Convolutional Neural Networks: Plant Seedlings Classification

---

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 **use a deep learning model to classify plant seedlings through supervised learning.**

Data Description:

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 at several growth stages.**

You are provided with a dataset of images of plant seedlings at various stages of growth.

- Each image has a filename that is its unique id.
- The dataset comprises of 12 plant species.
- The goal of the project is to create a classifier capable of determining a plant's species from a photo.

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

Dataset:

- The dataset can be download from Olympus.
- The data file names are:
  - images.npy
  - Label.csv
- The original files are from Kaggle. 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.
- Kaggle project link: https://www.kaggle.com/c/plant-seedlings-classification/data?select=train

Note: Download the dataset provided on Olympus

Learning Outcomes:

- Pre-processing of image data
- Visualization of images
- Building the CNN
- Evaluating the Model

Note: We are covering some additional concepts for the case study(Masking, HSV Conversion, Data Augmentation etc.)

Importing the necessary libraries

!pip install opencv-python

!pip install opencv-contrib-python

In [1]:
```
import os
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import math
import cv2
from glob import glob #
import itertools
import seaborn as sns
import numpy as np


import tensorflow as tf
from tensorflow.keras.models import Sequential,Model # Sequential api for se
from tensorflow.keras.layers import Dense, Dropout, Flatten, Conv2D, MaxPool
from tensorflow.keras.layers import BatchNormalization, Activation, Input, L
from tensorflow.keras import backend as K
from tensorflow.keras.utils import to_categorical # To perform one-hot encod
from tensorflow.keras import losses, optimizers
from tensorflow.keras.preprocessing.image import ImageDataGenerator

from sklearn import preprocessing
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix

seed = 42
```

Reading the dataset

In [2]:
```
# Load the image file of dataset
images = np.load('images.npy')

# Load the labels file of dataset
labels = pd.read_csv('Labels.csv')
```

Overview of the dataset

Let's print the shape of the images and labels

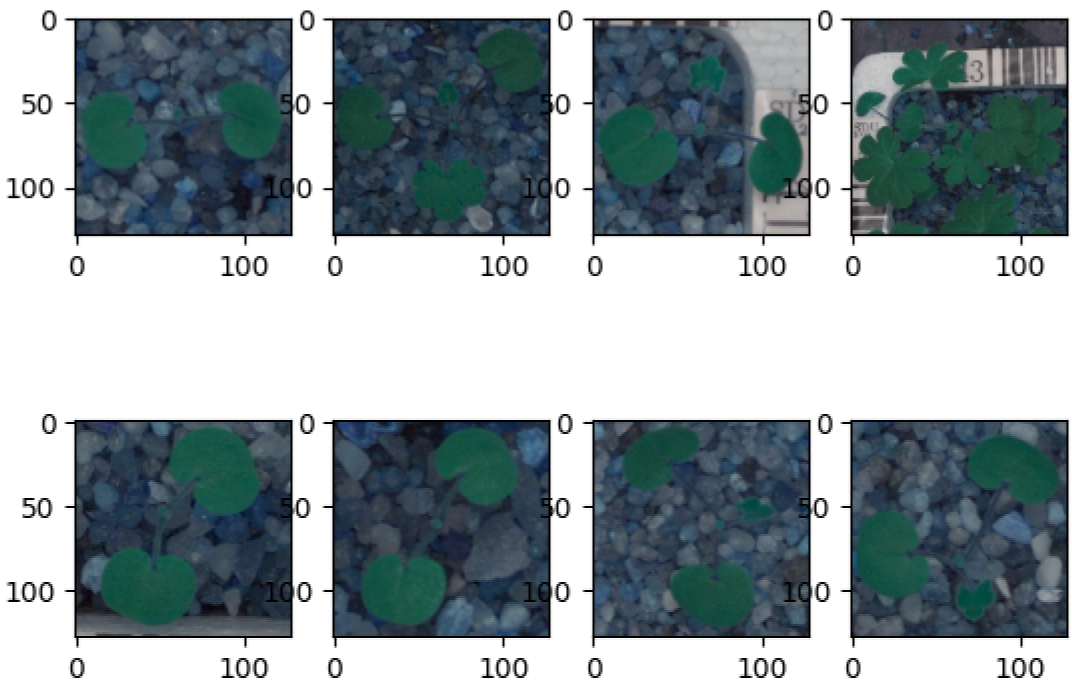
In [3]:
```
print(images.shape)
print(labels.shape)
```

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

- There are 4750 RGB images of shape 128 x 128 each. As mentioned, each image is an RGB image having 3 channels

Let's plot the first 8 images

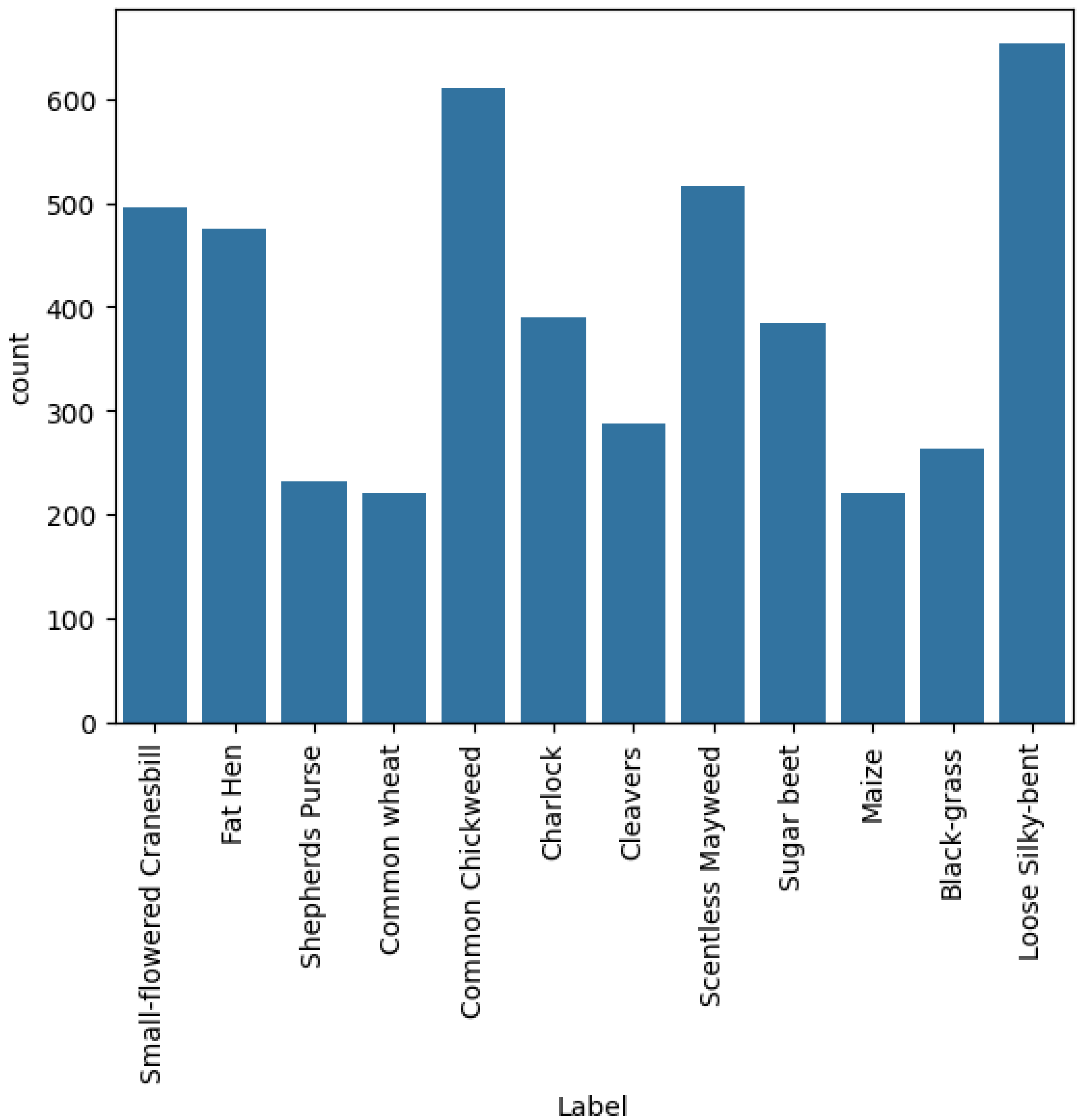
In [4]:
```
# Show some example images
for i in range(8):
    plt.subplot(2, 4, i + 1)
    plt.imshow(images[i])
```



Let's understand if the dataset is imbalanced or not

In [5]:
```
sns.countplot(x=labels['Label'])
plt.xticks(rotation='vertical')
```

Out[5]:
```
([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11],
 [Text(0, 0, 'Small-flowered Cranesbill'),
  Text(1, 0, 'Fat Hen'),
  Text(2, 0, 'Shepherds Purse'),
  Text(3, 0, 'Common wheat'),
  Text(4, 0, 'Common Chickweed'),
  Text(5, 0, 'Charlock'),
  Text(6, 0, 'Cleavers'),
  Text(7, 0, 'Scentless Mayweed'),
  Text(8, 0, 'Sugar beet'),
  Text(9, 0, 'Maize'),
  Text(10, 0, 'Black-grass'),
  Text(11, 0, 'Loose Silky-bent')])
```



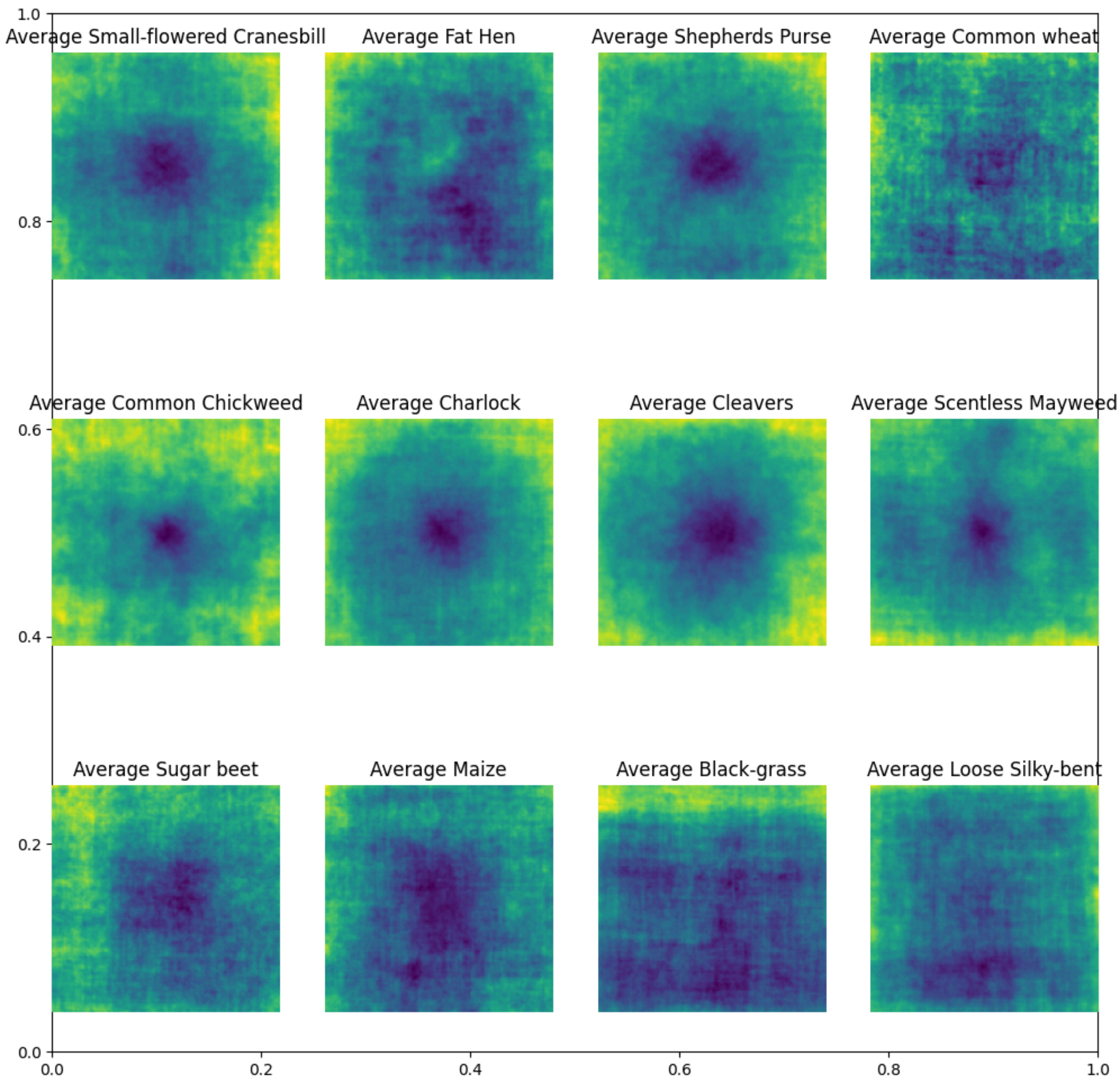
- As you can see from the above plot, the dataset is quite balanced. But, we might need to stratify the train_test_split.

Exploratory Data Analysis

Plotting mean images

In [6]:
```
def find_mean_img(full_mat):
    # Calculate the average
    mean_img = np.mean(full_mat, axis = 0)
    # Reshape it back to a matrix
    mean_img = mean_img.reshape((150,150))

    return mean_img

CATEGORIES=labels['Label'].unique()
```

```
d={ i:[] for i in CATEGORIES}

for i in labels.index:
  gray = cv2.cvtColor(images[i], cv2.COLOR_BGR2GRAY)
  gray = cv2.resize(gray,(150,150))
  d[labels['Label'][i]].append(gray)

l=[]
for i in d.keys():
  l.append(find_mean_img(d[i]))

plt.subplots(figsize=(12,12))
for i in range(len(l)):
    plt.subplot(3,4,i + 1,title='Average '+list(d.keys())[i])
    plt.imshow(l[i])
    plt.axis('off')
```



- From the above plots, we can say that **small-flowered cranesbill**, **Shepherds Purse**, **Charlock**, and **Cleavers** have similar kinds of shapes like width and length of

the leaves. They mostly have smaller lengths and widths.
- From the **Average Fat Hen graph**, we can observe that the length of the leaf is large and the width of the leaf is small.
- **Common Wheat**, **Silky bent**, and **Blackgrass** possess a similar kind of structure, their width is narrower and the length of the leaf is longer as compared to others. It's a kind of grass, not a leaf.
- **Mayweed** and **Common Chickweed** have a similar kind of structure, Their leaves have smaller lengths and widths comparing to the **Charlock** category.
- **Sugar beet** and **Maize** have a similar structure where the length of the leaves is more but the width is less.

Data Preprocessing: Image Preprocessing

Applying image processing

It is a very important step to perform image preprocessing. Preprocessing an image is an intensive task. We will be performing the following steps in order to process the images

Convert the RGB images into HSV

The HSV model describes colors similarly to how the human eye tends to perceive color. RGB defines color in terms of a combination of primary colors. In situations where color description plays an integral role, the HSV color model is often preferred over the RGB model.

'Hue' represents the color, **'Saturation'** represents the amount to which that respective color is mixed with white and **'Value'** represents the amount to which that respective color is mixed with black (Gray level).

HSV color space is more often used in computer vision owing to its **superior performance** compared to RGB color space in varying illumination levels. Often thresholding and masking is done in HSV color space. So it is very important to know the HSV values of the color we want to filter out.

In RGB, we cannot separate color information from luminance. HSV or Hue Saturation Value is used to separate image luminance from color information.

In this problem, color is an important factor in identifying the plant species. Hence converting BGR TO HSV is a good idea.

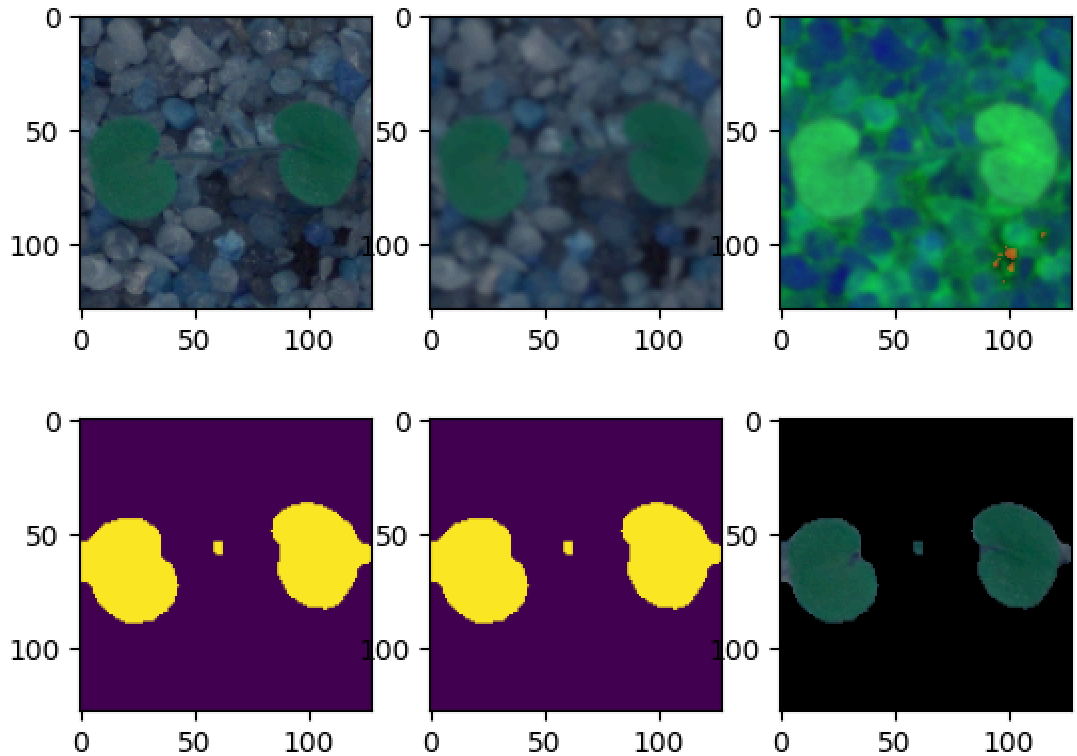
In order to remove the noise, we will have to blur the images (Gaussian Blurring)

In order to remove the background, we will have to create a mask.

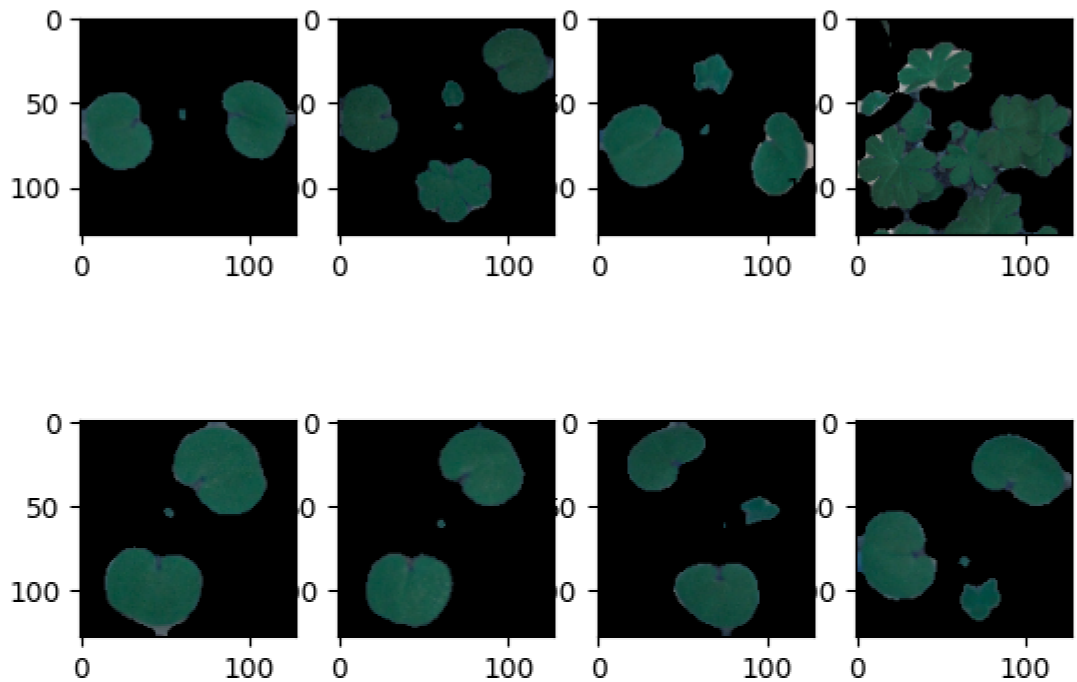
Creating a mask will remove the noise. We have then applied some morphological transformation (closing=Closing is reverse of Opening, Dilation followed by Erosion. It is useful in closing small holes inside the foreground objects, or small black points on the object.) to remove the imperfections from the image.

Note: OpenCV reads in images in the BGR format (instead of RGB) because when OpenCV was first being developed, the BGR color format was popular among camera manufacturers and image software providers. The red channel was considered one of the least important color channels, so was listed last, and many bitmaps use the BGR format for image storage. However, now the standard has changed and most image software and cameras use the RGB format, which is why, in programs, it's good practice to initially convert BGR images to RGB before analyzing or manipulating any images.

In [7]:
```
new_train = []
sets = []; getEx = True
for i in images:
    blurr = cv2.GaussianBlur(i,(5,5),0)
    hsv = cv2.cvtColor(blurr,cv2.COLOR_BGR2HSV) #Using BGR TO HSV conversior
    #HSV Bou daries for the Green color (GREEN PARAMETERS)
    lower = (25,40,50)
    upper = (75,255,255)
    mask = cv2.inRange(hsv,lower,upper) # create a mask
    struc = cv2.getStructuringElement(cv2.MORPH_ELLIPSE,(11,11)) #getting st
    mask = cv2.morphologyEx(mask,cv2.MORPH_CLOSE,struc) # applying morpholog
    boolean = mask>0
    new = np.zeros_like(i,np.uint8)
    new[boolean] = i[boolean]
    new_train.append(new)
    if getEx:
        plt.subplot(2,3,1);plt.imshow(i) # ORIGINAL
        plt.subplot(2,3,2);plt.imshow(blurr) # BLURRED
        plt.subplot(2,3,3);plt.imshow(hsv) # HSV CONVERTED
        plt.subplot(2,3,4);plt.imshow(mask) # MASKED
        plt.subplot(2,3,5);plt.imshow(boolean) # BOOLEAN MASKED
        plt.subplot(2,3,6);plt.imshow(new) # NEW PROCESSED IMAGE
        plt.show()
        getEx = False
new_train = np.asarray(new_train)
print("# CLEANED IMAGES")
for i in range(8):
    plt.subplot(2,4,i+1)
    plt.imshow(new_train[i])
```



```
# CLEANED IMAGES
```



Normalizing the dataset

In [8]:
```
# Normalize image data.
new_train = new_train / 255
```

Making the data compatible

- Convert labels from digits to one hot vectors.

- Check the shape of the data. Reshape the data into shapes compatible with Keras models, if not already compatible.

In [9]:
```
# Convert labels from digits to one hot vectors.
from sklearn.preprocessing import LabelBinarizer

enc = LabelBinarizer()
y = enc.fit_transform(labels)
```

In [10]:
```
print(y.shape)
print(new_train.shape)
```

```
(4750, 12)
(4750, 128, 128, 3)
```

Splitting the dataset

In this step, we are going to split the training dataset for validation. We are using the train_test_split() function from scikit-learn. Here we are splitting the dataset keeping the test_size=0.1. It means 10% of total data is used as testing data and the other 90% as training data. Check the below code for splitting the dataset.

In [11]:
```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(new_train,y , test_size=
```

In [12]:
```
print(X_train.shape)
print(y_train.shape)
print(X_test.shape)
print(y_test.shape)
```

```
(4275, 128, 128, 3)
(4275, 12)
(475, 128, 128, 3)
(475, 12)
```

We have used **stratify** to get an almost equal distribution of classes in the test and train set. Checking the class proportion for both the sets.

In [13]:
```
pd.DataFrame(y_train.argmax(axis=1)).value_counts()/pd.DataFrame(y_train.arg
```

Out[13]:
```
0
6     0.137778
3     0.128655
8     0.108538
10    0.104327
5     0.100117
1     0.082105
11    0.080936
2     0.060351
0     0.055439
9     0.048655
4     0.046550
7     0.046550
Name: count, dtype: float64
```

In [14]:
```
pd.DataFrame(y_test.argmax(axis=1)).value_counts()/pd.DataFrame(y_test.argma
```

Out[14]:
```
0
6     0.136842
3     0.128421
8     0.109474
10    0.105263
5     0.098947
1     0.082105
11    0.082105
2     0.061053
0     0.054737
9     0.048421
4     0.046316
7     0.046316
Name: count, dtype: float64
```

So, we can see above that the data was already compatible with Keras, as the shape of the data before and after reshaping is the same.

Building the CNN

- Define layers.
- Set the optimizer and loss function.

#Convolutional Layer-1

- This code was replaced bellow due to errors generated.
- model.add(Conv2D(filters = 32, kernel_size = (5,5),padding = 'Same', activation ='relu', batch_input_shape = (None,128, 128, 3)))
- ValueError: Unrecognized keyword arguments passed to Conv2D: {'batch_input_shape': (None, 128, 128, 3)}

Set the CNN model

model = Sequential()

Convolutional Layer-1

model.add(Conv2D(filters = 32, kernel_size = (5,5),padding = 'Same', activation ='relu', batch_input_shape = (None,128, 128, 3)))

Convolutional Layer-2

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

Max Pooling

model.add(MaxPool2D(pool_size=(2,2))) model.add(Dropout(0.2

Convolutional Layer-3

model.add(Conv2D(filters = 64, kernel_size = (3,3),padding = 'Same', activation ='relu'))

Convolutional Layer-4

model.add(Conv2D(filters = 64, kernel_size = (3,3),padding = 'same', activation ='relu'))

Max Pooling

model.add(MaxPool2D(pool_size=(2,2), strides=(2,2))) model.add(Dropout(0.3))

Convolutional Layer-5

model.add(Conv2D(filters = 128, kernel_size = (3,3),padding = 'Same', activation ='relu'))

Convolutional Layer-6

model.add(Conv2D(filters = 128, kernel_size = (3,3),padding = 'Same', activation ='relu'))

Max Pooling

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")) model.summary()

Key Changes & Notes:

```
*1.batch_input_shape → input_shape:
    Use input_shape=(128, 128, 3) only in the first layer.
Subsequent
    layers automatically infer the shape.

*2.Consistent Padding:
    Changed all padding to 'same' for consistency (both
'Same' and 'same' work, but lowercase is standard).

*3.MaxPooling & Dropout Layers:
    No changes needed here; they were correctly placed after
convolutional blocks.

*4.Imports:
    Included necessary imports for clarity.

*5.Model Initialization:
    Make sure to initialize Sequential() before adding
layers.
```

In [15]:

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPool2D, Dropout

# Initialize the model
model = Sequential()

# Convolutional Layer-1
model.add(Conv2D(filters=32, kernel_size=(5, 5), padding='same',
                 activation='relu', input_shape=(128, 128, 3)))

# Convolutional Layer-2
model.add(Conv2D(filters=32, kernel_size=(5, 5), padding='same',
```

```python
                 activation='relu'))

# Max Pooling + Dropout
model.add(MaxPool2D(pool_size=(2, 2)))
model.add(Dropout(0.2))

# Convolutional Layer-3
model.add(Conv2D(filters=64, kernel_size=(3, 3), padding='same',
                 activation='relu'))

# Convolutional Layer-4
model.add(Conv2D(filters=64, kernel_size=(3, 3), padding='same',
                 activation='relu'))

# Max Pooling + Dropout
model.add(MaxPool2D(pool_size=(2, 2), strides=(2, 2)))
model.add(Dropout(0.3))

# Convolutional Layer-5
model.add(Conv2D(filters=128, kernel_size=(3, 3), padding='same',
                 activation='relu'))

# Convolutional Layer-6
model.add(Conv2D(filters=128, kernel_size=(3, 3), padding='same',
                 activation='relu'))

# Max Pooling + Dropout
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"))
model.summary()
```

```
/Users/obaozai/miniconda3/envs/my_env/lib/python3.11/site-packages/keras/sr
c/layers/convolutional/base_conv.py:107: UserWarning: Do not pass an `input_
shape`/`input_dim` argument to a layer. When using Sequential models, prefer
using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
2025-02-20 06:25:22.302096: I metal_plugin/src/device/metal_device.cc:1154]
Metal device set to: Apple M3 Max
2025-02-20 06:25:22.302122: I metal_plugin/src/device/metal_device.cc:296] s
ystemMemory: 36.00 GB
2025-02-20 06:25:22.302130: I metal_plugin/src/device/metal_device.cc:313] m
axCacheSize: 13.50 GB
WARNING: All log messages before absl::InitializeLog() is called are written
to STDERR
I0000 00:00:1740054322.302142 20592676 pluggable_device_factory.cc:305] Coul
d not identify NUMA node of platform GPU ID 0, defaulting to 0. Your kernel
may not have been built with NUMA support.
I0000 00:00:1740054322.302161 20592676 pluggable_device_factory.cc:271] Crea
ted TensorFlow device (/job:localhost/replica:0/task:0/device:GPU:0 with 0 M
B memory) -> physical PluggableDevice (device: 0, name: METAL, pci bus id: <
undefined>)
```

Model: "sequential"

Layer (type)	**Output Shape**	**Par**
conv2d (Conv2D)	(None, 128, 128, 32)	2
conv2d_1 (Conv2D)	(None, 128, 128, 32)	25
max_pooling2d (MaxPooling2D)	(None, 64, 64, 32)	
dropout (Dropout)	(None, 64, 64, 32)	
conv2d_2 (Conv2D)	(None, 64, 64, 64)	18
conv2d_3 (Conv2D)	(None, 64, 64, 64)	36
max_pooling2d_1 (MaxPooling2D)	(None, 32, 32, 64)	
dropout_1 (Dropout)	(None, 32, 32, 64)	
conv2d_4 (Conv2D)	(None, 32, 32, 128)	73
conv2d_5 (Conv2D)	(None, 32, 32, 128)	147
max_pooling2d_2 (MaxPooling2D)	(None, 16, 16, 128)	
dropout_2 (Dropout)	(None, 16, 16, 128)	
global_max_pooling2d (GlobalMaxPooling2D)	(None, 128)	
dense (Dense)	(None, 256)	33
dropout_3 (Dropout)	(None, 256)	
dense_1 (Dense)	(None, 12)	3

Total params: 341,036 (1.30 MB)
Trainable params: 341,036 (1.30 MB)
Non-trainable params: 0 (0.00 B)

Defining thE optimizer and loss function

model.compile(optimizer = optimizers.legacy.RMSprop(learning_rate=0.001, rho=0.9, epsilon=1e-08, decay=0.0), loss = "categorical_crossentropy", metrics = ["accuracy"])

In Keras 3, the legacy submodule is deprecated.

- Use optimizers.RMSprop directly instead of optimizers.legacy.RMSprop.

•   Your parameters (learning_rate, rho, epsilon, decay) remain valid in the updated optimizer.

In [16]:
```
from tensorflow.keras import optimizers

# Compile the model using the updated RMSprop optimizer
model.compile(
    optimizer=optimizers.RMSprop(learning_rate=0.001, rho=0.9, epsilon=1e-08
    loss="categorical_crossentropy",
    metrics=["accuracy"]
)
```

In [17]:
```
# Fitting the model with epochs = 50
history=model.fit(X_train, y_train, epochs = 50, validation_split=0.1,batch_
```

```
Epoch 1/50
```

```
2025-02-20 06:25:23.679853: I tensorflow/core/grappler/optimizers/custom_gra
ph_optimizer_registry.cc:117] Plugin optimizer for device_type GPU is enable
d.
```

```
121/121 ━━━━━━━━━━━━━━━━━━━━ 9s 45ms/step - accuracy: 0.1732 - loss: 2.3709
- val_accuracy: 0.3037 - val_loss: 2.0931
Epoch 2/50
121/121 ━━━━━━━━━━━━━━━━━━━━ 4s 36ms/step - accuracy: 0.2536 - loss: 2.0934
- val_accuracy: 0.3201 - val_loss: 1.8421
Epoch 3/50
121/121 ━━━━━━━━━━━━━━━━━━━━ 4s 36ms/step - accuracy: 0.3404 - loss: 1.8936
- val_accuracy: 0.3972 - val_loss: 1.6188
Epoch 4/50
121/121 ━━━━━━━━━━━━━━━━━━━━ 4s 37ms/step - accuracy: 0.4198 - loss: 1.7079
- val_accuracy: 0.4346 - val_loss: 1.6485
Epoch 5/50
121/121 ━━━━━━━━━━━━━━━━━━━━ 4s 37ms/step - accuracy: 0.4661 - loss: 1.5781
- val_accuracy: 0.4206 - val_loss: 1.7587
Epoch 6/50
121/121 ━━━━━━━━━━━━━━━━━━━━ 4s 37ms/step - accuracy: 0.5119 - loss: 1.4544
- val_accuracy: 0.5397 - val_loss: 1.2833
Epoch 7/50
121/121 ━━━━━━━━━━━━━━━━━━━━ 4s 37ms/step - accuracy: 0.5631 - loss: 1.3228
- val_accuracy: 0.4182 - val_loss: 1.6725
Epoch 8/50
121/121 ━━━━━━━━━━━━━━━━━━━━ 4s 37ms/step - accuracy: 0.5879 - loss: 1.2568
- val_accuracy: 0.5280 - val_loss: 1.2474
Epoch 9/50
121/121 ━━━━━━━━━━━━━━━━━━━━ 4s 37ms/step - accuracy: 0.6505 - loss: 1.1088
- val_accuracy: 0.6192 - val_loss: 1.0817
Epoch 10/50
121/121 ━━━━━━━━━━━━━━━━━━━━ 4s 37ms/step - accuracy: 0.6592 - loss: 1.0520
- val_accuracy: 0.7523 - val_loss: 0.7874
Epoch 11/50
121/121 ━━━━━━━━━━━━━━━━━━━━ 4s 37ms/step - accuracy: 0.6817 - loss: 0.9623
- val_accuracy: 0.6589 - val_loss: 0.9343
Epoch 12/50
121/121 ━━━━━━━━━━━━━━━━━━━━ 4s 37ms/step - accuracy: 0.7265 - loss: 0.8333
- val_accuracy: 0.7360 - val_loss: 0.7973
Epoch 13/50
121/121 ━━━━━━━━━━━━━━━━━━━━ 4s 37ms/step - accuracy: 0.7205 - loss: 0.8957
- val_accuracy: 0.6308 - val_loss: 1.1155
Epoch 14/50
121/121 ━━━━━━━━━━━━━━━━━━━━ 4s 37ms/step - accuracy: 0.7340 - loss: 0.8602
- val_accuracy: 0.6846 - val_loss: 0.9436
Epoch 15/50
121/121 ━━━━━━━━━━━━━━━━━━━━ 4s 37ms/step - accuracy: 0.7553 - loss: 0.7996
- val_accuracy: 0.6636 - val_loss: 0.8467
Epoch 16/50
121/121 ━━━━━━━━━━━━━━━━━━━━ 4s 37ms/step - accuracy: 0.7591 - loss: 0.7668
- val_accuracy: 0.7827 - val_loss: 0.6645
Epoch 17/50
121/121 ━━━━━━━━━━━━━━━━━━━━ 4s 37ms/step - accuracy: 0.7559 - loss: 0.7904
- val_accuracy: 0.4743 - val_loss: 2.3915
Epoch 18/50
121/121 ━━━━━━━━━━━━━━━━━━━━ 4s 37ms/step - accuracy: 0.7508 - loss: 0.9204
- val_accuracy: 0.6005 - val_loss: 1.2696
Epoch 19/50
121/121 ━━━━━━━━━━━━━━━━━━━━ 4s 37ms/step - accuracy: 0.7703 - loss: 0.7477
- val_accuracy: 0.7640 - val_loss: 0.6375
```

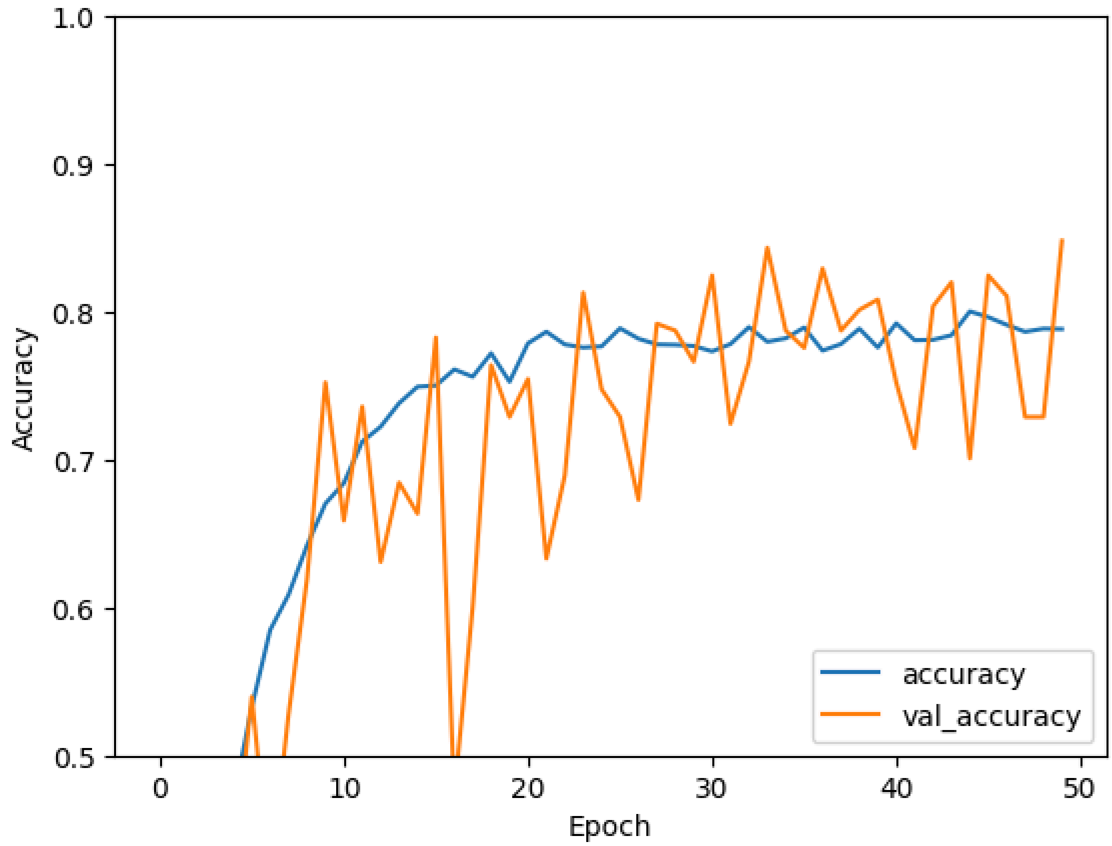
```
Epoch 20/50
121/121 ━━━━━━━━━━━━━━━━━━━━ 5s 37ms/step - accuracy: 0.7428 - loss: 0.8855
- val_accuracy: 0.7290 - val_loss: 0.7595
Epoch 21/50
121/121 ━━━━━━━━━━━━━━━━━━━━ 4s 37ms/step - accuracy: 0.7749 - loss: 0.8040
- val_accuracy: 0.7547 - val_loss: 0.7045
Epoch 22/50
121/121 ━━━━━━━━━━━━━━━━━━━━ 4s 37ms/step - accuracy: 0.7879 - loss: 0.7751
- val_accuracy: 0.6332 - val_loss: 1.5918
Epoch 23/50
121/121 ━━━━━━━━━━━━━━━━━━━━ 4s 37ms/step - accuracy: 0.7679 - loss: 0.9563
- val_accuracy: 0.6893 - val_loss: 1.6671
Epoch 24/50
121/121 ━━━━━━━━━━━━━━━━━━━━ 4s 37ms/step - accuracy: 0.7706 - loss: 1.0224
- val_accuracy: 0.8131 - val_loss: 0.5145
Epoch 25/50
121/121 ━━━━━━━━━━━━━━━━━━━━ 4s 37ms/step - accuracy: 0.7869 - loss: 0.9806
- val_accuracy: 0.7477 - val_loss: 0.9157
Epoch 26/50
121/121 ━━━━━━━━━━━━━━━━━━━━ 4s 37ms/step - accuracy: 0.7805 - loss: 0.9983
- val_accuracy: 0.7290 - val_loss: 1.0600
Epoch 27/50
121/121 ━━━━━━━━━━━━━━━━━━━━ 4s 37ms/step - accuracy: 0.7795 - loss: 1.1553
- val_accuracy: 0.6729 - val_loss: 1.8591
Epoch 28/50
121/121 ━━━━━━━━━━━━━━━━━━━━ 4s 37ms/step - accuracy: 0.7795 - loss: 1.4001
- val_accuracy: 0.7921 - val_loss: 0.8027
Epoch 29/50
121/121 ━━━━━━━━━━━━━━━━━━━━ 4s 37ms/step - accuracy: 0.7839 - loss: 1.7227
- val_accuracy: 0.7874 - val_loss: 1.2654
Epoch 30/50
121/121 ━━━━━━━━━━━━━━━━━━━━ 4s 37ms/step - accuracy: 0.7676 - loss: 2.1092
- val_accuracy: 0.7664 - val_loss: 1.5557
Epoch 31/50
121/121 ━━━━━━━━━━━━━━━━━━━━ 4s 37ms/step - accuracy: 0.7752 - loss: 2.5090
- val_accuracy: 0.8248 - val_loss: 1.2898
Epoch 32/50
121/121 ━━━━━━━━━━━━━━━━━━━━ 4s 37ms/step - accuracy: 0.7861 - loss: 2.5294
- val_accuracy: 0.7243 - val_loss: 2.6438
Epoch 33/50
121/121 ━━━━━━━━━━━━━━━━━━━━ 4s 37ms/step - accuracy: 0.7904 - loss: 2.7482
- val_accuracy: 0.7664 - val_loss: 2.5996
Epoch 34/50
121/121 ━━━━━━━━━━━━━━━━━━━━ 4s 37ms/step - accuracy: 0.7675 - loss: 3.5309
- val_accuracy: 0.8435 - val_loss: 1.6206
Epoch 35/50
121/121 ━━━━━━━━━━━━━━━━━━━━ 4s 37ms/step - accuracy: 0.7808 - loss: 3.7319
- val_accuracy: 0.7874 - val_loss: 2.6072
Epoch 36/50
121/121 ━━━━━━━━━━━━━━━━━━━━ 4s 37ms/step - accuracy: 0.7830 - loss: 3.9903
- val_accuracy: 0.7757 - val_loss: 3.4421
Epoch 37/50
121/121 ━━━━━━━━━━━━━━━━━━━━ 4s 37ms/step - accuracy: 0.7617 - loss: 5.9804
- val_accuracy: 0.8294 - val_loss: 3.3807
Epoch 38/50
121/121 ━━━━━━━━━━━━━━━━━━━━ 4s 37ms/step - accuracy: 0.7837 - loss: 6.4872
```

```
- val_accuracy: 0.7874 - val_loss: 5.1810
Epoch 39/50
121/121 ━━━━━━━━━━━━━━━━━━━━ 4s 37ms/step - accuracy: 0.7826 - loss: 6.8066
- val_accuracy: 0.8014 - val_loss: 5.3672
Epoch 40/50
121/121 ━━━━━━━━━━━━━━━━━━━━ 4s 37ms/step - accuracy: 0.7862 - loss: 9.8135
- val_accuracy: 0.8084 - val_loss: 4.4081
Epoch 41/50
121/121 ━━━━━━━━━━━━━━━━━━━━ 4s 37ms/step - accuracy: 0.7876 - loss: 10.5496
- val_accuracy: 0.7523 - val_loss: 11.9285
Epoch 42/50
121/121 ━━━━━━━━━━━━━━━━━━━━ 4s 37ms/step - accuracy: 0.7804 - loss: 13.0562
- val_accuracy: 0.7079 - val_loss: 11.5712
Epoch 43/50
121/121 ━━━━━━━━━━━━━━━━━━━━ 4s 37ms/step - accuracy: 0.7730 - loss: 14.7564
- val_accuracy: 0.8037 - val_loss: 8.9708
Epoch 44/50
121/121 ━━━━━━━━━━━━━━━━━━━━ 4s 37ms/step - accuracy: 0.7866 - loss: 18.7229
- val_accuracy: 0.8201 - val_loss: 12.4372
Epoch 45/50
121/121 ━━━━━━━━━━━━━━━━━━━━ 4s 37ms/step - accuracy: 0.7992 - loss: 21.0525
- val_accuracy: 0.7009 - val_loss: 26.2403
Epoch 46/50
121/121 ━━━━━━━━━━━━━━━━━━━━ 5s 38ms/step - accuracy: 0.7901 - loss: 29.4615
- val_accuracy: 0.8248 - val_loss: 16.2593
Epoch 47/50
121/121 ━━━━━━━━━━━━━━━━━━━━ 5s 38ms/step - accuracy: 0.8013 - loss: 34.8615
- val_accuracy: 0.8107 - val_loss: 15.4588
Epoch 48/50
121/121 ━━━━━━━━━━━━━━━━━━━━ 5s 38ms/step - accuracy: 0.7976 - loss: 38.9606
- val_accuracy: 0.7290 - val_loss: 55.9055
Epoch 49/50
121/121 ━━━━━━━━━━━━━━━━━━━━ 5s 39ms/step - accuracy: 0.7734 - loss: 57.6031
- val_accuracy: 0.7290 - val_loss: 69.3651
Epoch 50/50
121/121 ━━━━━━━━━━━━━━━━━━━━ 5s 39ms/step - accuracy: 0.7875 - loss: 56.8515
- val_accuracy: 0.8481 - val_loss: 26.7953
```

Plotting the training and validation accuracy

In [18]:

```
plt.plot(history.history['accuracy'], label='accuracy')
plt.plot(history.history['val_accuracy'], label = 'val_accuracy')

plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.ylim([0.5, 1])
plt.legend(loc='lower right');
```



In [19]:
```
# Evaluate the model.

score = model.evaluate(X_test, y_test, verbose=0, batch_size = 32)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

```
Test loss: 24.999555587768555
Test accuracy: 0.8568421006202698
```

- As we see from the learning graph the model is learning well, but it's **slightly overfitting after the 35th epoch.** Therefore, let's use some techniques to prevent overfitting.
- Validation accuracy is approximately constant after the 25th epoch. **Let's reduce the learning rate when the validation loss does not change.**

Additional Content

Preventing Overfitting

Overfitting is a problem in machine learning in which our model performs very well on the training data but performs poorly on testing data.

The problem of overfitting is severe in deep learning, since the neural network automatically detects several features and builds a complex mathematical model with several layers to map the input to the output. The tendency of deep neural networks to overfit to the training dataset could affects our end result badly, so this issue has to be addressed immediately.

There are many **regularization techniques** to overcome overfitting:

1) Dropout

2) Data Augmentation

3) Batch Normalization (weak regularizer)

Let's use Data augmentation here to prevent overfitting and make the model more robust for inference.

In this problem, we are using the ImageDataGenerator() function which randomly changes the characteristics of images and provides randomness in the data. To avoid overfitting, we need a function. This function randomly changes the image characteristics. Check the below code on how to reduce overfitting.

Reducing the Learning Rate:

ReduceLRonPlateau() is a 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.

In [20]:
```
from tensorflow.keras.callbacks import ReduceLROnPlateau

learning_rate_reduction = ReduceLROnPlateau(monitor='val_accuracy',
                                            patience=3,
                                            verbose=1,
                                            factor=0.5,
                                            min_lr=0.00001)
epochs = 30
batch_size = 38
```

Data Augmentation (Regularization)

Data Augmentation is a strategy that enables practitioners to significantly increase the diversity of data available for training models, without actually collecting new data. Data Augmentation techniques such as cropping, padding, and horizontal flipping are commonly used to train large neural networks in order to help regularize them and reduce overfitting.

Please read about Imagedatagenerator from here

In [21]:
```
# With data augmentation to prevent overfitting (accuracy 0.99286)

datagen = ImageDataGenerator(
        featurewise_center=False,  # set input mean to 0 over the dataset
        samplewise_center=False,  # set each sample mean to 0
        featurewise_std_normalization=False,  # divide inputs by std of the
        samplewise_std_normalization=False,  # divide each input by its std
        zca_whitening=False,  # apply ZCA whitening
        rotation_range=10,  # randomly rotate images in the range (degrees,
        zoom_range = 0.1, # Randomly zoom image
        width_shift_range=0.1,  # randomly shift images horizontally (fracti
        height_shift_range=0.1,  # randomly shift images vertically (fractic
        horizontal_flip=False,  # randomly flip images
        vertical_flip=False)  # randomly flip images


datagen.fit(X_train)
```

Dividing the training data into train and validation/dev set from X_train

In [22]:
```
from sklearn.model_selection import train_test_split
X_train, X_val, y_train, y_val = train_test_split(X_train,y_train , test_siz
```

In [23]:
```
X_train.shape
```

Out[23]:
```
(3847, 128, 128, 3)
```

Defining the model

Set the CNN model

model1 = Sequential()

model1.add(Conv2D(filters = 32, kernel_size = (5,5),padding = 'Same', activation ='relu', batch_input_shape = (None,128,128, 3)))

model1.add(Conv2D(filters = 32, kernel_size = (5,5),padding = 'Same', activation ='relu')) model1.add(MaxPool2D(pool_size=(2,2))) model1.add(Dropout(0.2))

model1.add(Conv2D(filters = 64, kernel_size = (3,3),padding = 'Same', activation ='relu')) model1.add(Conv2D(filters = 64, kernel_size = (3,3),padding = 'same', activation ='relu')) model1.add(MaxPool2D(pool_size=(2,2), strides=(2,2))) model1.add(Dropout(0.3))

model1.add(Conv2D(filters = 128, kernel_size = (3,3),padding = 'Same', activation ='relu')) model1.add(Conv2D(filters = 128, kernel_size = (3,3),padding = 'Same', activation ='relu')) model1.add(MaxPool2D(pool_size=(2,2), strides=(2,2))) model1.add(Dropout(0.4))

model1.add(GlobalMaxPooling2D()) model1.add(Dense(256, activation = "relu")) model1.add(Dropout(0.5)) model1.add(Dense(12, activation = "softmax")) model1.summary()

In [24]:
```
# Set the CNN model
model1 = Sequential()

model1.add(Conv2D(filters=32, kernel_size=(5, 5), padding='same',
                  activation='relu', input_shape=(128, 128, 3)))

model1.add(Conv2D(filters=32, kernel_size=(5, 5), padding='same',
                  activation='relu'))
model1.add(MaxPool2D(pool_size=(2, 2)))
model1.add(Dropout(0.2))

model1.add(Conv2D(filters=64, kernel_size=(3, 3), padding='same',
                  activation='relu'))
model1.add(Conv2D(filters=64, kernel_size=(3, 3), padding='same',
                  activation='relu'))
model1.add(MaxPool2D(pool_size=(2, 2), strides=(2, 2)))
model1.add(Dropout(0.3))

model1.add(Conv2D(filters=128, kernel_size=(3, 3), padding='same',
                  activation='relu'))
model1.add(Conv2D(filters=128, kernel_size=(3, 3), padding='same',
                  activation='relu'))
model1.add(MaxPool2D(pool_size=(2, 2), strides=(2, 2)))
model1.add(Dropout(0.4))

model1.add(GlobalMaxPooling2D())
model1.add(Dense(256, activation='relu'))
model1.add(Dropout(0.5))
model1.add(Dense(12, activation='softmax'))

model1.summary()
```

```
/Users/obaozai/miniconda3/envs/my_env/lib/python3.11/site-packages/keras/sr
c/layers/convolutional/base_conv.py:107: UserWarning: Do not pass an `input_
shape`/`input_dim` argument to a layer. When using Sequential models, prefer
using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

Model: "sequential_1"

Layer (type)	Output Shape	Par
conv2d_6 (Conv2D)	(None, 128, 128, 32)	2
conv2d_7 (Conv2D)	(None, 128, 128, 32)	25
max_pooling2d_3 (MaxPooling2D)	(None, 64, 64, 32)	
dropout_4 (Dropout)	(None, 64, 64, 32)	
conv2d_8 (Conv2D)	(None, 64, 64, 64)	18
conv2d_9 (Conv2D)	(None, 64, 64, 64)	36
max_pooling2d_4 (MaxPooling2D)	(None, 32, 32, 64)	
dropout_5 (Dropout)	(None, 32, 32, 64)	
conv2d_10 (Conv2D)	(None, 32, 32, 128)	73
conv2d_11 (Conv2D)	(None, 32, 32, 128)	147
max_pooling2d_5 (MaxPooling2D)	(None, 16, 16, 128)	
dropout_6 (Dropout)	(None, 16, 16, 128)	
global_max_pooling2d_1 (GlobalMaxPooling2D)	(None, 128)	
dense_2 (Dense)	(None, 256)	33
dropout_7 (Dropout)	(None, 256)	
dense_3 (Dense)	(None, 12)	3

Total params: 341,036 (1.30 MB)

Trainable params: 341,036 (1.30 MB)

Non-trainable params: 0 (0.00 B)

In [25]:
```
from tensorflow.keras import optimizers

# Defining the optimizer and loss function
model1.compile(
    optimizer=optimizers.RMSprop(learning_rate=0.001, rho=0.9, epsilon=1e-08
    loss="categorical_crossentropy",
    metrics=["accuracy"]
)
```

```
/Users/obaozai/miniconda3/envs/my_env/lib/python3.11/site-packages/keras/sr
c/optimizers/base_optimizer.py:86: UserWarning: Argument `decay` is no longe
r supported and will be ignored.
  warnings.warn(
```

Defining the optimizer and loss function

model1.compile(optimizer = optimizers.legacy.RMSprop(learning_rate=0.001, rho=0.9, epsilon=1e-08, decay=0.0), loss = "categorical_crossentropy", metrics = ["accuracy"])

In [26]:
```
# Fitting the model using fit_generator function
batch_size = 32
history1 = model1.fit(datagen.flow(X_train,y_train, batch_size=batch_size),
                              epochs = epochs, validation_data = (X_val,y_va
                              verbose = 2, steps_per_epoch=X_train.shape[0]
                              , callbacks=[learning_rate_reduction])
```

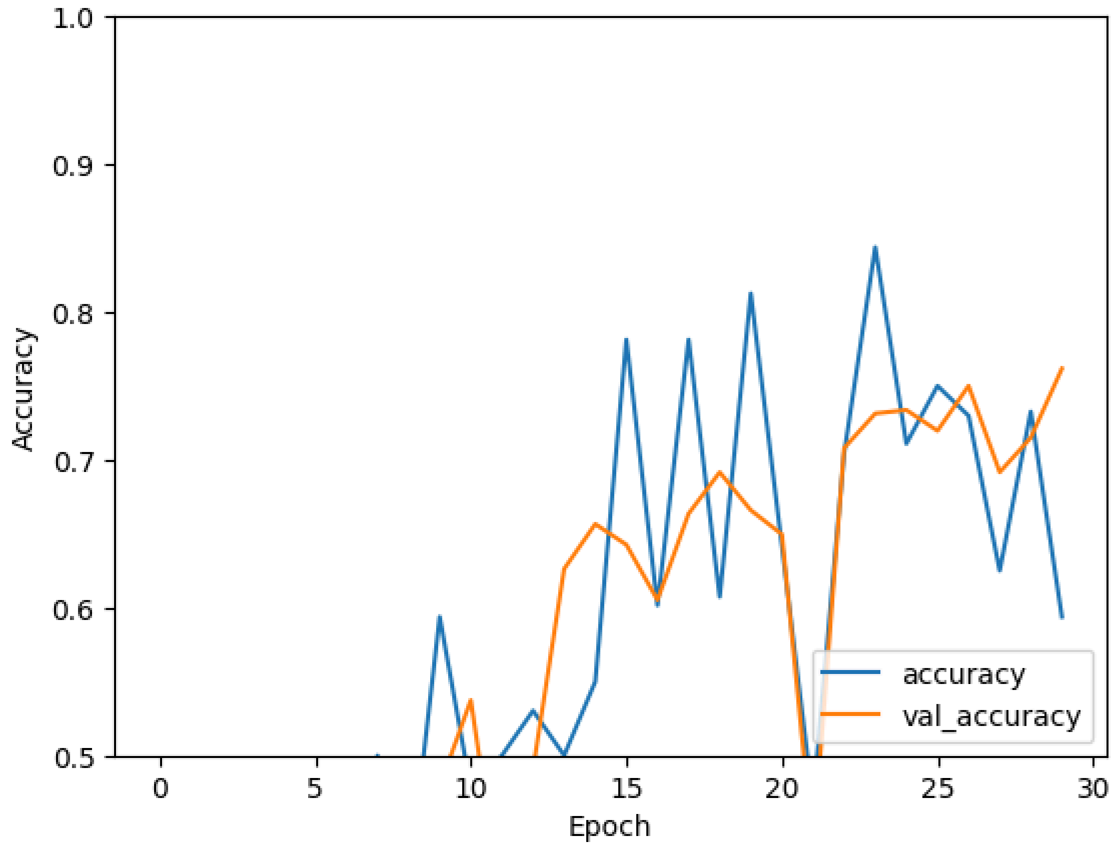
```
/Users/obaozai/miniconda3/envs/my_env/lib/python3.11/site-packages/keras/sr
c/trainers/data_adapters/py_dataset_adapter.py:121: UserWarning: Your `PyDat
aset` class should call `super().__init__(**kwargs)` in its constructor. `**
kwargs` can include `workers`, `use_multiprocessing`, `max_queue_size`. Do n
ot pass these arguments to `fit()`, as they will be ignored.
  self._warn_if_super_not_called()
Epoch 1/30
120/120 - 6s - 51ms/step - accuracy: 0.1987 - loss: 2.3497 - val_accuracy:
0.3061 - val_loss: 2.1934 - learning_rate: 1.0000e-03
Epoch 2/30
120/120 - 0s - 2ms/step - accuracy: 0.3125 - loss: 2.0550 - val_accuracy: 0.
2897 - val_loss: 2.1051 - learning_rate: 1.0000e-03
Epoch 3/30
/Users/obaozai/miniconda3/envs/my_env/lib/python3.11/site-packages/keras/sr
c/trainers/epoch_iterator.py:107: UserWarning: Your input ran out of data; i
nterrupting training. Make sure that your dataset or generator can generate
at least `steps_per_epoch * epochs` batches. You may need to use the `.repea
t()` function when building your dataset.
  self._interrupted_warning()
```

```
120/120 - 5s - 41ms/step - accuracy: 0.2692 - loss: 2.0904 - val_accuracy:
0.3364 - val_loss: 1.9624 - learning_rate: 1.0000e-03
Epoch 4/30
120/120 - 0s - 2ms/step - accuracy: 0.1562 - loss: 2.0952 - val_accuracy: 0.
3014 - val_loss: 1.9530 - learning_rate: 1.0000e-03
Epoch 5/30
120/120 - 5s - 42ms/step - accuracy: 0.2967 - loss: 1.9982 - val_accuracy:
0.3178 - val_loss: 1.8752 - learning_rate: 1.0000e-03
Epoch 6/30
120/120 - 0s - 2ms/step - accuracy: 0.4062 - loss: 1.7714 - val_accuracy: 0.
3949 - val_loss: 1.8169 - learning_rate: 1.0000e-03
Epoch 7/30
120/120 - 5s - 43ms/step - accuracy: 0.3541 - loss: 1.8305 - val_accuracy:
0.4276 - val_loss: 1.6525 - learning_rate: 1.0000e-03
Epoch 8/30
120/120 - 0s - 2ms/step - accuracy: 0.5000 - loss: 1.4058 - val_accuracy: 0.
4720 - val_loss: 1.5709 - learning_rate: 1.0000e-03
Epoch 9/30
120/120 - 5s - 44ms/step - accuracy: 0.4123 - loss: 1.6776 - val_accuracy:
0.4486 - val_loss: 1.5245 - learning_rate: 1.0000e-03
Epoch 10/30
120/120 - 0s - 2ms/step - accuracy: 0.5938 - loss: 1.3312 - val_accuracy: 0.
4790 - val_loss: 1.4961 - learning_rate: 1.0000e-03
Epoch 11/30
120/120 - 5s - 45ms/step - accuracy: 0.4776 - loss: 1.5177 - val_accuracy:
0.5374 - val_loss: 1.3198 - learning_rate: 1.0000e-03
Epoch 12/30
120/120 - 0s - 2ms/step - accuracy: 0.5000 - loss: 2.1388 - val_accuracy: 0.
3972 - val_loss: 1.5809 - learning_rate: 1.0000e-03
Epoch 13/30
120/120 - 6s - 46ms/step - accuracy: 0.5303 - loss: 1.4282 - val_accuracy:
0.4907 - val_loss: 1.3568 - learning_rate: 1.0000e-03
Epoch 14/30
120/120 - 0s - 2ms/step - accuracy: 0.5000 - loss: 1.6172 - val_accuracy: 0.
6262 - val_loss: 1.1539 - learning_rate: 1.0000e-03
Epoch 15/30
120/120 - 6s - 47ms/step - accuracy: 0.5499 - loss: 1.3691 - val_accuracy:
0.6565 - val_loss: 1.0676 - learning_rate: 1.0000e-03
Epoch 16/30
120/120 - 0s - 2ms/step - accuracy: 0.7812 - loss: 0.6474 - val_accuracy: 0.
6425 - val_loss: 1.0927 - learning_rate: 1.0000e-03
Epoch 17/30
120/120 - 6s - 48ms/step - accuracy: 0.6016 - loss: 1.2927 - val_accuracy:
0.6051 - val_loss: 1.1144 - learning_rate: 1.0000e-03
Epoch 18/30
120/120 - 0s - 2ms/step - accuracy: 0.7812 - loss: 0.8663 - val_accuracy: 0.
6636 - val_loss: 1.0172 - learning_rate: 1.0000e-03
Epoch 19/30
120/120 - 6s - 49ms/step - accuracy: 0.6073 - loss: 1.2136 - val_accuracy:
0.6916 - val_loss: 0.9562 - learning_rate: 1.0000e-03
Epoch 20/30
120/120 - 0s - 2ms/step - accuracy: 0.8125 - loss: 0.5487 - val_accuracy: 0.
6659 - val_loss: 0.9744 - learning_rate: 1.0000e-03
Epoch 21/30
120/120 - 6s - 50ms/step - accuracy: 0.6417 - loss: 1.1616 - val_accuracy:
0.6495 - val_loss: 1.0120 - learning_rate: 1.0000e-03
```

```
Epoch 22/30

Epoch 22: ReduceLROnPlateau reducing learning rate to 0.0005000000237487257.
120/120 - 0s - 2ms/step - accuracy: 0.4688 - loss: 2.0462 - val_accuracy: 0.
4369 - val_loss: 1.7533 - learning_rate: 1.0000e-03
Epoch 23/30
120/120 - 6s - 52ms/step - accuracy: 0.7043 - loss: 0.9017 - val_accuracy:
0.7079 - val_loss: 0.8443 - learning_rate: 5.0000e-04
Epoch 24/30
120/120 - 0s - 2ms/step - accuracy: 0.8438 - loss: 0.6000 - val_accuracy: 0.
7313 - val_loss: 0.8176 - learning_rate: 5.0000e-04
Epoch 25/30
120/120 - 6s - 53ms/step - accuracy: 0.7109 - loss: 0.9030 - val_accuracy:
0.7336 - val_loss: 0.8160 - learning_rate: 5.0000e-04
Epoch 26/30
120/120 - 0s - 2ms/step - accuracy: 0.7500 - loss: 0.7505 - val_accuracy: 0.
7196 - val_loss: 0.8628 - learning_rate: 5.0000e-04
Epoch 27/30
120/120 - 6s - 54ms/step - accuracy: 0.7298 - loss: 0.8428 - val_accuracy:
0.7500 - val_loss: 0.7117 - learning_rate: 5.0000e-04
Epoch 28/30
120/120 - 0s - 2ms/step - accuracy: 0.6250 - loss: 1.2370 - val_accuracy: 0.
6916 - val_loss: 0.8951 - learning_rate: 5.0000e-04
Epoch 29/30
120/120 - 7s - 55ms/step - accuracy: 0.7326 - loss: 0.8496 - val_accuracy:
0.7150 - val_loss: 0.8208 - learning_rate: 5.0000e-04
Epoch 30/30
120/120 - 0s - 2ms/step - accuracy: 0.5938 - loss: 0.8998 - val_accuracy: 0.
7617 - val_loss: 0.7364 - learning_rate: 5.0000e-04
```

In [27]:

```
plt.plot(history1.history['accuracy'], label='accuracy')
plt.plot(history1.history['val_accuracy'], label = 'val_accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.ylim([0.5, 1])
plt.legend(loc='lower right');
```



As you can see from the learning graph, **the overfitting has been reduced.** The model is **generalizing well to the validation set.**

In [28]:
```
# Evaluate the model.
score = model1.evaluate(X_test, y_test, verbose=0, batch_size = 38)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

```
Test loss: 0.6713515520095825
Test accuracy: 0.7831578850746155
```

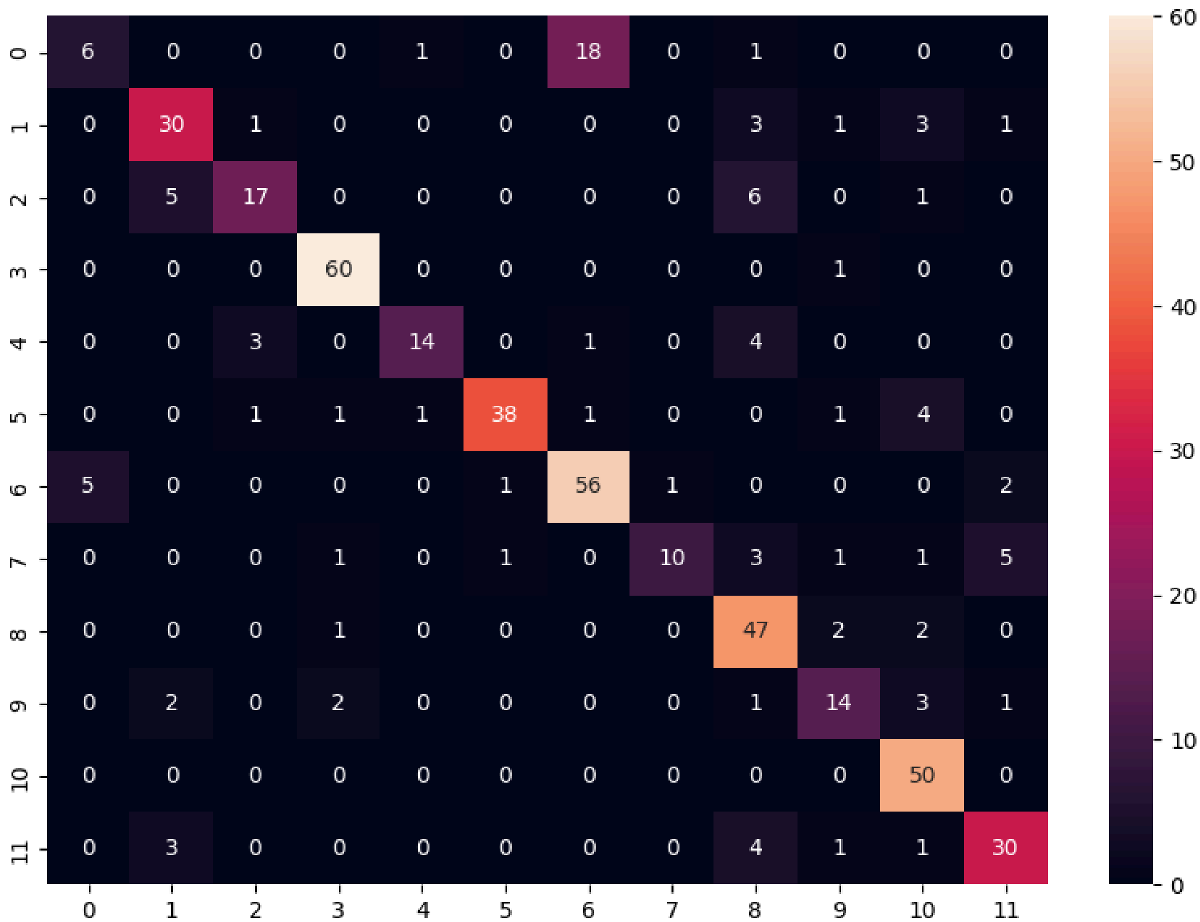
- As you can see, we are getting better inference accuracy here.

Plotting the confusion matrix

In [29]:
```
# Predict the values from the validation dataset
Y_pred = model1.predict(X_test)
# Convert predictions classes to one hot vectors
result = np.argmax(Y_pred, axis=1)
# Convert validation observations to one hot vectors
Y_true = np.argmax(y_test, axis=1)

import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import confusion_matrix

conf_mat = confusion_matrix(Y_true, result)
```

```
df_cm = pd.DataFrame(conf_mat, index = [i for i in range(0, 12)],
                     columns = [i for i in range(0, 12)])
plt.figure(figsize = (10,7))
sns.heatmap(df_cm, annot=True, fmt='g');
```

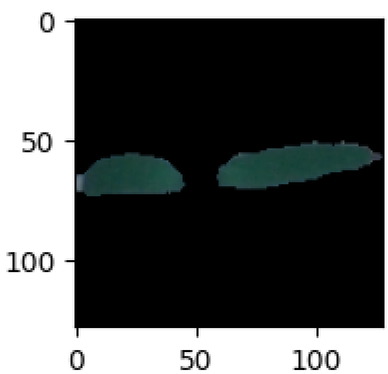
15/15 ━━━━━━━━━━━━━━━━━━━━ **0s** 13ms/step



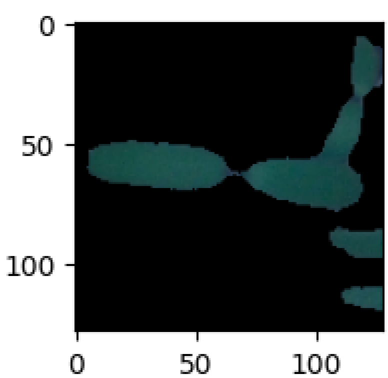
Visualizing the prediction

In [30]:
```
import numpy as np

plt.figure(figsize=(2,2))
plt.imshow(X_test[3],cmap="gray")
plt.show()
print('Predicted Label', np.argmax(model1.predict(X_test[3].reshape(1,128,12
print('True Label', np.argmax(y_test[3]))

plt.figure(figsize=(2,2))
plt.imshow(X_test[2],cmap="gray")
plt.show()
print('Predicted Label', np.argmax(model1.predict(X_test[2].reshape(1,128,12
print('True Label', np.argmax(y_test[2]))

plt.figure(figsize=(2,2))
plt.imshow(X_test[33],cmap="gray")
plt.show()
```

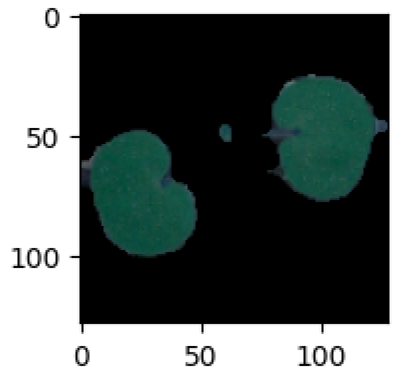
```
print('Predicted Label', np.argmax(model1.predict(X_test[33].reshape(1,128,1
print('True Label', np.argmax(y_test[33]))

plt.figure(figsize=(2,2))
plt.imshow(X_test[59],cmap="gray")
plt.show()
print('Predicted Label', np.argmax(model1.predict(X_test[59].reshape(1,128,1
print('True Label', np.argmax(y_test[59]))

plt.figure(figsize=(2,2))
plt.imshow(X_test[36],cmap="gray")
plt.show()
print('Predicted Label', np.argmax(model1.predict(X_test[36].reshape(1,128,1
print('True Label', np.argmax(y_test[36]))
```



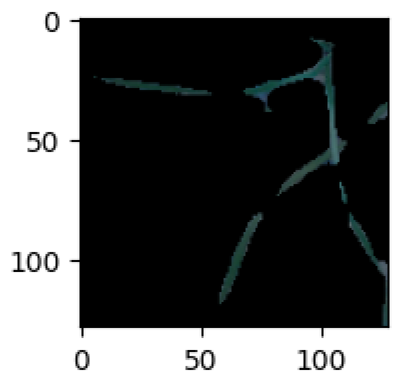

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 436ms/step
Predicted Label 5
True Label 5
```



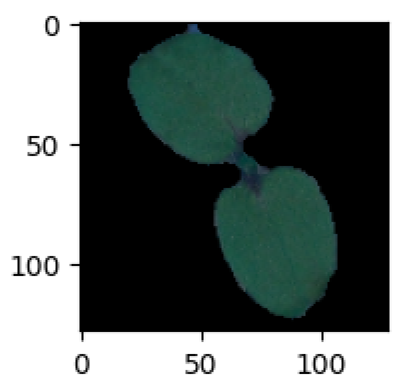

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step
Predicted Label 5
True Label 5
```



```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step
Predicted Label 10
True Label 10
```



```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step
Predicted Label 6
True Label 6
```



```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step
Predicted Label 1
True Label 2
```

Conclusion

- **The model accuracy becomes constant after the 25th epoch**, reducing learning rate works well at this time.
- **Data Augmentation works as a regularizer** and helps to reduce the variance in the training and increases the generalization of the model.