Introduction to Computer Vision: Plant Seedlings Classification

Problem Statement

Context

In recent times, the field of agriculture has been in urgent need of modernizing, since the amount of manual work people need to put in to check if plants are growing correctly is still highly extensive. Despite several advances in agricultural technology, people working in the agricultural industry still need to have the ability to sort and recognize different plants and weeds, which takes a lot of time and effort in the long term. The potential is ripe for this trillion-dollar industry to be greatly impacted by technological innovations that cut down on the requirement for manual labor, and this is where Artificial Intelligence can actually benefit the workers in this field, as **the time and energy required to identify plant seedlings will be greatly shortened by the use of AI and Deep Learning.** The ability to do so far more efficiently and even more effectively than experienced manual labor, could lead to better crop yields, the freeing up of human inolvement for higher-order agricultural decision making, and in the long term will result in more sustainable environmental practices in agriculture as well.

Objective

The aim of this project is to Build a Convolutional Neural Netowrk to classify plant seedlings into their respective categories.

Data Dictionary

The Aarhus University Signal Processing group, in collaboration with the University of Southern Denmark, has recently released a dataset containing **images of unique plants belonging to 12 different species.**

- The dataset can be download from Olympus.
- The data file names are:
  - images.npy
  - Labels.csv
- Due to the large volume of data, the images were converted to the images.npy file and the labels are also put into Labels.csv, so that you can work on the data/project seamlessly without having to worry about the high data volume.
- The goal of the project is to create a classifier capable of determining a plant's species from an image.

List of Species

- Black-grass
- Charlock
- Cleavers
- Common Chickweed
- Common Wheat
- Fat Hen
- Loose Silky-bent
- Maize
- Scentless Mayweed
- Shepherds Purse
- Small-flowered Cranesbill
- Sugar beet

Note: Please use GPU runtime on Google Colab to execute the code faster.

Importing necessary libraries

```
#Installing the libraries with the specified version.
#uncomment and run the following line if Google Colab is being used
!pip install tensorflow==2.15.0 scikit-learn==1.2.2 seaborn==0.13.1
matplotlib==3.7.1 numpy==1.25.2 pandas==1.5.3 opencv-python==4.8.0.76
-q --user

  WARNING: The scripts f2py, f2py3 and f2py3.10 are installed in
'/root/.local/bin' which is not on PATH.
  Consider adding this directory to PATH or, if you prefer to suppress
this warning, use --no-warn-script-location.
  WARNING: The script tensorboard is installed in '/root/.local/bin'
which is not on PATH.
  Consider adding this directory to PATH or, if you prefer to suppress
this warning, use --no-warn-script-location.
  WARNING: The scripts estimator_ckpt_converter,
import_pb_to_tensorboard, saved_model_cli, tensorboard, tf_upgrade_v2,
tflite_convert, toco and toco_from_protos are installed in
'/root/.local/bin' which is not on PATH.
  Consider adding this directory to PATH or, if you prefer to suppress
this warning, use --no-warn-script-location.
ERROR: pip's dependency resolver does not currently take into account
all the packages that are installed. This behaviour is the source of
the following dependency conflicts.
xgboost 2.1.1 requires nvidia-nccl-cu12; platform_system == "Linux"
and platform_machine != "aarch64", which is not installed.
albucore 0.0.13 requires typing-extensions>=4.9.0, but you have
typing-extensions 4.5.0 which is incompatible.
albumentations 1.4.13 requires typing-extensions>=4.9.0, but you have
typing-extensions 4.5.0 which is incompatible.
cudf-cu12 24.4.1 requires pandas<2.2.2dev0,>=2.0, but you have pandas
```

```
1.5.3 which is incompatible.
google-colab 1.0.0 requires pandas==2.1.4, but you have pandas 1.5.3
which is incompatible.
pandas-stubs 2.1.4.231227 requires numpy>=1.26.0; python_version <
"3.13", but you have numpy 1.25.2 which is incompatible.
tensorstore 0.1.64 requires ml-dtypes>=0.3.1, but you have ml-dtypes
0.2.0 which is incompatible.
tf-keras 2.17.0 requires tensorflow<2.18,>=2.17, but you have
tensorflow 2.15.0 which is incompatible.
xarray 2024.6.0 requires pandas>=2.0, but you have pandas 1.5.3 which
is incompatible.
```

```
# Installing the libraries with the specified version.
# uncomment and run the following lines if Jupyter Notebook is being
used
!pip install tensorflow==2.13.0 scikit-learn==1.2.2 seaborn==0.11.1
matplotlib==3.3.4 numpy==1.24.3 pandas==1.5.2 opencv-python==4.8.0.76
-q --user
```

```
  WARNING: The scripts f2py, f2py3 and f2py3.10 are installed in
'/root/.local/bin' which is not on PATH.
  Consider adding this directory to PATH or, if you prefer to suppress
this warning, use --no-warn-script-location.
  WARNING: The script tensorboard is installed in '/root/.local/bin'
which is not on PATH.
  Consider adding this directory to PATH or, if you prefer to suppress
this warning, use --no-warn-script-location.
  WARNING: The scripts estimator_ckpt_converter,
import_pb_to_tensorboard, saved_model_cli, tensorboard, tf_upgrade_v2,
tflite_convert, toco and toco_from_protos are installed in
'/root/.local/bin' which is not on PATH.
  Consider adding this directory to PATH or, if you prefer to suppress
this warning, use --no-warn-script-location.
ERROR: pip's dependency resolver does not currently take into account
all the packages that are installed. This behaviour is the source of
the following dependency conflicts.
xgboost 2.1.1 requires nvidia-nccl-cu12; platform_system == "Linux"
and platform_machine != "aarch64", which is not installed.
albucore 0.0.13 requires numpy<2,>=1.24.4, but you have numpy 1.24.3
which is incompatible.
albucore 0.0.13 requires typing-extensions>=4.9.0, but you have
typing-extensions 4.5.0 which is incompatible.
albumentations 1.4.13 requires numpy>=1.24.4, but you have numpy
1.24.3 which is incompatible.
albumentations 1.4.13 requires typing-extensions>=4.9.0, but you have
typing-extensions 4.5.0 which is incompatible.
arviz 0.18.0 requires matplotlib>=3.5, but you have matplotlib 3.3.4
which is incompatible.
bigframes 1.13.0 requires matplotlib>=3.7.1, but you have matplotlib
3.3.4 which is incompatible.
```

```
cudf-cu12 24.4.1 requires pandas<2.2.2dev0,>=2.0, but you have pandas 1.5.2 which is incompatible.
google-colab 1.0.0 requires pandas==2.1.4, but you have pandas 1.5.2 which is incompatible.
mizani 0.9.3 requires matplotlib>=3.5.0, but you have matplotlib 3.3.4 which is incompatible.
pandas-stubs 2.1.4.231227 requires numpy>=1.26.0; python_version < "3.13", but you have numpy 1.24.3 which is incompatible.
plotnine 0.12.4 requires matplotlib>=3.6.0, but you have matplotlib 3.3.4 which is incompatible.
tensorstore 0.1.64 requires ml-dtypes>=0.3.1, but you have ml-dtypes 0.2.0 which is incompatible.
tf-keras 2.17.0 requires tensorflow<2.18,>=2.17, but you have tensorflow 2.13.0 which is incompatible.
xarray 2024.6.0 requires pandas>=2.0, but you have pandas 1.5.2 which is incompatible.
```

Note: *After running the above cell, kindly restart the notebook kernel and run all cells sequentially from the start again.*

```python
import os
import numpy as np
# Importing numpy for Matrix Operations
import pandas as pd
# Importing pandas to read CSV files
import matplotlib.pyplot as plt
# Importting matplotlib for Plotting and visualizing images
import math
# Importing math module to perform mathematical operations
import cv2
# Importing openCV for image processing
import seaborn as sns
# Importing seaborn to plot graphs


# Tensorflow modules
import tensorflow as tf
from tensorflow.keras.preprocessing.image import ImageDataGenerator
# Importing the ImageDataGenerator for data augmentation
from tensorflow.keras.models import Sequential
# Importing the sequential module to define a sequential model
from tensorflow.keras.layers import Dense,Dropout,Flatten,Conv2D,MaxPooling2D,BatchNormalization # Defining all the layers to build our CNN Model
from tensorflow.keras.optimizers import Adam,SGD
# Importing the optimizers which can be used in our model
from sklearn import preprocessing
```

```
# Importing the preprocessing module to preprocess the data
from sklearn.model_selection import train_test_split
# Importing train_test_split function to split the data into train and
test
from sklearn.metrics import confusion_matrix
# Importing confusion_matrix to plot the confusion matrix

# Display images using OpenCV
from google.colab.patches import cv2_imshow
# Importing cv2_imshow from google.patches to display images

# Ignore warnings
import warnings
warnings.filterwarnings('ignore')
```

Loading the dataset

```
# Uncomment and run the below code if you are using google colab
# from google.colab import drive
# drive.mount('/content/drive')

# Load in Images for plants
images = np.load('/content/images.npy')

# Load in labels file
labels = pd.read_csv('/content/Labels.csv')
```

Data Overview

Understand the shape of the dataset

```
print(images.shape)
print(labels.shape)

(4750, 128, 128, 3)
(4750, 1)
```

There are 4,750 RGB images of shape 128 x 128 X 3, each image having 3 channels.

Exploratory Data Analysis

- EDA is an important part of any project involving data.
- It is important to investigate and understand the data better before building a model with it.
- A few questions have been mentioned below which will help you understand the data better.
- A thorough analysis of the data, in addition to the questions mentioned below, should be done.

1. How are these different category plant images different from each other?

2. Is the dataset provided an imbalance? (Check with using bar plots)

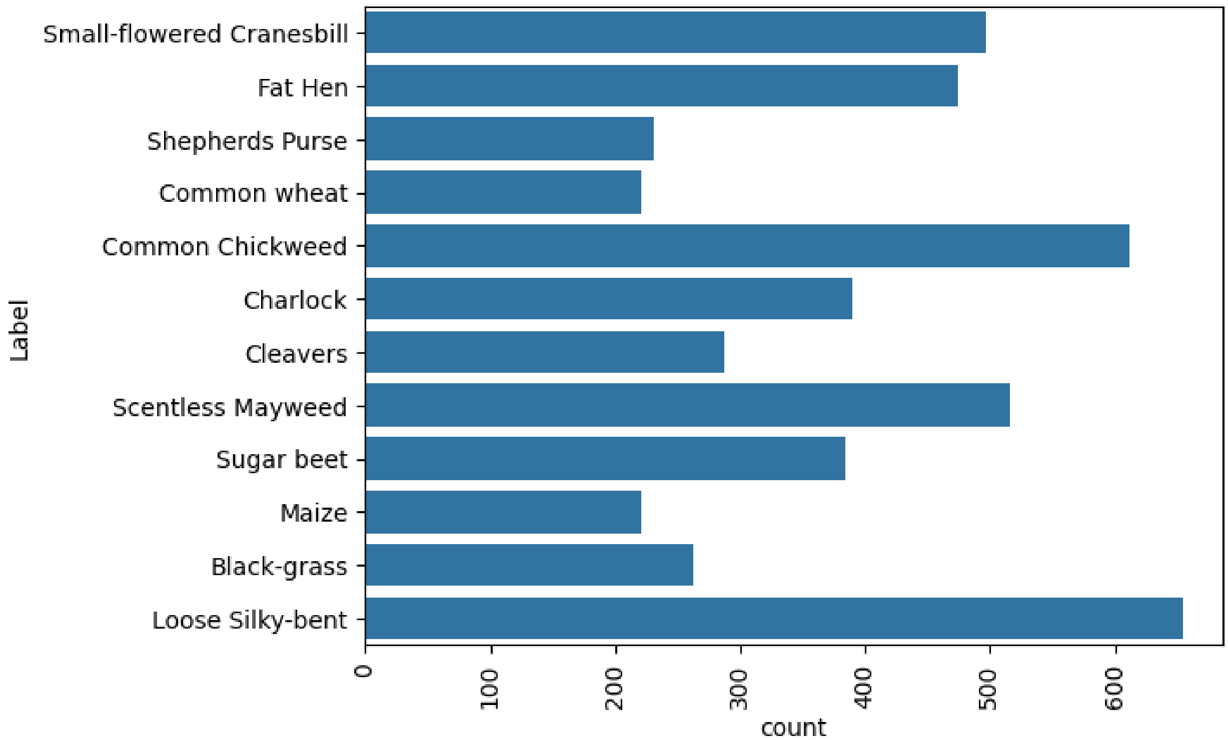
Function for EDA

```
def plot_images(images,labels):
  num_classes=10
# Number of Classes
  categories=np.unique(labels)
  keys=dict(labels['Label'])
# Obtaing the unique classes from y_train
  rows = 3
# Defining number of rows=3
  cols = 4
# Defining number of columns=4
  fig = plt.figure(figsize=(10, 8))
# Defining the figure size to 10x8
  for i in range(cols):
      for j in range(rows):
          random_index = np.random.randint(0, len(labels))
# Generating random indices from the data and plotting the images
          ax = fig.add_subplot(rows, cols, i * rows + j + 1)
# Adding subplots with 3 rows and 4 columns
          ax.imshow(images[random_index, :])
# Plotting the image
          ax.set_title(keys[random_index])
  plt.show()

labels.value_counts()
```

```
Label
Loose Silky-bent             654
Common Chickweed             611
Scentless Mayweed            516
Small-flowered Cranesbill    496
Fat Hen                      475
Charlock                     390
Sugar beet                   385
Cleavers                     287
Black-grass                  263
Shepherds Purse              231
Maize                        221
Common wheat                 221
Name: count, dtype: int64
```

- Class imbalance occurs from Cleavers to common wheat
- All int64 dtypes

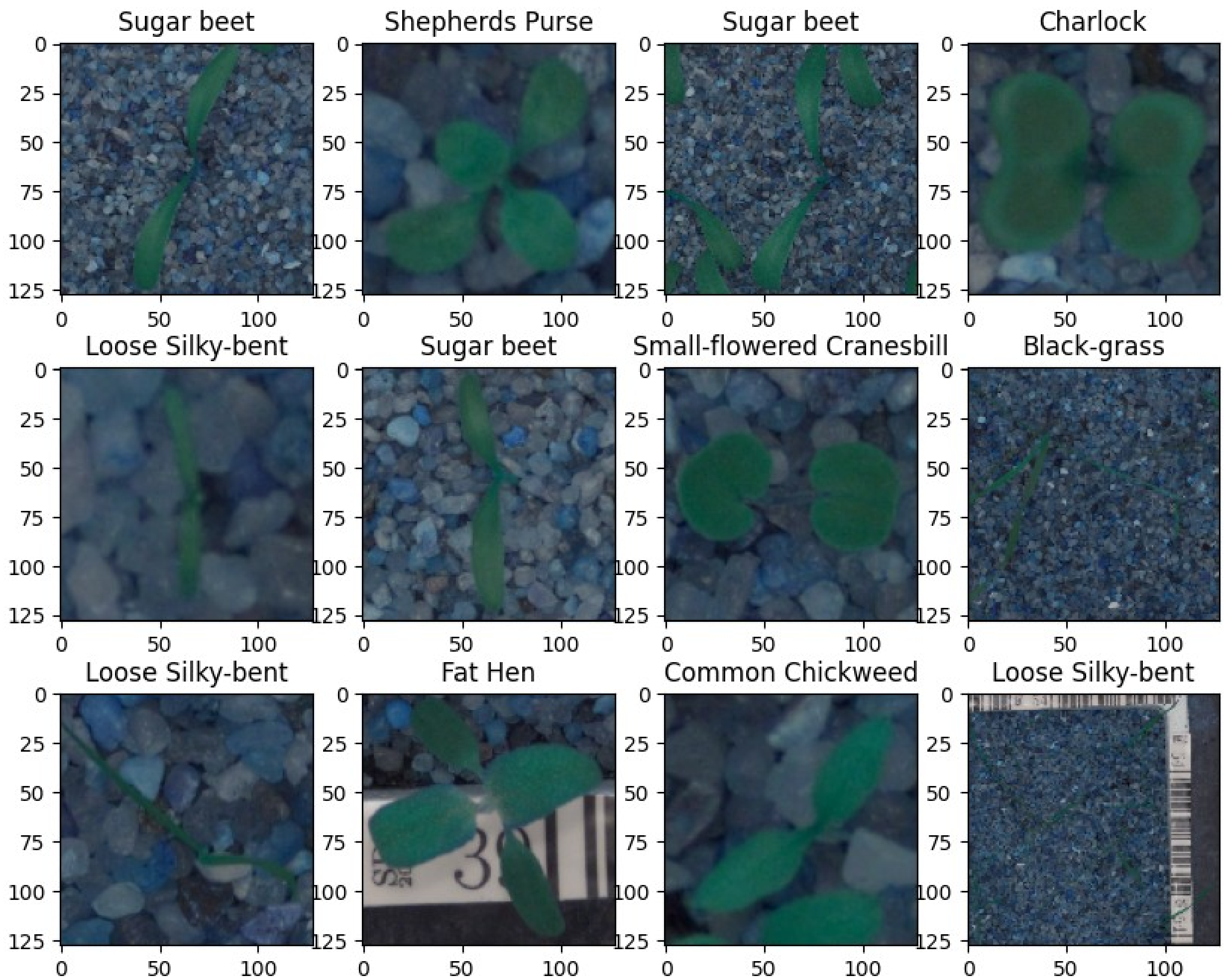
```
sns.countplot(labels['Label'])
plt.xticks(rotation=90)
plt.show()
```



Insight

- Loose Silky-bent has the highest count with 654, followed by chickweed and Scentless Mayweed
- Lowest count for common Wheat and Maize
- Will need to address class imblanaces with data augmentation

```
# Call EDA function plot images
plot_images(images,labels)
```



Data Pre-Processing

Convert the BGR images to RGB images.

```
# Converting the images from BGR to RGB using cvtColor function of
OpenCV
for i in range(len(images)):
  images[i] = cv2.cvtColor(images[i], cv2.COLOR_BGR2RGB)
```

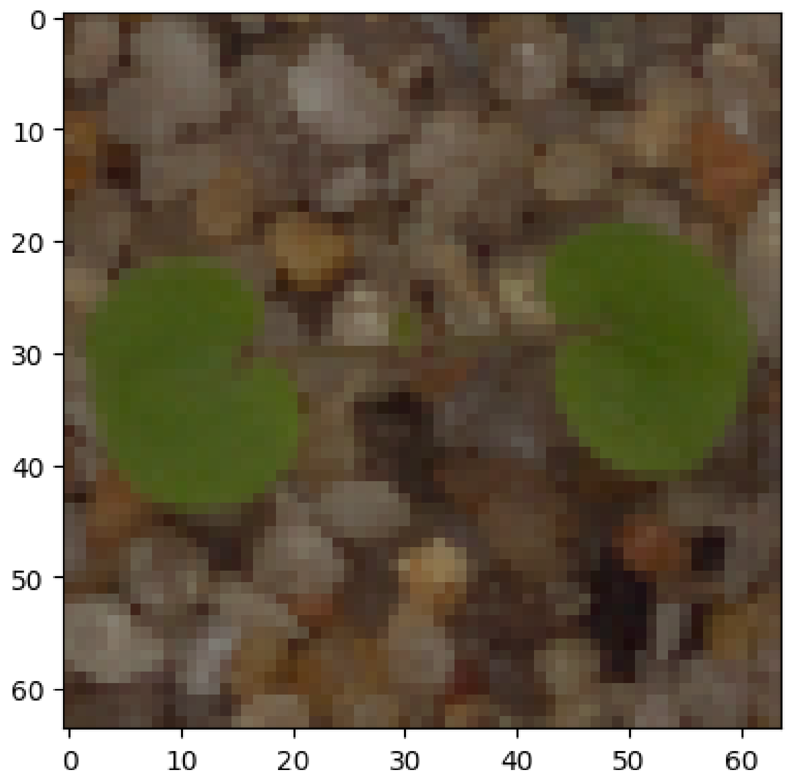
Resize the images

As the size of the images is large, it may be computationally expensive to train on these larger images; therefore, it is preferable to reduce the image size from 128 to 64.

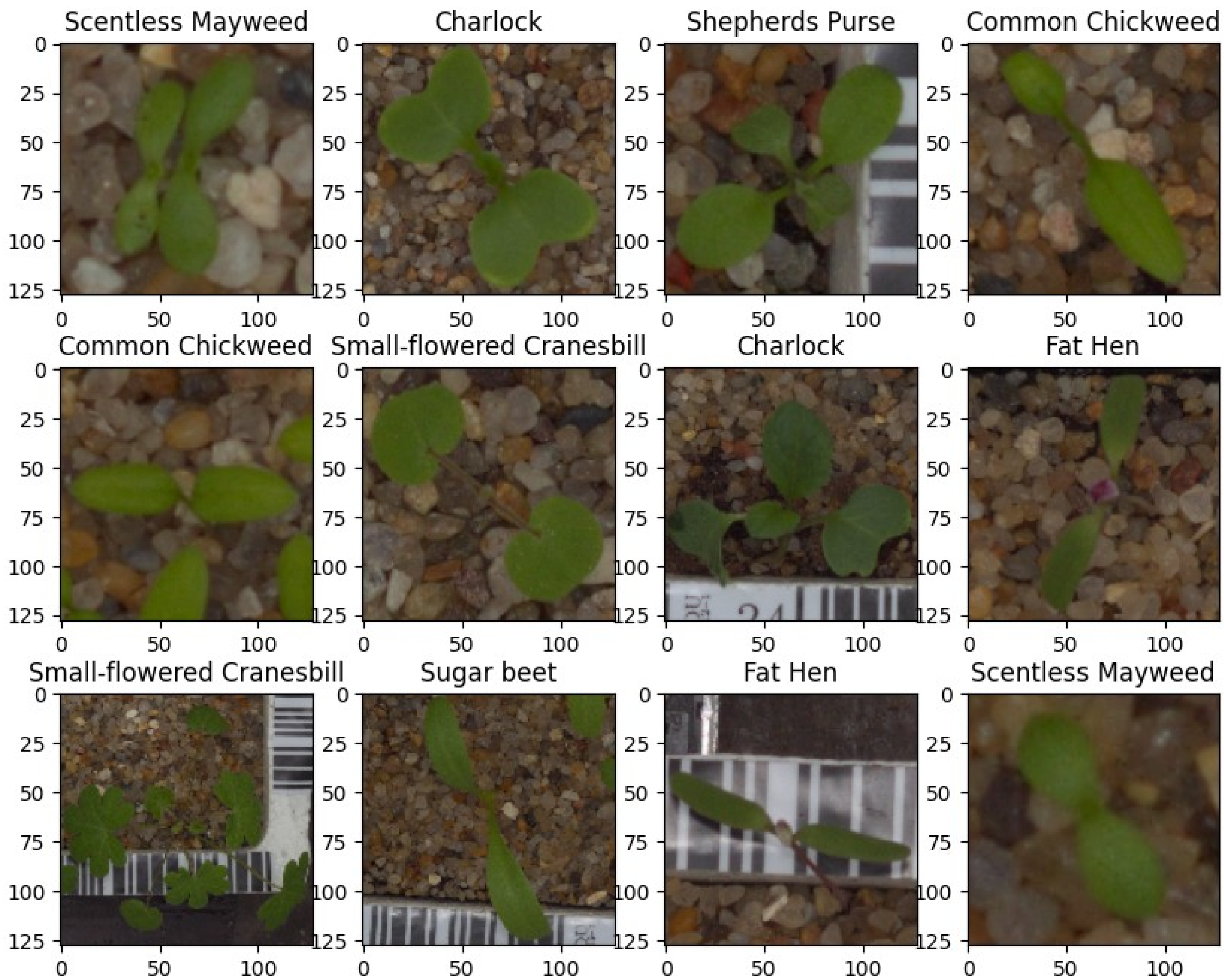
```
# Resize images to 64 from 128
images_decreased=[]
height = 64
width = 64
dimensions = (width, height)
```

```
for i in range(len(images)):
  images_decreased.append( cv2.resize(images[i], dimensions,
interpolation=cv2.INTER_LINEAR))

# Show images after changes to check quality
plt.imshow(images_decreased[0])
```

```
<matplotlib.image.AxesImage at 0x796f7b4952a0>
```



```
# Call EDA function plot images
plot_images(images,labels)
```



Data Preparation for Modeling

- Before you proceed to build a model, you need to split the data into train, test, and validation to be able to evaluate the model that you build on the train data
- You'll have to encode categorical features and scale the pixel values.
- You will build a model using the train data and then check its performance

Split the dataset

```
from sklearn.model_selection import train_test_split
X_temp, X_test, y_temp, y_test =
train_test_split(np.array(images_decreased),labels , test_size=0.1,
random_state=42,stratify=labels)
X_train, X_val, y_train, y_val = train_test_split(X_temp,y_temp ,
test_size=0.1, random_state=42,stratify=y_temp)

# Check Shape after split
print(X_train.shape,y_train.shape)
```

```
print(X_val.shape,y_val.shape)
print(X_test.shape,y_test.shape)
```

```
(3847, 64, 64, 3) (3847, 1)
(428, 64, 64, 3) (428, 1)
(475, 64, 64, 3) (475, 1)
```

Encode the target labels

```
# Convert labels from names to one hot vectors.
# We have already used encoding methods like onehotencoder and labelencoder earlier so now we will be using a new encoding method called labelBinarizer.
# Labelbinarizer works similar to onehotencoder

from sklearn.preprocessing import LabelBinarizer
enc = LabelBinarizer()
y_train_encoded = enc.fit_transform(y_train)
y_val_encoded=enc.transform(y_val)
y_test_encoded=enc.transform(y_test)
```

Data Normalization

```
# Normalizing the image pixels
X_train_normalized = X_train.astype('float32')/255.0
X_val_normalized = X_val.astype('float32')/255.0
X_test_normalized = X_test.astype('float32')/255.0
```

Model Building

We will build various models and compare them. Since we have a class imblance one model will use Data Augmetation method.

```
# Clearing backend
from tensorflow.keras import backend
backend.clear_session()

# Fixing the seed for random number generators
import random
np.random.seed(42)
random.seed(42)
tf.random.set_seed(42)
```

CNN Model - Model 1

- The Feature Extraction layers which are comprised of convolutional and pooling layers.

- The Fully Connected classification layers for prediction.

```
# Intializing a sequential model
model = Sequential()

# Adding first conv layer with 64 filters and kernel size 3x3 ,
padding 'same' provides the output size same as the input size
# Input_shape denotes input image dimension of images
model.add(Conv2D(64, (3, 3), activation='relu', padding="same",
input_shape=(64, 64, 3)))

# Adding max pooling to reduce the size of output of first conv layer
model.add(MaxPooling2D((2, 2), padding = 'same'))

model.add(Conv2D(32, (3, 3), activation='relu', padding="same"))
model.add(MaxPooling2D((2, 2), padding = 'same'))

# flattening the output of the conv layer after max pooling to make it
ready for creating dense connections
model.add(Flatten())

# Adding a fully connected dense layer with 100 neurons
model.add(Dense(16, activation='relu'))
model.add(Dropout(0.3))
# Adding the output layer with 12 neurons and activation functions as
softmax since this is a multi-class classification problem
model.add(Dense(12, activation='softmax'))

# Using SGD Optimizer
# opt = SGD(learning_rate=0.01, momentum=0.9)
opt=Adam()
# Compile model
model.compile(optimizer=opt, loss='categorical_crossentropy',
metrics=['accuracy'])

# Generating the summary of the model
model.summary()
```

```
Model: "sequential"

┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳
━━━━━━━━━━━━━━━━━┓
┃ Layer (type)                          ┃ Output Shape               ┃
Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇
━━━━━━━━━━━━━━━━━┩
│ conv2d (Conv2D)                       │ (None, 64, 64, 64)         │
1,792 │
├───────────────────────────────────────┼────────────────────────────┼
─────────────────┤
```

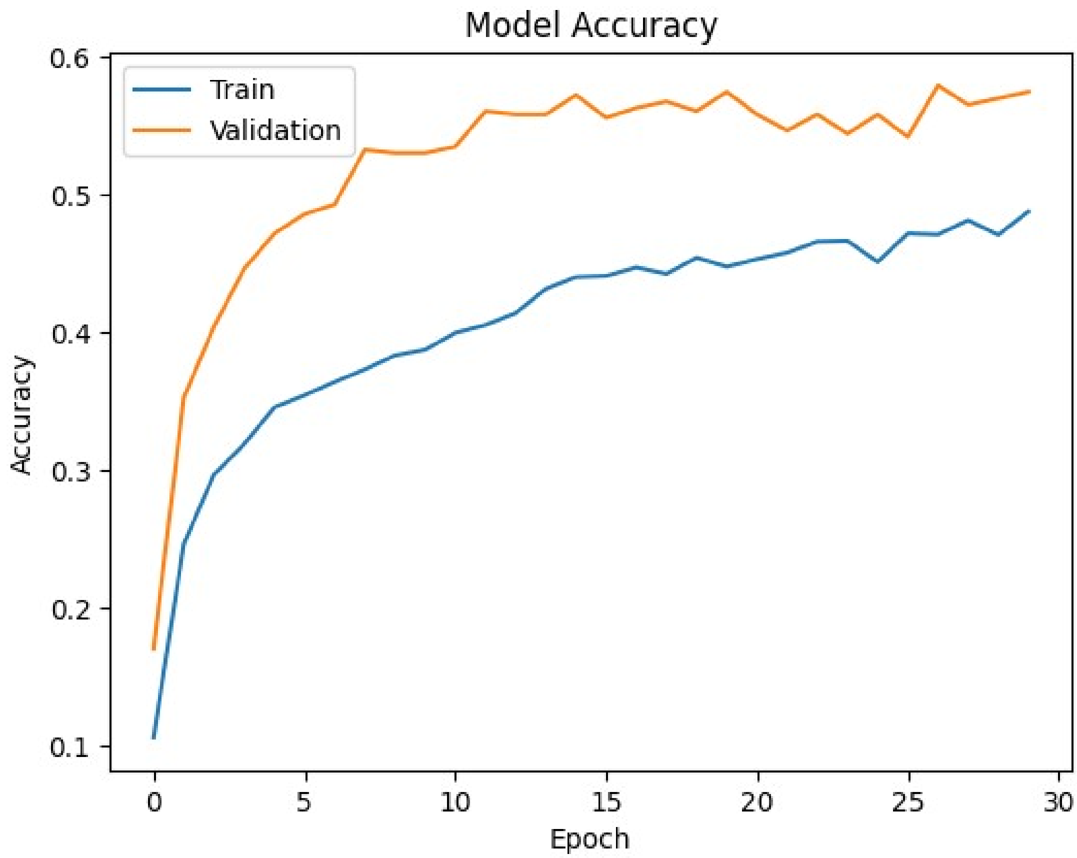
```
│ max_pooling2d (MaxPooling2D)         │ (None, 32, 32, 64)          │
0 │
├──────────────────────────────────────┼─────────────────────────────┤
─────────────────┤
│ conv2d_1 (Conv2D)                    │ (None, 32, 32, 32)          │
18,464 │
├──────────────────────────────────────┼─────────────────────────────┤
─────────────────┤
│ max_pooling2d_1 (MaxPooling2D)       │ (None, 16, 16, 32)          │
0 │
├──────────────────────────────────────┼─────────────────────────────┤
─────────────────┤
│ flatten (Flatten)                    │ (None, 8192)                │
0 │
├──────────────────────────────────────┼─────────────────────────────┤
─────────────────┤
│ dense (Dense)                        │ (None, 16)                  │
131,088 │
├──────────────────────────────────────┼─────────────────────────────┤
─────────────────┤
│ dropout (Dropout)                    │ (None, 16)                  │
0 │
├──────────────────────────────────────┼─────────────────────────────┤
─────────────────┤
│ dense_1 (Dense)                      │ (None, 12)                  │
204 │
└──────────────────────────────────────┴─────────────────────────────┴
─────────────────┘

 Total params: 151,548 (591.98 KB)

 Trainable params: 151,548 (591.98 KB)

 Non-trainable params: 0 (0.00 B)

# Fit the model on training data
history_1 = model.fit(
            X_train_normalized, y_train_encoded,
            epochs=30,
            validation_data=(X_val_normalized,y_val_encoded),
            batch_size=32,
            verbose=2
)

Epoch 1/30
121/121 - 12s - 98ms/step - accuracy: 0.1061 - loss: 2.4619 -
val_accuracy: 0.1706 - val_loss: 2.4107
Epoch 2/30
121/121 - 1s - 11ms/step - accuracy: 0.2464 - loss: 2.2628 -
val_accuracy: 0.3528 - val_loss: 2.0694
```

```
Epoch 3/30
121/121 - 1s - 12ms/step - accuracy: 0.2969 - loss: 2.0144 -
val_accuracy: 0.4042 - val_loss: 1.8244
Epoch 4/30
121/121 - 1s - 12ms/step - accuracy: 0.3189 - loss: 1.8949 -
val_accuracy: 0.4463 - val_loss: 1.6612
Epoch 5/30
121/121 - 1s - 7ms/step - accuracy: 0.3457 - loss: 1.8019 -
val_accuracy: 0.4720 - val_loss: 1.6009
Epoch 6/30
121/121 - 1s - 9ms/step - accuracy: 0.3546 - loss: 1.7384 -
val_accuracy: 0.4860 - val_loss: 1.5016
Epoch 7/30
121/121 - 1s - 11ms/step - accuracy: 0.3642 - loss: 1.7140 -
val_accuracy: 0.4930 - val_loss: 1.5305
Epoch 8/30
121/121 - 1s - 10ms/step - accuracy: 0.3733 - loss: 1.6633 -
val_accuracy: 0.5327 - val_loss: 1.4118
Epoch 9/30
121/121 - 1s - 9ms/step - accuracy: 0.3834 - loss: 1.6371 -
val_accuracy: 0.5304 - val_loss: 1.3958
Epoch 10/30
121/121 - 1s - 9ms/step - accuracy: 0.3876 - loss: 1.6144 -
val_accuracy: 0.5304 - val_loss: 1.3965
Epoch 11/30
121/121 - 1s - 7ms/step - accuracy: 0.3998 - loss: 1.5706 -
val_accuracy: 0.5350 - val_loss: 1.3471
Epoch 12/30
121/121 - 1s - 7ms/step - accuracy: 0.4055 - loss: 1.5434 -
val_accuracy: 0.5607 - val_loss: 1.3094
Epoch 13/30
121/121 - 2s - 16ms/step - accuracy: 0.4141 - loss: 1.5488 -
val_accuracy: 0.5584 - val_loss: 1.2965
Epoch 14/30
121/121 - 3s - 24ms/step - accuracy: 0.4318 - loss: 1.5232 -
val_accuracy: 0.5584 - val_loss: 1.2902
Epoch 15/30
121/121 - 1s - 12ms/step - accuracy: 0.4403 - loss: 1.4929 -
val_accuracy: 0.5724 - val_loss: 1.2290
Epoch 16/30
121/121 - 1s - 10ms/step - accuracy: 0.4411 - loss: 1.4846 -
val_accuracy: 0.5561 - val_loss: 1.2905
Epoch 17/30
121/121 - 1s - 5ms/step - accuracy: 0.4474 - loss: 1.4621 -
val_accuracy: 0.5631 - val_loss: 1.2736
Epoch 18/30
121/121 - 1s - 5ms/step - accuracy: 0.4427 - loss: 1.4557 -
val_accuracy: 0.5678 - val_loss: 1.2564
Epoch 19/30
```

```
121/121 - 1s - 5ms/step - accuracy: 0.4544 - loss: 1.4377 -
val_accuracy: 0.5607 - val_loss: 1.2426
Epoch 20/30
121/121 - 1s - 5ms/step - accuracy: 0.4479 - loss: 1.4453 -
val_accuracy: 0.5748 - val_loss: 1.2598
Epoch 21/30
121/121 - 1s - 5ms/step - accuracy: 0.4533 - loss: 1.4100 -
val_accuracy: 0.5584 - val_loss: 1.2353
Epoch 22/30
121/121 - 1s - 10ms/step - accuracy: 0.4580 - loss: 1.3969 -
val_accuracy: 0.5467 - val_loss: 1.2749
Epoch 23/30
121/121 - 1s - 5ms/step - accuracy: 0.4661 - loss: 1.3832 -
val_accuracy: 0.5584 - val_loss: 1.2434
Epoch 24/30
121/121 - 1s - 5ms/step - accuracy: 0.4666 - loss: 1.3703 -
val_accuracy: 0.5444 - val_loss: 1.2733
Epoch 25/30
121/121 - 1s - 5ms/step - accuracy: 0.4513 - loss: 1.3887 -
val_accuracy: 0.5584 - val_loss: 1.2450
Epoch 26/30
121/121 - 1s - 5ms/step - accuracy: 0.4721 - loss: 1.3638 -
val_accuracy: 0.5421 - val_loss: 1.2824
Epoch 27/30
121/121 - 1s - 6ms/step - accuracy: 0.4713 - loss: 1.3324 -
val_accuracy: 0.5794 - val_loss: 1.2730
Epoch 28/30
121/121 - 1s - 10ms/step - accuracy: 0.4814 - loss: 1.3376 -
val_accuracy: 0.5654 - val_loss: 1.2453
Epoch 29/30
121/121 - 1s - 10ms/step - accuracy: 0.4710 - loss: 1.3394 -
val_accuracy: 0.5701 - val_loss: 1.2569
Epoch 30/30
121/121 - 1s - 5ms/step - accuracy: 0.4879 - loss: 1.3096 -
val_accuracy: 0.5748 - val_loss: 1.1931
```

Model Evaluation

```
plt.plot(history_1.history['accuracy'])
plt.plot(history_1.history['val_accuracy'])
plt.title('Model Accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='upper left')
plt.show()
```



Insight

- The model's performance seems to have improved over time, but it also shows signs of overfitting.
- Class imbalance can create a bias and lead to overfitting.

Evaluating the model on the test data

```
accuracy = model.evaluate(X_test_normalized, y_test_encoded,
verbose=2)

15/15 - 0s - 21ms/step - accuracy: 0.5853 - loss: 1.2155
```

Insight

- Accuracy is relatively low
- loss is moderate indicates an error between the prediction and actual values.

Generating the predictions using test data

```
# Here we would get the output as probablities for each category
y_pred=model.predict(X_test_normalized)
```

```
15/15 ━━━━━━━━━━━━━━━━━━━━ 0s 13ms/step
```

```
y_pred
```

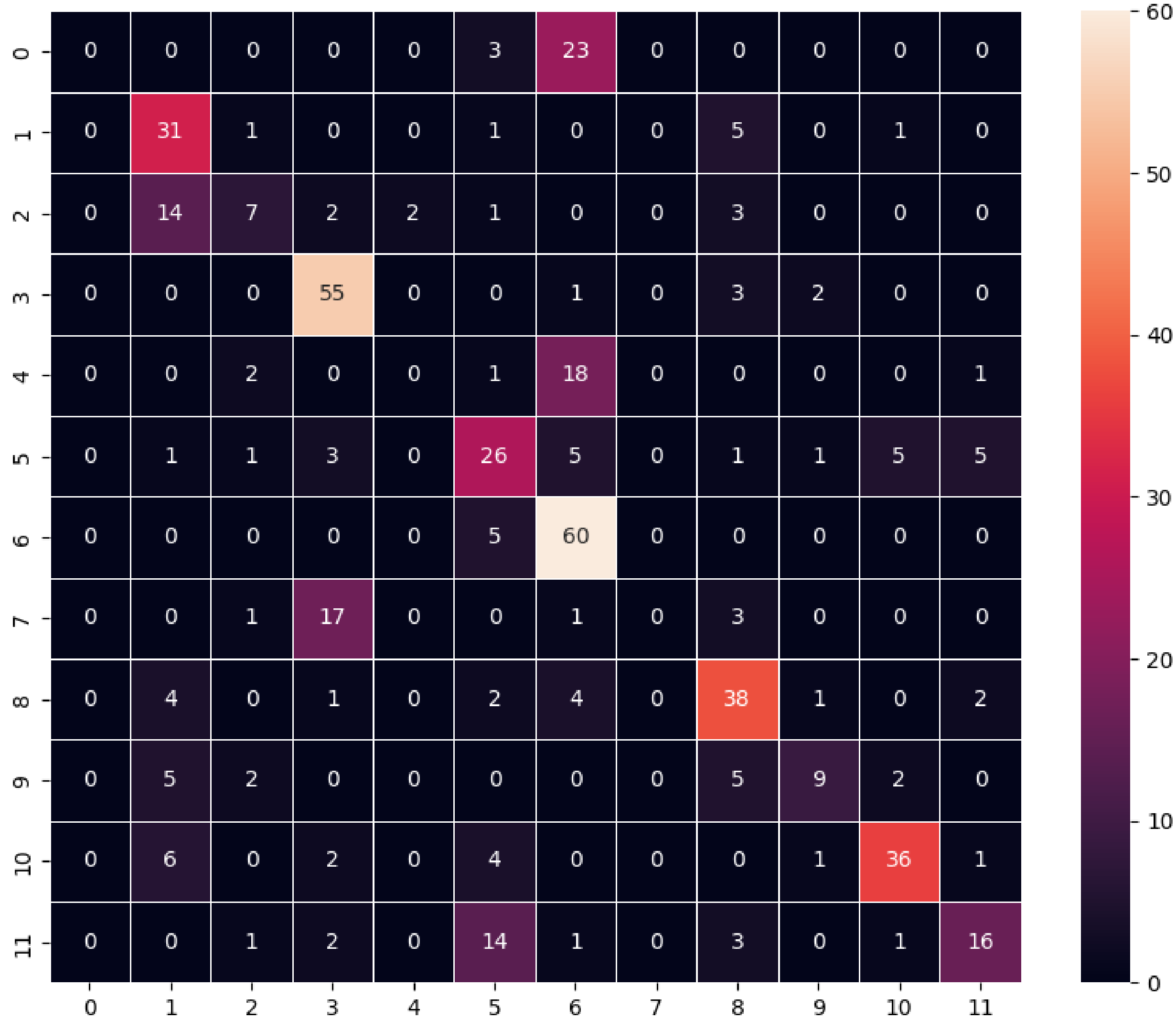
```
array([[1.07087409e-07, 4.32500002e-07, 5.03546049e-09, ...,
        2.47111544e-02, 1.09773588e-07, 1.56598017e-02],
       [1.34122292e-12, 6.51235580e-02, 2.73913350e-02, ...,
        1.23621814e-01, 6.93322957e-01, 2.30864380e-02],
       [1.29904143e-10, 3.93184386e-02, 2.26452537e-02, ...,
        7.70480558e-02, 6.77627563e-01, 3.43532860e-02],
       ...,
       [2.13238493e-01, 2.37728837e-10, 3.92930320e-04, ...,
        4.96607588e-10, 3.59613228e-10, 2.96505041e-05],
       [1.32720984e-07, 2.19241250e-03, 3.41512641e-04, ...,
        7.53059387e-02, 4.79581877e-06, 7.72175007e-03],
       [6.06402883e-09, 4.96775031e-01, 2.01263517e-01, ...,
        2.08794907e-01, 5.97561263e-02, 1.10067604e-02]],
dtype=float32)
```

Plotting the Confusion Matrix

Reason for use:

- A confusion matrix is a powerful tool for evaluating the performance of a classification model. It provides a detailed breakdown of how well the model is performing across different classes.

```
# Obtaining the categorical values from y_test_encoded and y_pred
y_pred_arg=np.argmax(y_pred,axis=1)
y_test_arg=np.argmax(y_test_encoded,axis=1)

# Plotting the Confusion Matrix using confusion matrix() function
which is also predefined tensorflow module
confusion_matrix = tf.math.confusion_matrix(y_test_arg,y_pred_arg)
f, ax = plt.subplots(figsize=(10, 8))
sns.heatmap(
    confusion_matrix,
    annot=True,
    linewidths=.4,
    fmt="d",
    square=True,
    ax=ax
)
plt.show()
```



Insight

- Classes 3, 5, and 8 seem to be classified quite well
- Classes 1, 2, and 10 appear to have more misclassifications

Further action

As we can see, our initial model appears to overfit. Therefore we'll try to address this problem with data augmentation.

Model Performance Improvement

CNN with Data Augmentation Model - Model 2

Reason for use:

- Used to artificially increase the size and diversity of a dataset by applying various transformations to existing data points. It's a powerful tool for improving the performance of machine learning models, especially in scenarios with limited data.

Reducing the Learning Rate:

Hint: Use **ReduceLRonPlateau()** function that will be used to decrease the learning rate by some factor, if the loss is not decreasing for some time. This may start decreasing the loss at a smaller learning rate. There is a possibility that the loss may still not decrease. This may lead to executing the learning rate reduction again in an attempt to achieve a lower loss.

Data Augmentation

Remember, **data augmentation should not be used in the validation/test data set**.

```
# Clearing backend
from tensorflow.keras import backend
backend.clear_session()

# Fixing the seed for random number generators
import random
np.random.seed(42)
random.seed(42)
tf.random.set_seed(42)

# All images to be rescaled by 1/255.
train_datagen = ImageDataGenerator(
                              rotation_range=20,
                              fill_mode='nearest'
                              )
test_datagen  = ImageDataGenerator(rescale = 1.0/255.)

# Intializing a sequential model
model = Sequential()

# Adding first conv layer with 64 filters and kernel size 3x3 ,
padding 'same' provides the output size same as the input size
# Input_shape denotes input image dimension images
model.add(Conv2D(64, (3, 3), activation='relu', padding="same",
input_shape=(64, 64, 3)))

# Adding max pooling to reduce the size of output of first conv layer
model.add(MaxPooling2D((2, 2), padding = 'same'))
# model.add(BatchNormalization())
```

```
model.add(Conv2D(32, (3, 3), activation='relu', padding="same"))
model.add(MaxPooling2D((2, 2), padding = 'same'))
model.add(BatchNormalization())
# flattening the output of the conv layer after max pooling to make it
ready for creating dense connections
model.add(Flatten())

# Adding a fully connected dense layer with 100 neurons
model.add(Dense(16, activation='relu'))
model.add(Dropout(0.3))
# Adding the output layer with 12 neurons and activation functions as
softmax since this is a multi-class classification problem
model.add(Dense(12, activation='softmax'))

# Using SGD Optimizer
# opt = SGD(learning_rate=0.01, momentum=0.9)
opt=Adam()
# Compile model
model.compile(optimizer=opt, loss='categorical_crossentropy',
metrics=['accuracy'])

# Generating the summary of the model
model.summary()
```

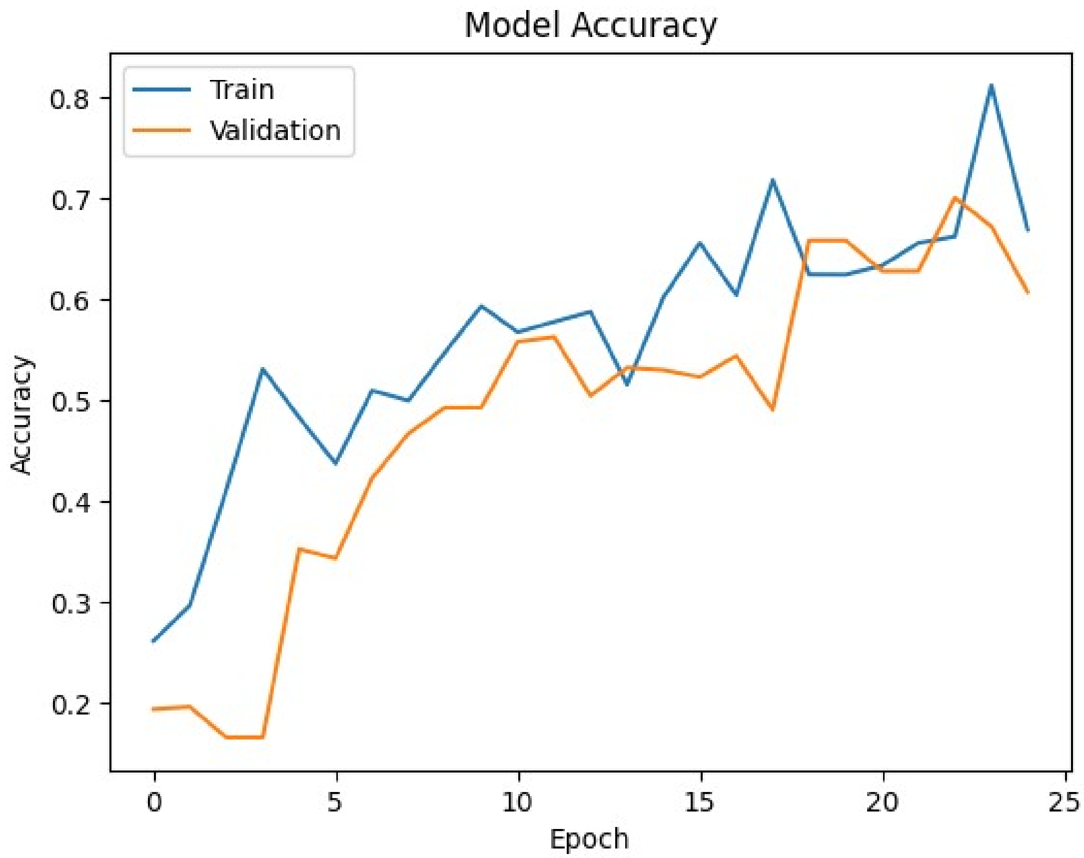
```
Model: "sequential"

┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━┓
┃ Layer (type)                         ┃ Output Shape                ┃         Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━┩
│ conv2d (Conv2D)                      │ (None, 64, 64, 64)          │           1,792 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ max_pooling2d (MaxPooling2D)         │ (None, 32, 32, 64)          │               0 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ conv2d_1 (Conv2D)                    │ (None, 32, 32, 32)          │          18,464 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ max_pooling2d_1 (MaxPooling2D)       │ (None, 16, 16, 32)          │               0 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ batch_normalization                  │ (None, 16, 16, 32)          │             128 │
```

```
│ (BatchNormalization)                    │                              │
│
├─────────────────────────────────────────┼──────────────────────────────┼
─────────────────┤
│ flatten (Flatten)                       │ (None, 8192)                 │
0 │
├─────────────────────────────────────────┼──────────────────────────────┼
─────────────────┤
│ dense (Dense)                           │ (None, 16)                   │
131,088 │
├─────────────────────────────────────────┼──────────────────────────────┼
─────────────────┤
│ dropout (Dropout)                       │ (None, 16)                   │
0 │
├─────────────────────────────────────────┼──────────────────────────────┼
─────────────────┤
│ dense_1 (Dense)                         │ (None, 12)                   │
204 │
└─────────────────────────────────────────┴──────────────────────────────┴
─────────────────┘

 Total params: 151,676 (592.48 KB)

 Trainable params: 151,612 (592.23 KB)

 Non-trainable params: 64 (256.00 B)

# Epochs
epochs = 25
# Batch size
batch_size = 64

history =
model.fit(train_datagen.flow(X_train_normalized,y_train_encoded,
                                       batch_size=batch_size,
                                       seed=42,
                                       shuffle=False),
                    epochs=epochs,
                    steps_per_epoch=X_train_normalized.shape[0] //
batch_size,
                    validation_data=(X_val_normalized,y_val_encoded),
                    verbose=1)

Epoch 1/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 11s 123ms/step - accuracy: 0.1748 - loss:
2.3630 - val_accuracy: 0.1939 - val_loss: 2.4077
Epoch 2/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.2969 - loss:
1.9539 - val_accuracy: 0.1963 - val_loss: 2.4155
Epoch 3/25
```

```
60/60 ━━━━━━━━━━━━━━━━━━━━ 6s 71ms/step - accuracy: 0.3999 - loss:
1.8059 - val_accuracy: 0.1659 - val_loss: 2.3553
Epoch 4/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 956us/step - accuracy: 0.5312 - loss:
1.5377 - val_accuracy: 0.1659 - val_loss: 2.3551
Epoch 5/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 6s 92ms/step - accuracy: 0.4760 - loss:
1.6079 - val_accuracy: 0.3528 - val_loss: 2.2425
Epoch 6/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 973us/step - accuracy: 0.4375 - loss:
1.7566 - val_accuracy: 0.3435 - val_loss: 2.2487
Epoch 7/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 5s 72ms/step - accuracy: 0.5001 - loss:
1.5149 - val_accuracy: 0.4229 - val_loss: 2.0922
Epoch 8/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.5000 - loss:
1.3775 - val_accuracy: 0.4673 - val_loss: 2.0932
Epoch 9/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 5s 71ms/step - accuracy: 0.5474 - loss:
1.3218 - val_accuracy: 0.4930 - val_loss: 1.9000
Epoch 10/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.5938 - loss:
1.3005 - val_accuracy: 0.4930 - val_loss: 1.8995
Epoch 11/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 6s 78ms/step - accuracy: 0.5620 - loss:
1.2685 - val_accuracy: 0.5584 - val_loss: 1.6318
Epoch 12/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.5781 - loss:
1.3014 - val_accuracy: 0.5631 - val_loss: 1.5794
Epoch 13/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 5s 70ms/step - accuracy: 0.5845 - loss:
1.2190 - val_accuracy: 0.5047 - val_loss: 1.4572
Epoch 14/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.5156 - loss:
1.4824 - val_accuracy: 0.5327 - val_loss: 1.4722
Epoch 15/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 6s 94ms/step - accuracy: 0.5941 - loss:
1.1558 - val_accuracy: 0.5304 - val_loss: 1.3672
Epoch 16/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.6562 - loss:
1.0981 - val_accuracy: 0.5234 - val_loss: 1.3814
Epoch 17/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 9s 71ms/step - accuracy: 0.6014 - loss:
1.0940 - val_accuracy: 0.5444 - val_loss: 1.3524
Epoch 18/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 992us/step - accuracy: 0.7188 - loss:
0.7883 - val_accuracy: 0.4907 - val_loss: 1.4266
Epoch 19/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 8s 111ms/step - accuracy: 0.6348 - loss:
```

```
1.0468 - val_accuracy: 0.6589 - val_loss: 1.1378
Epoch 20/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.6250 - loss:
1.0028 - val_accuracy: 0.6589 - val_loss: 1.1407
Epoch 21/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 9s 105ms/step - accuracy: 0.6350 - loss:
1.0045 - val_accuracy: 0.6285 - val_loss: 1.1081
Epoch 22/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.6562 - loss:
0.8854 - val_accuracy: 0.6285 - val_loss: 1.1521
Epoch 23/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 5s 67ms/step - accuracy: 0.6724 - loss:
0.9358 - val_accuracy: 0.7009 - val_loss: 0.8952
Epoch 24/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8125 - loss:
0.7255 - val_accuracy: 0.6729 - val_loss: 0.9110
Epoch 25/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 5s 69ms/step - accuracy: 0.6667 - loss:
0.9166 - val_accuracy: 0.6075 - val_loss: 1.2436
```

```
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('Model Accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='upper left')
plt.show()
```




```
accuracy = model.evaluate(X_test_normalized, y_test_encoded,
verbose=2)
```

```
15/15 - 1s - 35ms/step - accuracy: 0.6042 - loss: 1.2251
```

Insight

- accuracy plateaus and even starts to decline after around epoch 15
- The model is becoming too specialized in the training data and is not generalizing well to unseen data.

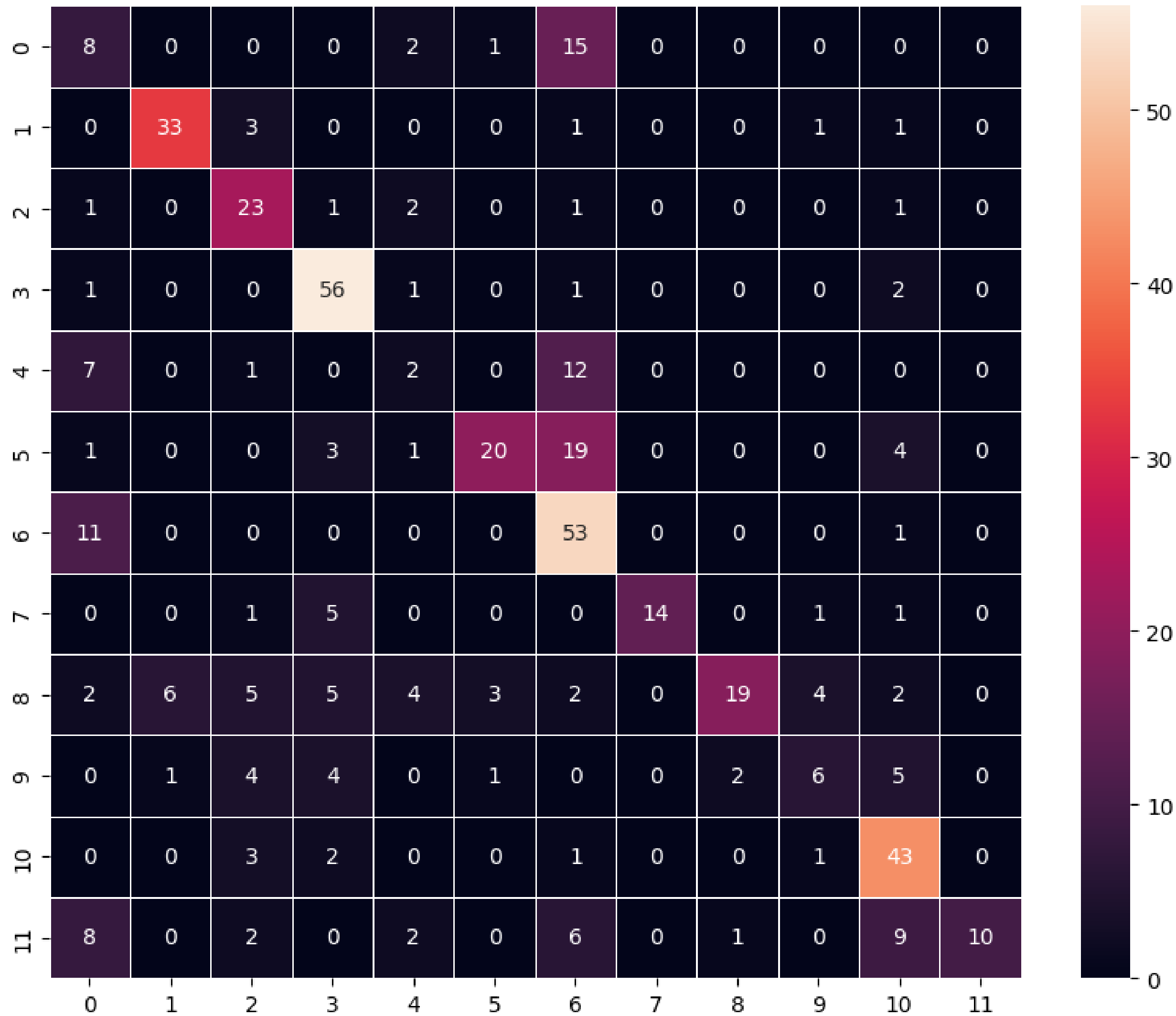
```
# Here we would get the output as probablities for each category
y_pred=model.predict(X_test_normalized)
```

```
15/15 ━━━━━━━━━━━━━━━━━━━━ 1s 22ms/step
```

```
y_pred
```

```
array([[1.04253047e-06, 1.00008594e-06, 1.89458713e-12, ...,
        6.45659748e-05, 2.23802665e-08, 2.27571419e-07],
       [1.84510282e-05, 4.25735041e-02, 8.90863314e-02, ...,
        1.43439084e-01, 6.73182726e-01, 2.54156697e-03],
```

```
       [2.37951226e-05, 9.76844691e-04, 5.53146092e-05, ...,
        6.83593936e-03, 9.82581556e-01, 6.22221996e-05],
       ...,
       [3.88498813e-01, 6.16831812e-07, 7.64684938e-09, ...,
        8.38404162e-07, 9.35458957e-07, 4.16157627e-06],
       [1.22530796e-02, 3.78918611e-02, 2.56179661e-01, ...,
        4.93127592e-02, 1.16993040e-02, 7.94723406e-02],
       [1.40206376e-02, 2.92008054e-02, 1.24283694e-01, ...,
        1.64484799e-01, 3.28473836e-01, 5.50264865e-03]],
dtype=float32)

# Obtaining the categorical values from y_test_encoded and y_pred
y_pred_arg=np.argmax(y_pred,axis=1)
y_test_arg=np.argmax(y_test_encoded,axis=1)

# Plotting the Confusion Matrix using confusion matrix() function
which is also predefined tensorflow module
confusion_matrix = tf.math.confusion_matrix(y_test_arg,y_pred_arg)
f, ax = plt.subplots(figsize=(10, 8))
sns.heatmap(
    confusion_matrix,
    annot=True,
    linewidths=.4,
    fmt="d",
    square=True,
    ax=ax
)
plt.show()
```



Insight

- 3, 5, 6, and 7, exhibit exceptionally high accuracy, with minimal misclassifications.
- 1, 2, and 10, show a higher frequency of misclassifications.

Model 1 and Model 2 Conclusion

- Based on the metrics, Model 2 has a better result with accuracy. It achieved an accuracy of 0.60, which is higher than Model 1's accuracy of 0.58

Model 3 - CNN Data Augmentation Learning Rate Model 3

How to improve it:

- Adjust learning rate: Lower or higher learning rates can impact convergence.

```
# Clearing backend
from tensorflow.keras import backend
backend.clear_session()

# Fixing the seed for random number generators
import random
np.random.seed(42)
random.seed(42)
tf.random.set_seed(42)

# All images to be rescaled by 1/255.
train_datagen = ImageDataGenerator(
                              rotation_range=20,
                              fill_mode='nearest'
                              )
test_datagen  = ImageDataGenerator(rescale = 1.0/255.)

import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten,
Dense, Dropout
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import EarlyStopping,
LearningRateScheduler

# Intializing a sequential model
model = Sequential()

# Adding first conv layer with 64 filters and kernel size 3x3 ,
padding 'same' provides the output size same as the input size
# Input_shape denotes input image dimension images
model.add(Conv2D(64, (3, 3), activation='relu', padding="same",
input_shape=(64, 64, 3)))

# Adding max pooling to reduce the size of output of first conv layer
model.add(MaxPooling2D((2, 2), padding = 'same'))
# model.add(BatchNormalization())
model.add(Conv2D(32, (3, 3), activation='relu', padding="same"))
model.add(MaxPooling2D((2, 2), padding = 'same'))
model.add(BatchNormalization())
# flattening the output of the conv layer after max pooling to make it
ready for creating dense connections
```

```python
model.add(Flatten())

# Adding a fully connected dense layer with 100 neurons
model.add(Dense(16, activation='relu'))
model.add(Dropout(0.3))
# Adding the output layer with 12 neurons and activation functions as
softmax since this is a multi-class classification problem
model.add(Dense(12, activation='softmax'))

# Using Adam Optimizer with learning rate scheduling
def lr_schedule(epoch):
  lr = 0.001 * 0.95 ** epoch
  return lr

optimizer = Adam(learning_rate=lr_schedule(0))

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

# Early stopping
early_stop = EarlyStopping(monitor='val_loss', patience=5,
restore_best_weights=True)

# Learning rate scheduler
lr_scheduler = LearningRateScheduler(lr_schedule)

# Generating the summary of the model
model.summary()
```

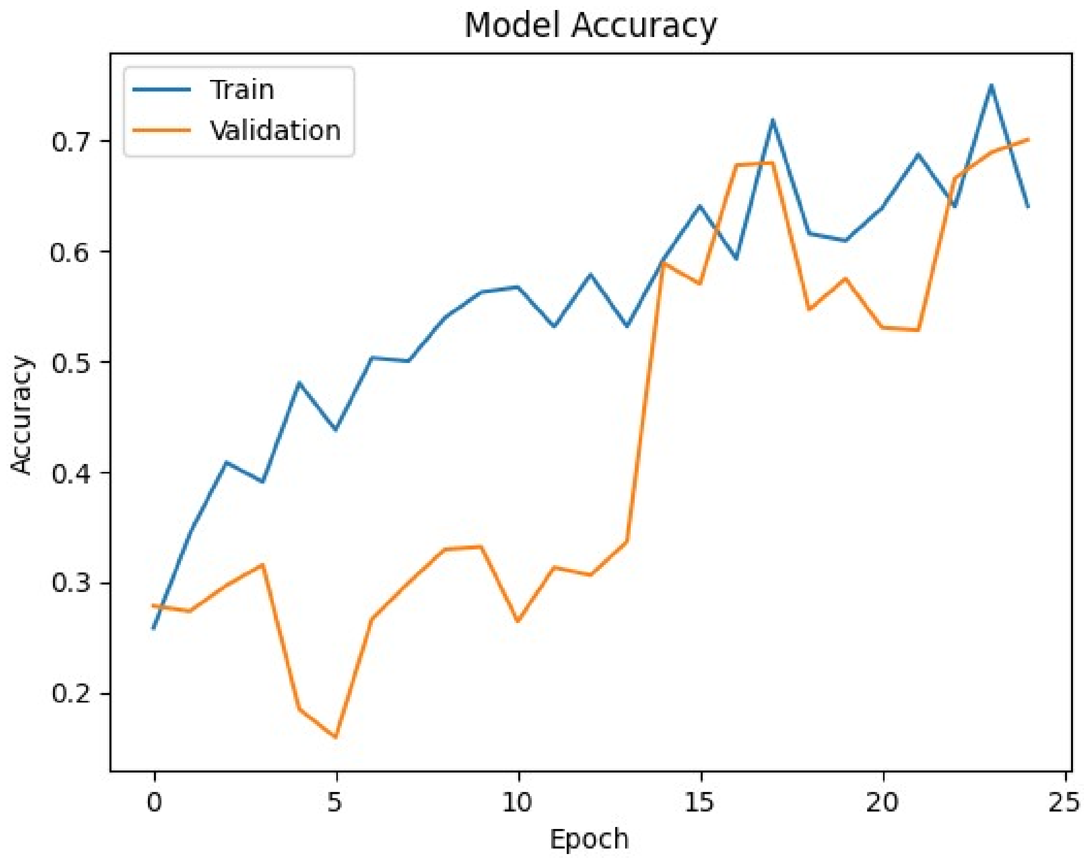
```
Model: "sequential"

┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳
━━━━━━━━━━━━━━━━━┓
┃ Layer (type)                          ┃ Output Shape                ┃
Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇
━━━━━━━━━━━━━━━━━┩
│ conv2d (Conv2D)                       │ (None, 64, 64, 64)          │
1,792 │
├───────────────────────────────────────┼─────────────────────────────┼
─────────────────┤
│ max_pooling2d (MaxPooling2D)          │ (None, 32, 32, 64)          │
0 │
├───────────────────────────────────────┼─────────────────────────────┼
─────────────────┤
│ conv2d_1 (Conv2D)                     │ (None, 32, 32, 32)          │
18,464 │
├───────────────────────────────────────┼─────────────────────────────┼
─────────────────┤
```

```
│ max_pooling2d_1 (MaxPooling2D)        │ (None, 16, 16, 32)          │
0 │
├───────────────────────────────────────┼─────────────────────────────┼
─────────────────┤
│ batch_normalization                   │ (None, 16, 16, 32)          │
128 │
│ (BatchNormalization)                  │                             │
│
├───────────────────────────────────────┼─────────────────────────────┼
─────────────────┤
│ flatten (Flatten)                     │ (None, 8192)                │
0 │
├───────────────────────────────────────┼─────────────────────────────┼
─────────────────┤
│ dense (Dense)                         │ (None, 16)                  │
131,088 │
├───────────────────────────────────────┼─────────────────────────────┼
─────────────────┤
│ dropout (Dropout)                     │ (None, 16)                  │
0 │
├───────────────────────────────────────┼─────────────────────────────┼
─────────────────┤
│ dense_1 (Dense)                       │ (None, 12)                  │
204 │
└───────────────────────────────────────┴─────────────────────────────┴
─────────────────┘

 Total params: 151,676 (592.48 KB)

 Trainable params: 151,612 (592.23 KB)

 Non-trainable params: 64 (256.00 B)

# Epochs
epochs = 25
# Batch size
batch_size = 64

history =
model.fit(train_datagen.flow(X_train_normalized,y_train_encoded,
                                       batch_size=batch_size,
                                       seed=42,
                                       shuffle=False),
                    epochs=epochs,
                    steps_per_epoch=X_train_normalized.shape[0] //
batch_size,
                    validation_data=(X_val_normalized,y_val_encoded),
                    verbose=1)
```

```
Epoch 1/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 10s 120ms/step - accuracy: 0.1755 - loss:
2.3582 - val_accuracy: 0.2780 - val_loss: 2.3887
Epoch 2/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.3438 - loss:
1.9618 - val_accuracy: 0.2734 - val_loss: 2.3943
Epoch 3/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 5s 69ms/step - accuracy: 0.3961 - loss:
1.8145 - val_accuracy: 0.2967 - val_loss: 2.3307
Epoch 4/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.3906 - loss:
1.5476 - val_accuracy: 0.3154 - val_loss: 2.3180
Epoch 5/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 5s 68ms/step - accuracy: 0.4693 - loss:
1.6201 - val_accuracy: 0.1846 - val_loss: 2.2624
Epoch 6/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 967us/step - accuracy: 0.4375 - loss:
1.7866 - val_accuracy: 0.1589 - val_loss: 2.2632
Epoch 7/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 6s 95ms/step - accuracy: 0.5024 - loss:
1.5258 - val_accuracy: 0.2664 - val_loss: 2.1215
Epoch 8/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.5000 - loss:
1.4375 - val_accuracy: 0.2991 - val_loss: 2.1046
Epoch 9/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 9s 74ms/step - accuracy: 0.5324 - loss:
1.3520 - val_accuracy: 0.3294 - val_loss: 1.9175
Epoch 10/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.5625 - loss:
1.3885 - val_accuracy: 0.3318 - val_loss: 1.9221
Epoch 11/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 5s 77ms/step - accuracy: 0.5694 - loss:
1.2966 - val_accuracy: 0.2640 - val_loss: 1.9284
Epoch 12/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.5312 - loss:
1.3053 - val_accuracy: 0.3131 - val_loss: 1.8081
Epoch 13/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 5s 69ms/step - accuracy: 0.5738 - loss:
1.2727 - val_accuracy: 0.3061 - val_loss: 1.7082
Epoch 14/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.5312 - loss:
1.4995 - val_accuracy: 0.3364 - val_loss: 1.6507
Epoch 15/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 8s 117ms/step - accuracy: 0.5957 - loss:
1.1764 - val_accuracy: 0.5888 - val_loss: 1.2755
Epoch 16/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 970us/step - accuracy: 0.6406 - loss:
1.1907 - val_accuracy: 0.5701 - val_loss: 1.3230
Epoch 17/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 5s 70ms/step - accuracy: 0.5857 - loss:
```

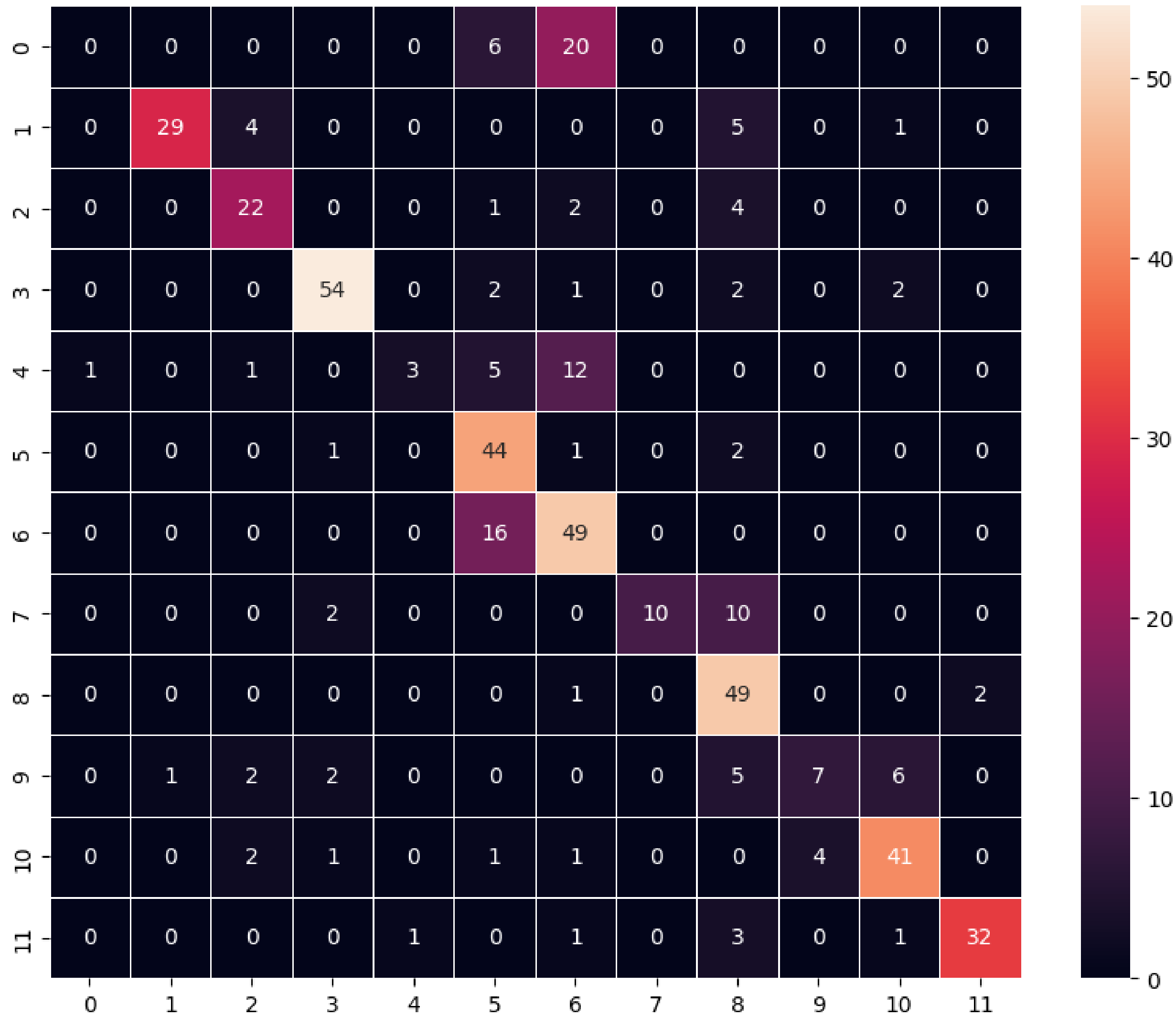
```
1.1483 - val_accuracy: 0.6776 - val_loss: 1.1073
Epoch 18/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 995us/step - accuracy: 0.7188 - loss:
0.8111 - val_accuracy: 0.6799 - val_loss: 1.1265
Epoch 19/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 6s 91ms/step - accuracy: 0.6282 - loss:
1.0950 - val_accuracy: 0.5467 - val_loss: 1.4638
Epoch 20/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.6094 - loss:
1.2294 - val_accuracy: 0.5748 - val_loss: 1.3174
Epoch 21/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 5s 68ms/step - accuracy: 0.6370 - loss:
1.0418 - val_accuracy: 0.5304 - val_loss: 1.5204
Epoch 22/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 977us/step - accuracy: 0.6875 - loss:
0.8498 - val_accuracy: 0.5280 - val_loss: 1.5492
Epoch 23/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 5s 72ms/step - accuracy: 0.6360 - loss:
1.0262 - val_accuracy: 0.6659 - val_loss: 0.9600
Epoch 24/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.7500 - loss:
0.8239 - val_accuracy: 0.6893 - val_loss: 0.8959
Epoch 25/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 6s 81ms/step - accuracy: 0.6408 - loss:
0.9688 - val_accuracy: 0.7009 - val_loss: 0.9536
```

```
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('Model Accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='upper left')
plt.show()
```




```
accuracy = model.evaluate(X_test_normalized, y_test_encoded,
verbose=2)

15/15 - 0s - 23ms/step - accuracy: 0.7158 - loss: 0.9187

# Here we would get the output as probablities for each category
y_pred=model.predict(X_test_normalized)

15/15 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step

y_pred

array([[2.07682410e-06, 1.19081105e-05, 1.67142119e-07, ...,
         3.83235654e-03, 1.99179908e-06, 1.12077410e-04],
       [7.19623358e-07, 1.29899293e-01, 1.02901600e-01, ...,
         8.98196623e-02, 6.69840395e-01, 2.05935142e-03],
       [1.66886693e-05, 2.10221251e-03, 1.18040561e-03, ...,
         5.97689413e-02, 8.42763662e-01, 7.15308299e-04],
       ...,
       [9.96675491e-02, 2.34500547e-07, 1.44479063e-05, ...,
         2.34288996e-06, 3.23602876e-06, 5.94713401e-06],
       [9.37740140e-07, 6.67120796e-04, 2.17379490e-03, ...,
         1.01750400e-02, 2.61887344e-05, 1.46889454e-03],
       [8.95417994e-04, 1.23590380e-02, 4.74839993e-02, ...,
```

```
        1.83344856e-01, 5.42227805e-01, 4.38706763e-03]],
dtype=float32)

# Obtaining the categorical values from y_test_encoded and y_pred
y_pred_arg=np.argmax(y_pred,axis=1)
y_test_arg=np.argmax(y_test_encoded,axis=1)

# Plotting the Confusion Matrix using confusion matrix() function
which is also predefined tensorflow module
confusion_matrix = tf.math.confusion_matrix(y_test_arg,y_pred_arg)
f, ax = plt.subplots(figsize=(10, 8))
sns.heatmap(
    confusion_matrix,
    annot=True,
    linewidths=.4,
    fmt="d",
    square=True,
    ax=ax
)
plt.show()
```



Insight

- Accuracy of 0.71 and higher than model 1 and 2. While the confusion matrix shows some correct classifications, the high error rate suggests that the model is failing to distinguish between classes.

Model 4- CNN Data Augmentation Drop out [0.5, 0.3]

```
# Clearing backend
from tensorflow.keras import backend
backend.clear_session()

# Fixing the seed for random number generators
```

```python
import random
np.random.seed(42)
random.seed(42)
tf.random.set_seed(42)

# All images to be rescaled by 1/255.
train_datagen = ImageDataGenerator(
                              rotation_range=20,
                              fill_mode='nearest'
                              )
test_datagen  = ImageDataGenerator(rescale = 1.0/255.)

# Intializing a sequential model
model = Sequential()

# Adding first conv layer with 64 filters and kernel size 3x3 ,
padding 'same' provides the output size same as the input size
# Input_shape denotes input image dimension images
model.add(Conv2D(64, (3, 3), activation='relu', padding="same",
input_shape=(64, 64, 3)))

# Adding max pooling to reduce the size of output of first conv layer
model.add(MaxPooling2D((2, 2), padding = 'same'))
# model.add(BatchNormalization())
model.add(Conv2D(32, (3, 3), activation='relu', padding="same"))
model.add(MaxPooling2D((2, 2), padding = 'same'))
model.add(BatchNormalization())
# flattening the output of the conv layer after max pooling to make it
ready for creating dense connections
model.add(Flatten())

# Adding a fully connected dense layer with 100 neurons
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))  # Add dropout after the first dense layer
model.add(Dense(64, activation='relu'))
model.add(Dropout(0.3))  # Add dropout after the second dense layer
# Adding the output layer with 12 neurons and activation functions as
softmax since this is a multi-class classification problem
model.add(Dense(12, activation='softmax'))

# Using SGD Optimizer
# opt = SGD(learning_rate=0.01, momentum=0.9)
opt=Adam()
# Compile model
model.compile(optimizer=opt, loss='categorical_crossentropy',
metrics=['accuracy'])

# Generating the summary of the model
model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 64, 64, 64)	1,792
max_pooling2d (MaxPooling2D)	(None, 32, 32, 64)	0
conv2d_1 (Conv2D)	(None, 32, 32, 32)	18,464
max_pooling2d_1 (MaxPooling2D)	(None, 16, 16, 32)	0
batch_normalization (BatchNormalization)	(None, 16, 16, 32)	128
flatten (Flatten)	(None, 8192)	0
dense (Dense)	(None, 128)	1,048,704
dropout (Dropout)	(None, 128)	0
dense_1 (Dense)	(None, 64)	8,256
dropout_1 (Dropout)	(None, 64)	0

```
│ dense_2 (Dense)                       │ (None, 12)                │
780 │
└───────────────────────────────────────┴───────────────────────────┴
─────────────────┘

 Total params: 1,078,124 (4.11 MB)

 Trainable params: 1,078,060 (4.11 MB)

 Non-trainable params: 64 (256.00 B)
```

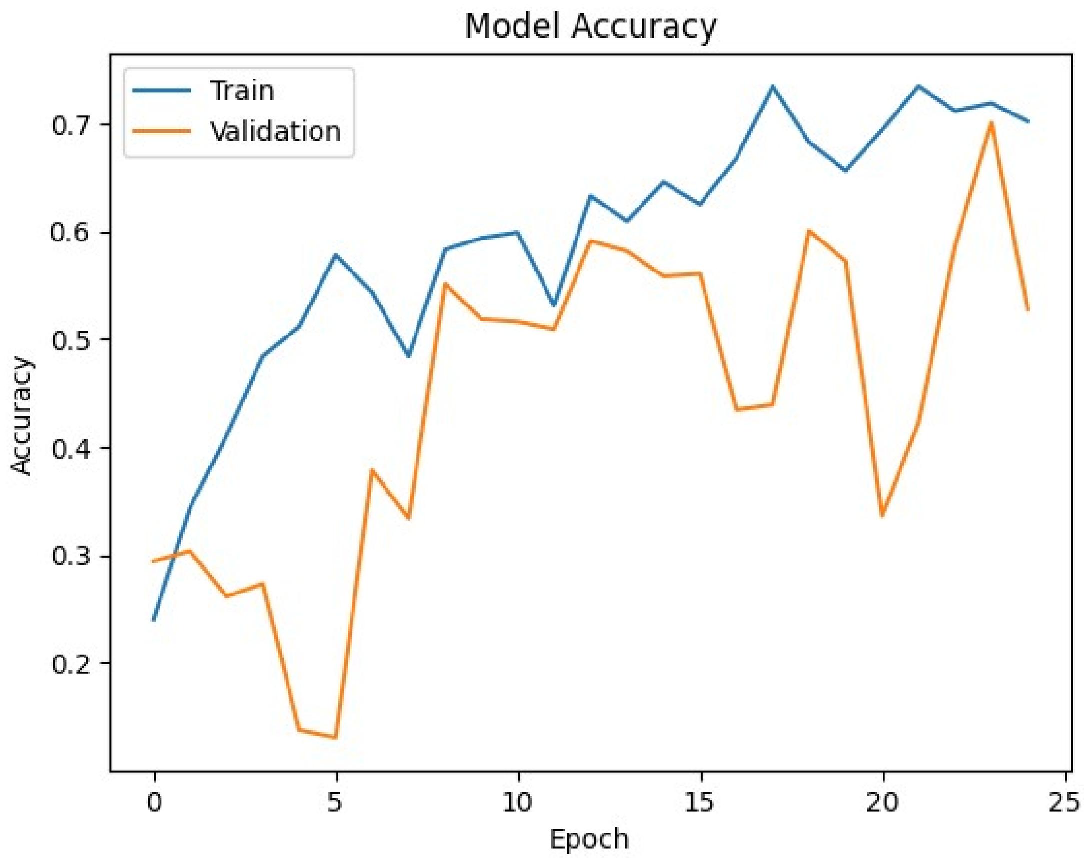
```
# Epochs
epochs = 25
# Batch size
batch_size = 64

history =
model.fit(train_datagen.flow(X_train_normalized,y_train_encoded,
                                       batch_size=batch_size,
                                       seed=42,
                                       shuffle=False),
                    epochs=epochs,
                    steps_per_epoch=X_train_normalized.shape[0] //
batch_size,
                    validation_data=(X_val_normalized,y_val_encoded),
                    verbose=1)
```

```
Epoch 1/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 11s 112ms/step - accuracy: 0.1794 - loss:
2.4540 - val_accuracy: 0.2944 - val_loss: 2.3947
Epoch 2/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.3438 - loss:
1.8678 - val_accuracy: 0.3037 - val_loss: 2.3933
Epoch 3/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 6s 101ms/step - accuracy: 0.3839 - loss:
1.8471 - val_accuracy: 0.2617 - val_loss: 2.3257
Epoch 4/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.4844 - loss:
1.5024 - val_accuracy: 0.2734 - val_loss: 2.3237
Epoch 5/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 9s 76ms/step - accuracy: 0.4908 - loss:
1.5148 - val_accuracy: 0.1379 - val_loss: 2.3324
Epoch 6/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.5781 - loss:
1.3759 - val_accuracy: 0.1308 - val_loss: 2.3426
Epoch 7/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 5s 78ms/step - accuracy: 0.5401 - loss:
1.3254 - val_accuracy: 0.3785 - val_loss: 2.0998
Epoch 8/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.4844 - loss:
```

```
1.2821 - val_accuracy: 0.3341 - val_loss: 2.0985
Epoch 9/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 5s 69ms/step - accuracy: 0.5903 - loss:
1.2578 - val_accuracy: 0.5514 - val_loss: 1.7286
Epoch 10/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 991us/step - accuracy: 0.5938 - loss:
1.1261 - val_accuracy: 0.5187 - val_loss: 1.7156
Epoch 11/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 6s 95ms/step - accuracy: 0.5883 - loss:
1.1787 - val_accuracy: 0.5164 - val_loss: 1.6232
Epoch 12/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.5312 - loss:
1.2611 - val_accuracy: 0.5093 - val_loss: 1.6306
Epoch 13/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 5s 71ms/step - accuracy: 0.6216 - loss:
1.0851 - val_accuracy: 0.5911 - val_loss: 1.2978
Epoch 14/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.6094 - loss:
1.1298 - val_accuracy: 0.5818 - val_loss: 1.3005
Epoch 15/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 6s 79ms/step - accuracy: 0.6446 - loss:
1.0288 - val_accuracy: 0.5584 - val_loss: 1.3052
Epoch 16/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.6250 - loss:
1.0892 - val_accuracy: 0.5607 - val_loss: 1.2848
Epoch 17/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 5s 72ms/step - accuracy: 0.6567 - loss:
0.9945 - val_accuracy: 0.4346 - val_loss: 1.7630
Epoch 18/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.7344 - loss:
0.8167 - val_accuracy: 0.4393 - val_loss: 1.6637
Epoch 19/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 5s 68ms/step - accuracy: 0.6805 - loss:
0.9252 - val_accuracy: 0.6005 - val_loss: 1.1811
Epoch 20/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.6562 - loss:
1.1324 - val_accuracy: 0.5724 - val_loss: 1.2191
Epoch 21/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 7s 95ms/step - accuracy: 0.6954 - loss:
0.8724 - val_accuracy: 0.3364 - val_loss: 2.1183
Epoch 22/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.7344 - loss:
0.8359 - val_accuracy: 0.4229 - val_loss: 1.7396
Epoch 23/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 9s 74ms/step - accuracy: 0.7083 - loss:
0.8540 - val_accuracy: 0.5864 - val_loss: 1.2627
Epoch 24/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.7188 - loss:
0.7790 - val_accuracy: 0.7009 - val_loss: 1.0006
```

```
Epoch 25/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 6s 78ms/step - accuracy: 0.7058 - loss:
0.8573 - val_accuracy: 0.5280 - val_loss: 1.7689
```

```
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('Model Accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='upper left')
plt.show()
```




```
accuracy = model.evaluate(X_test_normalized, y_test_encoded,
verbose=2)
```

```
15/15 - 0s - 22ms/step - accuracy: 0.4800 - loss: 1.9683
```

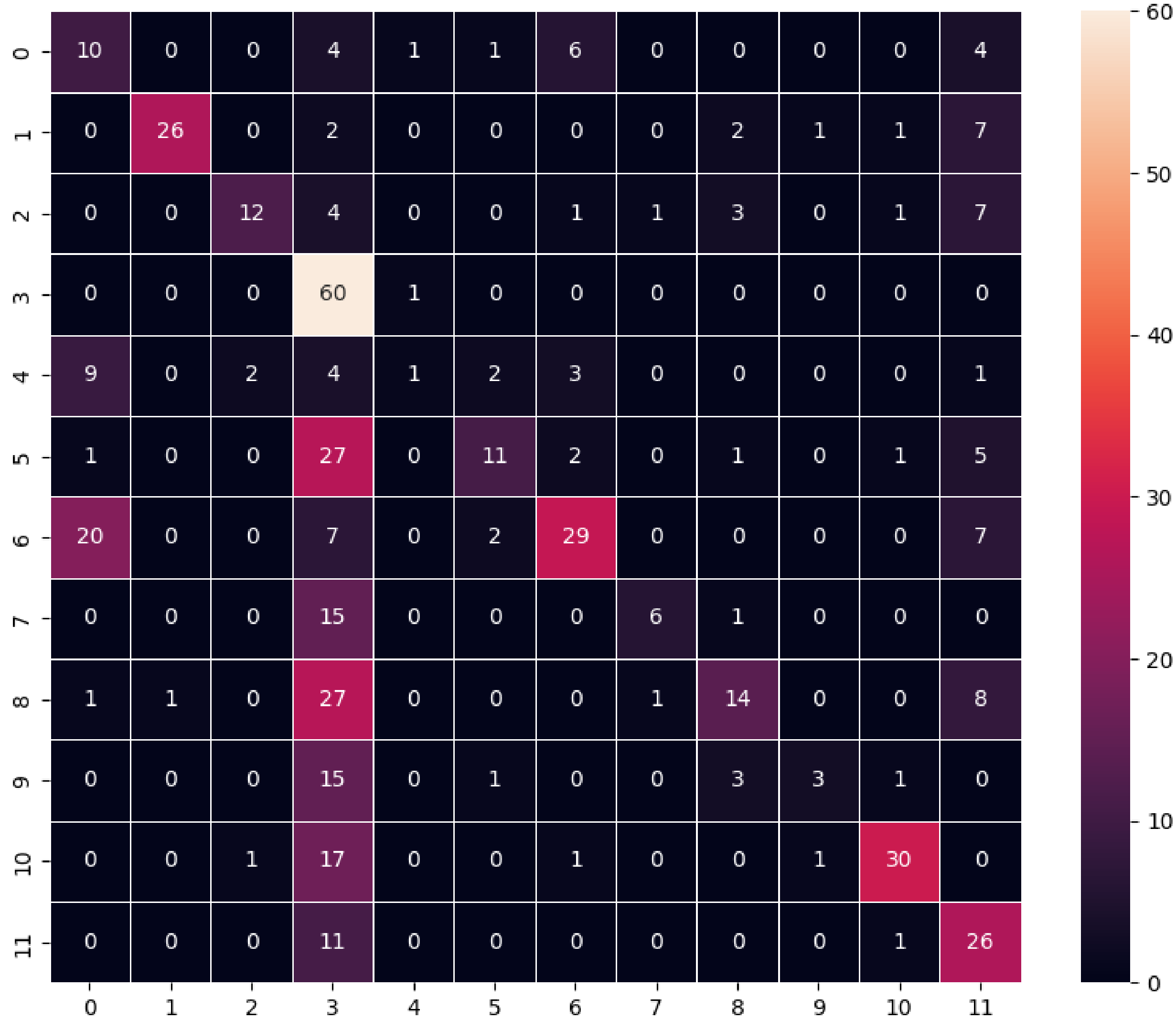
```
# Here we would get the output as probablities for each category
y_pred=model.predict(X_test_normalized)
```

```
15/15 ━━━━━━━━━━━━━━━━━━━━ 0s 16ms/step
```

```
y_pred
```

```
array([[3.95529998e-11, 5.42894549e-11, 9.10027804e-14, ...,
         1.18063633e-06, 2.36063649e-08, 1.02324265e-08],
       [2.69765343e-09, 6.27352492e-05, 2.02689989e-06, ...,
         5.64599298e-02, 9.39994752e-01, 2.35981322e-04],
       [5.36895243e-07, 7.89397236e-05, 9.47037552e-06, ...,
         1.48672655e-01, 6.33017421e-01, 5.24070160e-03],
       ...,
       [4.80278105e-01, 6.24360508e-10, 1.35395339e-08, ...,
         4.09103279e-07, 2.66701306e-07, 1.03751291e-03],
       [2.82798783e-06, 9.83096561e-06, 2.14139054e-05, ...,
         1.82944741e-02, 2.76299816e-05, 2.83887098e-03],
       [2.02286668e-04, 9.39405698e-04, 1.21023921e-04, ...,
         9.54436734e-02, 2.33823657e-01, 7.74171874e-02]],
dtype=float32)
```

```
# Obtaining the categorical values from y_test_encoded and y_pred
y_pred_arg=np.argmax(y_pred,axis=1)
y_test_arg=np.argmax(y_test_encoded,axis=1)

# Plotting the Confusion Matrix using confusion matrix() function
which is also predefined tensorflow module
confusion_matrix = tf.math.confusion_matrix(y_test_arg,y_pred_arg)
f, ax = plt.subplots(figsize=(10, 8))
sns.heatmap(
    confusion_matrix,
    annot=True,
    linewidths=.4,
    fmt="d",
    square=True,
    ax=ax
)
plt.show()
```



Insight

- Increasing the dropout rate [0.5, 0.3] reduced the accuracy of the model to 0.48. This model has the lowest accuracy amoung the models.

Models Conclusion

Model 1 - Convolutional Neural Network CNN: Accuracy 52%

Model 2 - Convolutional Neural Network CNN with Data Augmentation: Accuracy 66%

Model 3 - Convolutional Neural Network CNN with Data Augmentation learning rate: Accuracy 71%

Model 4 - Convolutional Neural Network CNN with Data Augmentation with Drop Out: Accuracy 48%

Model 3 has the highest accuracy and will be used on the final model.

Final Model

Model 3 - Convolutional Neural Network CNN with Data Augmentation learning rate: Accuracy 67%

Comment on the final model you have selected and use the same in the below code to visualize the image.

```
# Clearing backend
from tensorflow.keras import backend
backend.clear_session()

# Fixing the seed for random number generators
import random
np.random.seed(42)
random.seed(42)
tf.random.set_seed(42)

# All images to be rescaled by 1/255.
train_datagen = ImageDataGenerator(
                              rotation_range=20,
                              fill_mode='nearest'
                              )
test_datagen  = ImageDataGenerator(rescale = 1.0/255.)

import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten,
Dense, Dropout
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import EarlyStopping,
LearningRateScheduler

# Intializing a sequential model
model = Sequential()

# Adding first conv layer with 64 filters and kernel size 3x3 ,
padding 'same' provides the output size same as the input size
# Input_shape denotes input image dimension images
model.add(Conv2D(64, (3, 3), activation='relu', padding="same",
input_shape=(64, 64, 3)))

# Adding max pooling to reduce the size of output of first conv layer
model.add(MaxPooling2D((2, 2), padding = 'same'))
```

```
# model.add(BatchNormalization())
model.add(Conv2D(32, (3, 3), activation='relu', padding="same"))
model.add(MaxPooling2D((2, 2), padding = 'same'))
model.add(BatchNormalization())
# flattening the output of the conv layer after max pooling to make it
ready for creating dense connections
model.add(Flatten())

# Adding a fully connected dense layer with 100 neurons
model.add(Dense(16, activation='relu'))
model.add(Dropout(0.3))
# Adding the output layer with 12 neurons and activation functions as
softmax since this is a multi-class classification problem
model.add(Dense(12, activation='softmax'))

# Using Adam Optimizer with learning rate scheduling
def lr_schedule(epoch):
  lr = 0.001 * 0.95 ** epoch
  return lr

optimizer = Adam(learning_rate=lr_schedule(0))

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

# Early stopping
early_stop = EarlyStopping(monitor='val_loss', patience=5,
restore_best_weights=True)

# Learning rate scheduler
lr_scheduler = LearningRateScheduler(lr_schedule)

# Generating the summary of the model
model.summary()
```

```
Model: "sequential"

┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳
━━━━━━━━━━━━━━━━┓
┃ Layer (type)                          ┃ Output Shape               ┃
Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇
━━━━━━━━━━━━━━━━┩
│ conv2d (Conv2D)                       │ (None, 64, 64, 64)         │
1,792 │
├───────────────────────────────────────┼────────────────────────────┼
────────────────┤
│ max_pooling2d (MaxPooling2D)          │ (None, 32, 32, 64)         │
0 │
```

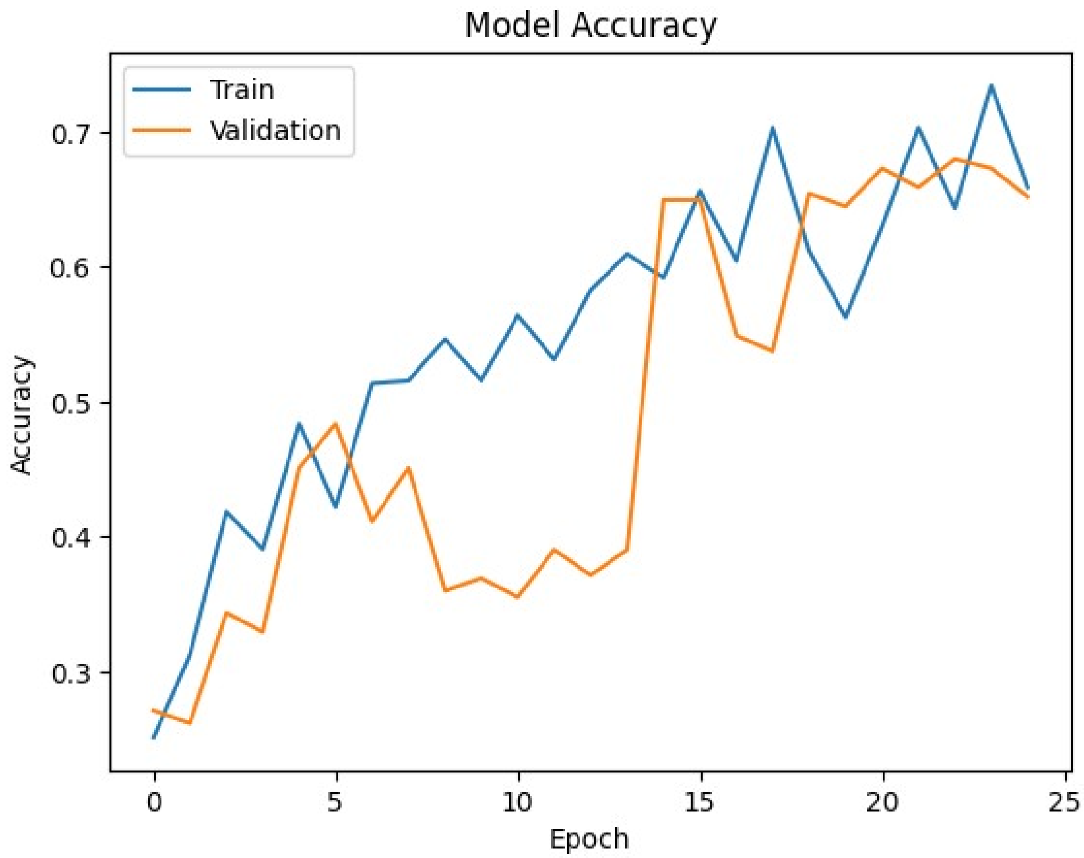
```
├─────────────────────────────────────┼───────────────────────────┼
────────────────┤
│ conv2d_1 (Conv2D)                   │ (None, 32, 32, 32)        │
18,464 │
├─────────────────────────────────────┼───────────────────────────┼
────────────────┤
│ max_pooling2d_1 (MaxPooling2D)      │ (None, 16, 16, 32)        │
0 │
├─────────────────────────────────────┼───────────────────────────┼
────────────────┤
│ batch_normalization                 │ (None, 16, 16, 32)        │
128 │
│ (BatchNormalization)                │                           │
│
├─────────────────────────────────────┼───────────────────────────┼
────────────────┤
│ flatten (Flatten)                   │ (None, 8192)              │
0 │
├─────────────────────────────────────┼───────────────────────────┼
────────────────┤
│ dense (Dense)                       │ (None, 16)                │
131,088 │
├─────────────────────────────────────┼───────────────────────────┼
────────────────┤
│ dropout (Dropout)                   │ (None, 16)                │
0 │
├─────────────────────────────────────┼───────────────────────────┼
────────────────┤
│ dense_1 (Dense)                     │ (None, 12)                │
204 │
└─────────────────────────────────────┴───────────────────────────┴
────────────────┘

 Total params: 151,676 (592.48 KB)

 Trainable params: 151,612 (592.23 KB)

 Non-trainable params: 64 (256.00 B)

# Epochs
epochs = 25
# Batch size
batch_size = 64

history =
model.fit(train_datagen.flow(X_train_normalized,y_train_encoded,
                                       batch_size=batch_size,
                                       seed=42,
                                       shuffle=False),
                    epochs=epochs,
```

```
                    steps_per_epoch=X_train_normalized.shape[0] //
batch_size,
                    validation_data=(X_val_normalized,y_val_encoded),
                    verbose=1)
```

```
Epoch 1/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 11s 125ms/step - accuracy: 0.1727 - loss:
2.3585 - val_accuracy: 0.2710 - val_loss: 2.3860
Epoch 2/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.3125 - loss:
1.9116 - val_accuracy: 0.2617 - val_loss: 2.3967
Epoch 3/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 5s 69ms/step - accuracy: 0.4057 - loss:
1.8079 - val_accuracy: 0.3435 - val_loss: 2.3202
Epoch 4/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 994us/step - accuracy: 0.3906 - loss:
1.5433 - val_accuracy: 0.3294 - val_loss: 2.2973
Epoch 5/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 6s 93ms/step - accuracy: 0.4748 - loss:
1.5934 - val_accuracy: 0.4509 - val_loss: 2.2449
Epoch 6/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.4219 - loss:
1.6922 - val_accuracy: 0.4836 - val_loss: 2.2265
Epoch 7/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 5s 71ms/step - accuracy: 0.5044 - loss:
1.5094 - val_accuracy: 0.4112 - val_loss: 2.0170
Epoch 8/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.5156 - loss:
1.4296 - val_accuracy: 0.4509 - val_loss: 2.0074
Epoch 9/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 5s 77ms/step - accuracy: 0.5463 - loss:
1.3345 - val_accuracy: 0.3598 - val_loss: 1.8660
Epoch 10/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.5156 - loss:
1.3747 - val_accuracy: 0.3692 - val_loss: 1.8596
Epoch 11/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 6s 79ms/step - accuracy: 0.5610 - loss:
1.2655 - val_accuracy: 0.3551 - val_loss: 1.8740
Epoch 12/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 980us/step - accuracy: 0.5312 - loss:
1.3712 - val_accuracy: 0.3902 - val_loss: 1.8299
Epoch 13/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 11s 94ms/step - accuracy: 0.5802 - loss:
1.2343 - val_accuracy: 0.3715 - val_loss: 1.8175
Epoch 14/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 976us/step - accuracy: 0.6094 - loss:
1.4721 - val_accuracy: 0.3902 - val_loss: 1.6694
Epoch 15/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 9s 70ms/step - accuracy: 0.5992 - loss:
1.1799 - val_accuracy: 0.6495 - val_loss: 1.2193
```

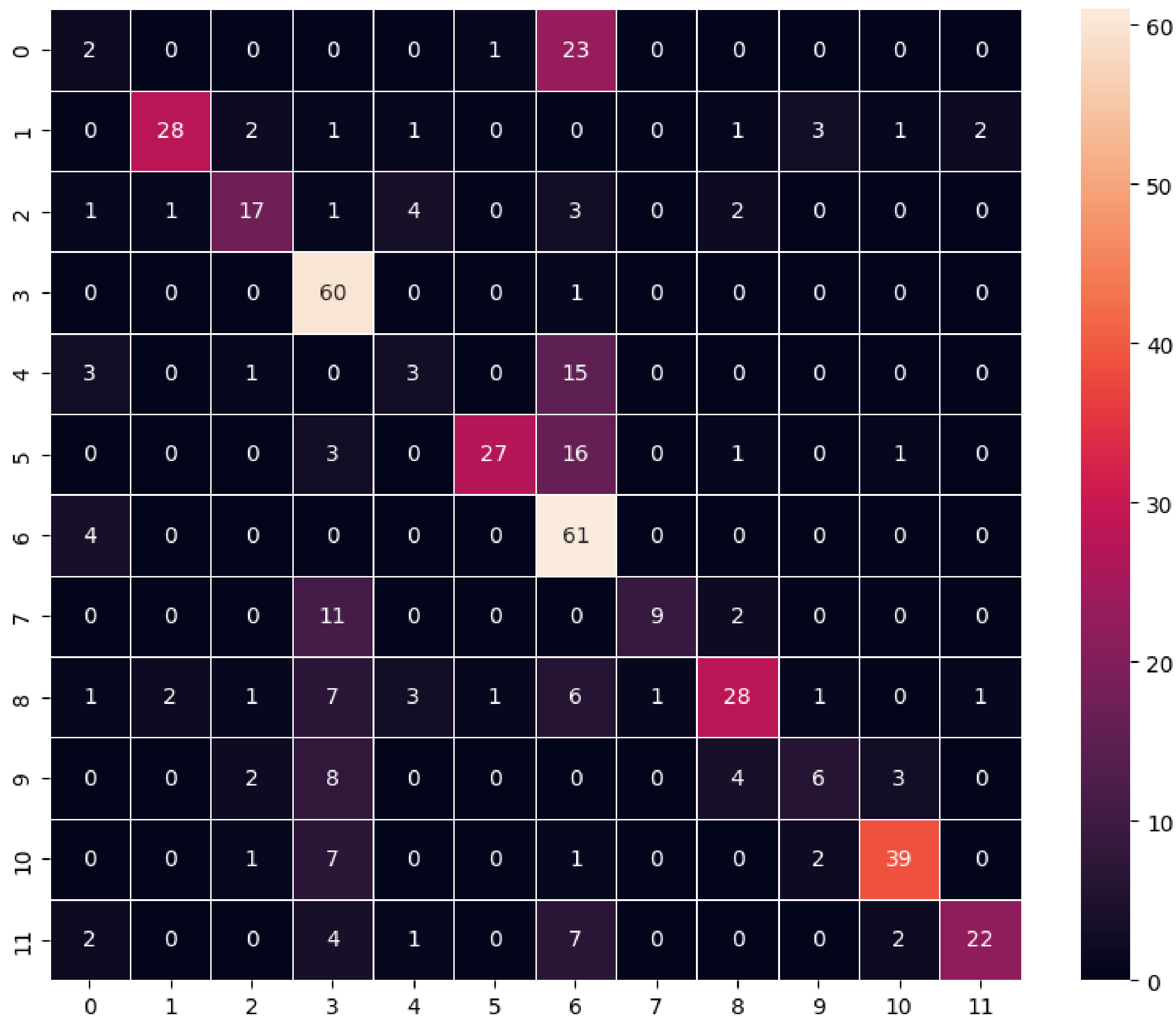
```
Epoch 16/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.6562 - loss:
1.0665 - val_accuracy: 0.6495 - val_loss: 1.2415
Epoch 17/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 6s 88ms/step - accuracy: 0.5939 - loss:
1.1235 - val_accuracy: 0.5491 - val_loss: 1.2722
Epoch 18/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 965us/step - accuracy: 0.7031 - loss:
0.8660 - val_accuracy: 0.5374 - val_loss: 1.3319
Epoch 19/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 6s 92ms/step - accuracy: 0.6222 - loss:
1.0941 - val_accuracy: 0.6542 - val_loss: 1.0582
Epoch 20/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 989us/step - accuracy: 0.5625 - loss:
1.1484 - val_accuracy: 0.6449 - val_loss: 1.0809
Epoch 21/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 9s 71ms/step - accuracy: 0.6269 - loss:
1.0525 - val_accuracy: 0.6729 - val_loss: 0.9668
Epoch 22/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.7031 - loss:
0.8454 - val_accuracy: 0.6589 - val_loss: 1.0205
Epoch 23/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 5s 75ms/step - accuracy: 0.6486 - loss:
1.0118 - val_accuracy: 0.6799 - val_loss: 0.9501
Epoch 24/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.7344 - loss:
0.7852 - val_accuracy: 0.6729 - val_loss: 0.9544
Epoch 25/25
60/60 ━━━━━━━━━━━━━━━━━━━━ 6s 78ms/step - accuracy: 0.6542 - loss:
0.9642 - val_accuracy: 0.6519 - val_loss: 1.1042
```

```python
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('Model Accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='upper left')
plt.show()
```




```
accuracy = model.evaluate(X_test_normalized, y_test_encoded,
verbose=2)

15/15 - 0s - 23ms/step - accuracy: 0.6358 - loss: 1.1600

# Here we would get the output as probablities for each category
y_pred=model.predict(X_test_normalized)

15/15 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step

# Obtaining the categorical values from y_test_encoded and y_pred
y_pred_arg=np.argmax(y_pred,axis=1)
y_test_arg=np.argmax(y_test_encoded,axis=1)

# Plotting the Confusion Matrix using confusion matrix() function
which is also predefined tensorflow module
confusion_matrix = tf.math.confusion_matrix(y_test_arg,y_pred_arg)
f, ax = plt.subplots(figsize=(10, 8))
sns.heatmap(
    confusion_matrix,
    annot=True,
    linewidths=.4,
    fmt="d",
    square=True,
```

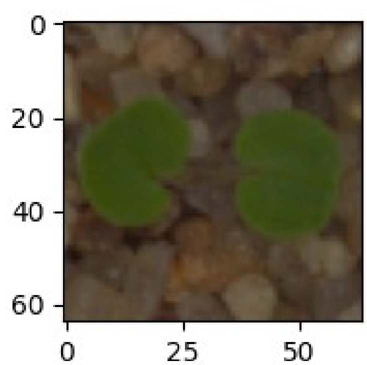
```
    ax=ax
)
plt.show()
```



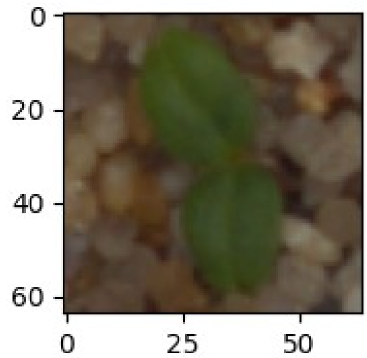
Visualizing the prediction

```
# Visualizing the predicted and correct label of images from test data
plt.figure(figsize=(2,2))
plt.imshow(X_test[2])
plt.show()
print('Predicted Label',
enc.inverse_transform(new_model.predict((X_test_normalized[2].reshape(
1,64,64,3)))))    # reshaping the input image as we are only trying to
predict using a single image
print('True Label', enc.inverse_transform(y_test_encoded)[2])
# using inverse_transform() to get the output label from the output
vector
```

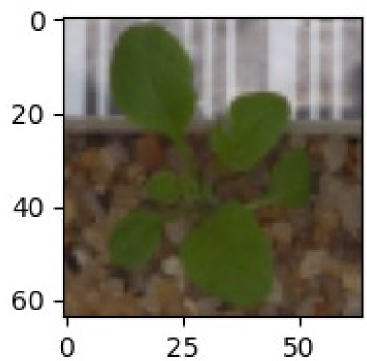
```
plt.figure(figsize=(2,2))
plt.imshow(X_test[33])
plt.show()
print('Predicted Label',
enc.inverse_transform(new_model.predict((X_test_normalized[33].reshape
(1,64,64,3)))))  # reshaping the input image as we are only trying to
predict using a single image
print('True Label', enc.inverse_transform(y_test_encoded)[33])
# using inverse_transform() to get the output label from the output
vector

plt.figure(figsize=(2,2))
plt.imshow(X_test[36])
plt.show()
print('Predicted Label',
enc.inverse_transform(new_model.predict((X_test_normalized[36].reshape
(1,64,64,3)))))  # reshaping the input image as we are only trying to
predict using a single image
print('True Label', enc.inverse_transform(y_test_encoded)[36])
# using inverse_transform() to get the output label from the output
vector
```




```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 61ms/step
Predicted Label ['Small-flowered Cranesbill']
True Label Small-flowered Cranesbill
```




```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 16ms/step
Predicted Label ['Fat Hen']
True Label Cleavers
```




```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 28ms/step
Predicted Label ['Charlock']
True Label Shepherds Purse
```

Model Conclusion & Improvement

While the model has achieved a reasonable accuracy of 63%, there's room for improvement.

Data-Related Improvements

- Data Augmentation: Increase the size and diversity of your training dataset by applying transformations like rotations, flips, and brightness adjustments.
- Data Quality: Ensure image quality is consistent and free from noise or artifacts.
- Class Balancing: If your dataset is imbalanced, consider techniques like oversampling or undersampling to address the issue.

Model Architecture Improvements

- Deeper Networks: Experiment with adding more convolutional layers or increasing the number of filters to capture more complex features.
- Residual Connections: Incorporate residual connections to improve training and accuracy for deeper networks.

Actionable Insights and Business Recommendations

Business Problem Context

Given the context of automating seedling identification to reduce manual labor in agriculture, a model accuracy of 63% is a promising starting point. It indicates that the model can reliably identify a significant portion of seedlings, which has the potential to streamline agricultural processes.

Actionable Insight

- Estimate the potential cost savings and time efficiency gains by implementing the model on a larger scale.
- Assess the impact of data quality on model performance. High-quality image data is crucial for accurate predictions.
- Consider the factors influencing farmer adoption of the technology, such as ease of use, cost-effectiveness, and perceived value.

Business Recommendations

- Develop a basic version of the seedling identification tool to test its feasibility and gather user feedback.
- Pilot Projects: Conduct pilot projects with farmers to evaluate the model's performance in real-world conditions.
- Data Collection: Continuously collect data to improve model accuracy over time.
- Cost-Benefit Analysis: Conduct a thorough cost-benefit analysis to determine the economic viability of the solution.
- Provide comprehensive training to farmers on how to use the tool effectively.

Convert to HTML