# Plants Seedling Classification

## Description

### Background and 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 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 involvement for higher-order agricultural decision making, and in the long term will result in more sustainable environmental practices in agriculture as well.

### Objective

The Aarhus University Signal Processing group, in collaboration with the University of Southern Denmark, has provided the data containing images of unique plants belonging to 12 different species. Being a data scientist, need to build a Convolutional Neural Network model which would classify the plant seedlings into their respective 12 categories.

### Data Description

This dataset contains images of unique plants belonging to 12 different species.

- The data file names are:
  - images.npy
  - Label.csv
- Due to the large volume of data, the images were converted to numpy arrays and stored in images.npy file and the corresponding labels are also put into Labels.csv.
- The goal of the project is to create a classifier capable of determining a plant's species from an image.

### List of Plant species:

- Black-grass
- Charlock
- Cleavers
- Common Chickweed
- Common Wheat
- Fat Hen
- Loose Silky-bent
- Maize

- Scentless Mayweed
- Shepherds Purse
- Small-flowered Cranesbill
- Sugar beet

## Importing the necessary libraries¶

In [1]:
```
import os
# Importing numpy for Matrix Operations
import numpy as np
# Importing pandas to read CSV files
import pandas as pd
# Importting matplotlib for Plotting and visualizing images
import matplotlib.pyplot as plt
# Importing math module to perform mathematical operations
import math
# Importing openCV for image processing
import cv2
# Importing seaborn to plot graphs
import seaborn as sns


# Tensorflow modules
import tensorflow as tf
# Importing the ImageDataGenerator for data augmentation
from tensorflow.keras.preprocessing.image import ImageDataGenerator
# Importing the sequential module to define a sequential model
from tensorflow.keras.models import Sequential
# Defining all the layers to build our CNN Model
from tensorflow.keras.layers import Dense,Dropout,Flatten,Conv2D,MaxPooling2D,BatchNormalization,MaxPool2D,GlobalMaxPooling2D
# Importing the optimizers which can be used in our model
from tensorflow.keras.optimizers import Adam,SGD,RMSprop
# Importing the preprocessing module to preprocess the data
from sklearn import preprocessing
# Importing train_test_split function to split the data into train and test
from sklearn.model_selection import train_test_split
# Importing confusion_matrix to plot the confusion matrix
from sklearn.metrics import confusion_matrix
# convert to one-hot-encoding
from keras.utils.np_utils import to_categorical

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


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

## Reading the dataset

In [2]:
```
# Mount Google drive to access the dataset (monkeys_dataset.zip)
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

In [3]:
```
# Load the image file of the dataset
images = np.load('/content/drive/MyDrive/Colab Notebooks/images.npy')

# Load the labels file of the dataset
labels = pd.read_csv('/content/drive/MyDrive/Colab Notebooks/Labels.cs
v')
```

## Overview of the dataset

Let's print the shape of the images and labels

In [4]:
```
print(images.shape)
print(labels.shape)
```

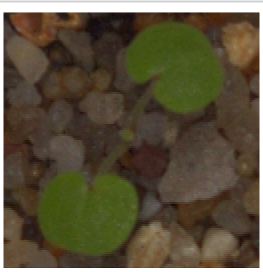
```
(4750, 128, 128, 3)
(4750, 1)
```

*Observations:*

- There are 4750 RGB images of shape 128 x 128 X 3, each image having 3 channels.

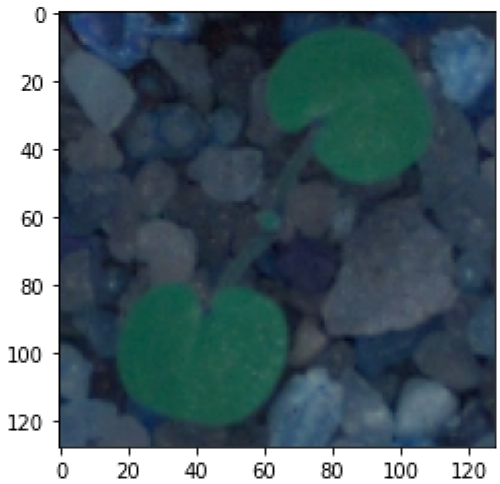
## Plotting images using OpenCV and matplotlib

In [ ]:
```
cv2_imshow(images[5])
```



In [ ]:
```
plt.imshow(images[5])
```

Out[ ]: <matplotlib.image.AxesImage at 0x7fee8c728090>




*Observations:*

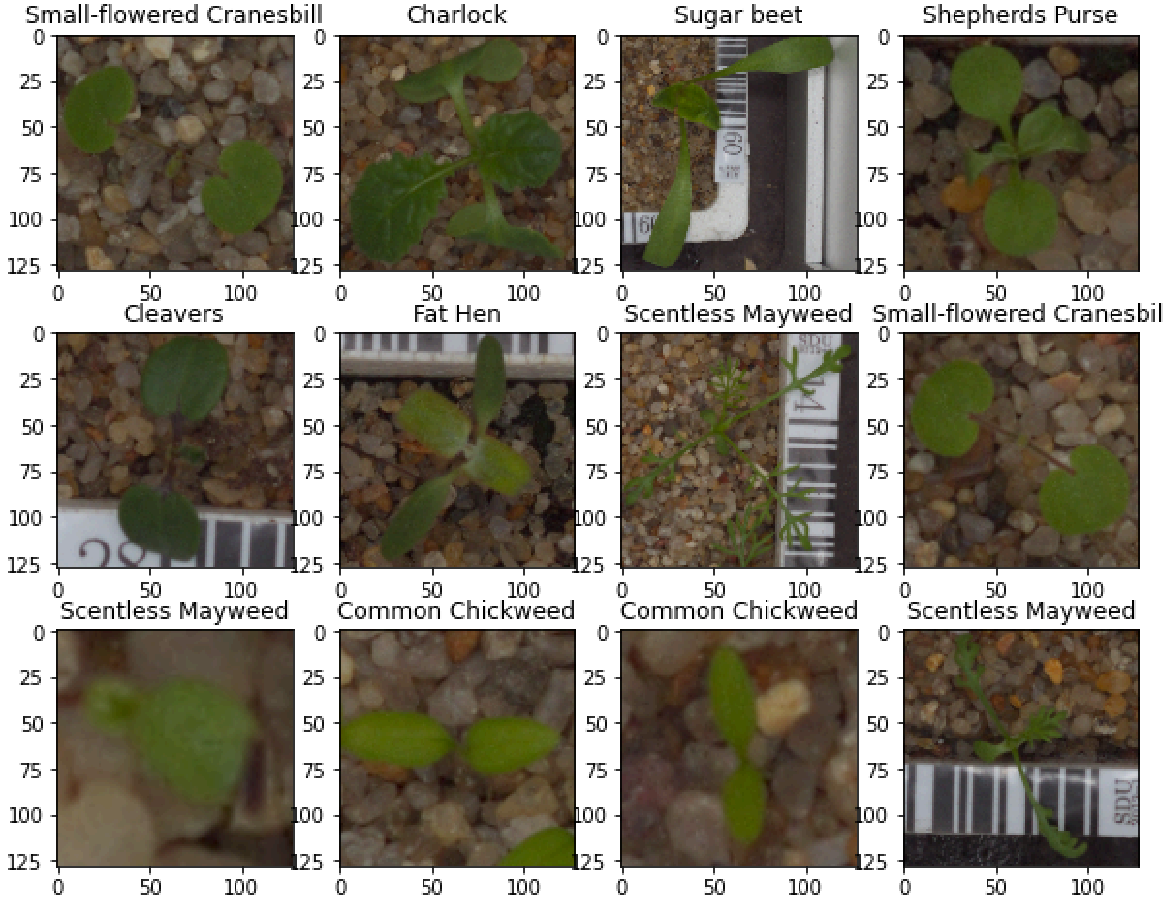
- We can observe that the images are being shown in different colors when plotted with openCV and matplotlib as OpenCV reads images in BGR format and this shows that the given numpy arrays were generated from the original images using OpenCV.
- Now we will convert these BGR images to RGB images so we could interpret them easily.

In [5]:
```
# Converting the images from BGR to RGB using cvtColor function of Ope
nCV
for i in range(len(images)):
  images[i] = cv2.cvtColor(images[i], cv2.COLOR_BGR2RGB)
```

# Exploratory Data Analysis

In [6]:
```
def plot_images(images,labels):
  # Number of Classes
  num_classes=12
  categories=np.unique(labels)
  # Obtaing the unique classes from y_train
  keys=dict(labels['Label'])
  # Defining number of rows=3
  rows = 3
  # Defining number of columns=4
  cols = 4
  # Defining the figure size to 10x8
  fig = plt.figure(figsize=(10, 8))
  for i in range(cols):
      for j in range(rows):
          # Generating random indices from the data and plotting the i
mages
          random_index = np.random.randint(0, len(labels))
          # Adding subplots with 3 rows and 4 columns
          ax = fig.add_subplot(rows, cols, i * rows + j + 1)
          # Plotting the image
          ax.imshow(images[random_index, :])
          ax.set_title(keys[random_index])
  plt.show()
```

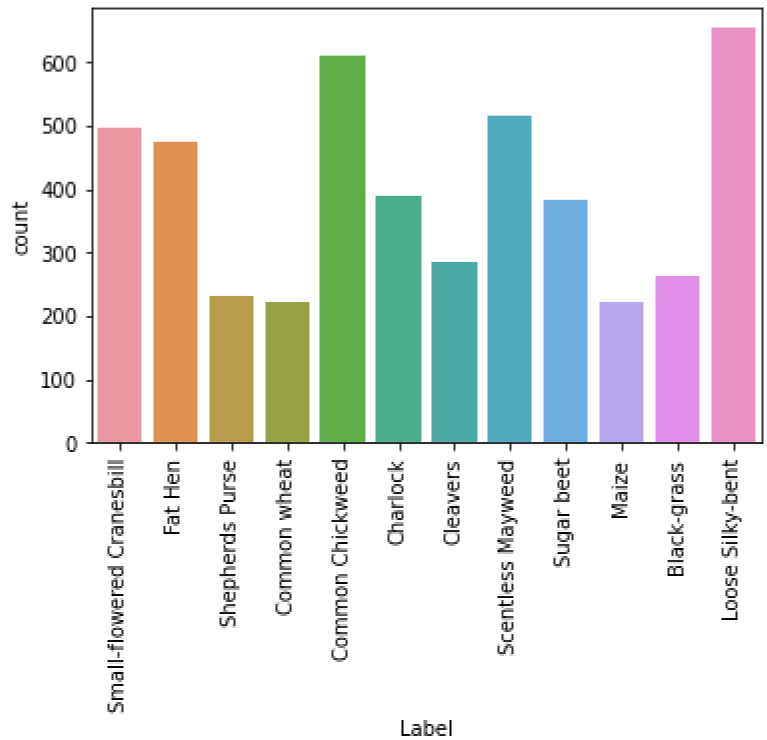
In [ ]:
```
plot_images(images,labels)
```



## Checking for data imbalance

In [ ]:
```
sns.countplot(labels['Label'])
plt.xticks(rotation='vertical')
```

Out[ ]:
```
(array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11]),
 <a list of 12 Text major ticklabel objects>)
```



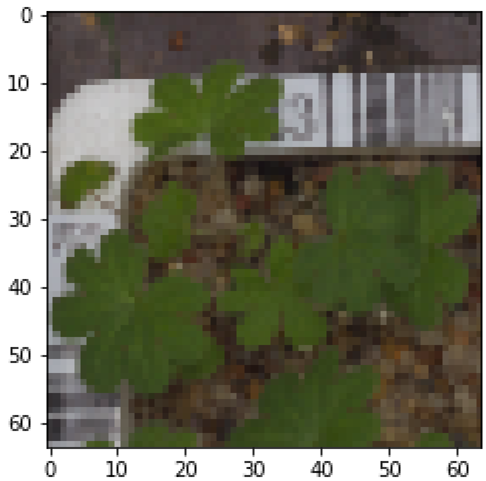

*Observations:*

- As you can see above, the classes are slightly imbalanced. It might be helpful to downsample or upsample to have a more balanced dataset.

## Resizing images

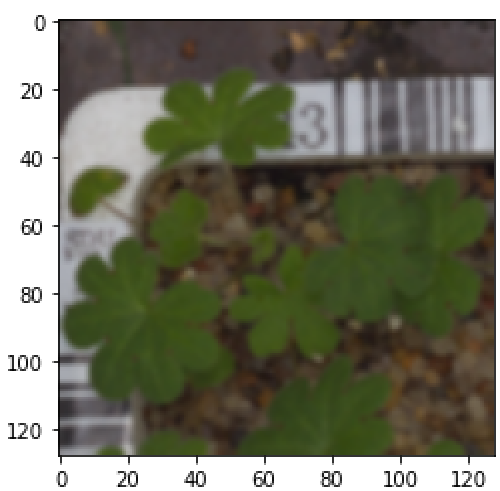
In [31]:
```
images_resized=[]
height = 64
width = 64
dimensions = (width, height)
for i in range(len(images)):
  images_resized.append( cv2.resize(images[i], dimensions, interpolati
on=cv2.INTER_LINEAR))
```

In [32]:
```
plt.imshow(images_resized[3])
```
Out[32]: <matplotlib.image.AxesImage at 0x7f040117b810>



# Visualizing images using Gaussian Blur

In [33]:
```
# Applying Gaussian Blur to denoise the images
images_gb=[]
for i in range(len(images)):
  # gb[i] = cv2.cvtColor(images[i], cv2.COLOR_BGR2RGB)
  images_gb.append(cv2.GaussianBlur(images[i], ksize =(3,3),sigmaX =
0))
```
In [34]:
```
plt.imshow(images_gb[3])
```
Out[34]: <matplotlib.image.AxesImage at 0x7f0404f90210>



*Observations:*

- It appears that GaussianBlur would be ineffective because the blurred or denoised image does not seem to contain any relevant information, and the model would struggle to categorize these blurred images.

## Splitting the dataset

As we have less images in our dataset, we will only use 10% of our data for testing and 90% of our data for training. We are using the train_test_split() function from scikit-learn. Here, we split the dataset while keeping the test size constant at 0.1. This means that 10% of total data is used for testing, while 90% is used for training.

In [35]:
```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(np.array(images_re
sized),labels , test_size=0.1, random_state=42,stratify=labels)
```

In [36]:
```
print(X_train.shape,y_train.shape)
print(X_test.shape,y_test.shape)
```

```
(4275, 64, 64, 3) (4275, 1)
(475, 64, 64, 3) (475, 1)
```

## Making the data compatible:

- Convert labels from names to one hot vectors
- Normalizing the values

## Encoding the target labels

In [37]:
```
# Convert labels from names to one hot vectors.
# We have already used encoding methods like onehotencoder and labelen
coder earlier so now we will be using a new encoding method called lab
elBinarizer.
# Labelbinarizer works similar to onehotencoder

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

## Data Normalization

Since the image pixel values range from 0-255, our method of normalization here will be scaling - we shall divide all the pixel values by 255 to standardize the images to have values between 0-1.

In [38]:
```
# Normalizing the image pixels
X_train_normalized = X_train.astype('float32')/255.0
X_test_normalized = X_test.astype('float32')/255.0
```

## Model Building - Convolutional Neural Network (CNN)

## Model-1

Let's create a CNN model sequentially, where we will be adding the layers one after another.

First, we need to clear the previous model's history from the session.

In Keras, we need a special command to clear the model's history, otherwise the previous model history remains in the backend.

Also, let's fix the seed again after clearing the backend.

Let's set the seed for random number generators in Numpy, the Random library in Python, and in TensorFlow to be able to reproduce the same results every time we run the code.

In [39]:
```
# Clearing backend
from tensorflow.keras import backend
backend.clear_session()
```

In [40]:
```
# Fixing the seed for random number generators
import random
np.random.seed(42)
random.seed(42)
tf.random.set_seed(42)
```

Now, let's build a CNN Model with the following 2 main parts -

1. The Feature Extraction layers which are comprised of convolutional and pooling layers.
2. The Fully Connected classification layers for prediction.

In [41]:
```
# Intializing a sequential model
model = Sequential()

# Adding first conv layer with 64 filters and kernel size 3x3 , paddin
g '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 10 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)        1792      
                                                                 
 max_pooling2d (MaxPooling2D  (None, 32, 32, 64)       0         
 )                                                               
                                                                 
 conv2d_1 (Conv2D)           (None, 32, 32, 32)        18464     
                                                                 
 max_pooling2d_1 (MaxPooling  (None, 16, 16, 32)       0         
 2D)                                                             
                                                                 
 flatten (Flatten)           (None, 8192)              0         
                                                                 
 dense (Dense)               (None, 16)                131088    
                                                                 
 dropout (Dropout)           (None, 16)                0         
                                                                 
 dense_1 (Dense)             (None, 12)                204       
                                                                 
=================================================================
Total params: 151,548
Trainable params: 151,548
Non-trainable params: 0
_________________________________________________________________
```

## Fitting the model on the train data

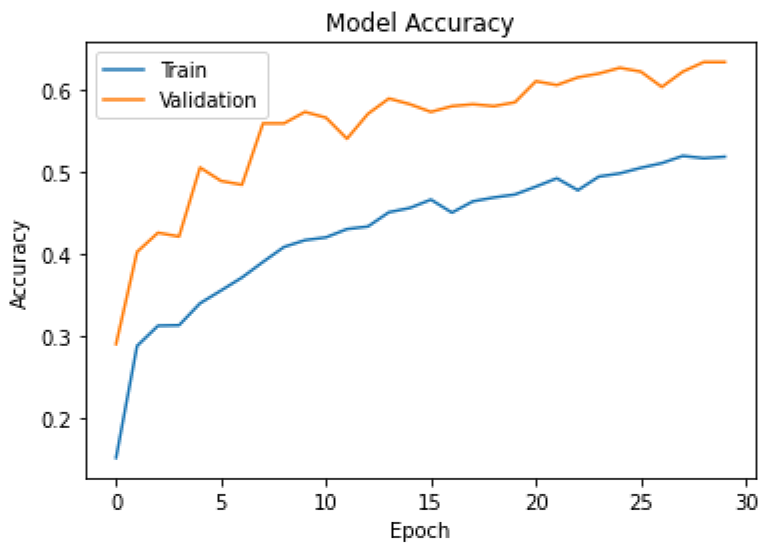
In [42]:
```
history_1 = model.fit(
            X_train_normalized, y_train_encoded,
            epochs=30,
            validation_split=0.1,
            # shuffle=True,
            batch_size=32,
            verbose=2
)
```

```
Epoch 1/30
121/121 - 31s - loss: 2.4027 - accuracy: 0.1510 - val_loss: 2.1484 - v
al_accuracy: 0.2897 - 31s/epoch - 257ms/step
Epoch 2/30
121/121 - 30s - loss: 2.1241 - accuracy: 0.2872 - val_loss: 1.9012 - v
al_accuracy: 0.4019 - 30s/epoch - 246ms/step
Epoch 3/30
121/121 - 31s - loss: 1.9969 - accuracy: 0.3119 - val_loss: 1.7580 - v
al_accuracy: 0.4252 - 31s/epoch - 257ms/step
Epoch 4/30
121/121 - 31s - loss: 1.9165 - accuracy: 0.3125 - val_loss: 1.7044 - v
al_accuracy: 0.4206 - 31s/epoch - 256ms/step
Epoch 5/30
121/121 - 30s - loss: 1.8505 - accuracy: 0.3392 - val_loss: 1.6272 - v
al_accuracy: 0.5047 - 30s/epoch - 245ms/step
Epoch 6/30
121/121 - 30s - loss: 1.7547 - accuracy: 0.3548 - val_loss: 1.5318 - v
al_accuracy: 0.4883 - 30s/epoch - 246ms/step
Epoch 7/30
121/121 - 31s - loss: 1.7072 - accuracy: 0.3707 - val_loss: 1.5327 - v
al_accuracy: 0.4836 - 31s/epoch - 252ms/step
Epoch 8/30
121/121 - 30s - loss: 1.6472 - accuracy: 0.3897 - val_loss: 1.3661 - v
al_accuracy: 0.5584 - 30s/epoch - 247ms/step
Epoch 9/30
121/121 - 30s - loss: 1.6056 - accuracy: 0.4079 - val_loss: 1.3708 - v
al_accuracy: 0.5584 - 30s/epoch - 246ms/step
Epoch 10/30
121/121 - 30s - loss: 1.5765 - accuracy: 0.4162 - val_loss: 1.3500 - v
al_accuracy: 0.5724 - 30s/epoch - 245ms/step
Epoch 11/30
121/121 - 29s - loss: 1.5711 - accuracy: 0.4195 - val_loss: 1.3185 - v
al_accuracy: 0.5654 - 29s/epoch - 242ms/step
Epoch 12/30
121/121 - 46s - loss: 1.5494 - accuracy: 0.4297 - val_loss: 1.3350 - v
al_accuracy: 0.5397 - 46s/epoch - 377ms/step
Epoch 13/30
121/121 - 30s - loss: 1.5422 - accuracy: 0.4328 - val_loss: 1.2813 - v
al_accuracy: 0.5701 - 30s/epoch - 246ms/step
Epoch 14/30
121/121 - 30s - loss: 1.4973 - accuracy: 0.4502 - val_loss: 1.2535 - v
al_accuracy: 0.5888 - 30s/epoch - 251ms/step
Epoch 15/30
121/121 - 42s - loss: 1.5035 - accuracy: 0.4554 - val_loss: 1.3042 - v
al_accuracy: 0.5818 - 42s/epoch - 346ms/step
Epoch 16/30
121/121 - 41s - loss: 1.4722 - accuracy: 0.4656 - val_loss: 1.2841 - v
al_accuracy: 0.5724 - 41s/epoch - 335ms/step
Epoch 17/30
121/121 - 40s - loss: 1.4818 - accuracy: 0.4497 - val_loss: 1.3108 - v
al_accuracy: 0.5794 - 40s/epoch - 329ms/step
Epoch 18/30
121/121 - 46s - loss: 1.4511 - accuracy: 0.4635 - val_loss: 1.2186 - v
al_accuracy: 0.5818 - 46s/epoch - 376ms/step
Epoch 19/30
121/121 - 35s - loss: 1.4495 - accuracy: 0.4682 - val_loss: 1.2460 - v
al_accuracy: 0.5794 - 35s/epoch - 291ms/step
```

```
Epoch 20/30
121/121 - 29s - loss: 1.4464 - accuracy: 0.4718 - val_loss: 1.2084 - v
al_accuracy: 0.5841 - 29s/epoch - 243ms/step
Epoch 21/30
121/121 - 30s - loss: 1.4095 - accuracy: 0.4814 - val_loss: 1.2131 - v
al_accuracy: 0.6098 - 30s/epoch - 248ms/step
Epoch 22/30
121/121 - 32s - loss: 1.3786 - accuracy: 0.4916 - val_loss: 1.1814 - v
al_accuracy: 0.6051 - 32s/epoch - 261ms/step
Epoch 23/30
121/121 - 42s - loss: 1.3832 - accuracy: 0.4770 - val_loss: 1.1808 - v
al_accuracy: 0.6145 - 42s/epoch - 351ms/step
Epoch 24/30
121/121 - 30s - loss: 1.3375 - accuracy: 0.4936 - val_loss: 1.1473 - v
al_accuracy: 0.6192 - 30s/epoch - 247ms/step
Epoch 25/30
121/121 - 29s - loss: 1.3480 - accuracy: 0.4975 - val_loss: 1.1397 - v
al_accuracy: 0.6262 - 29s/epoch - 243ms/step
Epoch 26/30
121/121 - 40s - loss: 1.3257 - accuracy: 0.5043 - val_loss: 1.1807 - v
al_accuracy: 0.6215 - 40s/epoch - 328ms/step
Epoch 27/30
121/121 - 45s - loss: 1.3051 - accuracy: 0.5100 - val_loss: 1.1523 - v
al_accuracy: 0.6028 - 45s/epoch - 370ms/step
Epoch 28/30
121/121 - 50s - loss: 1.2849 - accuracy: 0.5188 - val_loss: 1.1437 - v
al_accuracy: 0.6215 - 50s/epoch - 414ms/step
Epoch 29/30
121/121 - 31s - loss: 1.2587 - accuracy: 0.5160 - val_loss: 1.1194 - v
al_accuracy: 0.6332 - 31s/epoch - 258ms/step
Epoch 30/30
121/121 - 29s - loss: 1.2628 - accuracy: 0.5178 - val_loss: 1.1193 - v
al_accuracy: 0.6332 - 29s/epoch - 244ms/step
```

# Model Evaluation

In [43]:
```
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()
```




*Observations:*

- We can see from the above plot that the training accuracy of the model was good but the validation accuracy was not good.

# Evaluating the model on test data

In [44]:
```
accuracy = model.evaluate(X_test_normalized, y_test_encoded, verbose=2)
```

```
15/15 - 1s - loss: 1.1736 - accuracy: 0.6316 - 945ms/epoch - 63ms/step
```

# Generating the predictions using test data

In [45]:
```
# Here we would get the output as probablities for each category
y_pred=model.predict(X_test_normalized)
```

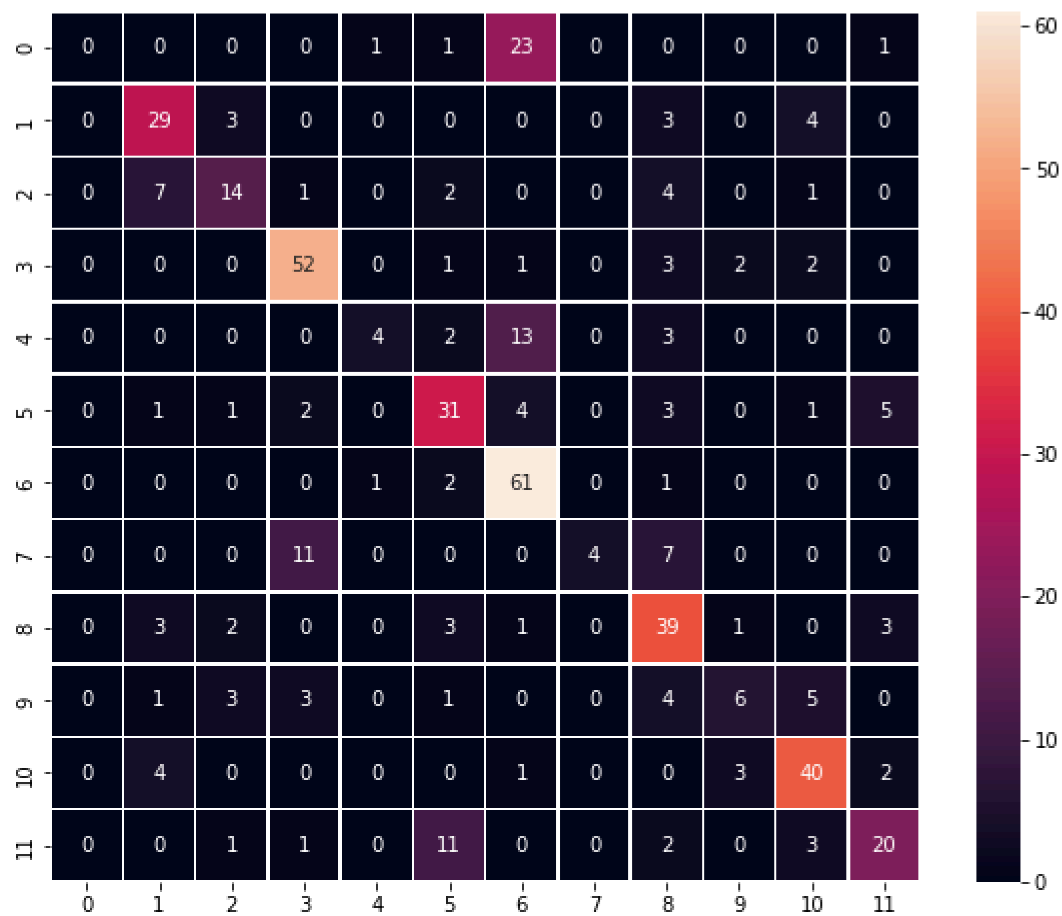
In [46]:
```
y_pred
```

Out[46]:
```
array([[1.09874218e-05, 1.22168786e-11, 6.73649980e-09, ...,
         5.27223758e-03, 4.38157399e-10, 2.48443172e-03],
       [1.66020725e-10, 8.90419334e-02, 2.86280550e-02, ...,
         6.74159452e-02, 6.81265354e-01, 5.55500947e-02],
       [5.89081765e-07, 1.45122074e-02, 4.66782972e-03, ...,
         1.07032210e-01, 6.92848265e-01, 3.05882618e-02],
       ...,
       [2.64065087e-01, 6.51067609e-08, 1.75980938e-04, ...,
         2.15251230e-08, 1.17872323e-09, 4.60183388e-03],
       [7.57237967e-06, 1.02935161e-03, 6.18628040e-02, ...,
         1.31721675e-01, 6.78519078e-04, 2.02117022e-02],
       [5.73528450e-06, 8.67484361e-02, 2.93071494e-02, ...,
         2.69943655e-01, 3.81976753e-01, 4.00505774e-02]], dtype=float3
2)
```

## Plotting the Confusion Matrix

- The Confusion matrix is also defined as an inbuilt function in the TensorFlow module, so we can use that for evaluating the classification model.
- The Confusion matrix expects categorical data as input. However, y_test_encoded is an encoded value, whereas y_pred has probabilities. So,we must retrieve the categorical values from the encoded values.
- We will use the argmax() function to obtain the maximum value over each category on both y_test_encoded and y_pred and obtain their respective classes.

In [47]:
```
# 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 whic
h 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()
```



## Model-2

As we can see, our initial model appears to overfit. Therefore we'll try to address this problem with data augmentation and Batch Normalization to check if we can improve the model's performance.

## Data Augmentation

In most of the real-world case studies, it is challenging to acquire a large number of images and then train CNNs. To overcome this problem, one approach we might consider is Data Augmentation. CNNs have the property of translational invariance, which means they can recognise an object even if its appearance shifts translationally in some way. Taking this attribute into account, we can augment the images using the techniques listed below -

1. Horizontal Flip (should be set to True/False)
2. Vertical Flip (should be set to True/False)
3. Height Shift (should be between 0 and 1)
4. Width Shift (should be between 0 and 1)
5. Rotation (should be between 0 and 180)
6. Shear (should be between 0 and 1)
7. Zoom (should be between 0 and 1) etc.

In [48]:
```
# 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)
```

In [49]:
```
# All images to be rescaled by 1/255.
train_datagen = ImageDataGenerator(
                              rotation_range=20,
                              fill_mode='nearest'
                              )
# test_datagen  = ImageDataGenerator(rescale = 1.0/255.)
```

In [50]:
```
# Intializing a sequential model
model = Sequential()

# Adding first conv layer with 64 filters and kernel size 3x3 , paddin
g '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)        1792

 max_pooling2d (MaxPooling2D  (None, 32, 32, 64)       0
 )

 conv2d_1 (Conv2D)           (None, 32, 32, 32)        18464

 max_pooling2d_1 (MaxPooling  (None, 16, 16, 32)       0
 2D)

 batch_normalization (BatchN  (None, 16, 16, 32)       128
 ormalization)

 flatten (Flatten)           (None, 8192)              0

 dense (Dense)               (None, 16)                131088

 dropout (Dropout)           (None, 16)                0

 dense_1 (Dense)             (None, 12)                204

=================================================================
Total params: 151,676
Trainable params: 151,612
Non-trainable params: 64
_________________________________________________________________
```

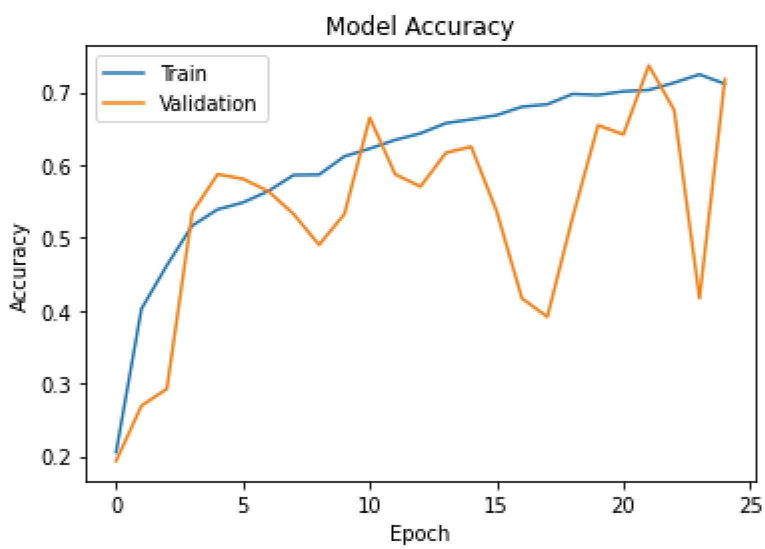
In [51]:
```
# Epochs
epochs = 25
# Batch size
batch_size = 64

history = model.fit(train_datagen.flow(X_train_normalized,y_train_enco
ded,
                                       batch_size=batch_size,
                                       seed=42,
                                       shuffle=False),
                    epochs=epochs,
                    steps_per_epoch=X_train_normalized.shape[0] // bat
ch_size,
                    validation_data=(X_test_normalized,y_test_encode
d),
                    verbose=1)
```

```
Epoch 1/25
66/66 [==============================] - 39s 577ms/step - loss: 2.2122
- accuracy: 0.2054 - val_loss: 2.4192 - val_accuracy: 0.1937
Epoch 2/25
66/66 [==============================] - 38s 568ms/step - loss: 1.6976
- accuracy: 0.4025 - val_loss: 2.2559 - val_accuracy: 0.2695
Epoch 3/25
66/66 [==============================] - 38s 575ms/step - loss: 1.5070
- accuracy: 0.4621 - val_loss: 2.1022 - val_accuracy: 0.2926
Epoch 4/25
66/66 [==============================] - 37s 565ms/step - loss: 1.3566
- accuracy: 0.5167 - val_loss: 1.9409 - val_accuracy: 0.5347
Epoch 5/25
66/66 [==============================] - 37s 564ms/step - loss: 1.2994
- accuracy: 0.5391 - val_loss: 1.7110 - val_accuracy: 0.5874
Epoch 6/25
66/66 [==============================] - 37s 563ms/step - loss: 1.2542
- accuracy: 0.5486 - val_loss: 1.7447 - val_accuracy: 0.5811
Epoch 7/25
66/66 [==============================] - 37s 566ms/step - loss: 1.2212
- accuracy: 0.5642 - val_loss: 1.6498 - val_accuracy: 0.5642
Epoch 8/25
66/66 [==============================] - 37s 564ms/step - loss: 1.1421
- accuracy: 0.5863 - val_loss: 1.3424 - val_accuracy: 0.5326
Epoch 9/25
66/66 [==============================] - 37s 560ms/step - loss: 1.1452
- accuracy: 0.5868 - val_loss: 1.4244 - val_accuracy: 0.4905
Epoch 10/25
66/66 [==============================] - 37s 564ms/step - loss: 1.0790
- accuracy: 0.6120 - val_loss: 1.3145 - val_accuracy: 0.5326
Epoch 11/25
66/66 [==============================] - 37s 563ms/step - loss: 1.0489
- accuracy: 0.6227 - val_loss: 1.0334 - val_accuracy: 0.6653
Epoch 12/25
66/66 [==============================] - 37s 566ms/step - loss: 0.9992
- accuracy: 0.6345 - val_loss: 1.5129 - val_accuracy: 0.5874
Epoch 13/25
66/66 [==============================] - 38s 567ms/step - loss: 0.9819
- accuracy: 0.6436 - val_loss: 1.2381 - val_accuracy: 0.5705
Epoch 14/25
66/66 [==============================] - 38s 569ms/step - loss: 0.9357
- accuracy: 0.6576 - val_loss: 1.5444 - val_accuracy: 0.6168
Epoch 15/25
66/66 [==============================] - 38s 570ms/step - loss: 0.9444
- accuracy: 0.6628 - val_loss: 1.1773 - val_accuracy: 0.6253
Epoch 16/25
66/66 [==============================] - 38s 577ms/step - loss: 0.9065
- accuracy: 0.6685 - val_loss: 2.2096 - val_accuracy: 0.5368
Epoch 17/25
66/66 [==============================] - 38s 575ms/step - loss: 0.8936
- accuracy: 0.6801 - val_loss: 3.0166 - val_accuracy: 0.4168
Epoch 18/25
66/66 [==============================] - 38s 578ms/step - loss: 0.8583
- accuracy: 0.6834 - val_loss: 2.8068 - val_accuracy: 0.3916
Epoch 19/25
66/66 [==============================] - 38s 579ms/step - loss: 0.8282
- accuracy: 0.6977 - val_loss: 1.5779 - val_accuracy: 0.5284
```

```
Epoch 20/25
66/66 [==============================] - 38s 576ms/step - loss: 0.8224
- accuracy: 0.6963 - val_loss: 1.3237 - val_accuracy: 0.6547
Epoch 21/25
66/66 [==============================] - 38s 571ms/step - loss: 0.8003
- accuracy: 0.7013 - val_loss: 1.1389 - val_accuracy: 0.6421
Epoch 22/25
66/66 [==============================] - 38s 575ms/step - loss: 0.8075
- accuracy: 0.7032 - val_loss: 0.9080 - val_accuracy: 0.7368
Epoch 23/25
66/66 [==============================] - 38s 567ms/step - loss: 0.7710
- accuracy: 0.7131 - val_loss: 1.0049 - val_accuracy: 0.6758
Epoch 24/25
66/66 [==============================] - 37s 566ms/step - loss: 0.7589
- accuracy: 0.7245 - val_loss: 2.8205 - val_accuracy: 0.4168
Epoch 25/25
66/66 [==============================] - 37s 564ms/step - loss: 0.7792
- accuracy: 0.7117 - val_loss: 0.8836 - val_accuracy: 0.7179
```

In [52]:

```
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()
```




In [53]:

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

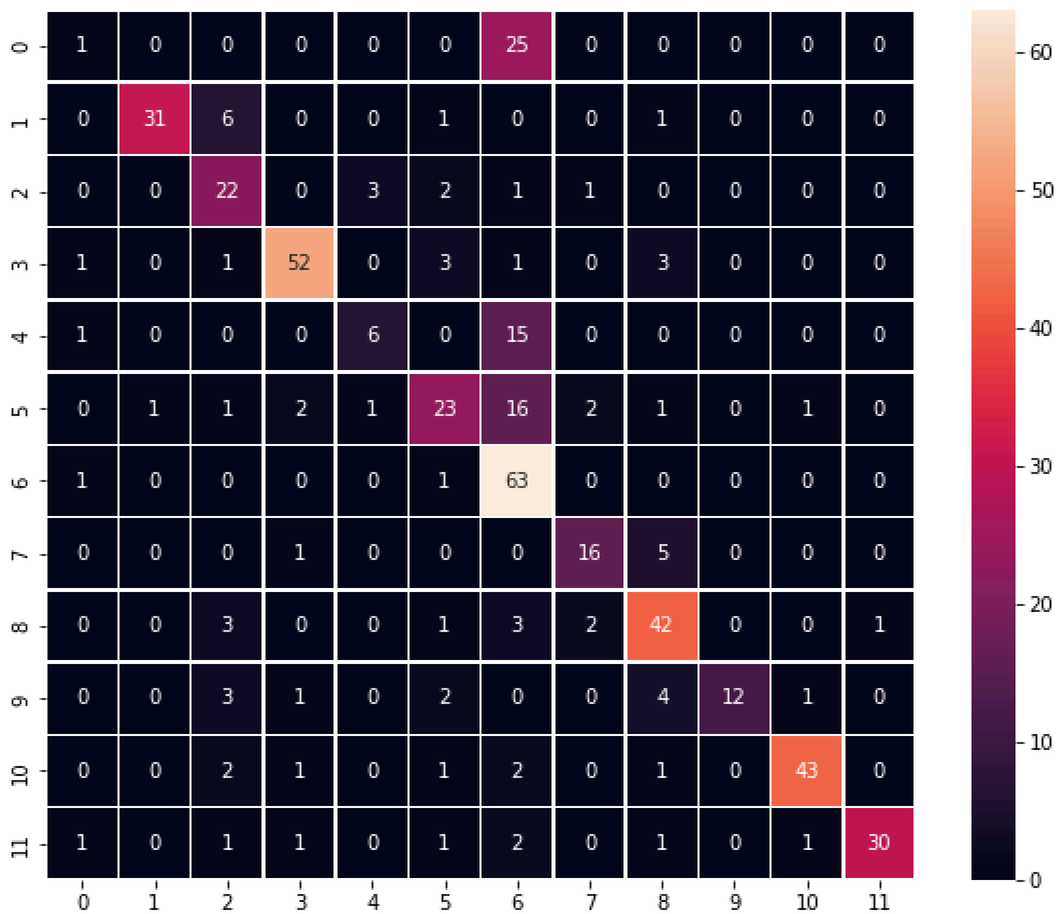
```
15/15 - 1s - loss: 0.8836 - accuracy: 0.7179 - 976ms/epoch - 65ms/step
```

*Observations:*

- We can observe that our accuracy has improved compared to our previous model.

In [54]:
```
# Here we would get the output as probablities for each category
y_pred=model.predict(X_test_normalized)
```

In [55]:
```
# 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 whic
h 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()
```




- We can observe that this model has outperformed our previous model.
- Let's try data augmentation model with different parameters.

## Model-3 with Data Augmentation (Different parameters)

Trying to create a new model with different set of parameters and image size

In [ ]:
```
#Resizing image to 256
images_resized=[]
height = 256
width = 256
dimensions = (width, height)
for i in range(len(images)):
  images_resized.append( cv2.resize(images[i], dimensions, interpolati
on=cv2.INTER_LINEAR))


# Applying Gaussian Blur to denoise the images
images_gb=[]
for i in range(len(images)):
  # gb[i] = cv2.cvtColor(images[i], cv2.COLOR_BGR2RGB)
  images_gb.append(cv2.GaussianBlur(images[i], ksize =(3,3),sigmaX =
0))

# Splitting
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(np.array(images_re
sized),labels , test_size=0.1, random_state=42,stratify=labels)

# Convert labels from names to one hot vectors.
# We have already used encoding methods like onehotencoder and labelen
coder earlier so now we will be using a new encoding method called lab
elBinarizer.
# Labelbinarizer works similar to onehotencoder
from sklearn.preprocessing import LabelBinarizer
enc = LabelBinarizer()
y_train_encoded = enc.fit_transform(y_train)
y_test_encoded=enc.transform(y_test)


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

In [ ]:
```
# Set the CNN model

batch_size = None

model = Sequential()

model.add(Conv2D(filters = 32, kernel_size = (5,5),padding = 'Same',
                 activation ='relu', batch_input_shape = (batch_size,256, 256, 3)))


model.add(Conv2D(filters = 32, kernel_size = (5,5),padding = 'Same',
                 activation ='relu'))
model.add(MaxPool2D(pool_size=(2,2)))
model.add(Dropout(0.2))


model.add(Conv2D(filters = 64, kernel_size = (3,3),padding = 'Same',
                 activation ='relu'))
model.add(Conv2D(filters = 64, kernel_size = (3,3),padding = 'same',
                 activation ='relu'))
model.add(MaxPool2D(pool_size=(2,2), strides=(2,2)))
model.add(Dropout(0.3))

model.add(Conv2D(filters = 128, kernel_size = (3,3),padding = 'Same',
                 activation ='relu'))
model.add(Conv2D(filters = 128, kernel_size = (3,3),padding = 'Same',
                 activation ='relu'))
model.add(MaxPool2D(pool_size=(2,2), strides=(2,2)))
model.add(Dropout(0.4))


model.add(GlobalMaxPooling2D())
model.add(Dense(256, activation = "relu"))
model.add(Dropout(0.5))
model.add(Dense(12, activation = "softmax"))
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, 256, 256, 32)      2432      
                                                                 
 conv2d_1 (Conv2D)           (None, 256, 256, 32)      25632     
                                                                 
 max_pooling2d (MaxPooling2D  (None, 128, 128, 32)     0         
 )                                                               
                                                                 
 dropout (Dropout)           (None, 128, 128, 32)      0         
                                                                 
 conv2d_2 (Conv2D)           (None, 128, 128, 64)      18496     
                                                                 
 conv2d_3 (Conv2D)           (None, 128, 128, 64)      36928     
                                                                 
 max_pooling2d_1 (MaxPooling  (None, 64, 64, 64)       0         
 2D)                                                             
                                                                 
 dropout_1 (Dropout)         (None, 64, 64, 64)        0         
                                                                 
 conv2d_4 (Conv2D)           (None, 64, 64, 128)       73856     
                                                                 
 conv2d_5 (Conv2D)           (None, 64, 64, 128)       147584    
                                                                 
 max_pooling2d_2 (MaxPooling  (None, 32, 32, 128)      0         
 2D)                                                             
                                                                 
 dropout_2 (Dropout)         (None, 32, 32, 128)       0         
                                                                 
 global_max_pooling2d (Globa  (None, 128)              0         
 lMaxPooling2D)                                                  
                                                                 
 dense (Dense)               (None, 256)               33024     
                                                                 
 dropout_3 (Dropout)         (None, 256)               0         
                                                                 
 dense_1 (Dense)             (None, 12)                3084      
                                                                 
=================================================================
Total params: 341,036
Trainable params: 341,036
Non-trainable params: 0
_________________________________________________________________
```

In [ ]:
```
train_datagen = ImageDataGenerator(
        # set input mean to 0 over the dataset
        featurewise_center=False,
        # set each sample mean to 0
        samplewise_center=False,
        # divide inputs by std of the dataset
        featurewise_std_normalization=False,
        # divide each input by its std
        samplewise_std_normalization=False,
        # apply ZCA whitening
        zca_whitening=False,
        # randomly rotate images in the range (degrees, 0 to 180)
        rotation_range=10,
        # Randomly zoom image
        zoom_range = 0.1,
        # randomly shift images horizontally (fraction of total width)
        width_shift_range=0.1,
        # randomly shift images vertically (fraction of total height)
        height_shift_range=0.1,
        # randomly flip images horizantally
        horizontal_flip=False,
        # randomly flip images vertically
        vertical_flip=False)


# test_datagen  = ImageDataGenerator(rescale = 1.0/255.)
```

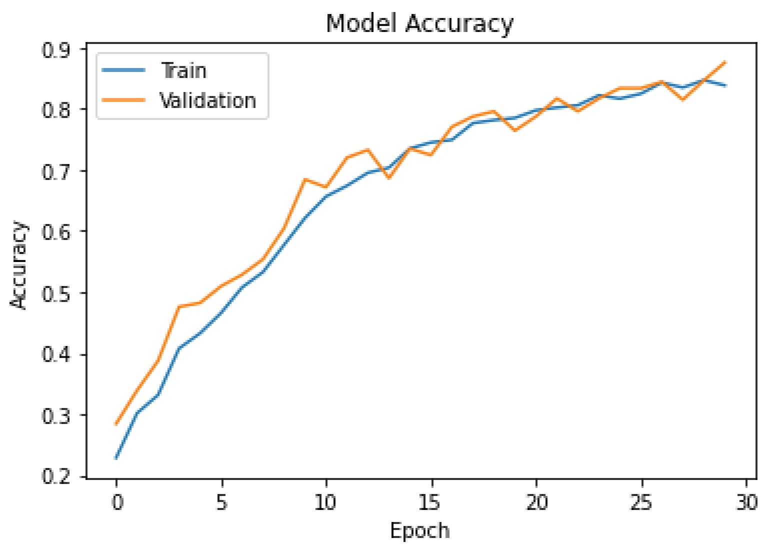
In [ ]:
```
# Epochs
epochs = 30
# Batch size
batch_size = 38

from keras.callbacks import ReduceLROnPlateau

learning_rate_reduction = ReduceLROnPlateau(monitor='val_accuracy',
                                            patience=3,
                                            verbose=1,
                                            factor=0.5,
                                            min_lr=0.00001)

history = model.fit(train_datagen.flow(X_train_normalized,y_train_enco
ded,
                                       batch_size=batch_size,
                                       seed=42,
                                       shuffle=False),
                    verbose = 1,
                    epochs=epochs,
                    steps_per_epoch=X_train_normalized.shape[0] // bat
ch_size,
                    validation_data=(X_test_normalized,y_test_encode
d),
                    callbacks=[learning_rate_reduction]
                    )
```

```
Epoch 1/30
112/112 [==============================] - 110s 830ms/step - loss: 2.1
967 - accuracy: 0.2285 - val_loss: 2.0857 - val_accuracy: 0.2842 - lr:
0.0010
Epoch 2/30
112/112 [==============================] - 82s 731ms/step - loss: 1.94
41 - accuracy: 0.3016 - val_loss: 1.9705 - val_accuracy: 0.3389 - lr:
0.0010
Epoch 3/30
112/112 [==============================] - 83s 736ms/step - loss: 1.85
40 - accuracy: 0.3314 - val_loss: 1.8695 - val_accuracy: 0.3874 - lr:
0.0010
Epoch 4/30
112/112 [==============================] - 81s 720ms/step - loss: 1.68
88 - accuracy: 0.4076 - val_loss: 1.7133 - val_accuracy: 0.4758 - lr:
0.0010
Epoch 5/30
112/112 [==============================] - 81s 723ms/step - loss: 1.61
17 - accuracy: 0.4324 - val_loss: 1.6310 - val_accuracy: 0.4821 - lr:
0.0010
Epoch 6/30
112/112 [==============================] - 82s 726ms/step - loss: 1.51
10 - accuracy: 0.4657 - val_loss: 1.5486 - val_accuracy: 0.5095 - lr:
0.0010
Epoch 7/30
112/112 [==============================] - 81s 724ms/step - loss: 1.41
58 - accuracy: 0.5074 - val_loss: 1.4271 - val_accuracy: 0.5284 - lr:
0.0010
Epoch 8/30
112/112 [==============================] - 81s 720ms/step - loss: 1.32
97 - accuracy: 0.5327 - val_loss: 1.4089 - val_accuracy: 0.5537 - lr:
0.0010
Epoch 9/30
112/112 [==============================] - 82s 730ms/step - loss: 1.24
02 - accuracy: 0.5771 - val_loss: 1.2205 - val_accuracy: 0.6042 - lr:
0.0010
Epoch 10/30
112/112 [==============================] - 82s 730ms/step - loss: 1.09
68 - accuracy: 0.6212 - val_loss: 1.0792 - val_accuracy: 0.6842 - lr:
0.0010
Epoch 11/30
112/112 [==============================] - 81s 722ms/step - loss: 0.98
24 - accuracy: 0.6564 - val_loss: 0.9993 - val_accuracy: 0.6716 - lr:
0.0010
Epoch 12/30
112/112 [==============================] - 83s 739ms/step - loss: 0.93
14 - accuracy: 0.6741 - val_loss: 0.8809 - val_accuracy: 0.7200 - lr:
0.0010
Epoch 13/30
112/112 [==============================] - 82s 732ms/step - loss: 0.87
88 - accuracy: 0.6953 - val_loss: 0.8996 - val_accuracy: 0.7326 - lr:
0.0010
Epoch 14/30
112/112 [==============================] - 83s 739ms/step - loss: 0.84
11 - accuracy: 0.7036 - val_loss: 0.9129 - val_accuracy: 0.6863 - lr:
0.0010
Epoch 15/30
```

```
112/112 [==============================] - 82s 731ms/step - loss: 0.76
43 - accuracy: 0.7350 - val_loss: 0.8144 - val_accuracy: 0.7347 - lr:
0.0010
Epoch 16/30
112/112 [==============================] - 83s 740ms/step - loss: 0.73
68 - accuracy: 0.7449 - val_loss: 0.8495 - val_accuracy: 0.7242 - lr:
0.0010
Epoch 17/30
112/112 [==============================] - 83s 741ms/step - loss: 0.70
73 - accuracy: 0.7491 - val_loss: 0.7565 - val_accuracy: 0.7705 - lr:
0.0010
Epoch 18/30
112/112 [==============================] - 84s 747ms/step - loss: 0.63
80 - accuracy: 0.7765 - val_loss: 0.6968 - val_accuracy: 0.7874 - lr:
0.0010
Epoch 19/30
112/112 [==============================] - 83s 738ms/step - loss: 0.63
42 - accuracy: 0.7814 - val_loss: 0.6435 - val_accuracy: 0.7958 - lr:
0.0010
Epoch 20/30
112/112 [==============================] - 84s 742ms/step - loss: 0.60
19 - accuracy: 0.7850 - val_loss: 0.7713 - val_accuracy: 0.7642 - lr:
0.0010
Epoch 21/30
112/112 [==============================] - 83s 738ms/step - loss: 0.57
24 - accuracy: 0.7975 - val_loss: 0.6836 - val_accuracy: 0.7874 - lr:
0.0010
Epoch 22/30
112/112 [==============================] - 83s 741ms/step - loss: 0.55
82 - accuracy: 0.8020 - val_loss: 0.6108 - val_accuracy: 0.8168 - lr:
0.0010
Epoch 23/30
112/112 [==============================] - 84s 743ms/step - loss: 0.54
26 - accuracy: 0.8058 - val_loss: 0.6366 - val_accuracy: 0.7958 - lr:
0.0010
Epoch 24/30
112/112 [==============================] - 84s 745ms/step - loss: 0.49
85 - accuracy: 0.8220 - val_loss: 0.6124 - val_accuracy: 0.8168 - lr:
0.0010
Epoch 25/30
112/112 [==============================] - 84s 743ms/step - loss: 0.51
11 - accuracy: 0.8166 - val_loss: 0.5653 - val_accuracy: 0.8337 - lr:
0.0010
Epoch 26/30
112/112 [==============================] - 84s 748ms/step - loss: 0.47
55 - accuracy: 0.8246 - val_loss: 0.5447 - val_accuracy: 0.8337 - lr:
0.0010
Epoch 27/30
112/112 [==============================] - 84s 745ms/step - loss: 0.42
50 - accuracy: 0.8428 - val_loss: 0.5220 - val_accuracy: 0.8442 - lr:
0.0010
Epoch 28/30
112/112 [==============================] - 84s 742ms/step - loss: 0.46
27 - accuracy: 0.8348 - val_loss: 0.5982 - val_accuracy: 0.8147 - lr:
0.0010
Epoch 29/30
112/112 [==============================] - 84s 743ms/step - loss: 0.43
```

```
96 - accuracy: 0.8468 - val_loss: 0.5478 - val_accuracy: 0.8463 - lr:
0.0010
Epoch 30/30
112/112 [==============================] - 84s 745ms/step - loss: 0.46
92 - accuracy: 0.8383 - val_loss: 0.4789 - val_accuracy: 0.8758 - lr:
0.0010
```

In [ ]:
```
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()
```



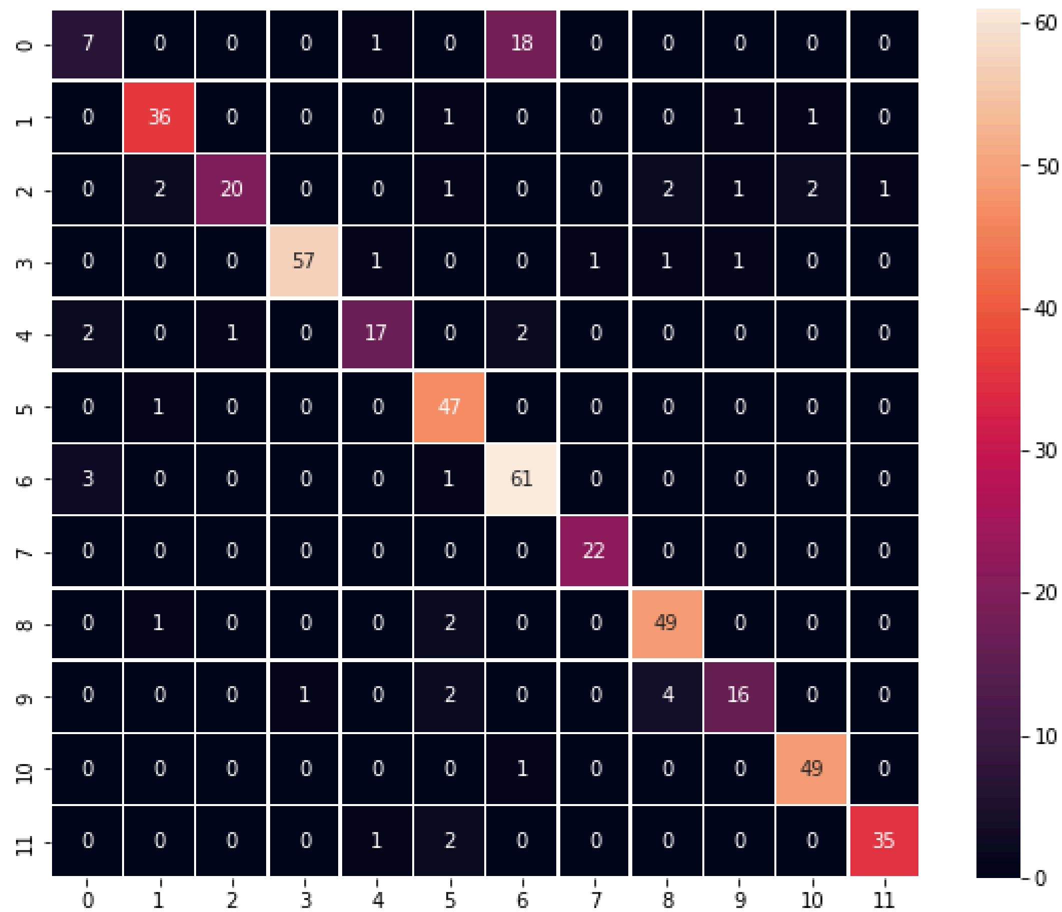

In [ ]:
```
accuracy = model.evaluate(X_test_normalized, y_test_encoded, verbose=
2)
```

```
15/15 - 2s - loss: 0.4789 - accuracy: 0.8758 - 2s/epoch - 150ms/step
```

We can observe that our accuracy has improved compared to our previous model.

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

In [ ]:
```
# 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 whic
h 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()
```




- We can observe that this model has outperformed our previous model.

# Transfer Learning using VGG16

Let's try again, but this time, we will be using the idea of Transfer Learning. We will be loading a pre-built architecture - VGG16, which was trained on the ImageNet dataset and is the runner-up in the ImageNet competition in 2014.

For training VGG16, we will directly use the convolutional and pooling layers and freeze their weights i.e. no training will be done on them. For classification, we will replace the existing fully-connected layers with FC layers created specifically for our problem.

In [ ]:
```
from tensorflow.keras.models import Model
from keras.applications.vgg16 import VGG16

vgg_model = VGG16(weights='imagenet', include_top = False, input_shape
= (256,256,3))
vgg_model.summary()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-
applications/vgg16/vgg16_weights_tf_dim_ordering_tf_kernels_notop.h5
58892288/58889256 [==============================] - 1s 0us/step
58900480/58889256 [==============================] - 1s 0us/step
Model: "vgg16"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 64, 64, 3)]       0

 block1_conv1 (Conv2D)       (None, 64, 64, 64)        1792

 block1_conv2 (Conv2D)       (None, 64, 64, 64)        36928

 block1_pool (MaxPooling2D)  (None, 32, 32, 64)        0

 block2_conv1 (Conv2D)       (None, 32, 32, 128)       73856

 block2_conv2 (Conv2D)       (None, 32, 32, 128)       147584

 block2_pool (MaxPooling2D)  (None, 16, 16, 128)       0

 block3_conv1 (Conv2D)       (None, 16, 16, 256)       295168

 block3_conv2 (Conv2D)       (None, 16, 16, 256)       590080

 block3_conv3 (Conv2D)       (None, 16, 16, 256)       590080

 block3_pool (MaxPooling2D)  (None, 8, 8, 256)         0

 block4_conv1 (Conv2D)       (None, 8, 8, 512)         1180160

 block4_conv2 (Conv2D)       (None, 8, 8, 512)         2359808

 block4_conv3 (Conv2D)       (None, 8, 8, 512)         2359808

 block4_pool (MaxPooling2D)  (None, 4, 4, 512)         0

 block5_conv1 (Conv2D)       (None, 4, 4, 512)         2359808

 block5_conv2 (Conv2D)       (None, 4, 4, 512)         2359808

 block5_conv3 (Conv2D)       (None, 4, 4, 512)         2359808

 block5_pool (MaxPooling2D)  (None, 2, 2, 512)         0

=================================================================
Total params: 14,714,688
Trainable params: 14,714,688
Non-trainable params: 0
_________________________________________________________________
```

In [ ]:

```
# Making all the layers of the VGG model non-trainable. i.e. freezing them
for layer in vgg_model.layers:
    layer.trainable = False
```

In [ ]:
```
new_model = Sequential()

# Adding the convolutional part of the VGG16 model from above
new_model.add(vgg_model)

# Flattening the output of the VGG16 model because it is from a convol
utional layer
new_model.add(Flatten())

# Adding a dense output layer
new_model.add(Dense(32, activation='relu'))
new_model.add(Dropout(0.2))
new_model.add(Dense(16, activation='relu'))
new_model.add(Dense(12, activation='softmax'))
opt=Adam()
# Compile model
new_model.compile(optimizer=opt, loss='categorical_crossentropy', metr
ics=['accuracy'])

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

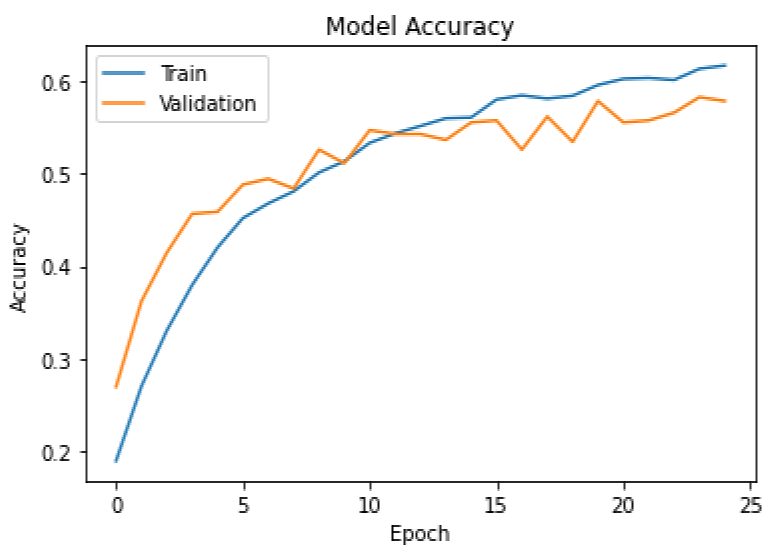
```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 vgg16 (Functional)          (None, 2, 2, 512)         14714688

 flatten_1 (Flatten)         (None, 2048)              0

 dense_2 (Dense)             (None, 32)                65568

 dropout_1 (Dropout)         (None, 32)                0

 dense_3 (Dense)             (None, 16)                528

 dense_4 (Dense)             (None, 12)                204

=================================================================
Total params: 14,780,988
Trainable params: 66,300
Non-trainable params: 14,714,688
_________________________________________________________________
```

In [ ]:
```
# Epochs
epochs = 25
# Batch size
batch_size = 64

history_vgg16 = new_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] // bat
ch_size,
                    validation_data=(X_test_normalized,y_test_encode
d),
                    verbose=1)
```

```
Epoch 1/25
66/66 [==============================] - 14s 166ms/step - loss: 2.3553
- accuracy: 0.1893 - val_loss: 2.1940 - val_accuracy: 0.2695
Epoch 2/25
66/66 [==============================] - 7s 108ms/step - loss: 2.1410
- accuracy: 0.2700 - val_loss: 2.0062 - val_accuracy: 0.3621
Epoch 3/25
66/66 [==============================] - 7s 109ms/step - loss: 1.9605
- accuracy: 0.3303 - val_loss: 1.8093 - val_accuracy: 0.4147
Epoch 4/25
66/66 [==============================] - 7s 110ms/step - loss: 1.8005
- accuracy: 0.3797 - val_loss: 1.6936 - val_accuracy: 0.4568
Epoch 5/25
66/66 [==============================] - 7s 108ms/step - loss: 1.6785
- accuracy: 0.4203 - val_loss: 1.6024 - val_accuracy: 0.4589
Epoch 6/25
66/66 [==============================] - 7s 110ms/step - loss: 1.5938
- accuracy: 0.4521 - val_loss: 1.5457 - val_accuracy: 0.4884
Epoch 7/25
66/66 [==============================] - 7s 108ms/step - loss: 1.5158
- accuracy: 0.4681 - val_loss: 1.4937 - val_accuracy: 0.4947
Epoch 8/25
66/66 [==============================] - 7s 108ms/step - loss: 1.4785
- accuracy: 0.4811 - val_loss: 1.4507 - val_accuracy: 0.4842
Epoch 9/25
66/66 [==============================] - 7s 108ms/step - loss: 1.4072
- accuracy: 0.5015 - val_loss: 1.3918 - val_accuracy: 0.5263
Epoch 10/25
66/66 [==============================] - 7s 108ms/step - loss: 1.3744
- accuracy: 0.5137 - val_loss: 1.3705 - val_accuracy: 0.5116
Epoch 11/25
66/66 [==============================] - 7s 110ms/step - loss: 1.3352
- accuracy: 0.5336 - val_loss: 1.3517 - val_accuracy: 0.5474
Epoch 12/25
66/66 [==============================] - 7s 108ms/step - loss: 1.2927
- accuracy: 0.5438 - val_loss: 1.3117 - val_accuracy: 0.5432
Epoch 13/25
66/66 [==============================] - 7s 108ms/step - loss: 1.2863
- accuracy: 0.5519 - val_loss: 1.3014 - val_accuracy: 0.5432
Epoch 14/25
66/66 [==============================] - 7s 108ms/step - loss: 1.2403
- accuracy: 0.5602 - val_loss: 1.3284 - val_accuracy: 0.5368
Epoch 15/25
66/66 [==============================] - 7s 108ms/step - loss: 1.2400
- accuracy: 0.5611 - val_loss: 1.2810 - val_accuracy: 0.5558
Epoch 16/25
66/66 [==============================] - 7s 108ms/step - loss: 1.2040
- accuracy: 0.5806 - val_loss: 1.2549 - val_accuracy: 0.5579
Epoch 17/25
66/66 [==============================] - 7s 108ms/step - loss: 1.1728
- accuracy: 0.5851 - val_loss: 1.2951 - val_accuracy: 0.5263
Epoch 18/25
66/66 [==============================] - 7s 111ms/step - loss: 1.1762
- accuracy: 0.5816 - val_loss: 1.2550 - val_accuracy: 0.5621
Epoch 19/25
66/66 [==============================] - 7s 108ms/step - loss: 1.1574
- accuracy: 0.5847 - val_loss: 1.2592 - val_accuracy: 0.5347
```

```
Epoch 20/25
66/66 [==============================] - 7s 111ms/step - loss: 1.1306
- accuracy: 0.5963 - val_loss: 1.1968 - val_accuracy: 0.5789
Epoch 21/25
66/66 [==============================] - 7s 109ms/step - loss: 1.1096
- accuracy: 0.6029 - val_loss: 1.2281 - val_accuracy: 0.5558
Epoch 22/25
66/66 [==============================] - 7s 109ms/step - loss: 1.1240
- accuracy: 0.6039 - val_loss: 1.2289 - val_accuracy: 0.5579
Epoch 23/25
66/66 [==============================] - 7s 108ms/step - loss: 1.0982
- accuracy: 0.6018 - val_loss: 1.2254 - val_accuracy: 0.5663
Epoch 24/25
66/66 [==============================] - 7s 111ms/step - loss: 1.0712
- accuracy: 0.6139 - val_loss: 1.2064 - val_accuracy: 0.5832
Epoch 25/25
66/66 [==============================] - 7s 108ms/step - loss: 1.0565
- accuracy: 0.6174 - val_loss: 1.1593 - val_accuracy: 0.5789
```

In [ ]:
```
plt.plot(history_vgg16.history['accuracy'])
plt.plot(history_vgg16.history['val_accuracy'])
plt.title('Model Accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='upper left')
plt.show()
```

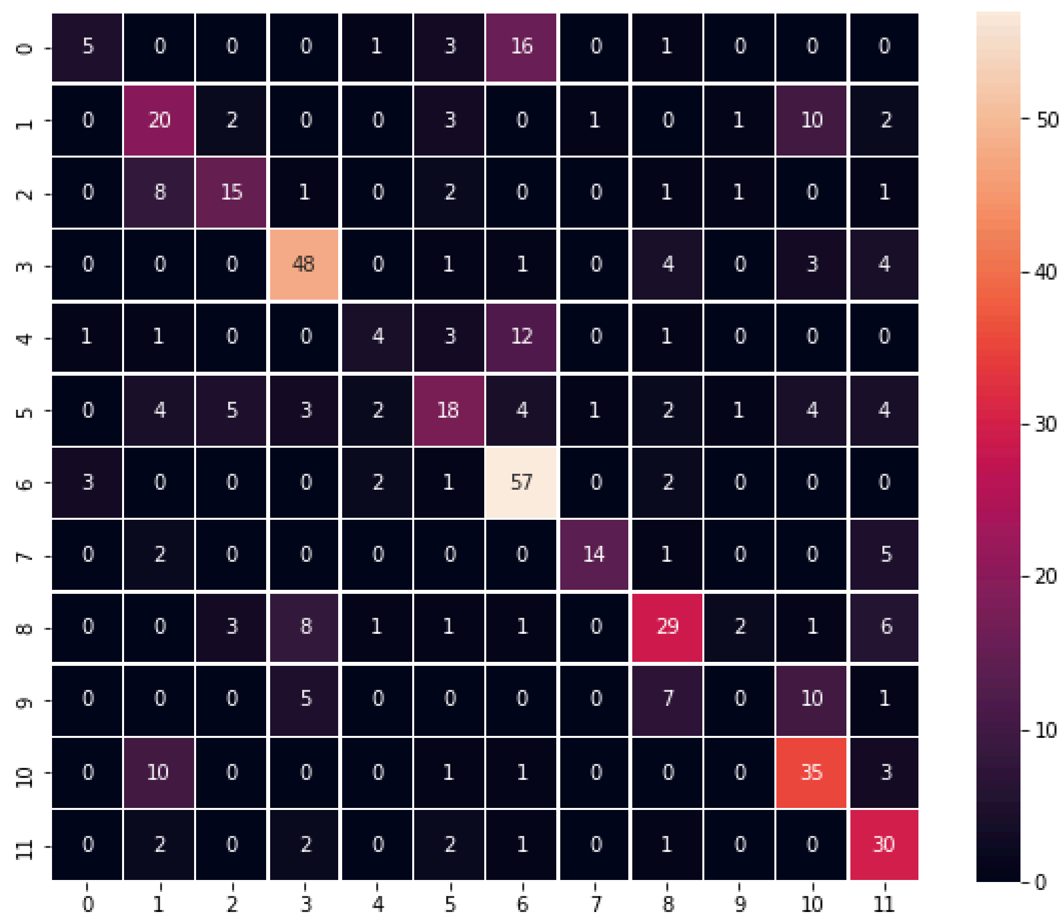



In [ ]:
```
accuracy = new_model.evaluate(X_test_normalized, y_test_encoded, verbo
se=2)
```

```
15/15 - 1s - loss: 1.1593 - accuracy: 0.5789 - 644ms/epoch - 43ms/step
```

In [ ]:
```
# Here we would get the output as probablities for each category
y_pred=new_model.predict(X_test_normalized)
```

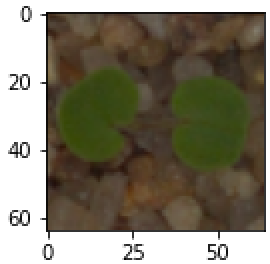
In [ ]:
```
# Obtaining the categorical values from y_test_encoded and y_pred
y_pred_arg=np.argmax(y_pred,axis=1)
y_test_arg=np.argmax(y_test_encoded,axis=1)

# Plotting the Confusion Matrix using confusion matrix() function whic
h 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()
```



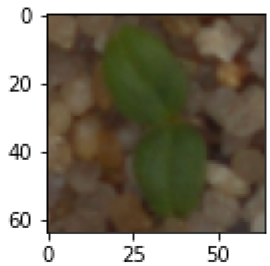

- According to the confusion matrix and accuracy curve, the VGG16 model does not outperform Model-2. This could be due to the data we're using; since we're using monkey species data, there's a chance that these images aren't in the ImageNet dataset, whose weights have been used to build our CNN model.
- Thus we can say that Model-3 is our best model and we can use this model to predict and visualize some test images.

## Visualizing the prediction:

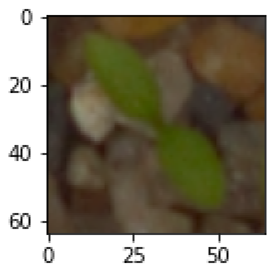
In [ ]:
```
# Visualizing the predicted and correct label of images from test data
plt.figure(figsize=(2,2))
plt.imshow(X_test[2])
plt.show()
# reshaping the input image as we are only trying to predict using a s
ingle image
print('Predicted Label', enc.inverse_transform(model.predict((X_test_n
ormalized[2].reshape(1,64,64,3)))))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

plt.figure(figsize=(2,2))
plt.imshow(X_test[33])
plt.show()
# reshaping the input image as we are only trying to predict using a s
ingle image
print('Predicted Label', enc.inverse_transform(model.predict((X_test_n
ormalized[33].reshape(1,64,64,3)))))
# using inverse_transform() to get the output label from the output ve
ctor
print('True Label', enc.inverse_transform(y_test_encoded)[33])

plt.figure(figsize=(2,2))
plt.imshow(X_test[59],)
plt.show()
# reshaping the input image as we are only trying to predict using a s
ingle image
print('Predicted Label', enc.inverse_transform(model.predict((X_test_n
ormalized[59].reshape(1,64,64,3)))))
# using inverse_transform() to get the output label from the output ve
ctor
print('True Label', enc.inverse_transform(y_test_encoded)[59])

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



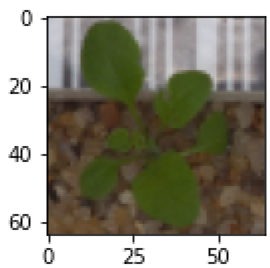
```
Predicted Label ['Small-flowered Cranesbill']
True Label Small-flowered Cranesbill
```



```
Predicted Label ['Cleavers']
True Label Cleavers
```



```
Predicted Label ['Common Chickweed']
True Label Common Chickweed
```



```
Predicted Label ['Shepherds Purse']
True Label Shepherds Purse
```

*Observations:*

- Out of four above predictions we got all 4 predictions right.

## Conclusion

- We can observe from the confusion matrix of all the models that our third model was the best model because it predicted the majority of the classes better than the other models.
- The test accuracy of the third model is approx 84%.
- Data Augmentation has also helped in improving the model.
- Although VGG16 did not outperform Model-3, it is evident that simply employing the transfer learning model can produce a better outcome than any ordinary CNN.

In [56]:

```
pd.DataFrame({'Models':['Base CNN Model','CNN Model with Data Augmentation','CNN Model with Data Augmentation - Diff parameter','Transfer Learning Model'],'Train Accuracy':['52%','71%','84%','62%'],'Test Accuracy':['63%','72%','87%','58%']})
```

Out[56]:

| | Models | Train Accuracy | Test Accuracy |
|---|---|---|---|
| **0** | Base CNN Model | 52% | 63% |
| **1** | CNN Model with Data Augmentation | 71% | 72% |
| **2** | CNN Model with Data Augmentation - Diff parameter | 84% | 87% |
| **3** | Transfer Learning Model | 62% | 58% |

## Scope of Improvement

- These models can be further improved by training with different filter sizes and different number of filters.
- These models can also be trained on the original image_size i.e 128 x 128 rather than being reduced to 64 and increased to 256.
- Data Augmentation can be performed more and dropout_rate can be changed to improve the model performance.
- Other Transfer Learning architectures can also be used to train the CNN model and these models can be used for classification.