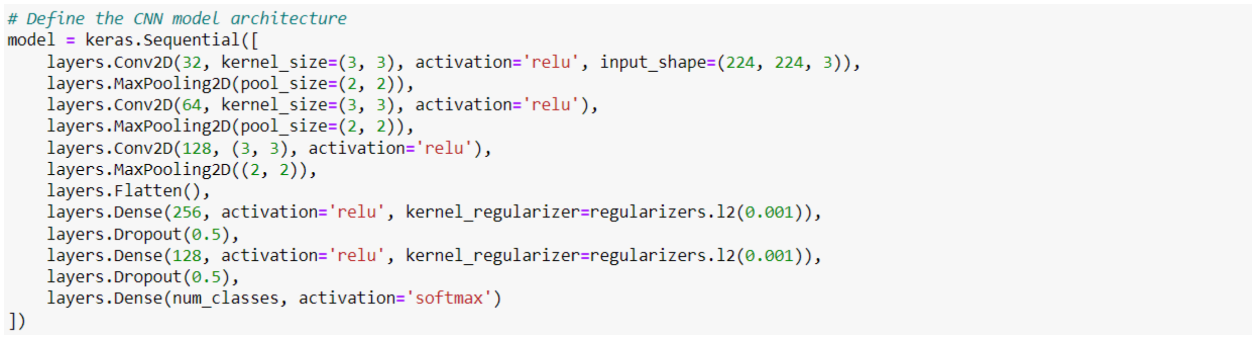
It can be observed that the number of instances varies across classes, ranging from 3 to 19.

Overall, the analysis of the classification report suggests that the model's performance is poor. The low precision, recall, and F1-score indicate that the model is struggling to effectively classify instances into the correct classes. This is evident from the low accuracy of 0.09 (9%) on the testing set.





CNN Model



```python
# Define the CNN model architecture
model = keras.Sequential([
    layers.Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=(224, 224, 3)),
    layers.MaxPooling2D(pool_size=(2, 2)),
    layers.Conv2D(64, kernel_size=(3, 3), activation='relu'),
    layers.MaxPooling2D(pool_size=(2, 2)),
    layers.Conv2D(128, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    layers.Flatten(),
    layers.Dense(256, activation='relu', kernel_regularizer=regularizers.l2(0.001)),
    layers.Dropout(0.5),
    layers.Dense(128, activation='relu', kernel_regularizer=regularizers.l2(0.001)),
    layers.Dropout(0.5),
    layers.Dense(num_classes, activation='softmax')
])
```

The model contains the following layers:

- *Conv2D layer with 32 filters of size 3x3 and ReLU activation.*
- *MaxPooling2D layer with a pool size of 2x2.*
- *Conv2D layer with 64 filters of size 3x3 and ReLU activation.*
- *MaxPooling2D layer with a pool size of 2x2.*
- *Conv2D layer with 128 filters of size 3x3 and ReLU activation.*
- *MaxPooling2D layer with a pool size of 2x2.*
- *Dense layer with 256 units and ReLU activation, with L2 regularization of 0.001.*
- *Dropout layer with a dropout rate of 0.5 to prevent overfitting.*
- *Dense layer with 128 units and ReLU activation, with L2 regularization of 0.001.*
- *Dropout layer with a dropout rate of 0.5.*
- *Dense layer with 47 units (output layer) and softmax activation.*

The training is performed for 20 epochs with a batch size of 128. Below are the results:

```
Epoch 1/20
36/36 [==============================] - 156s 4s/step - loss: 4.4337 - accuracy: 0.0224 -
val_loss: 4.1773 - val_accuracy: 0.0408
Epoch 2/20
36/36 [==============================] - 164s 5s/step - loss: 4.0893 - accuracy: 0.0268 -
val_loss: 4.0152 - val_accuracy: 0.0266
Epoch 3/20
36/36 [==============================] - 164s 5s/step - loss: 3.9661 - accuracy: 0.0299 -
val_loss: 3.9531 - val_accuracy: 0.0301
Epoch 4/20
36/36 [==============================] - 165s 5s/step - loss: 3.9176 - accuracy: 0.0370 -
val_loss: 3.9213 - val_accuracy: 0.0319
Epoch 5/20
36/36 [==============================] - 157s 4s/step - loss: 3.8951 - accuracy: 0.0397 -
val_loss: 3.9102 - val_accuracy: 0.0337
Epoch 6/20
36/36 [==============================] - 152s 4s/step - loss: 3.8761 - accuracy: 0.0401 -
val_loss: 3.9066 - val_accuracy: 0.0461
Epoch 7/20
36/36 [==============================] - 155s 4s/step - loss: 3.8623 - accuracy: 0.0468 -
val_loss: 3.8946 - val_accuracy: 0.0514
Epoch 8/20
36/36 [==============================] - 157s 4s/step - loss: 3.8446 - accuracy: 0.0543 -
val_loss: 3.8947 - val_accuracy: 0.0390
Epoch 9/20
36/36 [==============================] - 159s 4s/step - loss: 3.8133 - accuracy: 0.0516 -
val_loss: 3.8855 - val_accuracy: 0.0496
Epoch 10/20
36/36 [==============================] - 163s 5s/step - loss: 3.7857 - accuracy: 0.0678 -
val_loss: 3.8861 - val_accuracy: 0.0550
Epoch 11/20
36/36 [==============================] - 171s 5s/step - loss: 3.7723 - accuracy: 0.0731 -
val_loss: 3.8997 - val_accuracy: 0.0603
Epoch 12/20
36/36 [==============================] - 170s 5s/step - loss: 3.7425 - accuracy: 0.0889 -
val_loss: 3.9206 - val_accuracy: 0.0638
Epoch 13/20
36/36 [==============================] - 167s 5s/step - loss: 3.7133 - accuracy: 0.1053 -
val_loss: 3.9079 - val_accuracy: 0.0479
Epoch 14/20
36/36 [==============================] - 168s 5s/step - loss: 3.6562 - accuracy: 0.1197 -
val_loss: 3.9415 - val_accuracy: 0.0745
Epoch 15/20
36/36 [==============================] - 163s 5s/step - loss: 3.5938 - accuracy: 0.1463 -
val_loss: 3.9703 - val_accuracy: 0.0833
Epoch 16/20
36/36 [==============================] - 159s 4s/step - loss: 3.5603 - accuracy: 0.1653 -
val_loss: 3.9865 - val_accuracy: 0.0851
Epoch 17/20
36/36 [==============================] - 157s 4s/step - loss: 3.4730 - accuracy: 0.1886 -
val_loss: 4.0279 - val_accuracy: 0.0851
Epoch 18/20
36/36 [==============================] - 158s 4s/step - loss: 3.3745 - accuracy: 0.2152 -
val_loss: 4.0710 - val_accuracy: 0.0869
Epoch 19/20
36/36 [==============================] - 159s 4s/step - loss: 3.3252 - accuracy: 0.2431 -
val_loss: 4.0789 - val_accuracy: 0.0869
Epoch 20/20
36/36 [==============================] - 150s 4s/step - loss: 3.2689 - accuracy: 0.2538 -
val_loss: 4.0851 - val_accuracy: 0.1082
```

The training accuracy starts from a very low value of 0.0224 and gradually increases with each epoch, reaching 0.2538 in the final epoch. The validation accuracy also starts from a low value of 0.0408 and shows some improvement, reaching 0.1082 in the final epoch.

The training loss decreases with each epoch, indicating that the model is learning from the training data, while the validation loss is relatively stable.

These results suggest that the training process is still ongoing and has not reached convergence within the 20 epochs and that the model needs more epochs to improve.

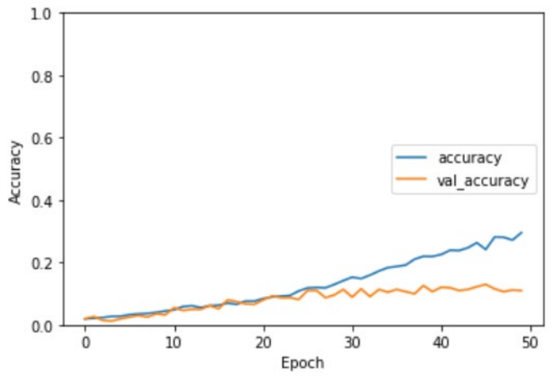
Below are the results with 50 epochs:

```
Epoch 1/50
36/36 [==============================] - 149s 4s/step - loss: 4.4766 - accuracy: 0.0188 -
val_loss: 4.1788 - val_accuracy: 0.0195
Epoch 2/50
36/36 [==============================] - 156s 4s/step - loss: 4.0969 - accuracy: 0.0215 -
val_loss: 4.0316 - val_accuracy: 0.0266
Epoch 3/50
36/36 [==============================] - 156s 4s/step - loss: 3.9966 - accuracy: 0.0233 -
val_loss: 3.9690 - val_accuracy: 0.0142
Epoch 4/50
36/36 [==============================] - 155s 4s/step - loss: 3.9480 - accuracy: 0.0270 -
val_loss: 3.9328 - val_accuracy: 0.0124
Epoch 5/50
36/36 [==============================] - 156s 4s/step - loss: 3.9111 - accuracy: 0.0277 -
val_loss: 3.8988 - val_accuracy: 0.0195
Epoch 6/50
36/36 [==============================] - 157s 4s/step - loss: 3.8857 - accuracy: 0.0326 -
val_loss: 3.8806 - val_accuracy: 0.0248
Epoch 7/50
36/36 [==============================] - 156s 4s/step - loss: 3.8592 - accuracy: 0.0350 -
val_loss: 3.8789 - val_accuracy: 0.0301
Epoch 8/50
36/36 [==============================] - 155s 4s/step - loss: 3.8459 - accuracy: 0.0366 -
val_loss: 3.8480 - val_accuracy: 0.0248
Epoch 9/50
36/36 [==============================] - 159s 4s/step - loss: 3.8149 - accuracy: 0.0399 -
val_loss: 3.8382 - val_accuracy: 0.0355
Epoch 10/50
36/36 [==============================] - 161s 4s/step - loss: 3.7899 - accuracy: 0.0441 -
val_loss: 3.8075 - val_accuracy: 0.0319
Epoch 11/50
36/36 [==============================] - 153s 4s/step - loss: 3.7721 - accuracy: 0.0481 -
val_loss: 3.7778 - val_accuracy: 0.0550
Epoch 12/50
36/36 [==============================] - 175s 5s/step - loss: 3.7468 - accuracy: 0.0585 -
val_loss: 3.7511 - val_accuracy: 0.0461
Epoch 13/50
36/36 [==============================] - 157s 4s/step - loss: 3.7195 - accuracy: 0.0612 -
val_loss: 3.7508 - val_accuracy: 0.0496
Epoch 14/50
36/36 [==============================] - 166s 5s/step - loss: 3.6933 - accuracy: 0.0550 -
val_loss: 3.7239 - val_accuracy: 0.0496
Epoch 15/50
36/36 [==============================] - 173s 5s/step - loss: 3.6571 - accuracy: 0.0614 -
val_loss: 3.7082 - val_accuracy: 0.0621
Epoch 16/50
36/36 [==============================] - 165s 5s/step - loss: 3.6413 - accuracy: 0.0627 -
val_loss: 3.6513 - val_accuracy: 0.0514
Epoch 17/50
36/36 [==============================] - 166s 5s/step - loss: 3.6190 - accuracy: 0.0698 -
val_loss: 3.6453 - val_accuracy: 0.0798
Epoch 18/50
36/36 [==============================] - 167s 5s/step - loss: 3.6045 - accuracy: 0.0656 -
val_loss: 3.6323 - val_accuracy: 0.0745
Epoch 19/50
36/36 [==============================] - 169s 5s/step - loss: 3.5804 - accuracy: 0.0758 -
val_loss: 3.7041 - val_accuracy: 0.0674
Epoch 20/50
36/36 [==============================] - 155s 4s/step - loss: 3.5598 - accuracy: 0.0758 -
val_loss: 3.6381 - val_accuracy: 0.0656
```

```
Epoch 21/50
36/36 [==============================] - 157s 4s/step - loss: 3.5369 - accuracy: 0.0840 -
val_loss: 3.6220 - val_accuracy: 0.0798
Epoch 22/50
36/36 [==============================] - 157s 4s/step - loss: 3.4940 - accuracy: 0.0891 -
val_loss: 3.6048 - val_accuracy: 0.0922
Epoch 23/50
36/36 [==============================] - 147s 4s/step - loss: 3.4633 - accuracy: 0.0915 -
val_loss: 3.6345 - val_accuracy: 0.0869
Epoch 24/50
36/36 [==============================] - 152s 4s/step - loss: 3.4180 - accuracy: 0.0938 -
val_loss: 3.6802 - val_accuracy: 0.0869
Epoch 25/50
36/36 [==============================] - 156s 4s/step - loss: 3.3909 - accuracy: 0.1086 -
val_loss: 3.6230 - val_accuracy: 0.0816
Epoch 26/50
36/36 [==============================] - 156s 4s/step - loss: 3.3713 - accuracy: 0.1181 -
val_loss: 3.6291 - val_accuracy: 0.1099
Epoch 27/50
36/36 [==============================] - 156s 4s/step - loss: 3.3405 - accuracy: 0.1197 -
val_loss: 3.6549 - val_accuracy: 0.1099
Epoch 28/50
36/36 [==============================] - 157s 4s/step - loss: 3.3247 - accuracy: 0.1186 -
val_loss: 3.6438 - val_accuracy: 0.0869
Epoch 29/50
36/36 [==============================] - 156s 4s/step - loss: 3.2885 - accuracy: 0.1294 -
val_loss: 3.6316 - val_accuracy: 0.0957
Epoch 30/50
36/36 [==============================] - 156s 4s/step - loss: 3.2517 - accuracy: 0.1416 -
val_loss: 3.6408 - val_accuracy: 0.1135
Epoch 31/50
36/36 [==============================] - 156s 4s/step - loss: 3.1863 - accuracy: 0.1527 -
val_loss: 3.7325 - val_accuracy: 0.0887
Epoch 32/50
36/36 [==============================] - 156s 4s/step - loss: 3.1833 - accuracy: 0.1480 -
val_loss: 3.6641 - val_accuracy: 0.1152
Epoch 33/50
36/36 [==============================] - 155s 4s/step - loss: 3.1235 - accuracy: 0.1594 -
val_loss: 3.6873 - val_accuracy: 0.0904
Epoch 34/50
36/36 [==============================] - 151s 4s/step - loss: 3.1272 - accuracy: 0.1724 -
val_loss: 3.7327 - val_accuracy: 0.1135
Epoch 35/50
36/36 [==============================] - 148s 4s/step - loss: 3.0488 - accuracy: 0.1833 -
val_loss: 3.7526 - val_accuracy: 0.1046
Epoch 36/50
36/36 [==============================] - 155s 4s/step - loss: 2.9972 - accuracy: 0.1871 -
val_loss: 3.8094 - val_accuracy: 0.1135
Epoch 37/50
36/36 [==============================] - 160s 4s/step - loss: 3.0107 - accuracy: 0.1915 -
val_loss: 3.6902 - val_accuracy: 0.1064
Epoch 38/50
36/36 [==============================] - 161s 4s/step - loss: 2.9664 - accuracy: 0.2103 -
val_loss: 3.7438 - val_accuracy: 0.0993
Epoch 39/50
36/36 [==============================] - 156s 4s/step - loss: 2.9222 - accuracy: 0.2196 -
val_loss: 3.7288 - val_accuracy: 0.1259
Epoch 40/50
36/36 [==============================] - 158s 4s/step - loss: 2.9081 - accuracy: 0.2192 -
val_loss: 3.6928 - val_accuracy: 0.1064
```

```
Epoch 41/50
36/36 [==============================] - 156s 4s/step - loss: 2.8655 - accuracy: 0.2254 -
val_loss: 3.8066 - val_accuracy: 0.1206
Epoch 42/50
36/36 [==============================] - 155s 4s/step - loss: 2.8092 - accuracy: 0.2391 -
val_loss: 3.8954 - val_accuracy: 0.1188
Epoch 43/50
36/36 [==============================] - 156s 4s/step - loss: 2.8246 - accuracy: 0.2380 -
val_loss: 3.8362 - val_accuracy: 0.1099
Epoch 44/50
36/36 [==============================] - 156s 4s/step - loss: 2.7875 - accuracy: 0.2471 -
val_loss: 3.8609 - val_accuracy: 0.1135
Epoch 45/50
36/36 [==============================] - 149s 4s/step - loss: 2.7892 - accuracy: 0.2631 -
val_loss: 3.8135 - val_accuracy: 0.1223
Epoch 46/50
36/36 [==============================] - 145s 4s/step - loss: 2.7752 - accuracy: 0.2416 -
val_loss: 3.9031 - val_accuracy: 0.1294
Epoch 47/50
36/36 [==============================] - 148s 4s/step - loss: 2.6770 - accuracy: 0.2817 -
val_loss: 3.9568 - val_accuracy: 0.1152
Epoch 48/50
36/36 [==============================] - 156s 4s/step - loss: 2.6802 - accuracy: 0.2808 -
val_loss: 3.9217 - val_accuracy: 0.1064
Epoch 49/50
36/36 [==============================] - 160s 4s/step - loss: 2.7268 - accuracy: 0.2715 -
val_loss: 4.0463 - val_accuracy: 0.1117
Epoch 50/50
36/36 [==============================] - 156s 4s/step - loss: 2.6147 - accuracy: 0.2954 -
val_loss: 3.9666 - val_accuracy: 0.1099
```

It seems that the model's performance did not improve significantly after adding the extra 30 epochs. The accuracy for both training and validation sets remained relatively stable throughout the epochs, with no clear upward or downward trend. The loss values also showed a similar pattern, where there was no significant decrease after the initial epochs.



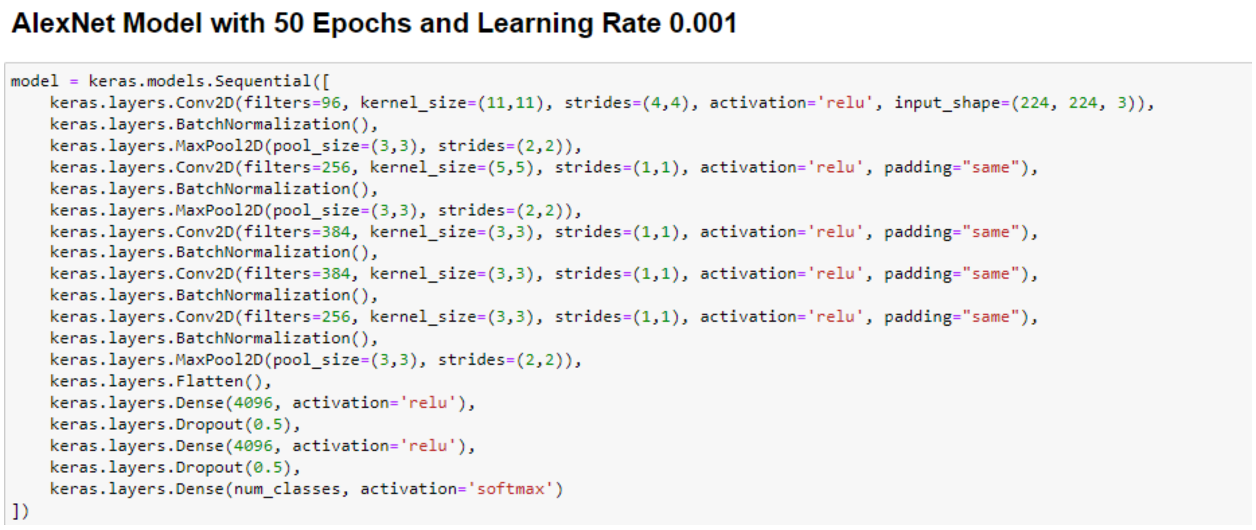
Based on these observations, it seems that the model reached a point of diminishing returns in terms of improvement. It did not show substantial progress in accuracy or loss after the initial epochs, suggesting that further training may not lead to significant performance gains. It appears that the model's performance plateaued around the early epochs (around 30 epochs) and did not exhibit substantial improvement thereafter.

However, this CNN model’s performance is still better than LeNet 5’s with a much lower test loss and slightly higher test accuracy.

```
# Evaluate the model on the testing set
score = model.evaluate(X_test, y_test, verbose=2)
print('Test Loss:', score[0])
print('Test accuracy:', score[1])
```

```
18/18 - 5s - loss: 3.9436 - accuracy: 0.1418 - 5s/epoch - 264ms/step
Test Loss: 3.9436161518096924
Test accuracy: 0.1418439745903015
```

AlexNet Model



AlexNet Model with 50 Epochs and Learning Rate 0.001

```
model = keras.models.Sequential([
    keras.layers.Conv2D(filters=96, kernel_size=(11,11), strides=(4,4), activation='relu', input_shape=(224, 224, 3)),
    keras.layers.BatchNormalization(),
    keras.layers.MaxPool2D(pool_size=(3,3), strides=(2,2)),
    keras.layers.Conv2D(filters=256, kernel_size=(5,5), strides=(1,1), activation='relu', padding="same"),
    keras.layers.BatchNormalization(),
    keras.layers.MaxPool2D(pool_size=(3,3), strides=(2,2)),
    keras.layers.Conv2D(filters=384, kernel_size=(3,3), strides=(1,1), activation='relu', padding="same"),
    keras.layers.BatchNormalization(),
    keras.layers.Conv2D(filters=384, kernel_size=(3,3), strides=(1,1), activation='relu', padding="same"),
    keras.layers.BatchNormalization(),
    keras.layers.Conv2D(filters=256, kernel_size=(3,3), strides=(1,1), activation='relu', padding="same"),
    keras.layers.BatchNormalization(),
    keras.layers.MaxPool2D(pool_size=(3,3), strides=(2,2)),
    keras.layers.Flatten(),
    keras.layers.Dense(4096, activation='relu'),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(4096, activation='relu'),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(num_classes, activation='softmax')
])
```

The model contains the following layers:

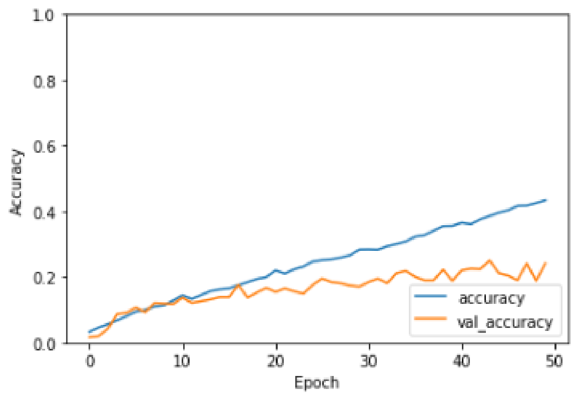
- *Conv2D layer with 96 filters of size 11x11, a stride of 4x4, and ReLU activation.*
- *BatchNormalization layer for normalization after the Conv2D layer.*
- *MaxPooling2D layer with a pool size of 3x3 and a stride of 2x2.*
- *Conv2D layer with 256 filters of size 5x5, a stride of 1x1, and ReLU activation.*
- *BatchNormalization layer for normalization after the Conv2D layer.*
- *MaxPooling2D layer with a pool size of 3x3 and a stride of 2x2.*
- *Conv2D layer with 384 filters of size 3x3, a stride of 1x1, and ReLU activation.*
- *BatchNormalization layer for normalization after the Conv2D layer.*
- *Conv2D layer with 384 filters of size 3x3, a stride of 1x1, and ReLU activation.*
- *BatchNormalization layer for normalization after the Conv2D layer.*
- *Conv2D layer with 256 filters of size 3x3, a stride of 1x1, and ReLU activation.*
- *BatchNormalization layer for normalization after the Conv2D layer.*
- *MaxPooling2D layer with a pool size of 3x3 and a stride of 2x2.*
- *Dense layer with 4096 units and ReLU activation.*
- *Dropout layer with a dropout rate of 0.5 to prevent overfitting.*
- *Dense layer with 4096 units and ReLU activation.*
- *Dropout layer with a dropout rate of 0.5.*
- *Dense layer with 47 units (output layer) and softmax activation.*

```
Epoch 1/50
141/141 [==============================] - 214s 2s/step - loss: 5.2604 - accuracy: 0.0328 -
val_loss: 3.9002 - val_accuracy: 0.0160
Epoch 2/50
141/141 [==============================] - 219s 2s/step - loss: 4.5072 - accuracy: 0.0459 -
val_loss: 3.8646 - val_accuracy: 0.0195
Epoch 3/50
141/141 [==============================] - 218s 2s/step - loss: 4.1889 - accuracy: 0.0559 -
val_loss: 3.7730 - val_accuracy: 0.0426
Epoch 4/50
141/141 [==============================] - 218s 2s/step - loss: 4.0252 - accuracy: 0.0674 -
val_loss: 3.6270 - val_accuracy: 0.0869
Epoch 5/50
141/141 [==============================] - 220s 2s/step - loss: 3.8617 - accuracy: 0.0802 -
val_loss: 3.6061 - val_accuracy: 0.0904
Epoch 6/50
141/141 [==============================] - 219s 2s/step - loss: 3.7654 - accuracy: 0.0940 -
val_loss: 3.5801 - val_accuracy: 0.1064
Epoch 7/50
141/141 [==============================] - 218s 2s/step - loss: 3.6971 - accuracy: 0.0991 -
val_loss: 3.6370 - val_accuracy: 0.0922
Epoch 8/50
141/141 [==============================] - 211s 1s/step - loss: 3.6016 - accuracy: 0.1095 -
val_loss: 3.5480 - val_accuracy: 0.1206
Epoch 9/50
141/141 [==============================] - 204s 1s/step - loss: 3.5590 - accuracy: 0.1130 -
val_loss: 3.4531 - val_accuracy: 0.1188
Epoch 10/50
141/141 [==============================] - 222s 2s/step - loss: 3.4700 - accuracy: 0.1281 -
val_loss: 3.5021 - val_accuracy: 0.1170
Epoch 11/50
141/141 [==============================] - 223s 2s/step - loss: 3.4365 - accuracy: 0.1438 -
val_loss: 3.4148 - val_accuracy: 0.1365
Epoch 12/50
141/141 [==============================] - 219s 2s/step - loss: 3.4060 - accuracy: 0.1328 -
val_loss: 3.4566 - val_accuracy: 0.1206
Epoch 13/50
141/141 [==============================] - 219s 2s/step - loss: 3.3826 - accuracy: 0.1445 -
val_loss: 3.4342 - val_accuracy: 0.1259
Epoch 14/50
141/141 [==============================] - 217s 2s/step - loss: 3.3205 - accuracy: 0.1567 -
val_loss: 3.4636 - val_accuracy: 0.1312
Epoch 15/50
141/141 [==============================] - 220s 2s/step - loss: 3.2457 - accuracy: 0.1620 -
val_loss: 3.4046 - val_accuracy: 0.1383
Epoch 16/50
141/141 [==============================] - 222s 2s/step - loss: 3.2403 - accuracy: 0.1651 -
val_loss: 3.4926 - val_accuracy: 0.1383
Epoch 17/50
141/141 [==============================] - 207s 1s/step - loss: 3.2008 - accuracy: 0.1749 -
val_loss: 3.2705 - val_accuracy: 0.1755
Epoch 18/50
141/141 [==============================] - 214s 2s/step - loss: 3.1591 - accuracy: 0.1842 -
val_loss: 3.4299 - val_accuracy: 0.1365
Epoch 19/50
141/141 [==============================] - 218s 2s/step - loss: 3.1296 - accuracy: 0.1937 -
val_loss: 3.3244 - val_accuracy: 0.1525
Epoch 20/50
141/141 [==============================] - 218s 2s/step - loss: 3.0948 - accuracy: 0.1992 -
val_loss: 3.3670 - val_accuracy: 0.1667
```

```
Epoch 21/50
141/141 [==============================] - 220s 2s/step - loss: 3.0409 - accuracy: 0.2194 -
val_loss: 3.3938 - val_accuracy: 0.1543
Epoch 22/50
141/141 [==============================] - 218s 2s/step - loss: 3.0080 - accuracy: 0.2099 -
val_loss: 3.2298 - val_accuracy: 0.1649
Epoch 23/50
141/141 [==============================] - 218s 2s/step - loss: 2.9915 - accuracy: 0.2230 -
val_loss: 3.3630 - val_accuracy: 0.1560
Epoch 24/50
141/141 [==============================] - 217s 2s/step - loss: 2.9364 - accuracy: 0.2314 -
val_loss: 3.5324 - val_accuracy: 0.1489
Epoch 25/50
141/141 [==============================] - 204s 1s/step - loss: 2.9078 - accuracy: 0.2469 -
val_loss: 3.2160 - val_accuracy: 0.1755
Epoch 26/50
141/141 [==============================] - 209s 1s/step - loss: 2.8707 - accuracy: 0.2509 -
val_loss: 3.2589 - val_accuracy: 0.1950
Epoch 27/50
141/141 [==============================] - 217s 2s/step - loss: 2.8422 - accuracy: 0.2535 -
val_loss: 3.1965 - val_accuracy: 0.1844
Epoch 28/50
141/141 [==============================] - 219s 2s/step - loss: 2.8176 - accuracy: 0.2586 -
val_loss: 3.2422 - val_accuracy: 0.1809
Epoch 29/50
141/141 [==============================] - 217s 2s/step - loss: 2.7699 - accuracy: 0.2657 -
val_loss: 3.2973 - val_accuracy: 0.1738
Epoch 30/50
141/141 [==============================] - 218s 2s/step - loss: 2.7338 - accuracy: 0.2832 -
val_loss: 3.2957 - val_accuracy: 0.1702
Epoch 31/50
141/141 [==============================] - 220s 2s/step - loss: 2.6969 - accuracy: 0.2837 -
val_loss: 3.3316 - val_accuracy: 0.1844
Epoch 32/50
141/141 [==============================] - 220s 2s/step - loss: 2.6938 - accuracy: 0.2828 -
val_loss: 3.1074 - val_accuracy: 0.1950
Epoch 33/50
141/141 [==============================] - 210s 1s/step - loss: 2.6624 - accuracy: 0.2932 -
val_loss: 3.2313 - val_accuracy: 0.1809
Epoch 34/50
141/141 [==============================] - 204s 1s/step - loss: 2.6107 - accuracy: 0.2999 -
val_loss: 3.0846 - val_accuracy: 0.2110
Epoch 35/50
141/141 [==============================] - 218s 2s/step - loss: 2.5728 - accuracy: 0.3074 -
val_loss: 3.1038 - val_accuracy: 0.2181
Epoch 36/50
141/141 [==============================] - 219s 2s/step - loss: 2.5344 - accuracy: 0.3227 -
val_loss: 3.0639 - val_accuracy: 0.2004
Epoch 37/50
141/141 [==============================] - 219s 2s/step - loss: 2.5124 - accuracy: 0.3271 -
val_loss: 3.2038 - val_accuracy: 0.1897
Epoch 38/50
141/141 [==============================] - 219s 2s/step - loss: 2.4633 - accuracy: 0.3409 -
val_loss: 3.1867 - val_accuracy: 0.1897
Epoch 39/50
141/141 [==============================] - 218s 2s/step - loss: 2.4330 - accuracy: 0.3546 -
val_loss: 3.0909 - val_accuracy: 0.2216
Epoch 40/50
141/141 [==============================] - 217s 2s/step - loss: 2.3823 - accuracy: 0.3553 -
val_loss: 3.2403 - val_accuracy: 0.1879
```

```
Epoch 41/50
141/141 [==============================] - 217s 2s/step - loss: 2.3737 - accuracy: 0.3650 -
val_loss: 3.0918 - val_accuracy: 0.2199
Epoch 42/50
141/141 [==============================] - 203s 1s/step - loss: 2.3417 - accuracy: 0.3606 -
val_loss: 3.0464 - val_accuracy: 0.2252
Epoch 43/50
141/141 [==============================] - 212s 2s/step - loss: 2.2965 - accuracy: 0.3752 -
val_loss: 3.0277 - val_accuracy: 0.2234
Epoch 44/50
141/141 [==============================] - 218s 2s/step - loss: 2.2651 - accuracy: 0.3856 -
val_loss: 2.9942 - val_accuracy: 0.2500
Epoch 45/50
141/141 [==============================] - 217s 2s/step - loss: 2.2616 - accuracy: 0.3956 -
val_loss: 3.1048 - val_accuracy: 0.2110
Epoch 46/50
141/141 [==============================] - 218s 2s/step - loss: 2.1995 - accuracy: 0.4029 -
val_loss: 3.0981 - val_accuracy: 0.2039
Epoch 47/50
141/141 [==============================] - 217s 2s/step - loss: 2.1449 - accuracy: 0.4167 -
val_loss: 3.0866 - val_accuracy: 0.1897
Epoch 48/50
141/141 [==============================] - 217s 2s/step - loss: 2.1445 - accuracy: 0.4176 -
val_loss: 2.9823 - val_accuracy: 0.2411
Epoch 49/50
141/141 [==============================] - 218s 2s/step - loss: 2.0908 - accuracy: 0.4253 -
val_loss: 3.5063 - val_accuracy: 0.1879
Epoch 50/50
141/141 [==============================] - 203s 1s/step - loss: 2.0701 - accuracy: 0.4335 -
val_loss: 3.0149 - val_accuracy: 0.2411
```

The training was performed for 50 epochs, with each epoch consisting of 141 steps. The training loss decreased gradually from an initial value of 5.2604 to 2.0701, while the training accuracy increased from 0.0328 to 0.4335. On the other hand, the validation loss decreased from 3.9002 to 3.0149, and the validation accuracy increased from 0.0160 to 0.2411.



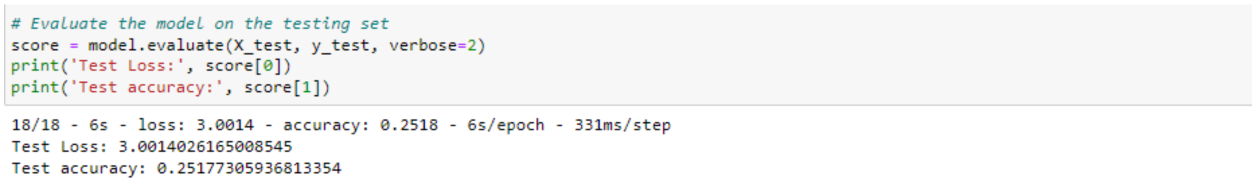
In this case, there is evidence of overfitting because the training accuracy (0.4335) is significantly higher than the validation accuracy (0.2411), and the training loss (2.0701) is lower than the validation loss (3.0149).

The model starts with a low accuracy of 0.0328, indicating that it makes mostly random predictions. As the training progresses, the accuracy improves, reaching 0.4335. However, the achieved accuracy is still relatively low, suggesting that the model may not have learned complex patterns in the data. The validation accuracy also shows a similar trend but lags behind the training accuracy, further indicating the model's difficulty in generalizing.

The loss values for both training and validation decrease over the epochs, which is a positive sign. However, it seems that the loss converges relatively early (around epoch 20) and doesn't decrease significantly afterwards. This suggests that the model's learning might have reached a plateau, and further training may not lead to significant improvements.

Based on these results, the following are recommendations to improve the model's performance:

- Consider using a more sophisticated neural network architecture, such as increasing the number of layers or utilizing pre-trained models like VGG16, ResNet, or Inception.
- Adjust the learning rate and try different optimization algorithms to optimize the training process further.



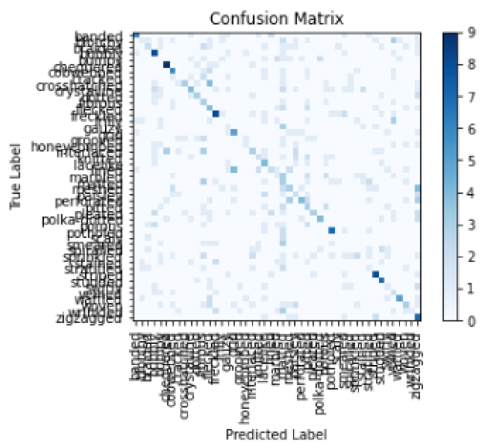
```
# Evaluate the model on the testing set
score = model.evaluate(X_test, y_test, verbose=2)
print('Test Loss:', score[0])
print('Test accuracy:', score[1])
```

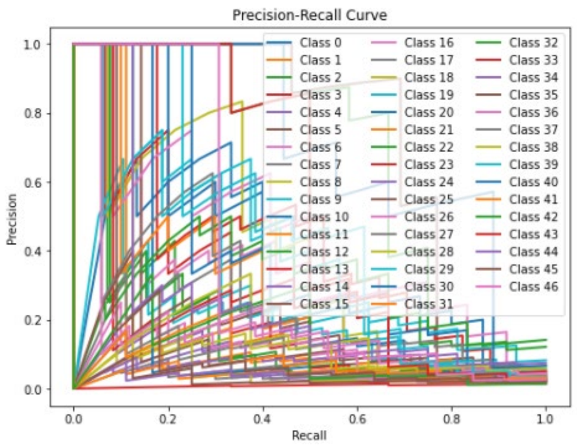
```
18/18 - 6s - loss: 3.0014 - accuracy: 0.2518 - 6s/epoch - 331ms/step
Test Loss: 3.0014026165008545
Test accuracy: 0.25177305936813354
```

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

           0       0.60      0.40      0.48        15
           1       0.20      0.06      0.10        16
           2       0.29      0.14      0.19        14
           3       0.35      0.53      0.42        15
           4       0.00      0.00      0.00        12
           5       0.64      0.69      0.67        13
           6       0.35      0.43      0.39        14
           7       0.00      0.00      0.00        10
           8       0.60      0.21      0.32        14
           9       0.36      0.21      0.27        19
          10       0.43      0.30      0.35        10
          11       0.12      0.33      0.18         9
          12       0.17      0.31      0.22        13
          13       0.47      0.47      0.47        17
          14       0.00      0.00      0.00        11
          15       0.10      0.14      0.12         7
          16       0.42      0.42      0.42        12
          17       0.00      0.00      0.00         9
          18       0.25      0.12      0.17         8
          19       0.25      0.19      0.21        16
          20       0.29      0.17      0.21        12
          21       0.19      0.40      0.26        10
          22       0.29      0.13      0.18        15
          23       0.11      0.08      0.10        12
          24       0.06      0.20      0.10        15
          25       0.23      0.19      0.21        16
          26       0.12      0.20      0.15        10
          27       0.50      0.24      0.32        17
          28       0.13      0.27      0.18        11
          29       0.50      0.19      0.27        16
          30       0.57      0.33      0.42        12
          31       0.00      0.00      0.00         9
          32       0.54      0.58      0.56        12
          33       0.00      0.00      0.00         3
          34       0.00      0.00      0.00         9
          35       0.00      0.00      0.00         8
          36       0.33      0.12      0.17        17
          37       0.00      0.00      0.00         9
          38       0.11      0.12      0.12         8
          39       0.53      0.62      0.57        13
          40       0.44      0.78      0.56         9
          41       0.14      0.11      0.12         9
          42       0.11      0.33      0.17         6
          43       0.71      0.42      0.53        12
          44       0.27      0.19      0.22        16
          45       1.00      0.09      0.17        11
          46       0.19      0.54      0.28        13

    accuracy                           0.25       564
   macro avg       0.28      0.24      0.23       564
weighted avg       0.30      0.25      0.25       564
```





The precision values range from 0.00 to 1.00, with an average weighted precision of 0.30. Some classes have relatively higher precision scores (e.g., class 45 with a precision of 1.00), while others have low scores (e.g., classes 4, 7, 14, 17, 33, 34, 35, 37, 41 with precision scores of 0.00).

The recall values range from 0.00 to 0.78, with an average weighted recall of 0.25. Some classes have relatively higher recall scores (e.g., class 40 with a recall of 0.78), while others have low scores (e.g., classes 1, 7, 14, 17, 33, 34, 35, 37 with recall scores of 0.00).

The F1-scores range from 0.00 to 0.67, with an average weighted F1-score of 0.25. Similar to precision and recall, some classes have higher F1-scores (e.g., class 40 with an F1-score of 0.56), while others have low scores (e.g., classes 4, 7, 14, 17, 33, 34, 35, 37, 41 with F1-scores of 0.00).

The support values range from 3 to 19. Some classes have a relatively low number of instances (e.g., classes 3, 7, 10, 11, 14, 15, 17, 18, 23, 27, 28, 31, 33, 34, 35, 37, 38, 41, 42, 45 with support less than 12), while others have a higher number of instances (e.g., classes 0, 1, 2, 5, 9, 13, 19, 26, 36, 39, 40, 43, 44, 46 with support greater than 12).

Overall, the model's performance seems to be quite low, with low precision, recall, and F1-scores for many classes. It indicates that the model is struggling to accurately classify instances across multiple classes. The class imbalance in your dataset might be contributing to poor performance, particularly for classes with low support.

The current model has slightly higher precision scores compared to the LeNet model. Although both models have low precision values overall, the current model shows a slightly better ability to correctly identify instances of certain classes.

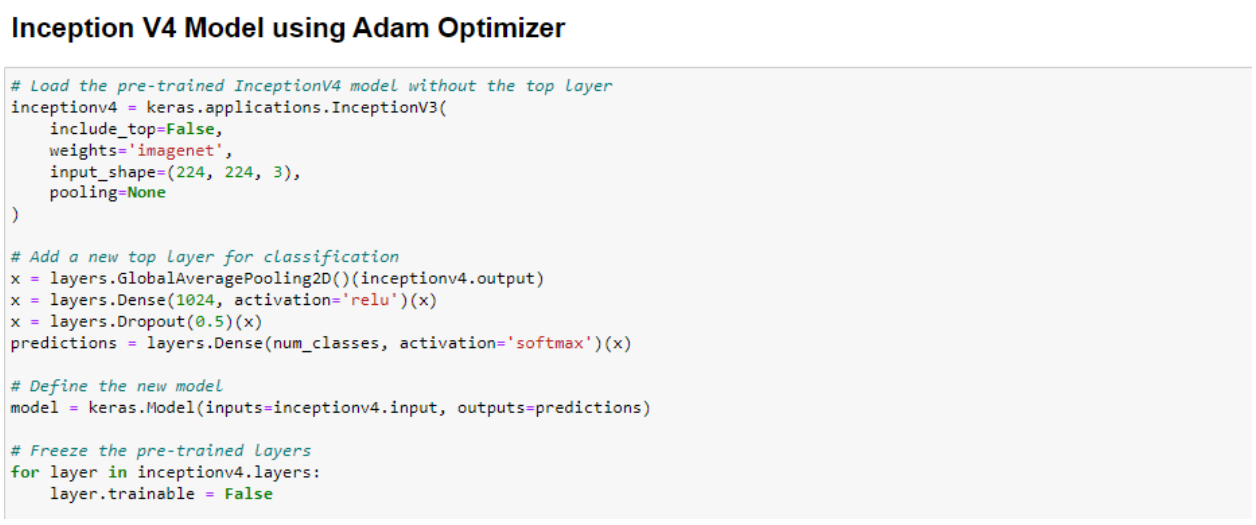
The recall scores of the current model are generally higher than those of the LeNet model. This indicates that the current model has a better ability to capture instances of each class compared to the previous model.

The F1-scores of the current model are also higher across most classes compared to the previous model. This suggests that the current model achieves a better balance between precision and recall than the previous model.

The accuracy of the current model is significantly higher than the LeNet model (0.25 vs. 0.09). This implies that the current model has an improved overall ability to correctly classify instances compared to the previous model.

Considering these metrics, it appears that the AlexNet model outperforms the LeNet model. It demonstrates higher precision, recall, F1-scores, and overall accuracy. Although both models still have relatively low performance, the current model shows a slight improvement in correctly identifying instances for most classes.

Inception V4 Model



Inception V4 Model using Adam Optimizer

```
# Load the pre-trained InceptionV4 model without the top layer
inceptionv4 = keras.applications.InceptionV3(
    include_top=False,
    weights='imagenet',
    input_shape=(224, 224, 3),
    pooling=None
)

# Add a new top layer for classification
x = layers.GlobalAveragePooling2D()(inceptionv4.output)
x = layers.Dense(1024, activation='relu')(x)
x = layers.Dropout(0.5)(x)
predictions = layers.Dense(num_classes, activation='softmax')(x)

# Define the new model
model = keras.Model(inputs=inceptionv4.input, outputs=predictions)

# Freeze the pre-trained layers
for layer in inceptionv4.layers:
    layer.trainable = False
```

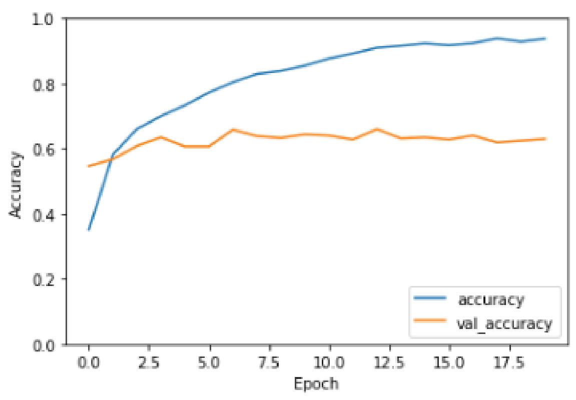
A breakdown of the layers:

1. *The pre-trained InceptionV3 model layers.*
2. *Two additional layers were added for classification:*
   - *GlobalAveragePooling2D layer*
   - *Dense layer with 1024 units and ReLU activation*
3. *One Dropout layer with a dropout rate of 0.5.*
4. *The final Dense layer with 47 units (output layer) and softmax activation.*

The layers from the pre-trained InceptionV3 model are frozen and not trainable, so they do not contribute to the trainable parameters of the model. Only the two additional layers and the final output layer are trainable.

```
Epoch 1/20
141/141 [==============================] - 195s 1s/step - loss: 2.6797 - accuracy: 0.3511 -
val_loss: 1.7677 - val_accuracy: 0.5461
Epoch 2/20
141/141 [==============================] - 191s 1s/step - loss: 1.5153 - accuracy: 0.5816 -
val_loss: 1.5624 - val_accuracy: 0.5674
Epoch 3/20
141/141 [==============================] - 194s 1s/step - loss: 1.2100 - accuracy: 0.6598 -
val_loss: 1.4754 - val_accuracy: 0.6082
Epoch 4/20
141/141 [==============================] - 195s 1s/step - loss: 1.0368 - accuracy: 0.6995 -
val_loss: 1.3463 - val_accuracy: 0.6348
Epoch 5/20
141/141 [==============================] - 188s 1s/step - loss: 0.8762 - accuracy: 0.7329 -
val_loss: 1.4532 - val_accuracy: 0.6064
Epoch 6/20
141/141 [==============================] - 188s 1s/step - loss: 0.7406 - accuracy: 0.7719 -
val_loss: 1.4581 - val_accuracy: 0.6064
Epoch 7/20
141/141 [==============================] - 188s 1s/step - loss: 0.6212 - accuracy: 0.8036 -
val_loss: 1.4069 - val_accuracy: 0.6578
Epoch 8/20
141/141 [==============================] - 178s 1s/step - loss: 0.5506 - accuracy: 0.8291 -
val_loss: 1.4487 - val_accuracy: 0.6383
Epoch 9/20
141/141 [==============================] - 180s 1s/step - loss: 0.5058 - accuracy: 0.8384 -
val_loss: 1.4913 - val_accuracy: 0.6330
Epoch 10/20
141/141 [==============================] - 188s 1s/step - loss: 0.4467 - accuracy: 0.8548 -
val_loss: 1.4454 - val_accuracy: 0.6436
Epoch 11/20
141/141 [==============================] - 188s 1s/step - loss: 0.3833 - accuracy: 0.8763 -
val_loss: 1.5249 - val_accuracy: 0.6401
Epoch 12/20
141/141 [==============================] - 188s 1s/step - loss: 0.3378 - accuracy: 0.8921 -
val_loss: 1.4971 - val_accuracy: 0.6277
Epoch 13/20
141/141 [==============================] - 190s 1s/step - loss: 0.2922 - accuracy: 0.9096 -
val_loss: 1.5887 - val_accuracy: 0.6596
Epoch 14/20
141/141 [==============================] - 188s 1s/step - loss: 0.2661 - accuracy: 0.9158 -
val_loss: 1.6276 - val_accuracy: 0.6312
Epoch 15/20
141/141 [==============================] - 190s 1s/step - loss: 0.2389 - accuracy: 0.9231 -
val_loss: 1.6050 - val_accuracy: 0.6348
Epoch 16/20
141/141 [==============================] - 188s 1s/step - loss: 0.2436 - accuracy: 0.9176 -
val_loss: 1.7046 - val_accuracy: 0.6277
Epoch 17/20
141/141 [==============================] - 187s 1s/step - loss: 0.2356 - accuracy: 0.9235 -
val_loss: 1.7589 - val_accuracy: 0.6401
Epoch 18/20
141/141 [==============================] - 174s 1s/step - loss: 0.1963 - accuracy: 0.9384 -
val_loss: 1.7652 - val_accuracy: 0.6188
Epoch 19/20
141/141 [==============================] - 185s 1s/step - loss: 0.2159 - accuracy: 0.9286 -
val_loss: 1.7723 - val_accuracy: 0.6241
Epoch 20/20
141/141 [==============================] - 188s 1s/step - loss: 0.1978 - accuracy: 0.9373 -
val_loss: 1.8616 - val_accuracy: 0.6294
```

The training process was executed for 20 epochs.



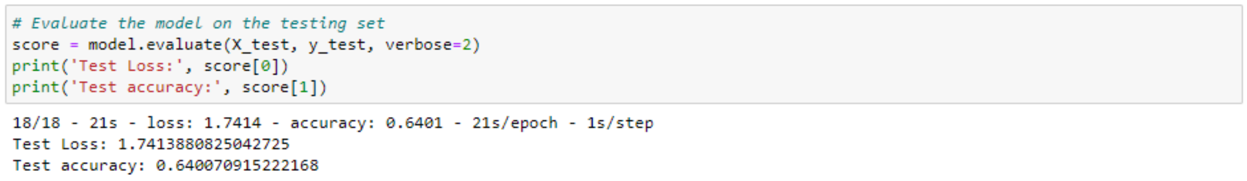
The training loss started at 2.6797 in the first epoch and steadily decreased over subsequent epochs. After 20 epochs, the loss reached 0.1978, indicating a significant reduction.

The training accuracy started at 35.11% in the first epoch and consistently improved throughout the training. After 20 epochs, the accuracy reached 93.73%, demonstrating substantial progress.

The validation loss started at 1.7677 in the first epoch and varied throughout the training. It showed some fluctuations but generally increased slightly over the epochs, reaching 1.8616 in the last epoch.

The validation accuracy began at 54.61% in the first epoch and fluctuated during the training process. It achieved a maximum accuracy of 65.96% in the 13th epoch but decreased slightly towards the end, ending at 62.94%.

Overall, the model shows promising performance. The training loss consistently decreased, indicating that the model was learning from the training data. The accuracy also steadily increased, implying that the model was getting better at predicting the correct classes. However, the validation loss and accuracy did not show consistent improvement, suggesting that the model might be slightly overfitting the training data.



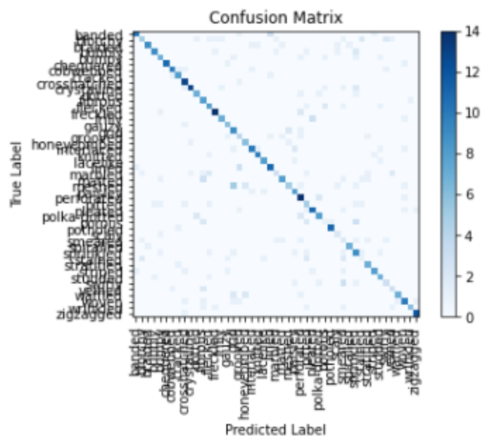
```
# Evaluate the model on the testing set
score = model.evaluate(X_test, y_test, verbose=2)
print('Test Loss:', score[0])
print('Test accuracy:', score[1])
```

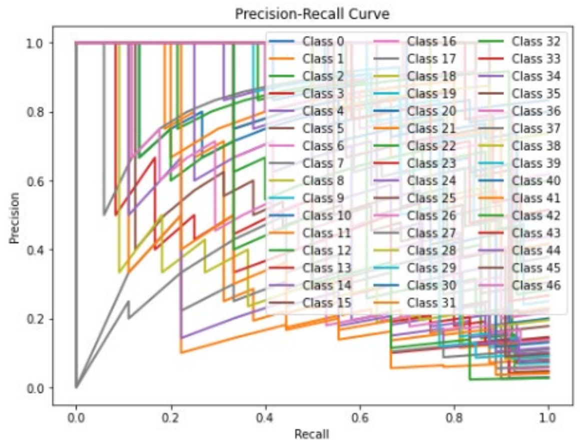
```
18/18 - 21s - loss: 1.7414 - accuracy: 0.6401 - 21s/epoch - 1s/step
Test Loss: 1.7413880825042725
Test accuracy: 0.640070915222168
```

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

           0       0.83      0.67      0.74        15
           1       0.56      0.31      0.40        16
           2       0.90      0.64      0.75        14
           3       0.82      0.60      0.69        15
           4       0.58      0.58      0.58        12
           5       1.00      0.85      0.92        13
           6       0.83      0.71      0.77        14
           7       0.73      0.80      0.76        10
           8       0.65      0.93      0.76        14
           9       0.87      0.68      0.76        19
          10       0.70      0.70      0.70        10
          11       0.44      0.89      0.59         9
          12       0.53      0.62      0.57        13
          13       0.82      0.82      0.82        17
          14       0.78      0.64      0.70        11
          15       0.60      0.86      0.71         7
          16       0.45      0.75      0.56        12
          17       0.36      0.44      0.40         9
          18       0.60      0.75      0.67         8
          19       0.77      0.62      0.69        16
          20       0.83      0.83      0.83        12
          21       0.80      0.80      0.80        10
          22       0.69      0.73      0.71        15
          23       0.50      0.25      0.33        12
          24       0.73      0.53      0.62        15
          25       0.62      0.31      0.42        16
          26       0.62      0.80      0.70        10
          27       0.56      0.82      0.67        17
          28       0.33      0.45      0.38        11
          29       0.73      0.69      0.71        16
          30       0.67      0.67      0.67        12
          31       0.20      0.11      0.14         9
          32       0.92      0.92      0.92        12
          33       0.40      0.67      0.50         3
          34       0.13      0.22      0.17         9
          35       0.88      0.88      0.88         8
          36       0.41      0.53      0.46        17
          37       0.17      0.11      0.13         9
          38       0.64      0.88      0.74         8
          39       0.80      0.62      0.70        13
          40       0.58      0.78      0.67         9
          41       0.43      0.33      0.38         9
          42       0.43      0.50      0.46         6
          43       0.88      0.58      0.70        12
          44       0.62      0.62      0.62        16
          45       0.86      0.55      0.67        11
          46       0.86      0.92      0.89        13

    accuracy                           0.64       564
   macro avg       0.64      0.64      0.63       564
weighted avg       0.66      0.64      0.64       564
```





The model's performance varies across the different classes. The highest precision score is 1.0, achieved by class 5, which means that when the model predicts an instance to belong to this class, it is very likely to be correct. On the other hand, the lowest precision score is 0.13, achieved by class 34, which means that when the model predicts an instance to belong to this class, it is very likely to be incorrect.

Similarly, the highest recall score is 0.93, achieved by class 8, which means that the model correctly identifies most instances of this class. The lowest recall score is 0.11, achieved by class 31, which means that the model has difficulty identifying instances of this class.

The F1-scores are a balance between precision and recall, and they show how well the model performs overall. The highest F1-score is 0.92, achieved by classes 5 and 32, indicating that the model performs very well for these classes. The lowest F1-score is 0.14, achieved by class 31, indicating that the model performs poorly for this class.

Overall, the weighted average F1-score for the model is 0.64, which is not very high. This suggests that the model's performance is inconsistent across the different classes, and there may be some classes for which the model needs more training data or better feature engineering to improve its performance. Additionally, the macro-average F1-score is 0.63, indicating that the model's performance is similar across the different classes.

Comparing the two classification reports, we can see that the previous AlexNet model performed significantly worse compared to this model.

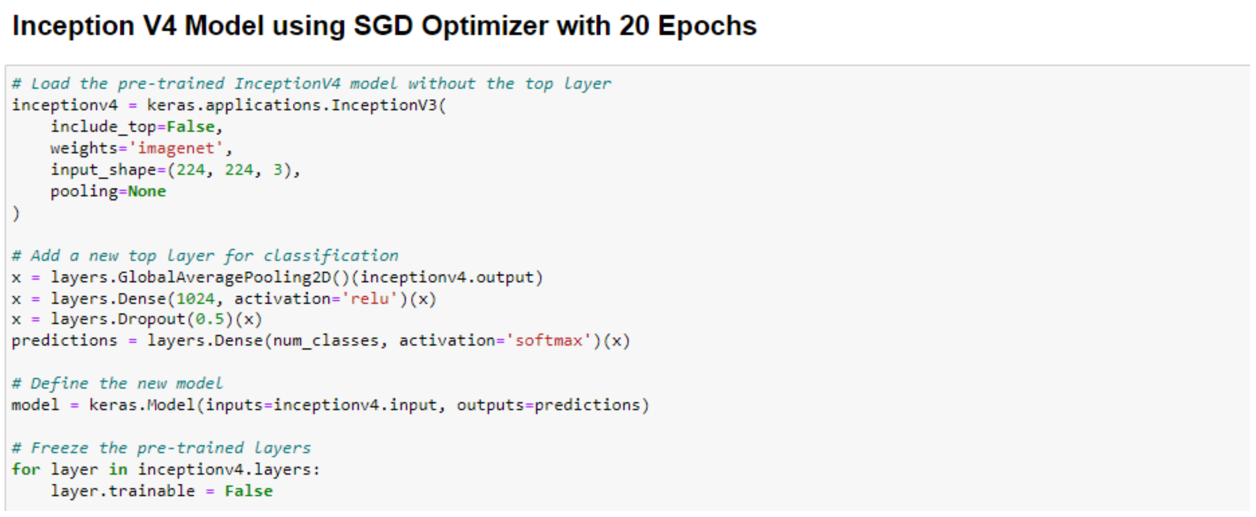
In terms of precision, the previous model achieved lower precision scores for most of the classes, indicating that it had a higher tendency to make false positive predictions. The updated model, on the other hand, generally achieved higher precision scores, suggesting that it made fewer false positive predictions.

Regarding recall, the previous model had lower recall scores for the majority of the classes, indicating that it struggled to correctly identify instances of those classes. The updated model, on the contrary, achieved higher recall scores, suggesting that it was better at identifying instances of the respective classes.

Looking at the F1-scores, which provide a balance between precision and recall, we can see that the updated model consistently outperformed the previous model. The F1-scores for the updated model are generally higher, indicating a better overall performance.

In terms of accuracy, the previous model had an accuracy of 0.25, while the updated model achieved an accuracy of 0.64. This shows a significant improvement in the model's ability to correctly classify instances.

In summary, the updated model with the provided classification report demonstrates better precision, recall, F1-scores, and accuracy compared to the previous AlexNet model. It suggests that the updated model has been trained to perform more effectively in classifying the given dataset.



Inception V4 Model using SGD Optimizer with 20 Epochs

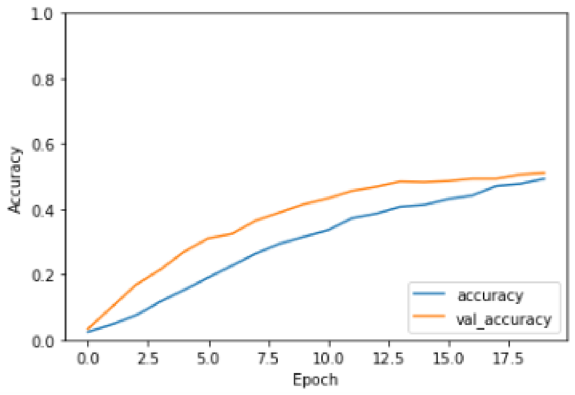
```
# Load the pre-trained InceptionV4 model without the top layer
inceptionv4 = keras.applications.InceptionV3(
    include_top=False,
    weights='imagenet',
    input_shape=(224, 224, 3),
    pooling=None
)

# Add a new top layer for classification
x = layers.GlobalAveragePooling2D()(inceptionv4.output)
x = layers.Dense(1024, activation='relu')(x)
x = layers.Dropout(0.5)(x)
predictions = layers.Dense(num_classes, activation='softmax')(x)

# Define the new model
model = keras.Model(inputs=inceptionv4.input, outputs=predictions)

# Freeze the pre-trained layers
for layer in inceptionv4.layers:
    layer.trainable = False
```

```
Epoch 1/20
141/141 [==============================] - 194s 1s/step - loss: 4.2191 - accuracy: 0.0244 -
val_loss: 3.8667 - val_accuracy: 0.0337
Epoch 2/20
141/141 [==============================] - 189s 1s/step - loss: 3.9424 - accuracy: 0.0468 -
val_loss: 3.6944 - val_accuracy: 0.1011
Epoch 3/20
141/141 [==============================] - 181s 1s/step - loss: 3.7428 - accuracy: 0.0749 -
val_loss: 3.5489 - val_accuracy: 0.1684
Epoch 4/20
141/141 [==============================] - 175s 1s/step - loss: 3.5939 - accuracy: 0.1168 -
val_loss: 3.4218 - val_accuracy: 0.2145
Epoch 5/20
141/141 [==============================] - 174s 1s/step - loss: 3.4284 - accuracy: 0.1525 -
val_loss: 3.3053 - val_accuracy: 0.2695
Epoch 6/20
141/141 [==============================] - 174s 1s/step - loss: 3.2982 - accuracy: 0.1902 -
val_loss: 3.1970 - val_accuracy: 0.3103
Epoch 7/20
141/141 [==============================] - 186s 1s/step - loss: 3.1808 - accuracy: 0.2272 -
val_loss: 3.0935 - val_accuracy: 0.3245
Epoch 8/20
141/141 [==============================] - 189s 1s/step - loss: 3.0475 - accuracy: 0.2644 -
val_loss: 2.9918 - val_accuracy: 0.3652
Epoch 9/20
141/141 [==============================] - 189s 1s/step - loss: 2.9452 - accuracy: 0.2943 -
val_loss: 2.8995 - val_accuracy: 0.3901
Epoch 10/20
141/141 [==============================] - 189s 1s/step - loss: 2.8348 - accuracy: 0.3154 -
val_loss: 2.8088 - val_accuracy: 0.4149
Epoch 11/20
141/141 [==============================] - 189s 1s/step - loss: 2.7562 - accuracy: 0.3353 -
val_loss: 2.7235 - val_accuracy: 0.4326
Epoch 12/20
141/141 [==============================] - 190s 1s/step - loss: 2.6477 - accuracy: 0.3723 -
val_loss: 2.6470 - val_accuracy: 0.4557
Epoch 13/20
141/141 [==============================] - 188s 1s/step - loss: 2.5714 - accuracy: 0.3859 -
val_loss: 2.5732 - val_accuracy: 0.4681
Epoch 14/20
141/141 [==============================] - 189s 1s/step - loss: 2.4900 - accuracy: 0.4067 -
val_loss: 2.5009 - val_accuracy: 0.4840
Epoch 15/20
141/141 [==============================] - 186s 1s/step - loss: 2.4252 - accuracy: 0.4127 -
val_loss: 2.4344 - val_accuracy: 0.4823
Epoch 16/20
141/141 [==============================] - 178s 1s/step - loss: 2.3608 - accuracy: 0.4304 -
val_loss: 2.3747 - val_accuracy: 0.4858
Epoch 17/20
141/141 [==============================] - 243s 2s/step - loss: 2.2846 - accuracy: 0.4415 -
val_loss: 2.3161 - val_accuracy: 0.4929
Epoch 18/20
141/141 [==============================] - 26649s 190s/step - loss: 2.2088 - accuracy: 0.4703
- val_loss: 2.2647 - val_accuracy: 0.4929
Epoch 19/20
141/141 [==============================] - 290s 2s/step - loss: 2.1726 - accuracy: 0.4770 -
val_loss: 2.2161 - val_accuracy: 0.5053
Epoch 20/20
141/141 [==============================] - 295s 2s/step - loss: 2.0925 - accuracy: 0.4925 -
val_loss: 2.1675 - val_accuracy: 0.5106
```



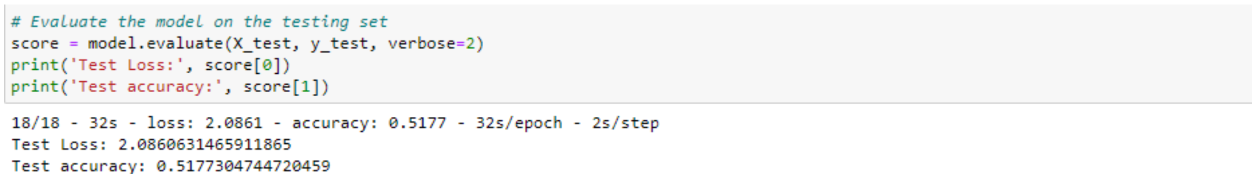
A higher loss value compared to the previous results using Adam optimizer at each epoch. This indicates that the model's predictions are less accurate in the new results.

Also, lower accuracy values throughout the training compared to the previous results. This suggests that the model's performance in correctly classifying the data has decreased.

The validation loss generally decreases with each epoch, indicating that the model is learning and improving over time. However, the validation loss in the previous results also showed a decreasing trend.

The validation accuracy shows a slight increase over the epochs, indicating some improvement in the model's ability to generalize to unseen data. However, the validation accuracy in the previous results was consistently higher.

Overall, the previous results with Adam optimizer seem to indicate a better model performance with higher accuracy and lower loss compared to the new results.



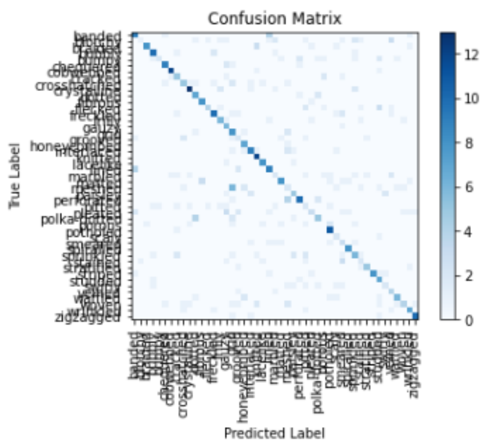
```
# Evaluate the model on the testing set
score = model.evaluate(X_test, y_test, verbose=2)
print('Test Loss:', score[0])
print('Test accuracy:', score[1])
```

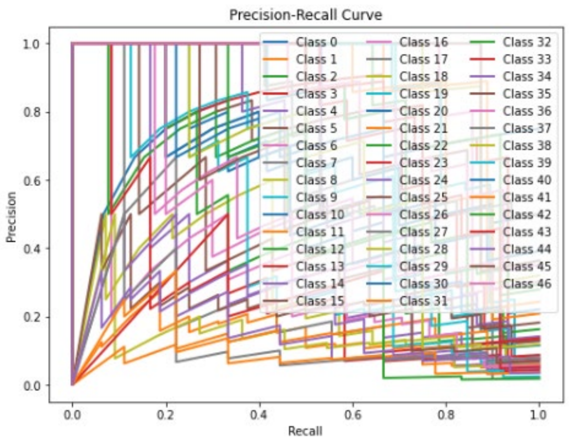
```
18/18 - 32s - loss: 2.0861 - accuracy: 0.5177 - 32s/epoch - 2s/step
Test Loss: 2.0860631465911865
Test accuracy: 0.5177304744720459
```

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

           0       0.53      0.60      0.56        15
           1       0.50      0.06      0.11        16
           2       0.89      0.57      0.70        14
           3       0.67      0.67      0.67        15
           4       0.67      0.17      0.27        12
           5       0.75      0.69      0.72        13
           6       0.86      0.86      0.86        14
           7       0.42      0.50      0.45        10
           8       0.50      0.36      0.42        14
           9       0.72      0.68      0.70        19
          10       0.47      0.70      0.56        10
          11       0.73      0.89      0.80         9
          12       0.67      0.31      0.42        13
          13       0.71      0.59      0.65        17
          14       0.55      0.55      0.55        11
          15       0.33      0.86      0.48         7
          16       0.30      0.67      0.41        12
          17       0.20      0.22      0.21         9
          18       0.54      0.88      0.67         8
          19       0.60      0.56      0.58        16
          20       0.60      1.00      0.75        12
          21       0.73      0.80      0.76        10
          22       0.50      0.60      0.55        15
          23       0.22      0.17      0.19        12
          24       0.67      0.53      0.59        15
          25       0.13      0.12      0.13        16
          26       0.45      0.50      0.48        10
          27       0.67      0.59      0.62        17
          28       0.15      0.18      0.17        11
          29       0.43      0.19      0.26        16
          30       0.45      0.42      0.43        12
          31       0.00      0.00      0.00         9
          32       0.61      0.92      0.73        12
          33       0.25      0.67      0.36         3
          34       0.20      0.33      0.25         9
          35       0.80      1.00      0.89         8
          36       0.40      0.35      0.38        17
          37       0.25      0.22      0.24         9
          38       0.71      0.62      0.67         8
          39       1.00      0.62      0.76        13
          40       0.37      0.78      0.50         9
          41       0.40      0.22      0.29         9
          42       0.33      0.67      0.44         6
          43       0.67      0.50      0.57        12
          44       0.57      0.25      0.35        16
          45       0.58      0.64      0.61        11
          46       0.83      0.77      0.80        13

    accuracy                           0.52       564
   macro avg       0.52      0.53      0.50       564
weighted avg       0.54      0.52      0.51       564
```





The previous model using Adam optimizer achieved an accuracy of 0.64, while the current model using SGD optimizer has an accuracy of 0.52. Therefore, the previous model had a higher overall accuracy.

In the previous model, the precision values ranged from 0.13 to 1.00, with a weighted average precision of 0.66. In the current model, the precision values range from 0.00 to 1.00, with a weighted average precision of 0.54. The previous model generally had higher precision values across most classes.

The recall values using Adam ranged from 0.11 to 0.93, with a weighted average recall of 0.64. The recall values using SGD range from 0.06 to 1.00, with a weighted average recall of 0.52. The previous model generally had higher recall values across most classes.

In the previous model, the F1-scores ranged from 0.13 to 0.92, with a weighted average F1-score of 0.64. In the current model, the F1-scores range from 0.00 to 0.89, with a weighted average F1-score of 0.51. The previous model generally had higher F1-scores across most classes.

Overall, the previous model performed better in terms of accuracy, precision, recall, and F1-score compared to the current model.

Overall Model Results

Model	Number of Layers	Number of Epochs	Training Loss / Accuracy	Validation Loss / Accuracy	Test Loss / Accuracy
LeNet 5	7	20	0.1253 0.9752	12.0452 0.0975	13.3695 0.0851
CNN	11	20	3.2689 0.2538	4.0851 0.1082	
CNN	11	50	2.6147 0.2954	3.9666 0.1099	3.9436 0.1418
AlexNet	18	50	2.0701 0.4335	3.0149 0.2411	3.0014 0.2518
Inception V4	**Adam Optimizer**	**20**	**0.1978** **0.9373**	**1.8616** **0.6294**	**1.7414** **0.6401**
Inception V4	SGD Optimizer	20	2.0925 0.4925	2.1675 0.5106	2.0861 0.5177

From the results provided, it appears that the Inception V4 model using Adam Optimizer has the lowest test loss (1.7417) and the highest test accuracy (0.6401), indicating strong performance on the test dataset. Additionally, the model achieved low training and validation loss values (0.1978 and 1.8616, respectively) and high training and validation accuracy values (0.9373 and 0.6294, respectively), further confirming its good performance.

Code

As attached in the folder. File name: JupyterNotebook_G1