SENSE : CLASSIFYING FABRIC PATTERNS USING DEEP LEARNING PATTERN

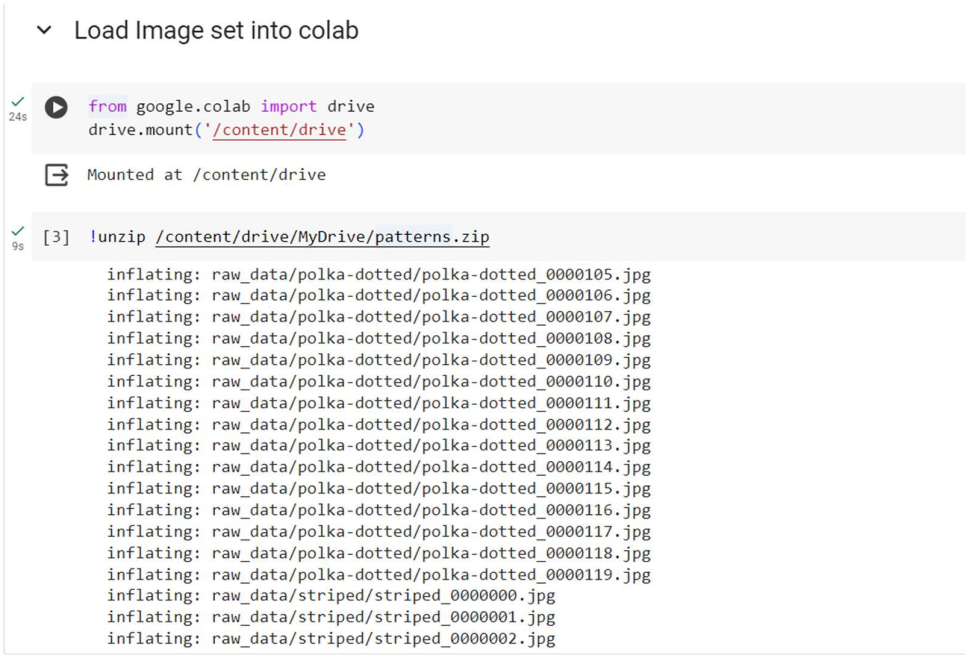
DATA COLLECTION

There are many popular open sources for collecting the data. Eg: kaggle.com, UCI repository, etc.

Download the dataset

It is the most crucial aspect that makes algorithm training possible. So this section allows you to download the required dataset.
There are many popular open sources for collecting the data. Eg: kaggle.com, UCI repository, etc.
In this project, we have used 'Common Fabric Pattern" data. This data is downloaded from kaggle.com.
Please refer to the link given below to download the dataset.
Link: https://www.kaggle.com/datasets/nguyngiabol/dress-pattern-dataset
As the dataset is downloaded. Let us read and understand the data properly with the help of some visualization techniques and some analyzing techniques.
Note: There are several techniques for understanding the data. But here we have used some of it. In an additional way, you can use multiple techniques
We are going to build our training model on Google Colab.

Upload the dataset into Google Drive and connect the Google Colab with drive using the below code



Create dataset

To build a DL model we have six classes in our dataset. But In the project dataset folder training and testing data are needed. So, in this case, we just have to assign a variable and pass the folder path to it.

Importing the libraries

```
[1]  import pandas as pd
     import numpy as np
     from numpy import random
     import os
     import matplotlib.pyplot as plt
```

Pandas and Numpy libraries are used for reading and manipulating the dataframes. Os is used to access the directory file. Matplotlib is a common Python library which is used for plotting graphs or images as the case may be.

Three different transfer learning models are used in our project and the best model is selected.

The image input size of the model is 255, 255, 3.

```
[ ]  def read_data(folder):
         label, paths = [], []
         for l in labels:
             path = f"{folder}/{l}/"
             folder_data = os.listdir(path)
             for image_path in folder_data:
                 label.append(l)
                 paths.append(os.path.join(directory, l, image_path))

         return label, paths
```

```
[ ]  all_labels, all_paths = read_data(directory)
```

```
[ ]  df = pd.DataFrame({
         'path': all_paths,
         'label': all_labels
     })
```

This Python function read_data(folder) aims to read image data from a specified folder structure. It takes a folder parameter representing the root directory containing subfolders, each corresponding to a particular label or category of images. It initializes two empty lists label and paths to store the labels and file paths respectively. The function iterates over each label in the labels list (which is assumed to be defined elsewhere), constructs the path to the corresponding subfolder within the specified folder, and retrieves a list of files within that subfolder using os.listdir(). For each image file within the subfolder, it appends the label to the label list and constructs the full file path by joining the root directory, label, and image filename using os.path.join(), finally appending this path to the paths list. Once all images from all labels are processed, the function returns the lists of labels and corresponding file paths.

Split into train, test, and validation sets

Split into train, test, and valid sets

```
[ ] from sklearn.model_selection import train_test_split
```

```
[ ] train_df, dummy_df=train_test_split(df, train_size=.8, random_state=123, shuffle=True, stratify=df['label'])
    valid_df, test_df=train_test_split(dummy_df, train_size=.5, random_state=123, shuffle=True, stratify=dummy_df['label'])
    print("Train dataset : ",len(train_df),"Test dataset : ",len(test_df),"Validation dataset : ",len(valid_df))
    train_balance=train_df['label'].value_counts()
    print('Train dataset value count: \n',train_df['label'].value_counts())
```

```
Train dataset :  672 Test dataset :  84 Validation dataset :  84
Train dataset value count:
 animal-print    96
striped         96
plain           96
paisley         96
zigzagged       96
chequered       96
polka-dotted    96
Name: label, dtype: int64
```

The code performs data splitting using the train_test_split function from scikit-learn. Initially, it splits the original dataset df into training (train_df) and test (dummy_df) sets, allocating 80% of the data to training and 20% to testing, while ensuring class balance through stratification based on the 'label' column. Then, it further divides the test set dummy_df into validation (valid_df) and final test (test_df) sets, each comprising 50% of the data. Finally, it prints the sizes of the three datasets and the value counts of the classes in the training set to verify the distribution.

Extracting labels from the files directory

```
[4] directory = '/content/raw_data'
```

Saving Labels

```
labels = os.listdir(directory)
labels
```

```
['polka-dotted',
 'plain',
 'chequered',
 'paisley',
 'animal-print',
 'striped',
 'zigzagged']
```

```
[6] labels.sort()
```

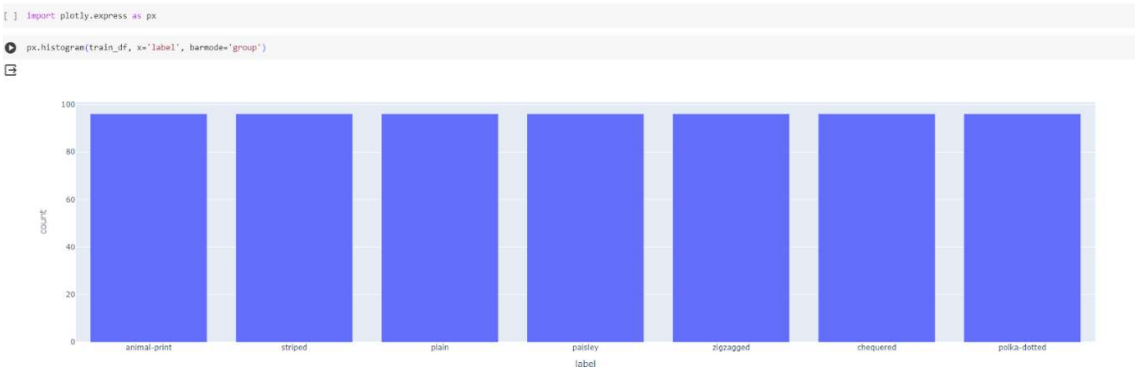
A

This code snippet retrieves a list of labels or classes from a directory. It uses the `os.listdir()` function to get the list of all items (files and directories) in the specified `directory`. The `labels` variable then stores this list, which typically represents the class names or categories used in a classification task. Each item in the `labels` list corresponds to a unique class or

category in the dataset. We then sort the list in alphabetical order to ensure that our model gives the correct results.

IMAGE AUGMENTATION

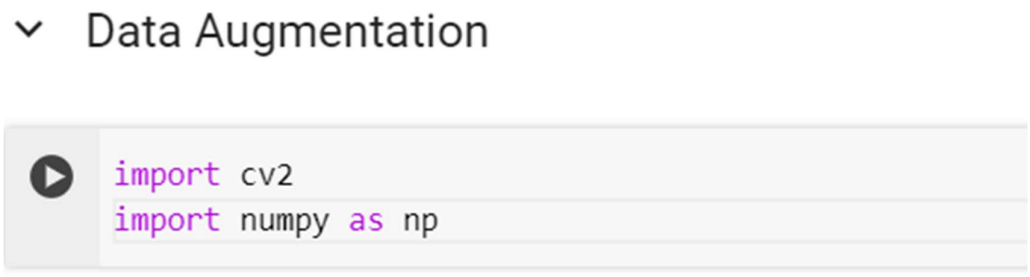
Image data augmentation helps to increase the size and diversity of the training dataset by applying transformations such as rotations, flips, and shifts to the original images. This helps prevent overfitting and improves the model's ability to generalize unseen data by exposing it to a wider range of variations and scenarios.



First we will start with visualising the counts of the different classes in the train set. The counts are equal so there is no need for data augmentation.

Importing the libraries

Import the necessary libraries as shown in the image



Cv2 is a Computer Vision library which will be used for data image manipulations.

Defining the augmentation function

We will create a function that will take an image and augment it and return it back to the calling statement.

```
[ ]  def apply_transform(image):

        # Rotate (random angle between -40 and 40 degrees)
        angle = np.random.uniform(-40, 40)
        rows, cols = image.shape[:2]
        M = cv2.getRotationMatrix2D((cols / 2, rows / 2), angle, 1)
        image = cv2.warpAffine(image, M, (cols, rows))

        # Horizontal Flip
        if np.random.rand() < 0.5:
            image = cv2.flip(image, 1)

        # Vertical Flip
        if np.random.rand() < 0.5:
            image = cv2.flip(image, 0)

        # Random Brightness and Contrast
        alpha = 1.0 + np.random.uniform(-0.2, 0.2)  # Brightness
        beta = 0.0 + np.random.uniform(-0.2, 0.2)   # Contrast
        image = cv2.convertScaleAbs(image, alpha=alpha, beta=beta)

        # Random Gamma Correction
        gamma = np.random.uniform(0.8, 1.2)
        image = np.clip((image / 255.0) ** gamma, 0, 1) * 255.0

        return image
```

The provided Python function `apply_transform(image)` performs image augmentation by applying random rotations (between -40 and 40 degrees), horizontal and vertical flips with a 50% probability, random adjustments to brightness and contrast, and random gamma correction. These transformations introduce variability to the input images, effectively increasing the diversity of the training dataset. This helps prevent overfitting and improves the model's ability to generalise unseen data by exposing it to a broader range of scenarios and variations.

Next we will create another augmentation function that calls the apply_transform method.

```
[ ]  def apply_augmentation(image_path, label):
        image = cv2.imread(image_path)
        image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
        augmented_image = apply_transform(image=image)
        return augmented_image, label
```

The function `apply_augmentation(image_path, label)` reads an image from the specified `image_path` using OpenCV, converts it to RGB color space, and then applies augmentation transformations using the `apply_transform()` function. Finally, it returns the augmented image along with the corresponding label. This function serves to augment a single image with transformations such as rotation, flipping, brightness and contrast adjustments, and

gamma correction,thereby enhancing the diversity of the training dataset for improved model generalization and performance.

IMAGE PREPROCESSING

In this milestone, we will be improving the image data that suppresses unwilling distortions or enhances some image features important for further processing, although performing some geometric transformations of images like rotation, scaling, translation, etc

Importing the libraries

Import the necessary libraries as shown in the image

.

```
[ ]  from keras.preprocessing.image import ImageDataGenerator
```

Keras.ImageDataGenerator is a powerful tool used for data augmentation in computer vision tasks, particularly image classification.

Configure ImageDataGenerator class

ImageDataGenerator class is instantiated and the configuration for the types of data augmentation

There are five main types of data augmentation techniques for image data; specifically:

- Image shifts via the width_shift_range and height_shift_range arguments.
- The image flips via the horizontal_flip and vertical_flip arguments.
- Image rotations via the rotation_range argument
- Image brightness via the brightness_range argument.
- Image zoom via the zoom_range argument.

An instance of the ImageDataGenerator class can be constructed as:

```
[ ]  gen=ImageDataGenerator()
```

Apply ImageDataGenerator functionality to train_df, valid_df, test_df

The provided code segment utilizes the flow_from_dataframe() method from a data augmentation generator (gen) to generate batches of augmented image data for training, validation, and testing purposes in a deep learning pipeline.

For the train_gen, balanced_df DataFrame is used as the source of training data. The 'path' column specifies the paths to the images, while the 'label' column indicates the corresponding labels. Images are resized to a target size of (255,255) pixels. The seed parameter ensures reproducibility by setting the random seed for shuffling. class_mode is set to 'categorical' as the labels are one-hot encoded. color_mode is specified as 'rgb' to indicate that images are in RGB color space. The shuffle parameter is set to True to shuffle the data after each epoch, enhancing randomness during training. Finally, batch_size is set to 32, determining the number of samples per batch during training.

Similarly, for the val_gen and test_gen, the valid_df and test_df DataFrames are used as the sources of validation and testing data, respectiv**Creating and Compiling the model**

```
[ ]  model=Sequential()
     model.add(Convolution2D(filters=32, kernel_size=3, padding='same', activation="relu",
                             input_shape=(255, 255, 3)))
     model.add(MaxPooling2D(strides=2, pool_size=2, padding= "valid"))
     model.add(Convolution2D(filters=32, kernel_size=2, padding='same', activation="relu"))
     model.add(MaxPooling2D(strides=2, pool_size=2, padding="valid"))
     model.add(Dropout(0.5))
     model.add(Convolution2D(filters=32, kernel_size=2, padding='same', activation="relu"))
     model.add(MaxPooling2D(strides=2, pool_size=2, padding="valid"))
     model.add(Dropout(0.5))
     model.add(Flatten())
     model.add(Dense(128, activation='relu'))
     model.add(Dropout(0.5))
     model.add(Dense(9, activation='softmax'))
```

This code defines a convolutional neural network (CNN) using the Keras framework. Let's break down each part:

- `from keras.models import Sequential`: This imports the Sequential model from the Keras library, which allows us to build a linear stack of layers.
- `from keras.layers import Conv2D, MaxPool2D, Flatten, Dense, InputLayer, BatchNormalization, Dropout`: This imports various layer types from Keras that will be used to define the architecture of the CNN.
- `model = Sequential()`: This initializes a Sequential model, which is an empty neural network architecture.
- `model.add(Conv2D(32, (3, 3), activation='relu', padding='same', name='conv_1', input_shape=(255,255,3)))`: This adds a convolutional layer with 32 filters of size 3x3, using ReLU (Rectified Linear Unit) activation function. The 'same' padding is used to maintain the spatial dimensions of the input. This layer is named 'conv_1', and it expects input images of size 255x255 with 3 channels (RGB).

- `model.add(MaxPooling2D((2, 2), name='maxpool_1'))`: This adds a max-pooling layer with a pool size of 2x2. Max-pooling reduces the spatial dimensions of the feature maps while retaining the most important information.

- This sequence of adding convolutional and max-pooling layers is repeated three more times with increasing numbers of filters (64, 128, 128), resulting in a hierarchical representation of the input images.

- `model.add(Flatten())`: This adds a flatten layer that converts the 3D feature maps into a 1D vector, preparing them to be input into a dense (fully connected) layer.

- `model.add(Dropout(0.5))`: This adds a dropout layer with a dropout rate of 0.5, which helps prevent overfitting by randomly setting a fraction of input units to zero during training.

- `model.add(Dense(128, activation='relu', name='dense_2'))`: This adds a fully connected (dense) layer with 128 units and ReLU activation function.

- `model.add(Dense(7, activation='softmax', name='output'))`: This adds the output layer with 7 units (assuming a classification task with 7 classes) and softmax activation function, which outputs probabilities for each class.

- `model.compile(loss='categorical_crossentropy',optimizer='Adam',metrics=['accuracy'])`: This compiles the model, specifying the loss function (categorical crossentropy), optimizer (Adam), and metrics to be evaluated during training (accuracy).

In summary, this code defines a CNN architecture for image classification with convolutional, max-pooling, dropout, and fully connected layers, suitable for processing RGB images of size 255x255 pixels.

```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv_1 (Conv2D)             (None, 255, 255, 32)      896

 maxpool_1 (MaxPooling2D)    (None, 127, 127, 32)      0

 conv_2 (Conv2D)             (None, 127, 127, 64)      18496

 maxpool_2 (MaxPooling2D)    (None, 63, 63, 64)        0

 conv_3 (Conv2D)             (None, 63, 63, 128)       73856

 maxpool_3 (MaxPooling2D)    (None, 31, 31, 128)       0

 conv_4 (Conv2D)             (None, 31, 31, 128)       147584

 maxpool_4 (MaxPooling2D)    (None, 15, 15, 128)       0

 flatten (Flatten)           (None, 28800)             0

 dropout (Dropout)           (None, 28800)             0

 dense_2 (Dense)             (None, 128)               3686528

 output (Dense)              (None, 7)                 903

=================================================================
Total params: 3928263 (14.99 MB)
Trainable params: 3928263 (14.99 MB)
Non-trainable params: 0 (0.00 Byte)
```

Using model.summary(), we can view all the layers in the CNN model.

```
[ ]  model_checkpoint_callback = tf.keras.callbacks.ModelCheckpoint(
        filepath="model_cnn.h5",
        monitor='val_accuracy',
        mode='max',
        save_best_only=True,
        verbose=1)
```

The provided code snippet configures a ModelCheckpoint callback in TensorFlow/Keras, which is a useful tool for automatically saving the best model during training. By specifying parameters such as the file path to save the model, the monitored metric (in this case, validation accuracy), and whether to save only the best model observed, the callback ensures that the most optimal model is preserved. This is particularly beneficial for preventing overfitting and ensuring that the model generalizes well to unseen data. Additionally, the verbosity parameter controls the level of output displayed during training, allowing users to monitor the process.

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

The provided code snippet compiles a Keras model using the Adam optimizer and categorical cross-entropy loss function, which is commonly used for multi-class classification tasks. Additionally, the accuracy metric is specified to monitor the performance of the model during training and evaluation. This configuration prepares the model for the training process, where it learns to minimize the specified loss function using the optimizer while maximizing the chosen metric to improve its classification accuracy.

Parameters such as shuffle and batch_size are adjusted accordingly, with shuffle set to False for validation and testing data to ensure consistency during evaluation.

MODEL BUILDING

Now it's time to build our model. First, we will create a CNN (Convolutional Neural Network) on our own and test its performance

Importing the libraries

The following libraries will be required

```
from keras.models import Sequential
from keras.layers import Dense, Activation, Dropout, Flatten, BatchNormalization
from keras.layers import Convolution2D, MaxPooling2D
from keras import regularizers
from keras.models import Model
from keras.optimizers import Adam, Adamax
import tensorflow as tf
```

The keras library is used to import all the layers that will be needed to create the CNN.

Creating and Compiling the model

```
[ ]  model=Sequential()
     model.add(Convolution2D(filters=32, kernel_size=3, padding='same', activation="relu",
                             input_shape=(255, 255, 3)))
     model.add(MaxPooling2D(strides=2, pool_size=2, padding= "valid"))
     model.add(Convolution2D(filters=32, kernel_size=2, padding='same', activation="relu"))
     model.add(MaxPooling2D(strides=2, pool_size=2, padding="valid"))
     model.add(Dropout(0.5))
     model.add(Convolution2D(filters=32, kernel_size=2, padding='same', activation="relu"))
     model.add(MaxPooling2D(strides=2, pool_size=2, padding="valid"))
     model.add(Dropout(0.5))
     model.add(Flatten())
     model.add(Dense(128, activation='relu'))
     model.add(Dropout(0.5))
     model.add(Dense(9, activation='softmax'))
```

This code defines a convolutional neural network (CNN) using the Keras framework. Let's break down each part:

- `from keras.models import Sequential`: This imports the Sequential model from the Keras library, which allows us to build a linear stack of layers.
- `from keras.layers import Conv2D, MaxPool2D, Flatten, Dense, InputLayer, BatchNormalization, Dropout`: This imports various layer types from Keras that will be used to define the architecture of the CNN.
- `model = Sequential()`: This initializes a Sequential model, which is an empty neural network architecture.
- `model.add(Conv2D(32, (3, 3), activation='relu', padding='same', name='conv_1', input_shape=(255,255,3)))`: This adds a convolutional layer with 32 filters of size 3x3, using ReLU (Rectified Linear Unit) activation function. The 'same' padding is used to maintain the spatial dimensions of the input. This layer is named 'conv_1', and it expects input images of size 255x255 with 3 channels (RGB).

- `model.add(MaxPooling2D((2, 2), name='maxpool_1'))`: This adds a max-pooling layer with a pool size of 2x2. Max-pooling reduces the spatial dimensions of the feature maps while retaining the most important information.

- This sequence of adding convolutional and max-pooling layers is repeated three more times with increasing numbers of filters (64, 128, 128), resulting in a hierarchical representation of the input images.

- `model.add(Flatten())`: This adds a flatten layer that converts the 3D feature maps into a 1D vector, preparing them to be input into a dense (fully connected) layer.

- `model.add(Dropout(0.5))`: This adds a dropout layer with a dropout rate of 0.5, which helps prevent overfitting by randomly setting a fraction of input units to zero during training.

- `model.add(Dense(128, activation='relu', name='dense_2'))`: This adds a fully connected (dense) layer with 128 units and ReLU activation function.

- `model.add(Dense(7, activation='softmax', name='output'))`: This adds the output layer with 7 units (assuming a classification task with 7 classes) and softmax activation function, which outputs probabilities for each class.

- `model.compile(loss='categorical_crossentropy',optimizer='Adam',metrics=['accuracy'])`: This compiles the model, specifying the loss function (categorical crossentropy), optimizer (Adam), and metrics to be evaluated during training (accuracy).

In summary, this code defines a CNN architecture for image classification with convolutional, max-pooling, dropout, and fully connected layers, suitable for processing RGB images of size 255x255 pixels.

```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv_1 (Conv2D)             (None, 255, 255, 32)      896

 maxpool_1 (MaxPooling2D)    (None, 127, 127, 32)      0

 conv_2 (Conv2D)             (None, 127, 127, 64)      18496

 maxpool_2 (MaxPooling2D)    (None, 63, 63, 64)        0

 conv_3 (Conv2D)             (None, 63, 63, 128)       73856

 maxpool_3 (MaxPooling2D)    (None, 31, 31, 128)       0

 conv_4 (Conv2D)             (None, 31, 31, 128)       147584

 maxpool_4 (MaxPooling2D)    (None, 15, 15, 128)       0

 flatten (Flatten)           (None, 28800)             0

 dropout (Dropout)           (None, 28800)             0

 dense_2 (Dense)             (None, 128)               3686528

 output (Dense)              (None, 7)                 903

=================================================================
Total params: 3928263 (14.99 MB)
Trainable params: 3928263 (14.99 MB)
Non-trainable params: 0 (0.00 Byte)
```

Using model.summary(), we can view all the layers in the CNN model.

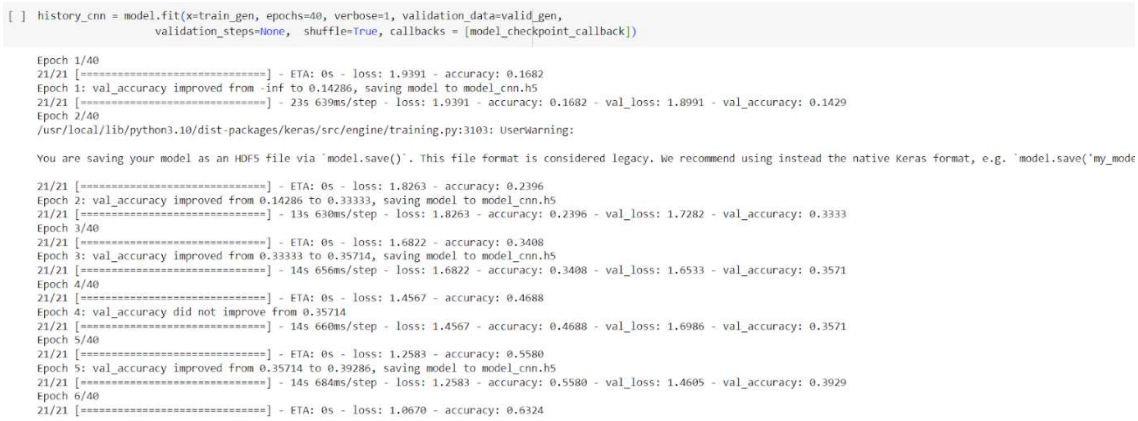
```
[ ]  model_checkpoint_callback = tf.keras.callbacks.ModelCheckpoint(
        filepath="model_cnn.h5",
        monitor='val_accuracy',
        mode='max',
        save_best_only=True,
        verbose=1)
```

The provided code snippet configures a ModelCheckpoint callback in TensorFlow/Keras, which is a useful tool for automatically saving the best model during training. By specifying parameters such as the file path to save the model, the monitored metric (in this case, validation accuracy), and whether to save only the best model observed, the callback ensures that the most optimal model is preserved. This is particularly beneficial for preventing overfitting and ensuring that the model generalizes well to unseen data. Additionally, the verbosity parameter controls the level of output displayed during training, allowing users to monitor the process.

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

The provided code snippet compiles a Keras model using the Adam optimizer and categorical cross-entropy loss function, which is commonly used for multi-class classification tasks. Additionally, the accuracy metric is specified to monitor the performance of the model during training and evaluation. This configuration prepares the model for the training process, where it learns to minimize the specified loss function using the optimizer while maximizing the chosen metric to improve its classification accuracy.

Running and Evaluating the model



```
[ ] history_cnn = model.fit(x=train_gen, epochs=40, verbose=1, validation_data=valid_gen,
                            validation_steps=None,  shuffle=True, callbacks = [model_checkpoint_callback])

    Epoch 1/40
    21/21 [==============================] - ETA: 0s - loss: 1.9391 - accuracy: 0.1682
    Epoch 1: val_accuracy improved from -inf to 0.14286, saving model to model_cnn.h5
    21/21 [==============================] - 23s 639ms/step - loss: 1.9391 - accuracy: 0.1682 - val_loss: 1.8991 - val_accuracy: 0.1429
    Epoch 2/40
    /usr/local/lib/python3.10/dist-packages/keras/src/engine/training.py:3103: UserWarning:

    You are saving your model as an HDF5 file via `model.save()`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_mode

    21/21 [==============================] - ETA: 0s - loss: 1.8263 - accuracy: 0.2396
    Epoch 2: val_accuracy improved from 0.14286 to 0.33333, saving model to model_cnn.h5
    21/21 [==============================] - 13s 630ms/step - loss: 1.8263 - accuracy: 0.2396 - val_loss: 1.7282 - val_accuracy: 0.3333
    Epoch 3/40
    21/21 [==============================] - ETA: 0s - loss: 1.6822 - accuracy: 0.3408
    Epoch 3: val_accuracy improved from 0.33333 to 0.35714, saving model to model_cnn.h5
    21/21 [==============================] - 14s 656ms/step - loss: 1.6822 - accuracy: 0.3408 - val_loss: 1.6533 - val_accuracy: 0.3571
    Epoch 4/40
    21/21 [==============================] - ETA: 0s - loss: 1.4567 - accuracy: 0.4688
    Epoch 4: val_accuracy did not improve from 0.35714
    21/21 [==============================] - 14s 660ms/step - loss: 1.4567 - accuracy: 0.4688 - val_loss: 1.6986 - val_accuracy: 0.3571
    Epoch 5/40
    21/21 [==============================] - ETA: 0s - loss: 1.2583 - accuracy: 0.5580
    Epoch 5: val_accuracy improved from 0.35714 to 0.39286, saving model to model_cnn.h5
    21/21 [==============================] - 14s 684ms/step - loss: 1.2583 - accuracy: 0.5580 - val_loss: 1.4605 - val_accuracy: 0.3929
    Epoch 6/40
    21/21 [==============================] - ETA: 0s - loss: 1.0670 - accuracy: 0.6324
```

The provided code snippet trains a convolutional neural network (CNN) model using TensorFlow's Keras API. It utilizes data generators `train_generator` and `valid_generator` to feed augmented image data batches into the model for training and validation, respectively. The model is trained over 10 epochs, with each epoch iterating through the entire training dataset and adjusting the model's weights to minimize categorical cross-entropy loss. The validation data is used to evaluate the model's performance on unseen data after each epoch. Training progress and metrics are displayed on the console (`verbose=1`). The `history` object returned by the `fit()` method stores information about training and validation loss and accuracy for further analysis. Overall, this process facilitates the training of the CNN model, allowing it to learn from the training data and improve its performance over successive epochs.

However, we can observe that our CNN is not performing well. It is giving a training accuracy of 97% and validation accuracy of about 71%.

ResNet50 Model Initialisation

ResNet-50, short for Residual Network with 50 layers, is a deep convolutional neural network architecture that has significantly impacted the field of computer vision. Introduced by Microsoft Research in 2015, ResNet-50 is renowned for its remarkable depth while mitigating the vanishing gradient problem commonly encountered in deep networks. The key innovation of ResNet-50 lies in the use of residual connections, or skip connections, which allow the network to learn residual mappings instead of directly fitting the desired

underlying mapping. This enables the training of very deep networks, up to 50 layers or more, by mitigating the degradation problem caused by the increased depth. ResNet-50 has demonstrated superior performance on various visual recognition tasks, including image classification, object detection, and semantic segmentation, and it serves as a foundational architecture in modern deep learning frameworks and applications.

```
[ ] base_model=tf.keras.applications.ResNet50(include_top=False, weights="imagenet",input_shape=(255,255,3))
    print('Created ResNet50  model')

    Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/resnet/resnet50_weights_tf_
    94765736/94765736 [==============================] - 0s 0us/step
    Created ResNet50  model
```

Adding Dense Layers and Compiling the Model

A dense layer is a deeply connected neural network layer. It is the most common and frequently used layer.

```
for layer in base_model.layers:
    layer.trainable = False
```

+ Code + Text

```
[ ] for layer in base_model.layers[173:]:
        layer.trainable = True
```

```
[ ] x1 = base_model.output

    #Global average pool to reduce number of features and Flatten the output
    x2 = tf.keras.layers.GlobalAveragePooling2D()(x1)
```

```
[ ] # adding extra layers
    x3 = tf.keras.layers.Dense(1024,activation='relu',kernel_initializer= "he_uniform")(x2)
    x4 = tf.keras.layers.Dropout(0.4)(x3)
    x5=  tf.keras.layers.Dense(512,activation='relu',kernel_initializer= "he_uniform")(x4)
```

In the provided code snippet, a pre-trained base model is used, and its layers are frozen by setting `trainable` to `False` for all layers. Then, starting from the 173rd layer, the layers are unfrozen to allow fine-tuning of these layers during training.

The output of the base model (`base_model.output`) is passed through a global average pooling layer (`GlobalAveragePooling2D`) to reduce the number of features and flatten the output. Subsequently, two dense layers with 1024 and 512 units, respectively, are added, each followed by ReLU activation and dropout regularization with a dropout rate of 0.4. These dense layers serve to further process the features extracted by the base model before the final classification layer. The `kernel_initializer` parameter is set to "he_uniform" for both

dense layers, which initializes the layer weights using a He uniform distribution.

```
[ ]  #Add output layer
     prediction = tf.keras.layers.Dense(7,activation='softmax')(x5)
```

```
     final_model = tf.keras.models.Model(inputs=base_model.input, #Pre-trained model input as input layer
                                         outputs=prediction)
```

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

```
[ ]  model_checkpoint_callback_rs = tf.keras.callbacks.ModelCheckpoint(
         filepath="model_50.h5",
         monitor='val_accuracy',
         mode='max',
         save_best_only=True,
         verbose=1)
```

In this code snippet, a final prediction layer is added to the model (`final_model`) after the last dense layer (`x5`). This prediction layer consists of a dense layer with 9 units (assuming it's a classification task with 9 classes) and softmax activation function, which is suitable for multi-class classification problems.

The `final_model` is then defined using the `tf.keras.models.Model` class, specifying the inputs as the input layer of the pre-trained model (`base_model.input`) and the outputs as the prediction layer (`prediction`).

Finally, the model is compiled using the Adam optimizer, categorical crossentropy as the loss function (since it's a multi-class classification problem), and accuracy as the evaluation metric.

Create callbacks

Callbacks in convolutional neural networks (CNNs) are objects that can perform actions at various stages during the training process. They are commonly used in TensorFlow and Keras to monitor the training progress, adjust learning rates dynamically, save model checkpoints, and more.

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

The provided code snippet compiles a Keras model using the Adam optimizer and categorical cross-entropy loss function, which is commonly used for multi-class classification tasks. Additionally, the accuracy metric is specified to monitor the performance of the model during training and evaluation. This configuration prepares the model for the training

process, where it learns to minimize the specified loss function using the optimizer while maximizing the chosen metric to improve its classification accuracy.

Train the model

Now, let us train our model with our image dataset. The model is trained for 10 epochs. We can see that the training loss decreases in almost every epoch till 10 epochs and probably there is further scope to improve the model.

fit functions used to train a deep-learning neural network

Arguments:

epochs: an integer and number of epochs we want to train our model for.

validation_data can be either:

- an inputs and targets list

- a generator

- an inputs, targets, and sample_weights list which can be used to evaluate

the loss and metrics for any model after any epoch has ended.

validation_steps: only if the validation_data is a generator then only this argument

can be used. It specifies the total number of steps taken from the generator before it is

stopped at every epoch and its value is calculated as the total number of validation data points

in your dataset divided by the validation batch size.

```
history_resnet = final_model.fit(train_gen,
                    epochs = 20,
                    validation_data = valid_gen,callbacks = [model_checkpoint_callback_rs])
```

```
Epoch 1/20
21/21 [==============================] - ETA: 0s - loss: 2.2749 - accuracy: 0.2515
Epoch 1: val_accuracy improved from -inf to 0.29762, saving model to model_50.h5
21/21 [==============================] - 23s 813ms/step - loss: 2.2749 - accuracy: 0.2515 - val_loss: 1.6784 - val_accuracy: 0.2976
Epoch 2/20
21/21 [==============================] - ETA: 0s - loss: 1.5786 - accuracy: 0.3750
Epoch 2: val_accuracy improved from 0.29762 to 0.47619, saving model to model_50.h5
21/21 [==============================] - 14s 657ms/step - loss: 1.5786 - accuracy: 0.3750 - val_loss: 1.4757 - val_accuracy: 0.4762
Epoch 3/20
21/21 [==============================] - ETA: 0s - loss: 1.3548 - accuracy: 0.4985
Epoch 3: val_accuracy improved from 0.47619 to 0.48810, saving model to model_50.h5
21/21 [==============================] - 14s 626ms/step - loss: 1.3548 - accuracy: 0.4985 - val_loss: 1.3685 - val_accuracy: 0.4881
Epoch 4/20
21/21 [==============================] - ETA: 0s - loss: 1.2655 - accuracy: 0.5283
Epoch 4: val_accuracy did not improve from 0.48810
21/21 [==============================] - 14s 645ms/step - loss: 1.2655 - accuracy: 0.5283 - val_loss: 1.3939 - val_accuracy: 0.4881
Epoch 5/20
21/21 [==============================] - ETA: 0s - loss: 1.2197 - accuracy: 0.5491
Epoch 5: val_accuracy improved from 0.48810 to 0.57143, saving model to model_50.h5
21/21 [==============================] - 15s 704ms/step - loss: 1.2197 - accuracy: 0.5491 - val_loss: 1.2603 - val_accuracy: 0.5714
Epoch 6/20
21/21 [==============================] - ETA: 0s - loss: 1.1768 - accuracy: 0.5744
Epoch 6: val_accuracy did not improve from 0.57143
21/21 [==============================] - 14s 644ms/step - loss: 1.1768 - accuracy: 0.5744 - val_loss: 1.2398 - val_accuracy: 0.5357
Epoch 7/20
21/21 [==============================] - ETA: 0s - loss: 1.1414 - accuracy: 0.5685
Epoch 7: val_accuracy improved from 0.57143 to 0.58333, saving model to model_50.h5
21/21 [==============================] - 14s 693ms/step - loss: 1.1414 - accuracy: 0.5685 - val_loss: 1.1773 - val_accuracy: 0.5833
```

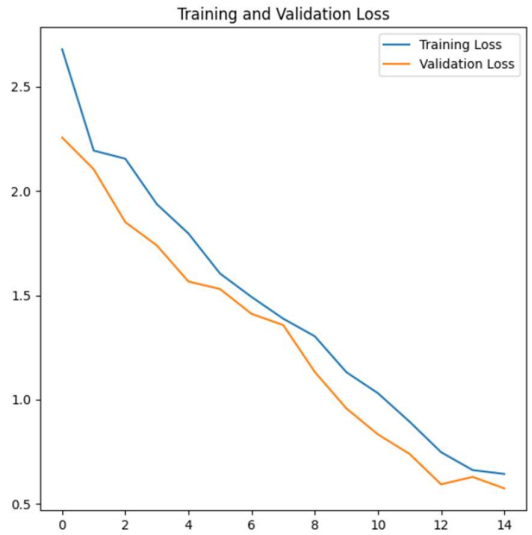
EVALUATING THE MODEL

Visualising Performance

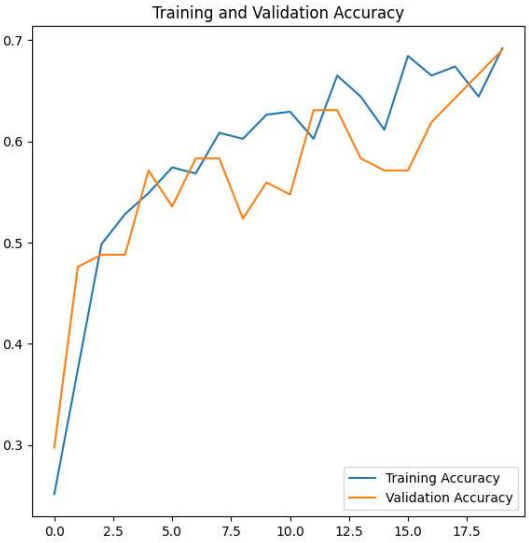
We will plot the accuracy and loss of both training and accuracy for all 40 epochs for the CNN model and 20 epochs of the transfer learning model.

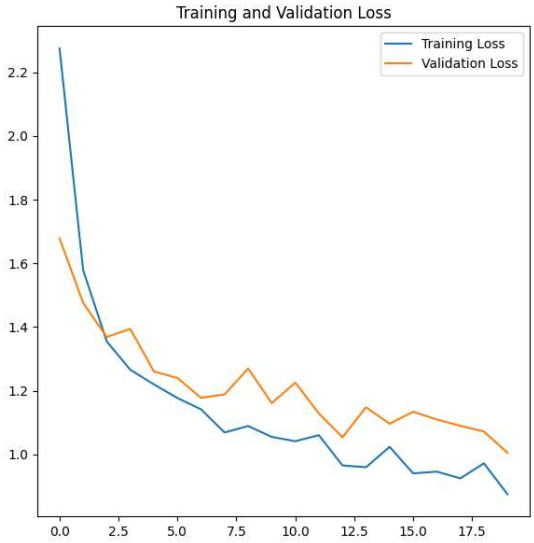
```
[ ] import matplotlib.pyplot as plt
```

```
[ ] acc = history.history['accuracy']
    val_acc = history.history['val_accuracy']
    loss = history.history['loss']
    val_loss = history.history['val_loss']

    epochs_range = range(5)

    plt.figure(figsize=(15, 15))
    plt.subplot(2, 2, 1)
    plt.plot(epochs_range, acc, label='Training Accuracy')
    plt.plot(epochs_range, val_acc, label='Validation Accuracy')
    plt.legend(loc='lower right')
    plt.title('Training and Validation Accuracy')

    plt.subplot(2, 2, 2)
    plt.plot(epochs_range, loss, label='Training Loss')
    plt.plot(epochs_range, val_loss, label='Validation Loss')
    plt.legend(loc='upper right')
    plt.title('Training and Validation Loss')
    plt.show()
```



From the above we can infer that the model is giving a training accuracy of around 80% and validation accuracy of 97% and loss is around 0.5 for both.





From the above we can infer that the ResNet model is giving a training accuracy of around 66% and validation accuracy of 69% and loss is around 0.5 for both.

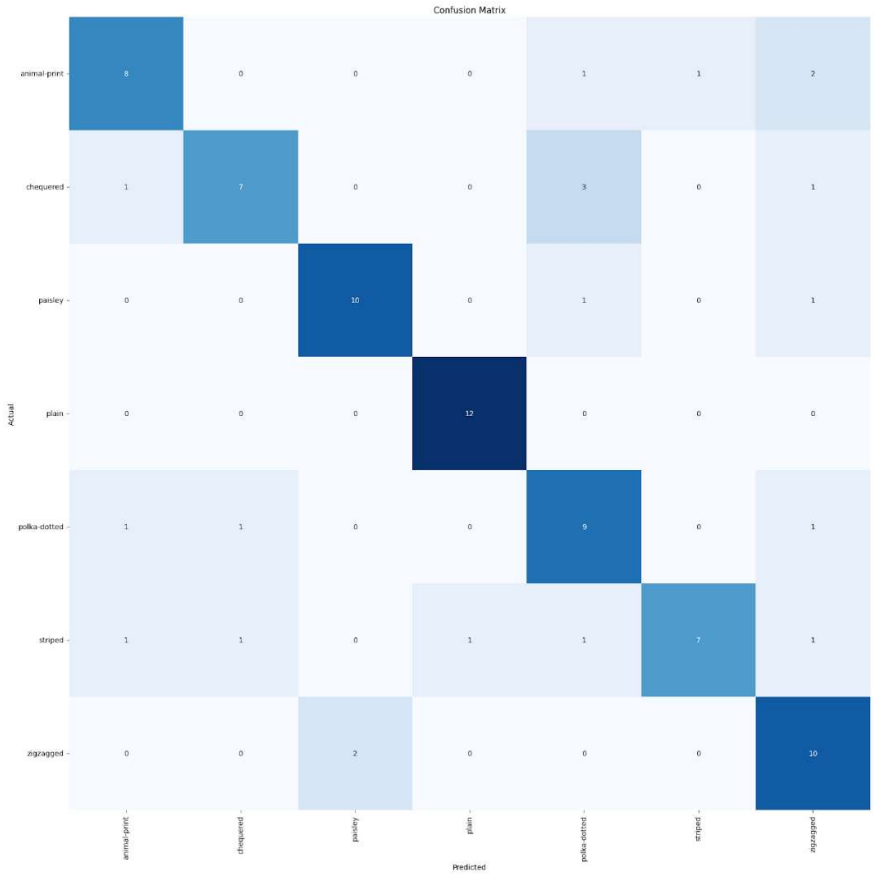
The CNN model is giving better performance so we will proceed with it.

Confusion Matrix on test_df

We will evaluate the model on the test_df and print the confusion matrix to visualise the correctly predicted labels by the model. We will also print its classification report

```python
def predictor(model,test_gen):
    classes=list(test_gen.class_indices.keys())
    class_count=len(classes)
    preds=model.predict(test_gen, verbose=1)
    errors=0
    pred_indices=[]
    test_count =len(preds)
    for i, p in enumerate (preds):
        pred_index=np.argmax(p)
        pred_indices.append(pred_index)
        true_index= test_gen.labels[i]
        if  pred_index != true_index:
            errors +=1
    accuracy = (test_count-errors)*100/test_count
    ytrue=np.array(test_gen.labels, dtype='int')
    ypred=np.array(pred_indices, dtype='int')
    msg=f'There were {errors} errors in {test_count} tests for an accuracy of {accuracy:6.2f}'
    print (msg)
    cm = confusion_matrix(ytrue, ypred )
    # plot the confusion matrix
    plt.figure(figsize=(20, 20))
    sns.heatmap(cm, annot=True, vmin=0, fmt='g', cmap='Blues', cbar=False)
    plt.xticks(np.arange(class_count)+.5, classes, rotation=90)
    plt.yticks(np.arange(class_count)+.5, classes, rotation=0)
    plt.xlabel("Predicted")
    plt.ylabel("Actual")
    plt.title("Confusion Matrix")
    plt.show()
    clr = classification_report(ytrue, ypred, target_names=classes, digits= 4) # create classification report
    print("Classification Report:\n----------------------\n", clr)
    return
```

The function `predictor` takes a trained model and a test data generator as inputs and performs prediction on the test data. It calculates the accuracy of the model, identifies errors in predictions, and generates a confusion matrix and classification report for evaluation. This function provides valuable insights into the model's performance, including its ability to correctly classify different classes and potential areas for improvement. Additionally, it visualizes the results using a heatmap for the confusion matrix, aiding in the interpretation of classification errors. Overall, the `predictor` function serves as a comprehensive tool for assessing the effectiveness of the trained model on unseen test data.



Manual Testing

The last step before saving our model is to perform manual testing to check our model's performance.

```
[ ]  from tensorflow.keras.preprocessing.image import load_img, img_to_array
```

```
def get_model_prediction(image_path):
    img = load_img(image_path, target_size=(255, 255))
    x = img_to_array(img)
    x = np.expand_dims(x, axis=0)
    predictions = model.predict(x, verbose=0)
    return labels[predictions.argmax()]
```

The function `get_model_prediction` takes the path of an image file as input. It loads the image, resizes it to a target size of 255x255 pixels, and converts it to a NumPy array. The array is then expanded to create a batch dimension, as the model expects input in batch format.

The model predicts the class probabilities for the input image using the `predict` method. The `verbose=0` parameter suppresses the verbosity during prediction.

Finally, the function returns the predicted label for the image, which is determined by the class with the highest probability according to the model's output.

Manual Testing

```
pred = []
for file in test_df['path'].values:
    pred.append(get_model_prediction(file))
```

```
[ ] fig, axes = plt.subplots(nrows=4, ncols=4, figsize=(15, 10))
    random_index = np.random.randint(0, len(test_gen), 16)

    for i, ax in enumerate(axes.ravel()):
        img_path = test_df['path'].iloc[random_index[i]]

        ax.imshow( load_img(img_path))
        ax.axis('off')

        if test_df['label'].iloc[random_index[i]] == pred[random_index[i]]:
            color = "green"
        else:
            color = "red"

        ax.set_title(f"True: {test_df['label'].iloc[random_index[i]]}\nPredicted: {pred[random_index[i]]}", color=color)

    plt.tight_layout()
    plt.show()
```

```
[ ] fig, axes = plt.subplots(nrows=4, ncols=4, figsize=(15, 10))
    random_index = np.random.randint(0, len(test_gen), 16)

    for i, ax in enumerate(axes.ravel()):
        img_path = test_df['path'].iloc[random_index[i]]

        ax.imshow( load_img(img_path))
        ax.axis('off')

        if test_df['label'].iloc[random_index[i]] == pred[random_index[i]]:
            color = "green"
        else:
            color = "red"

        ax.set_title(f"True: {test_df['label'].iloc[random_index[i]]}\nPredicted: {pred[random_index[i]]}", color=color)

    plt.tight_layout()
    plt.show()
```

The provided code segment generates predictions for a subset of images from the test dataset using the `get_model_prediction` function. It then visualizes the predicted results alongside the true labels for comparison.

A subplot grid with 4 rows and 4 columns is created using `plt.subplots()`, with a figure size of 15x10 inches. A random index is generated to select 16 images from the test dataset.

For each subplot, an image is loaded from the test dataset using the image path stored in `test_df['path']`. The image is displayed using `imshow()`, and the axis is turned off using `ax.axis('off')`.

The true label and predicted label for each image are displayed in the subplot title. If the true label matches the predicted label, the title text is displayed in green; otherwise, it is displayed in red.

Finally, `plt.tight_layout()` is called to adjust subplot parameters for better layout, and `plt.show()` displays the plot.

Overall, this code segment provides a visual representation of the model's predictions on a subset of test images, allowing for easy interpretation and assessment of the model's performance.

Save the Model

The model is saved with .h5 extension as follows.

An H5 file is a data file saved in the Hierarchical Data Format (HDF). It contains multidimensional arrays of scientific data. We have already saved the model with the best performance during the callbacks while fitting the model. The model is saved at model_cnn (2).h5.

APPLICATION BUILDING

In this section, we will be building a web application that is integrated to the model we built. A UI is provided for the uses where he has to enter the values for predictions. The enter values are given to the saved model and prediction is showcased on the UI.

This section has the following tasks

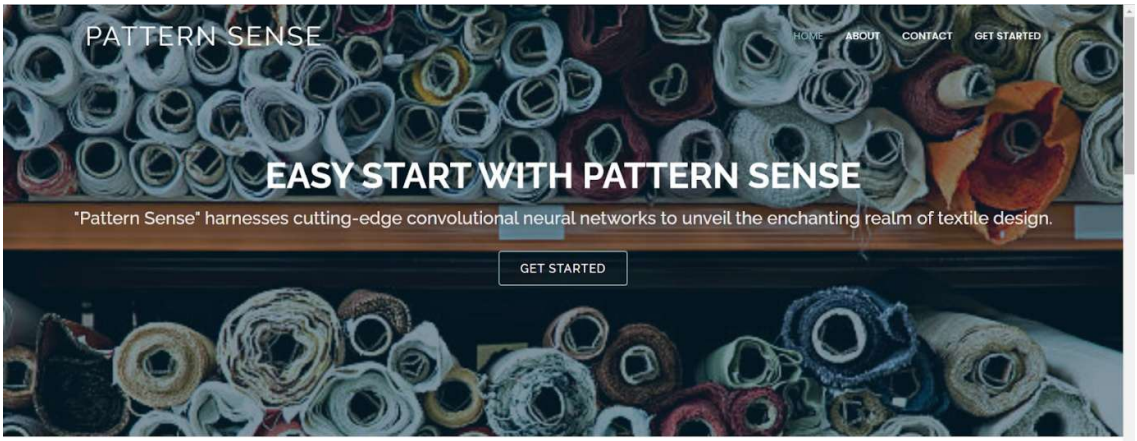
- Building HTML Pages
- Building server-side script

Building Html Page

For this project create one HTML file namely

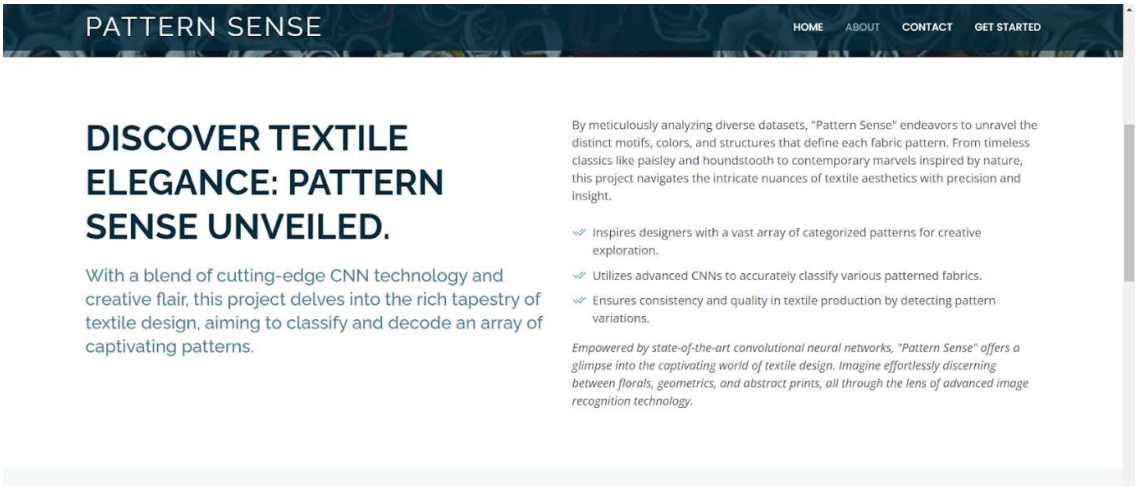
- home.html

Let’s see how our home.html page looks like:



DISCOVER TEXTILE

By meticulously analyzing diverse datasets, "Pattern Sense" endeavors to unravel the distinct motifs, colors, and structures that define each fabric pattern. From timeless



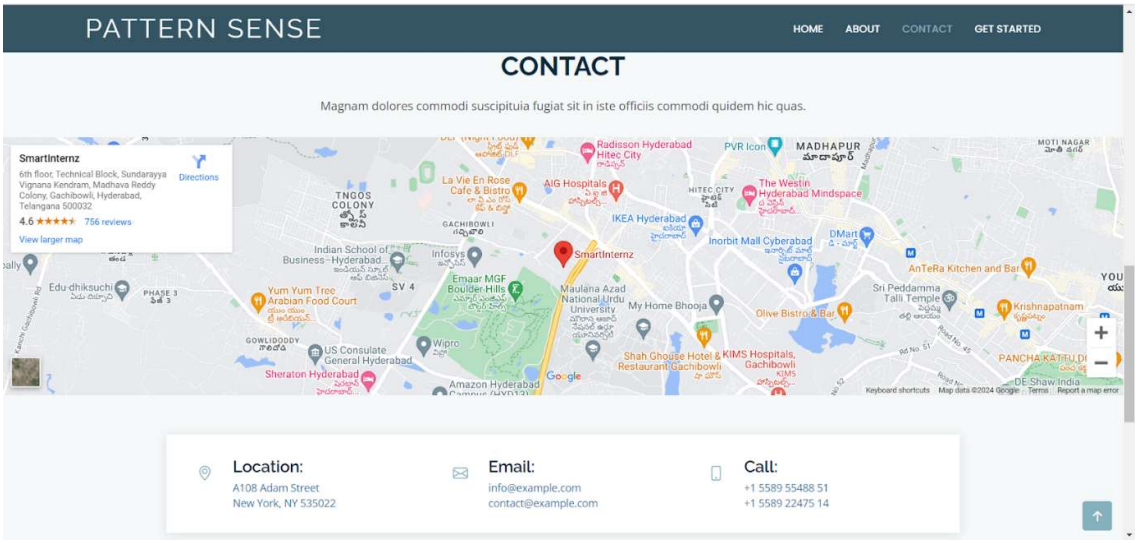
DISCOVER TEXTILE ELEGANCE: PATTERN SENSE UNVEILED.

With a blend of cutting-edge CNN technology and creative flair, this project delves into the rich tapestry of textile design, aiming to classify and decode an array of captivating patterns.

By meticulously analyzing diverse datasets, "Pattern Sense" endeavors to unravel the distinct motifs, colors, and structures that define each fabric pattern. From timeless classics like paisley and houndstooth to contemporary marvels inspired by nature, this project navigates the intricate nuances of textile aesthetics with precision and insight.

- Inspires designers with a vast array of categorized patterns for creative exploration.
- Utilizes advanced CNNs to accurately classify various patterned fabrics.
- Ensures consistency and quality in textile production by detecting pattern variations.

Empowered by state-of-the-art convolutional neural networks, "Pattern Sense" offers a glimpse into the captivating world of textile design. Imagine effortlessly discerning between florals, geometrics, and abstract prints, all through the lens of advanced image recognition technology.



Build Python code

- Import the libraries
- Loading the saved model and initializing the Flask app
- Render HTML pages:

- Once we upload the file into the app, then verifying the file uploaded properly or not. Here we will be using the declared constructor to route to the HTML page that we have created earlier.

- In the above example, '/' URL is bound with the home.html function. Hence, when the home page of the web server is opened in the browser, the HTML page will be rendered. Whenever you enter the values from the html page the values can be retrieved using POST Method.

```python
from flask import Flask, render_template, request
import tensorflow._api.v2 as tf
from keras.models import load_model
from keras.preprocessing.image import load_img, img_to_array
import numpy as np

app=Flask(__name__)

model=load_model('model_cnn (2).h5')

labels=['Chancellor Hall','Chancellor Tower','Clock Tower','Colorfull Stairway','DKP Baru','Library','Recital Hall','UMS Aquarium','UMS Mosque

def get_model_prediction(image_path):
    img = load_img(image_path, target_size=(255, 255))
    x = img_to_array(img)
    x = np.expand_dims(x, axis=0)
    predictions = model.predict(x, verbose=0)
    return labels[predictions.argmax()]

@app.route('/')
def Home():
    return render_template("home.html")

@app.route('/predict_page')
def predict():
    return render_template("predict.html")

@app.route('/predict',methods=['POST'])
def prediction():
    img = request.files['ump_image']
    img_path = "static/assets/uploads/" + img.filename
    img.save(img_path)
    p = get_model_prediction(img_path)
    return render_template("predictionpage.html",img_path=img_path,prediction=p)

if __name__ == "__main__":
    app.run(debug=True)
```

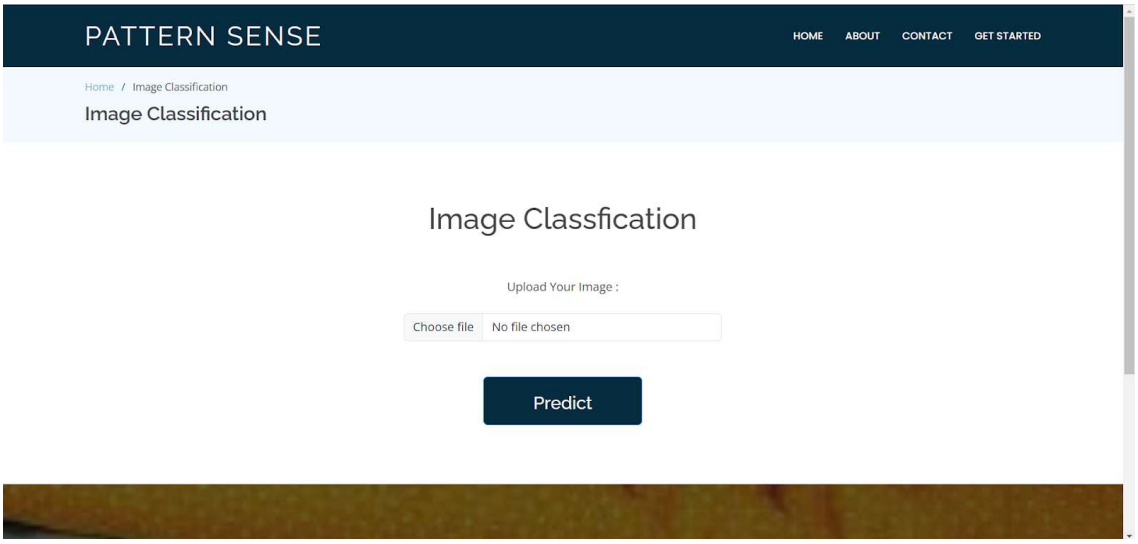
Here we are routing our app to the prediction function. This function retrieves all the values from the HTML page using a Post request. That is stored in an array. This array is passed to the model.predict() function. This function returns the prediction. This prediction value will be rendered to the text that we have mentioned in the index.html page earlier.

Run the application

- Open the Anaconda prompt from the start menu.
- Navigate to the folder where your Python script is.
- Now type the “python my_app.py” command.
- Navigate to the localhost where you can view your web page.
- Click on the predict button from the top right corner, enter the inputs, click on the submit button, and see the result/prediction on the web.

```
To enable the following instructions: SSE SSE2 SSE3 SSE4.1 SSE4.2 AVX AVX2 AVX512F AVX512_VNNI FMA, in other operations, rebuild TensorFlow with the appropriate
compiler flags.
 * Debugger is active!
 * Debugger PIN: 946-817-010
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

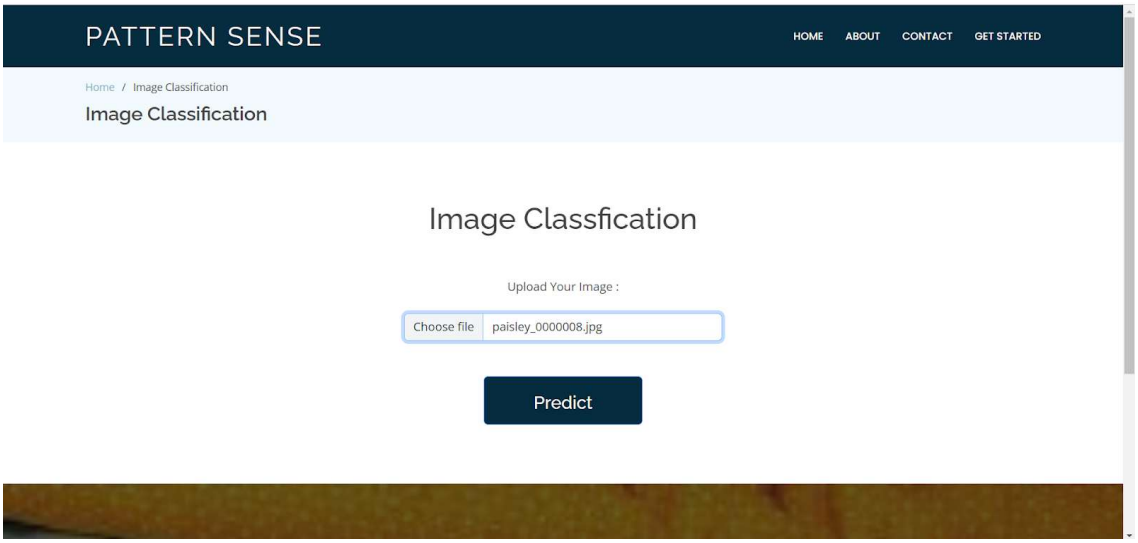
Click on Get Started at the top right corner to go to the inner page as below:



Upload the image

Click on Predict button

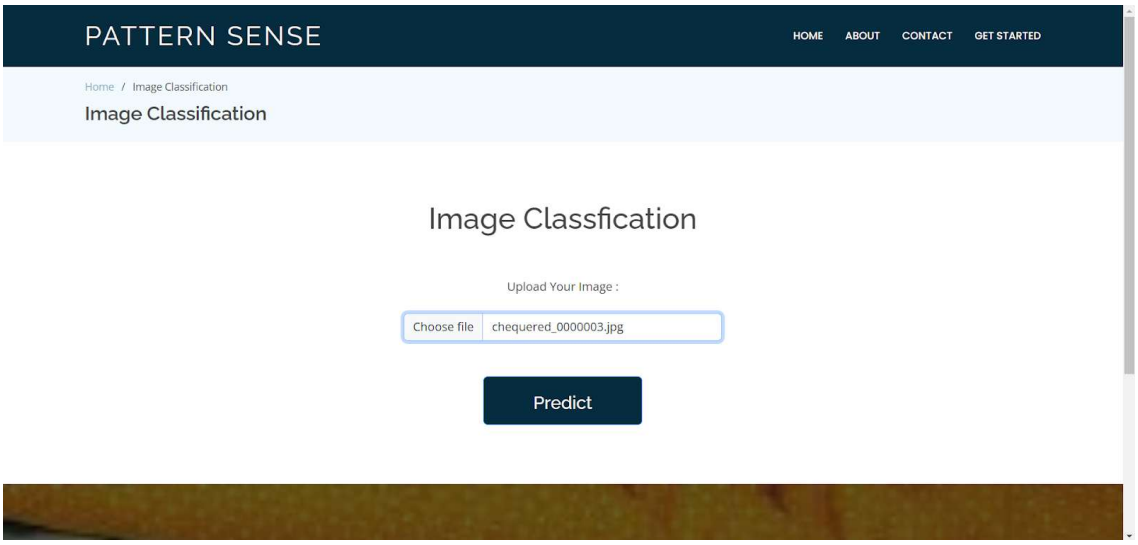
Input 1:



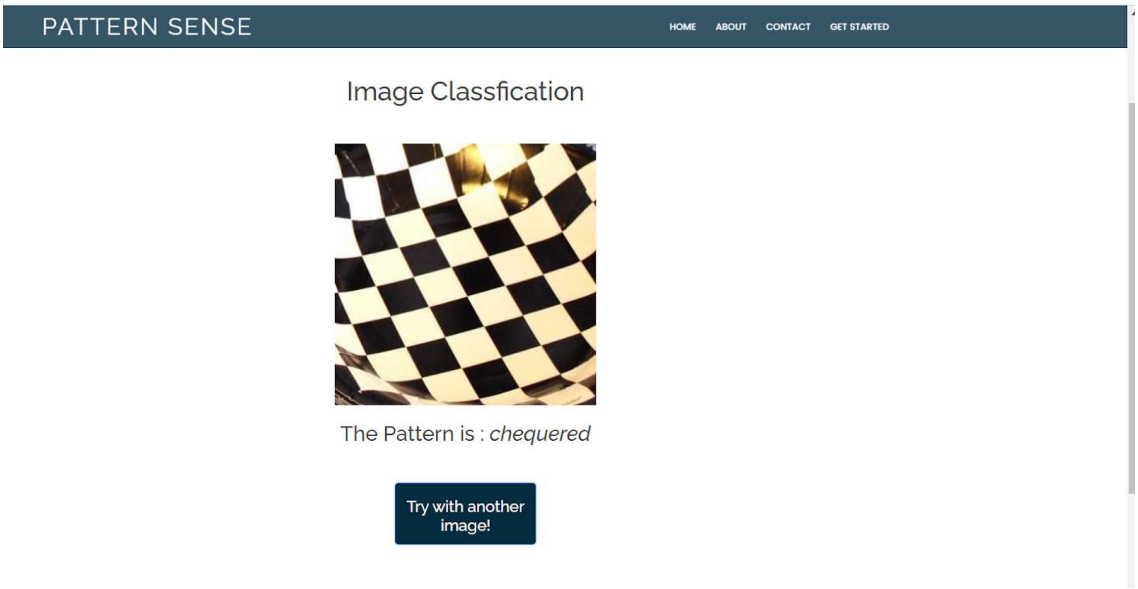
Once you upload the image and click on the submit button, the output will be displayed on the below page

Output:1

Input:2


PATTERN SENSE
HOME
ABOUT
CONTACT
GET STARTED
Home / Image Classification
Image Classification
Image Classfication
Upload Your Image :
Choose file
chequered_0000003.jpg
Predict

Output:2


PATTERN SENSE
HOME
ABOUT
CONTACT
GET STARTED
Image Classfication
The Pattern is : chequered
Try with another image!