Introduction to Computer Vision: Plant Seedlings Classification

Problem Statement

Context

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

Objective

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

Data Dictionary

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

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

List of Species

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

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

Importing necessary libraries

In [1]:
```
# Installing the libraries with the specified version.
# uncomment and run the following line if Google Colab is being used
!pip install tensorflow==2.15.0 scikit-learn==1.2.2 seaborn==0.13.1 matplotlib==3.7.1 numpy==1.25.2 pandas==1.5
```

```
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 475.2/475.2 MB 2.6 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 9.6/9.6 MB 38.2 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 294.8/294.8 kB 10.0 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 18.2/18.2 MB 21.7 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 12.1/12.1 MB 29.6 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 61.7/61.7 MB 7.0 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.7/1.7 MB 28.0 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.0/1.0 MB 41.2 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 5.5/5.5 MB 51.2 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 442.0/442.0 kB 16.5 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 77.9/77.9 kB 4.4 MB/s eta 0:00:00
  WARNING: The scripts f2py, f2py3 and f2py3.10 are installed in '/root/.local/bin' which is not on PATH.
  Consider adding this directory to PATH or, if you prefer to suppress this warning, use --no-warn-script-locat
ion.
  WARNING: The script tensorboard is installed in '/root/.local/bin' which is not on PATH.
  Consider adding this directory to PATH or, if you prefer to suppress this warning, use --no-warn-script-locat
ion.
  WARNING: The scripts estimator_ckpt_converter, import_pb_to_tensorboard, saved_model_cli, tensorboard, tf_upg
rade_v2, tflite_convert, toco and toco_from_protos are installed in '/root/.local/bin' which is not on PATH.
  Consider adding this directory to PATH or, if you prefer to suppress this warning, use --no-warn-script-locat
ion.
ERROR: pip's dependency resolver does not currently take into account all the packages that are installed. This
behaviour is the source of the following dependency conflicts.
cudf-cu12 24.6.1 requires pandas<2.2.3dev0,>=2.0, but you have pandas 1.5.3 which is incompatible.
google-colab 1.0.0 requires pandas==2.2.2, but you have pandas 1.5.3 which is incompatible.
mizani 0.11.4 requires pandas>=2.1.0, but you have pandas 1.5.3 which is incompatible.
plotnine 0.13.6 requires pandas<3.0.0,>=2.1.0, but you have pandas 1.5.3 which is incompatible.
tensorstore 0.1.66 requires ml-dtypes>=0.3.1, but you have ml-dtypes 0.2.0 which is incompatible.
tf-keras 2.17.0 requires tensorflow<2.18,>=2.17, but you have tensorflow 2.15.0 which is incompatible.
xarray 2024.9.0 requires pandas>=2.1, but you have pandas 1.5.3 which is incompatible.
```

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

In [3]:
```
import os
import numpy as np                                                                    # Importing nu
import pandas as pd                                                                   # Importing pa
import matplotlib.pyplot as plt                                                       # Importting m
import math                                                                           # Importing ma
import cv2                                                                            # Importing op
import seaborn as sns                                                                 # Importing se


# Tensorflow modules
import tensorflow as tf
from tensorflow.keras.preprocessing.image import ImageDataGenerator                  # Importing th
from tensorflow.keras.models import Sequential                                       # Importing th
from tensorflow.keras.layers import Dense,Dropout,Flatten,Conv2D,MaxPooling2D,BatchNormalization # Defining all
from tensorflow.keras.optimizers import Adam,SGD                                     # Importing th
from sklearn import preprocessing                                                    # Importing th
from sklearn.model_selection import train_test_split                                 # Importing tr
from sklearn.metrics import confusion_matrix                                         # Importing co

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

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

Loading the dataset

In [4]:
```
import os
from google.colab import drive

# Check if the drive is already mounted
if not os.path.ismount('/content/drive'):
    drive.mount('/content/drive')
else:
    print("Drive is already mounted")

import numpy as np
import pandas as pd

# Load the images
images = np.load('/content/drive/MyDrive/Colab Notebooks/images.npy')

# Load the labels
labels = pd.read_csv('/content/drive/MyDrive/Colab Notebooks/Labels.csv')
```

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

Data Overview

Understand the shape of the dataset

In [5]:
```
# Printing the shape of the images and the lables
print(images.shape)
print(labels.shape)
```

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

- There are 4750 RBG Images of shape 128 x 128 X 3, where each images has 3 channels.

Exploratory Data Analysis

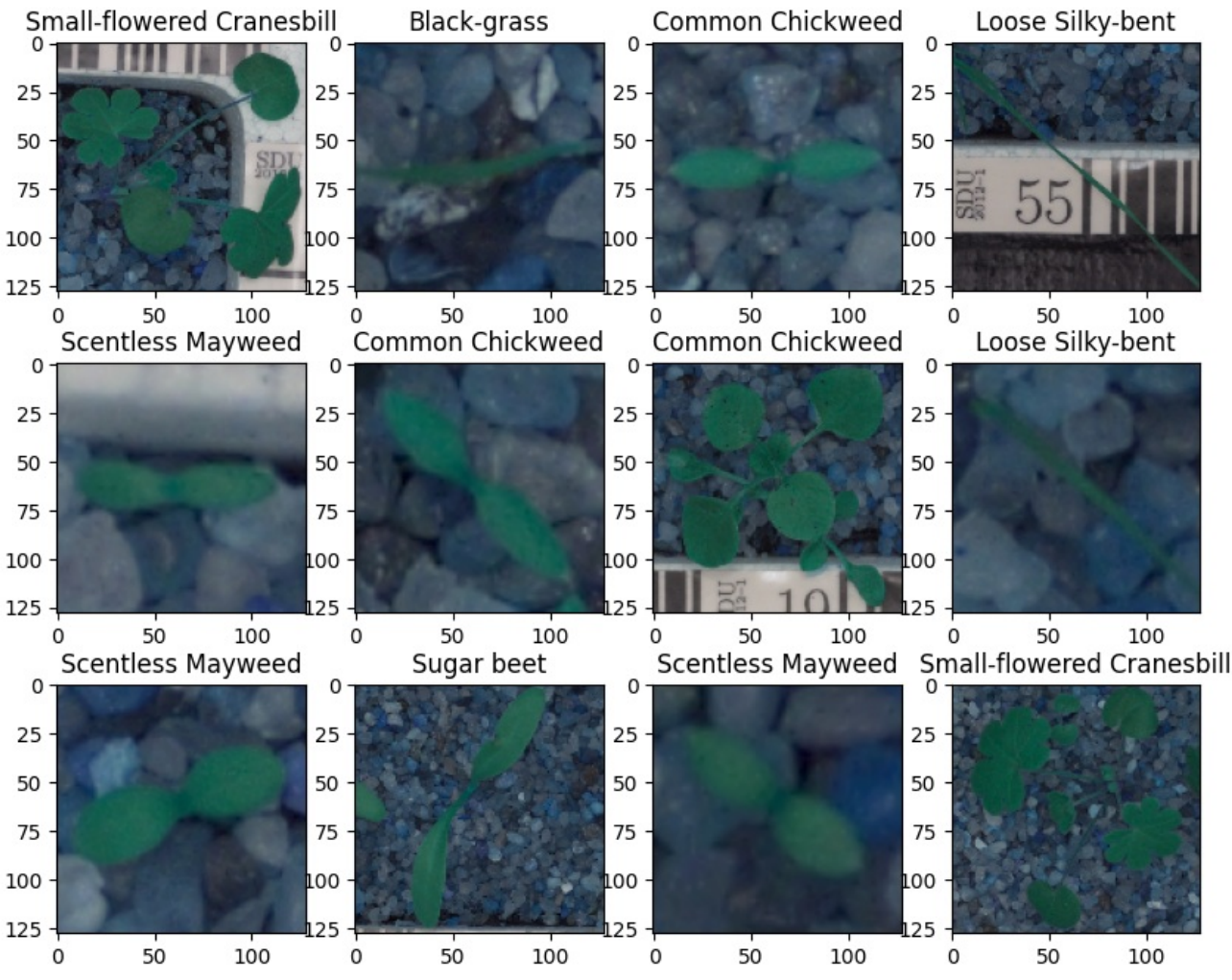
- EDA is an important part of any project involving data.
- It is important to investigate and understand the data better before building a model with it.
- A few questions have been mentioned below which will help you understand the data better.
- A thorough analysis of the data, in addition to the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

1. How are these different category plant images different from each other?
2. Is the dataset provided an imbalance? (Check with using bar plots)

Question 1

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

In [7]:
```
plot_images(images,labels)
```



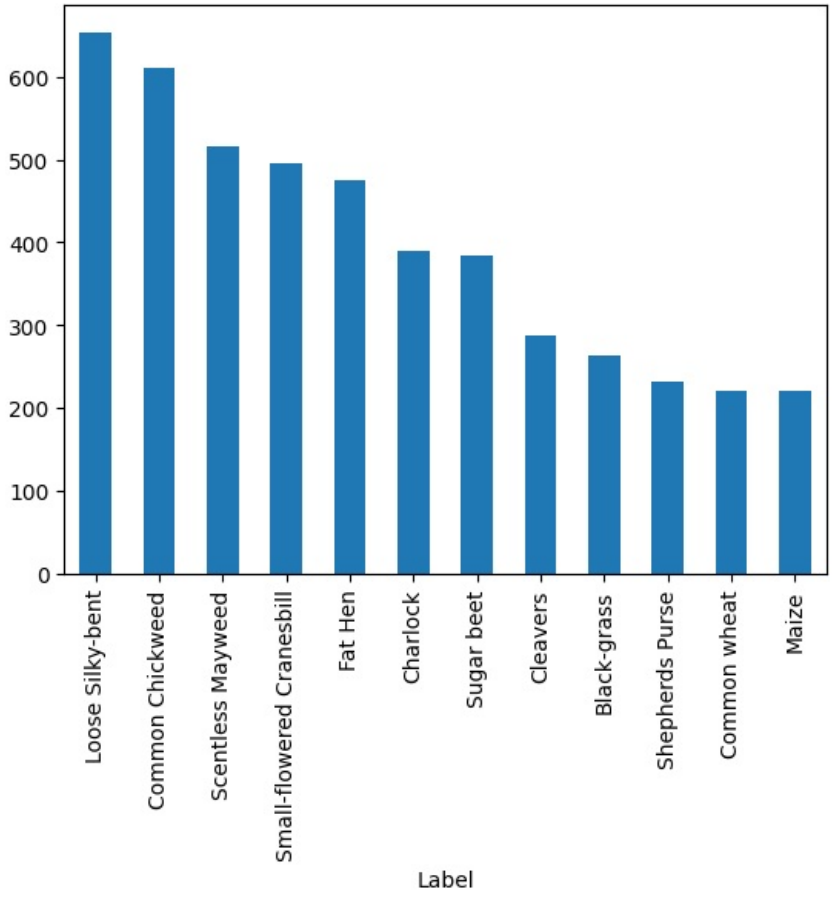
How are these different category plant images different from each other?

- All of the plant species are differnt from ecah other because of many factors.
  - One of the most importnat factors is the type of leaf they have. For instance, the Charlock compared to the Common Chickweed is really noticiable in terms of size and shape, where the Charlock have a much bigger, wider, and many more leafs in the plant compared to the Common Chickweed, where there is a maximum of two leafs on the opposite ends of each other. The color, is also a fcator for both of these types of plant species, the Common Chickweed seems to be a little bit lighter than the Charlock.

Question 2

In [8]:
```
# checking count with barplot for each category
labels['Label'].value_counts().plot(kind='bar')
```

Out[8]:
```
<Axes: xlabel='Label'>
```



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

- From the above bar plot, you can visualize that the data is very much imbalanced.
- The Loose Silky-bent plant species has the most out of them all, counting up to more than 650. Whereas, plant species like Shepherds Purse, Common wheat, and Maize have around 200.

Countplot for Plant Species

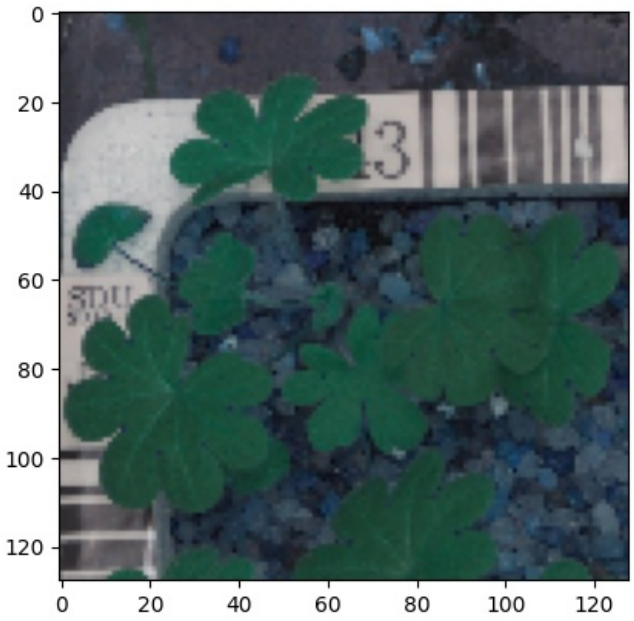
Data Pre-Processing

Convert the BGR images to RGB images.

In [9]:
```
# Plotting images using OpenCV and matplotlib
cv2_imshow(images[3])
```

In [10]:
```
plt.imshow(images[3])
```

Out[10]: <matplotlib.image.AxesImage at 0x7eca48643220>



- We can highlight that the images being shown in different colors when plotted with OpenCV and matplotlib as OpenCV reads images in BlueGreenRed (BGR) format, we could now convert to RBG images to depict them more fondly.

In [11]:
```
import cv2
import numpy as np

# Assuming images are loaded in BGR format, convert them to RGB
def convert_bgr_to_rgb(images):
    rgb_images = []
    for image in images:
        rgb_images.append(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
    return np.array(rgb_images)

# Convert the loaded images to RGB
rgb_images = convert_bgr_to_rgb(images)
```

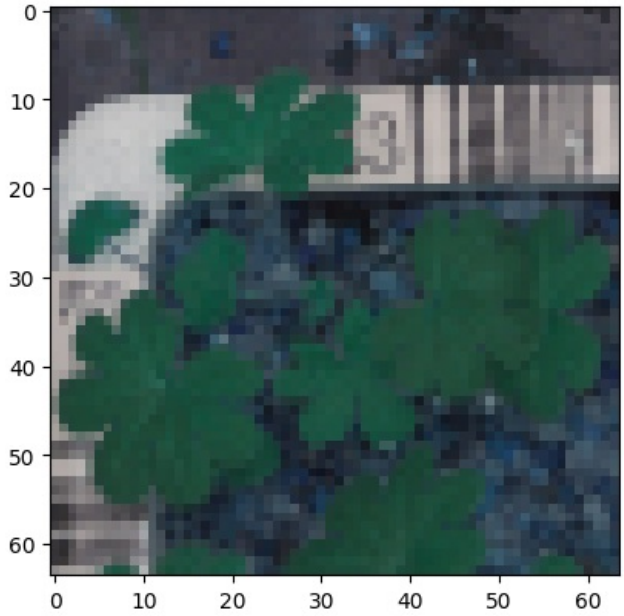
Resize the images

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

In [12]:
```
images_decreased=[]
height = 64
width = 64
dimensions = (width, height)
for i in range(len(images)):
  images_decreased.append( cv2.resize(images[i], dimensions, interpolation=cv2.INTER_LINEAR))
```

In [13]:
```
plt.imshow(images_decreased[3])
```

Out[13]: <matplotlib.image.AxesImage at 0x7eca484d92d0>

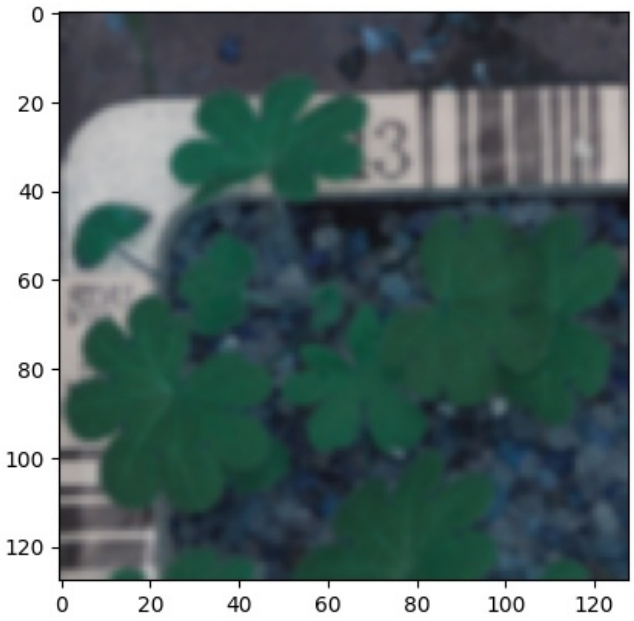


Visualizing images using Gaussian Blur

In [14]:
```
# Using 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 [15]:
```
plt.imshow(images_gb[3])
```

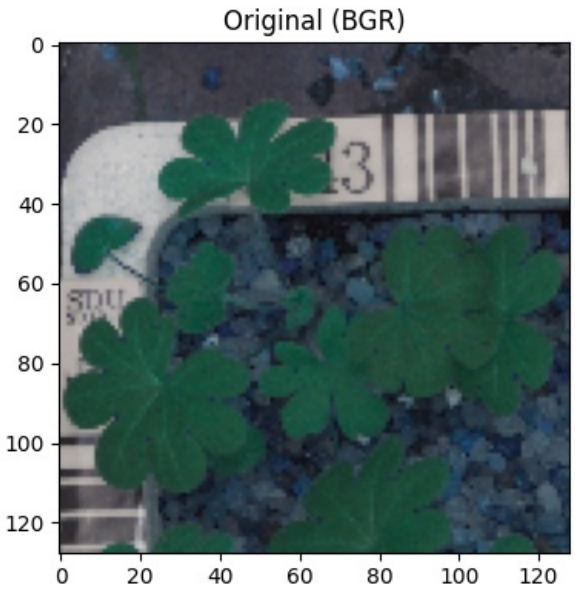
Out[15]:
```
<matplotlib.image.AxesImage at 0x7eca483bfca0>
```

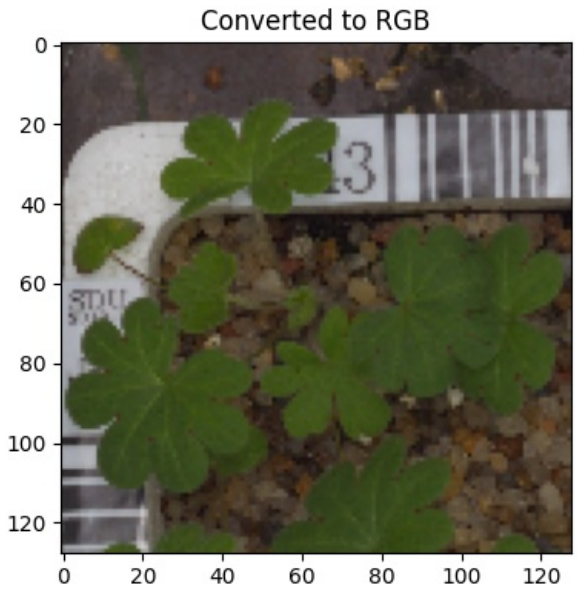


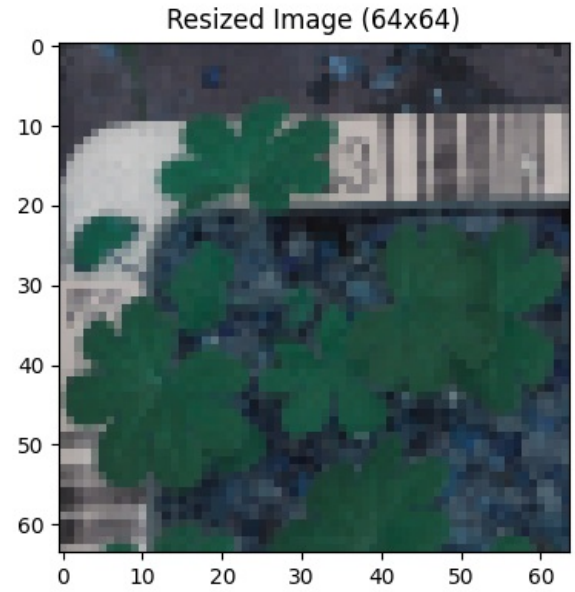
The Gaussian Blur does help the model categorize the image as compared to the previous image when it was resized.

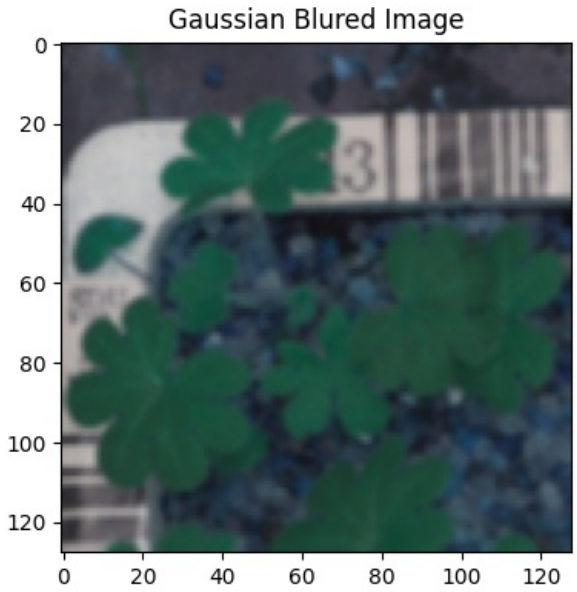
Images Before and After Pre-Processing Steps

In [16]:
```
import matplotlib.pyplot as plt

# Plot random images before and after pre-processing
plt.figure(figsize=(10, 8))

# Original image
plt.subplot(2, 2, 1)
plt.imshow(images[3])  # Original BGR image
plt.title('Original (BGR)')

# Converted to RGB
plt.subplot(2, 2, 2)
plt.imshow(rgb_images[3])  # Converted RGB image
plt.title('Converted to RGB')

# Resized image
plt.subplot(2, 2, 3)
plt.imshow(images_decreased[3])  # Resized image
plt.title('Resized Image (64x64)')

# Gaussian Blured Image
plt.subplot(2, 2, 4)
plt.imshow(images_gb[3])  # Gaussian Blured Image
plt.title('Gaussian Blured Image')

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









Data Preparation for Modeling

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

Split the dataset

In [17]:
```
from sklearn.model_selection import train_test_split

# 10% for testing, 90% for temp (train + validation)
X_temp, X_test, y_temp, y_test = train_test_split(np.array(images_decreased), labels,
                                                  test_size=0.1, random_state=42, stratify=labels)

# Split the remaining 90% into 80% training and 10% validation (10% of total data)
X_train, X_val, y_train, y_val = train_test_split(X_temp, y_temp,
                                                  test_size=0.111, random_state=42, stratify=y_temp)
```

In [18]:
```
print(X_train.shape,y_train.shape)
print(X_val.shape,y_val.shape)
print(X_test.shape,y_test.shape)
```

```
(3800, 64, 64, 3) (3800, 1)
(475, 64, 64, 3) (475, 1)
(475, 64, 64, 3) (475, 1)
```

Encode the target labels

In [19]:
```
# One Hot Encoding
from sklearn.preprocessing import LabelBinarizer
enc = LabelBinarizer()
y_train_encoded = enc.fit_transform(y_train)
```

```
y_val_encoded=enc.transform(y_val)
y_test_encoded=enc.transform(y_test)
```

In [20]:
```
y_train_encoded
```

Out[20]:
```
array([[0, 0, 0, ..., 1, 0, 0],
       [0, 0, 0, ..., 0, 1, 0],
       [0, 0, 0, ..., 0, 0, 0],
       ...,
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 1, 0, 0]])
```

Data Normalization

In [21]:
```
# Normalizing the Images
X_train_normalized = X_train.astype('float32')/255.0
X_val_normalized = X_val.astype('float32')/255.0
X_test_normalized = X_test.astype('float32')/255.0
```

Model Building

In [52]:
```
# Start off by Clearing the Backend
from tensorflow.keras import backend
backend.clear_session()
```

In [53]:
```
# Creating a Random Seed
import random
np.random.seed(42)
random.seed(42)
tf.random.set_seed(42)
```

In [54]:
```
# Intializing a sequential model
model = Sequential()

# Adding first conv layer with 64 filters and kernel size 3x3 , padding 'same' provides the output size same as
# 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 class
model.add(Dense(12, activation='softmax'))

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

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

Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 64, 64, 64)	1,792
max_pooling2d (MaxPooling2D)	(None, 32, 32, 64)	0
conv2d_1 (Conv2D)	(None, 32, 32, 32)	18,464
max_pooling2d_1 (MaxPooling2D)	(None, 16, 16, 32)	0
flatten (Flatten)	(None, 8192)	0
dense (Dense)	(None, 16)	131,088
dropout (Dropout)	(None, 16)	0
dense_1 (Dense)	(None, 12)	204

Total params: 151,548 (591.98 KB)

Trainable params: 151,548 (591.98 KB)

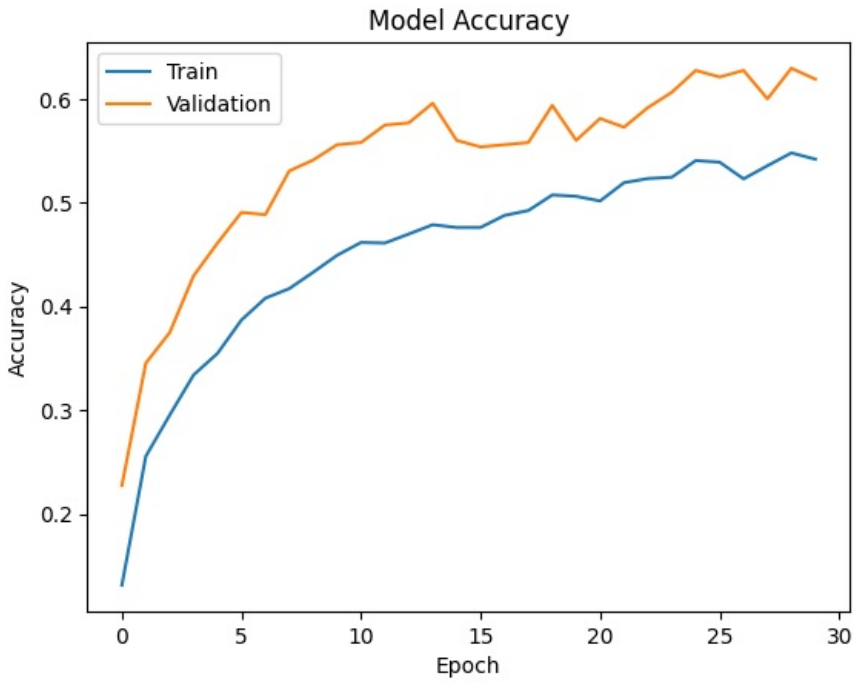
Non-trainable params: 0 (0.00 B)

In [55]:
```
# Training 30 epochs over the data with a batch size of 32
history_1 = model.fit(
            X_train_normalized, y_train_encoded,
            epochs=30,
            validation_data=(X_val_normalized,y_val_encoded),
            batch_size=32,
            verbose=2
)
```

```
Epoch 1/30
119/119 - 8s - 64ms/step - accuracy: 0.1313 - loss: 2.4182 - val_accuracy: 0.2274 - val_loss: 2.2688
Epoch 2/30
119/119 - 1s - 6ms/step - accuracy: 0.2555 - loss: 2.1483 - val_accuracy: 0.3453 - val_loss: 1.9223
Epoch 3/30
119/119 - 1s - 10ms/step - accuracy: 0.2953 - loss: 1.9537 - val_accuracy: 0.3747 - val_loss: 1.7320
Epoch 4/30
119/119 - 1s - 6ms/step - accuracy: 0.3339 - loss: 1.8464 - val_accuracy: 0.4295 - val_loss: 1.6887
Epoch 5/30
119/119 - 1s - 6ms/step - accuracy: 0.3547 - loss: 1.7696 - val_accuracy: 0.4611 - val_loss: 1.5380
Epoch 6/30
119/119 - 1s - 10ms/step - accuracy: 0.3868 - loss: 1.6948 - val_accuracy: 0.4905 - val_loss: 1.4686
Epoch 7/30
119/119 - 1s - 6ms/step - accuracy: 0.4079 - loss: 1.6518 - val_accuracy: 0.4884 - val_loss: 1.4156
Epoch 8/30
119/119 - 1s - 6ms/step - accuracy: 0.4171 - loss: 1.6201 - val_accuracy: 0.5305 - val_loss: 1.4188
Epoch 9/30
119/119 - 1s - 6ms/step - accuracy: 0.4329 - loss: 1.5667 - val_accuracy: 0.5411 - val_loss: 1.3956
Epoch 10/30
119/119 - 1s - 11ms/step - accuracy: 0.4492 - loss: 1.5237 - val_accuracy: 0.5558 - val_loss: 1.3318
Epoch 11/30
119/119 - 1s - 6ms/step - accuracy: 0.4616 - loss: 1.4916 - val_accuracy: 0.5579 - val_loss: 1.3674
Epoch 12/30
119/119 - 1s - 10ms/step - accuracy: 0.4611 - loss: 1.4750 - val_accuracy: 0.5747 - val_loss: 1.2926
Epoch 13/30
119/119 - 1s - 6ms/step - accuracy: 0.4697 - loss: 1.4338 - val_accuracy: 0.5768 - val_loss: 1.2969
Epoch 14/30
119/119 - 1s - 6ms/step - accuracy: 0.4787 - loss: 1.4201 - val_accuracy: 0.5958 - val_loss: 1.2471
Epoch 15/30
119/119 - 1s - 10ms/step - accuracy: 0.4761 - loss: 1.4019 - val_accuracy: 0.5600 - val_loss: 1.3536
Epoch 16/30
119/119 - 1s - 11ms/step - accuracy: 0.4761 - loss: 1.4102 - val_accuracy: 0.5537 - val_loss: 1.3302
Epoch 17/30
119/119 - 1s - 5ms/step - accuracy: 0.4876 - loss: 1.3578 - val_accuracy: 0.5558 - val_loss: 1.3495
Epoch 18/30
119/119 - 1s - 6ms/step - accuracy: 0.4924 - loss: 1.3516 - val_accuracy: 0.5579 - val_loss: 1.2927
Epoch 19/30
119/119 - 1s - 11ms/step - accuracy: 0.5074 - loss: 1.3002 - val_accuracy: 0.5937 - val_loss: 1.2294
Epoch 20/30
119/119 - 1s - 6ms/step - accuracy: 0.5061 - loss: 1.3099 - val_accuracy: 0.5600 - val_loss: 1.3006
Epoch 21/30
119/119 - 1s - 6ms/step - accuracy: 0.5016 - loss: 1.3050 - val_accuracy: 0.5811 - val_loss: 1.2364
Epoch 22/30
119/119 - 1s - 11ms/step - accuracy: 0.5192 - loss: 1.2704 - val_accuracy: 0.5726 - val_loss: 1.2508
Epoch 23/30
119/119 - 1s - 6ms/step - accuracy: 0.5232 - loss: 1.2541 - val_accuracy: 0.5916 - val_loss: 1.2032
Epoch 24/30
119/119 - 1s - 7ms/step - accuracy: 0.5245 - loss: 1.2376 - val_accuracy: 0.6063 - val_loss: 1.2190
Epoch 25/30
119/119 - 1s - 7ms/step - accuracy: 0.5405 - loss: 1.2012 - val_accuracy: 0.6274 - val_loss: 1.1741
Epoch 26/30
119/119 - 1s - 9ms/step - accuracy: 0.5389 - loss: 1.2056 - val_accuracy: 0.6211 - val_loss: 1.1826
Epoch 27/30
119/119 - 1s - 5ms/step - accuracy: 0.5229 - loss: 1.2145 - val_accuracy: 0.6274 - val_loss: 1.1519
Epoch 28/30
119/119 - 1s - 11ms/step - accuracy: 0.5355 - loss: 1.1892 - val_accuracy: 0.6000 - val_loss: 1.2241
Epoch 29/30
119/119 - 1s - 10ms/step - accuracy: 0.5479 - loss: 1.1685 - val_accuracy: 0.6295 - val_loss: 1.1664
Epoch 30/30
119/119 - 1s - 11ms/step - accuracy: 0.5418 - loss: 1.1823 - val_accuracy: 0.6189 - val_loss: 1.1935
```

Model Evaluation

In [56]:
```
# Displaying a plot to show how the model's train and validation data accuracy metric
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()
```



- As seen from the above graph, the validation accuracy performed a little better than the training accuracy.
- However, the scores are really close to each other marking a sign of not overfiting, with that being said the scores are still low which is not the best. But the model does seem to be generalized pretty well.

Evaluating the Test Data

In [58]:
```
accuracy = model.evaluate(X_test_normalized, y_test_encoded, verbose=2)
```

```
15/15 - 0s - 3ms/step - accuracy: 0.6168 - loss: 1.2285
```

- With the test data having an accuracy that is around the interval of the validation score, and not too far off from the training score, this shows that the model is not showing signs of overfitting, in fact, it is more generalized.

Making Predictions on the Test Data

In [59]:
```
y_pred=model.predict(X_test_normalized)
```

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

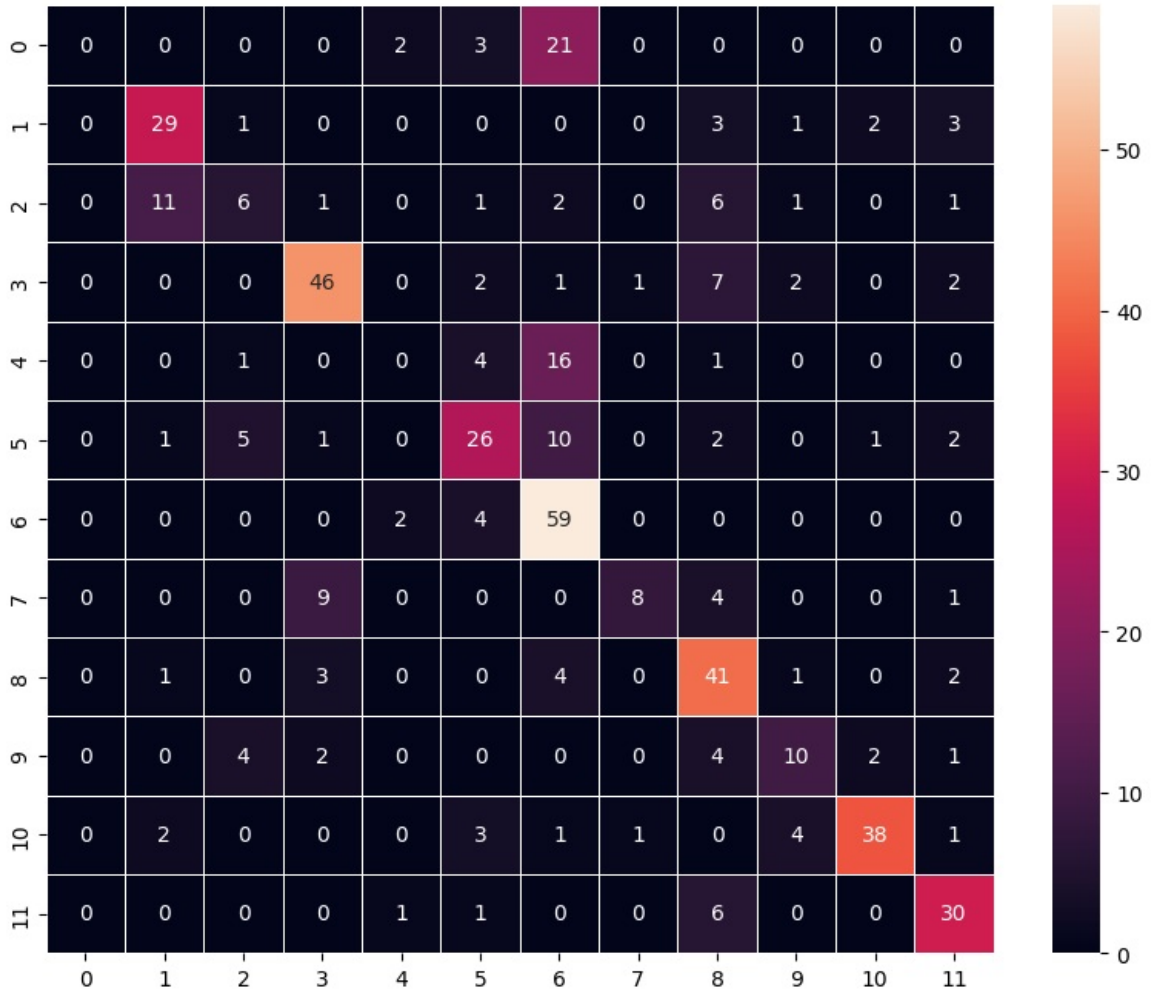
In [60]:
```
# Showcasing the results
y_pred
```

Out[60]:
```
array([[3.4451769e-10, 4.6323798e-09, 1.2246487e-07, ..., 1.1128162e-02,
        2.0847262e-09, 1.6418296e-04],
       [3.4492561e-13, 2.1680938e-02, 1.5308570e-02, ..., 1.2397158e-01,
        8.2719088e-01, 8.2468055e-04],
       [6.1466942e-08, 4.6987083e-02, 6.5204255e-02, ..., 1.5578106e-01,
        6.4039713e-01, 2.1238957e-02],
       ...,
       [3.0217871e-01, 1.2326831e-09, 2.3680632e-06, ..., 1.3313968e-10,
        9.6402152e-07, 4.4557378e-06],
       [4.0666146e-05, 1.5349439e-04, 2.8393960e-03, ..., 1.4500564e-02,
        2.8733739e-06, 3.7592895e-02],
       [1.8775779e-03, 1.3367867e-01, 1.8454260e-01, ..., 1.2448495e-01,
        1.6239774e-01, 7.4797370e-02]], dtype=float32)
```

Confusion Matrix

In [61]:
```
# Obtaining the categorical values from y_test_encoded and y_pred
y_pred_arg=np.argmax(y_pred,axis=1)
y_test_arg=np.argmax(y_test_encoded,axis=1)

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

```
plt.show()
```



- In class 6, there seems to be a big misclassification compared to the rest of the classes, especially on (0,6), as well as (4,6).
- Furthermore, in classes 1, 2, 3, 4, and 11, contain misclassifications.
- Some classes such as 0, 7, and 9 contain very minimal errors.

Model Performance Improvement

In [62]:
```
# 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)
```

Data Augmentation

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

In [63]:
```
from tensorflow.keras.preprocessing.image import ImageDataGenerator

# Data augmentation for training set
train_datagen = ImageDataGenerator(
    rotation_range=30,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,
    fill_mode='nearest')

# Apply augmentation on training data
train_generator = train_datagen.flow(X_train_normalized, y_train_encoded, batch_size=32)
```

In [64]:
```
model2 = Sequential()

# First Convolutional Layer
```

```
model2.add(Conv2D(64, (3, 3), activation='relu', padding='same', input_shape=(64, 64, 3)))
model2.add(MaxPooling2D(pool_size=(2, 2), padding='same'))

# Second Convolutional Layer
model2.add(Conv2D(32, (3, 3), activation='relu', padding='same'))
model2.add(MaxPooling2D(pool_size=(2, 2), padding='same'))

# Flatten the output
model2.add(Flatten())

# Fully connected layer
model2.add(Dense(128, activation='relu'))
model2.add(Dropout(0.3))

# Output layer for multi-class classification
model2.add(Dense(12, activation='softmax'))  # Assuming 12 classes

# Compile the Model ###
opt = Adam()
model2.compile(optimizer=opt, loss='categorical_crossentropy', metrics=['accuracy'])
```

Reducing the Learning Rate:

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

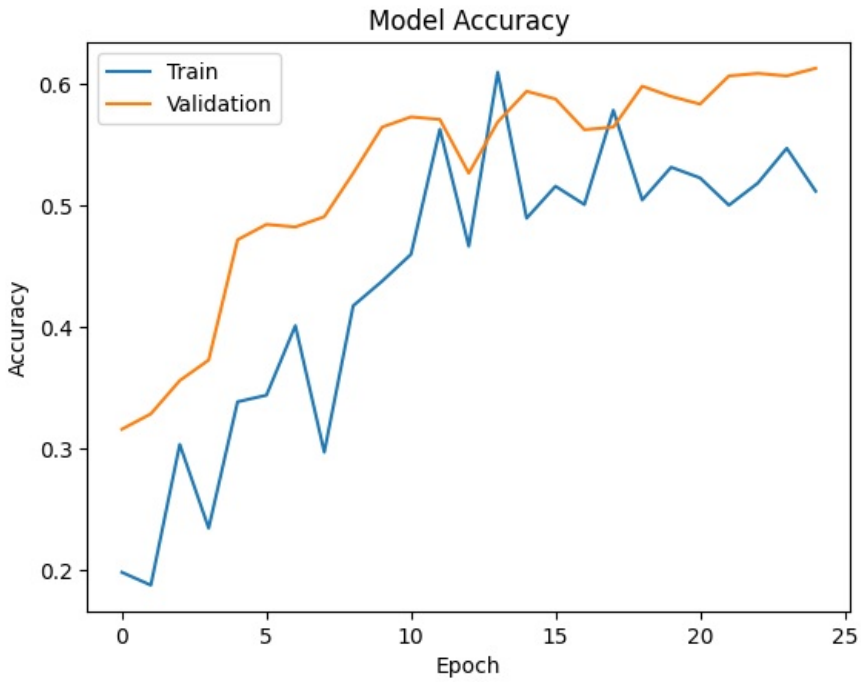
In [65]:
```
from tensorflow.keras.callbacks import ReduceLROnPlateau  # Import ReduceLROnPlateau

# Learning Rate Reduction
reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.2, patience=3, min_lr=0.0001)
```

In [66]:
```
# Epochs
epochs = 25
# Batch size
batch_size = 64

# Create the training data generator
train_generator = train_datagen.flow(X_train_normalized,
                                     y_train_encoded,
                                     batch_size=batch_size,
                                     seed=42,
                                     shuffle=False)

# Fit the model
history = model2.fit(train_generator, # Pass the generator here
                     epochs=epochs,
                     steps_per_epoch=X_train_normalized.shape[0] // batch_size,
                     validation_data=(X_val_normalized, y_val_encoded),
                     verbose=1,
                     callbacks=[reduce_lr])
```

```
Epoch 1/25
59/59 ━━━━━━━━━━━━━━━━━━━━ 14s 189ms/step - accuracy: 0.1603 - loss: 2.4159 - val_accuracy: 0.3158 - val_loss:
2.0827 - learning_rate: 0.0010
Epoch 2/25
59/59 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.1875 - loss: 2.2324 - val_accuracy: 0.3284 - val_loss: 2.0
488 - learning_rate: 0.0010
Epoch 3/25
59/59 ━━━━━━━━━━━━━━━━━━━━ 7s 104ms/step - accuracy: 0.2916 - loss: 2.0611 - val_accuracy: 0.3558 - val_loss: 1
.7856 - learning_rate: 0.0010
Epoch 4/25
59/59 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.2344 - loss: 1.9302 - val_accuracy: 0.3726 - val_loss: 1.7
578 - learning_rate: 0.0010
Epoch 5/25
59/59 ━━━━━━━━━━━━━━━━━━━━ 5s 74ms/step - accuracy: 0.3155 - loss: 1.9005 - val_accuracy: 0.4716 - val_loss: 1.
6290 - learning_rate: 0.0010
Epoch 6/25
59/59 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.3438 - loss: 1.8560 - val_accuracy: 0.4842 - val_loss: 1.5
450 - learning_rate: 0.0010
Epoch 7/25
59/59 ━━━━━━━━━━━━━━━━━━━━ 8s 124ms/step - accuracy: 0.3964 - loss: 1.7263 - val_accuracy: 0.4821 - val_loss: 1
.6090 - learning_rate: 0.0010
Epoch 8/25
59/59 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.2969 - loss: 2.0932 - val_accuracy: 0.4905 - val_loss: 1.5
586 - learning_rate: 0.0010
Epoch 9/25
59/59 ━━━━━━━━━━━━━━━━━━━━ 7s 101ms/step - accuracy: 0.4142 - loss: 1.6508 - val_accuracy: 0.5263 - val_loss: 1
.4612 - learning_rate: 0.0010
Epoch 10/25
59/59 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.4375 - loss: 1.6141 - val_accuracy: 0.5642 - val_loss: 1.3
982 - learning_rate: 0.0010
Epoch 11/25
59/59 ━━━━━━━━━━━━━━━━━━━━ 9s 95ms/step - accuracy: 0.4544 - loss: 1.5489 - val_accuracy: 0.5726 - val_loss: 1.
3038 - learning_rate: 0.0010
Epoch 12/25
59/59 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.5625 - loss: 1.3695 - val_accuracy: 0.5705 - val_loss: 1.3
365 - learning_rate: 0.0010
Epoch 13/25
59/59 ━━━━━━━━━━━━━━━━━━━━ 9s 74ms/step - accuracy: 0.4680 - loss: 1.5302 - val_accuracy: 0.5263 - val_loss: 1.
4111 - learning_rate: 0.0010
Epoch 14/25
59/59 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.6094 - loss: 1.3815 - val_accuracy: 0.5684 - val_loss: 1.3
357 - learning_rate: 0.0010
Epoch 15/25
59/59 ━━━━━━━━━━━━━━━━━━━━ 7s 100ms/step - accuracy: 0.4835 - loss: 1.4510 - val_accuracy: 0.5937 - val_loss: 1
.2702 - learning_rate: 2.0000e-04
Epoch 16/25
59/59 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.5156 - loss: 1.4084 - val_accuracy: 0.5874 - val_loss: 1.2
643 - learning_rate: 2.0000e-04
Epoch 17/25
59/59 ━━━━━━━━━━━━━━━━━━━━ 9s 74ms/step - accuracy: 0.4879 - loss: 1.4637 - val_accuracy: 0.5621 - val_loss: 1.
3117 - learning_rate: 2.0000e-04
Epoch 18/25
59/59 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.5781 - loss: 1.2705 - val_accuracy: 0.5642 - val_loss: 1.2
995 - learning_rate: 2.0000e-04
Epoch 19/25
59/59 ━━━━━━━━━━━━━━━━━━━━ 7s 99ms/step - accuracy: 0.5105 - loss: 1.4290 - val_accuracy: 0.5979 - val_loss: 1.
2337 - learning_rate: 2.0000e-04
Epoch 20/25
59/59 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.5312 - loss: 1.3614 - val_accuracy: 0.5895 - val_loss: 1.2
401 - learning_rate: 2.0000e-04
Epoch 21/25
59/59 ━━━━━━━━━━━━━━━━━━━━ 5s 74ms/step - accuracy: 0.5281 - loss: 1.3996 - val_accuracy: 0.5832 - val_loss: 1.
2699 - learning_rate: 2.0000e-04
Epoch 22/25
59/59 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.5000 - loss: 1.4596 - val_accuracy: 0.6063 - val_loss: 1.2
475 - learning_rate: 2.0000e-04
Epoch 23/25
59/59 ━━━━━━━━━━━━━━━━━━━━ 6s 97ms/step - accuracy: 0.5070 - loss: 1.4230 - val_accuracy: 0.6084 - val_loss: 1.
2533 - learning_rate: 1.0000e-04
Epoch 24/25
59/59 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.5469 - loss: 1.3759 - val_accuracy: 0.6063 - val_loss: 1.2
538 - learning_rate: 1.0000e-04
Epoch 25/25
59/59 ━━━━━━━━━━━━━━━━━━━━ 5s 75ms/step - accuracy: 0.5114 - loss: 1.4158 - val_accuracy: 0.6126 - val_loss: 1.
2112 - learning_rate: 1.0000e-04
```

In [67]:

```
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()
```



- The training and validation test scores still show some signs of overfitting, however, both scores performed lower and are much further apart compared to the first model.

Evaluating the Test Data

In [69]:
```
accuracy = model2.evaluate(X_test_normalized, y_test_encoded, verbose=2)
```

```
15/15 - 0s - 4ms/step - accuracy: 0.6253 - loss: 1.1975
```

- The test score still remains around the area of both the training and validation accuracy scores.

Making Predictions on the Test Data

In [70]:
```
y_pred=model2.predict(X_test_normalized)
```

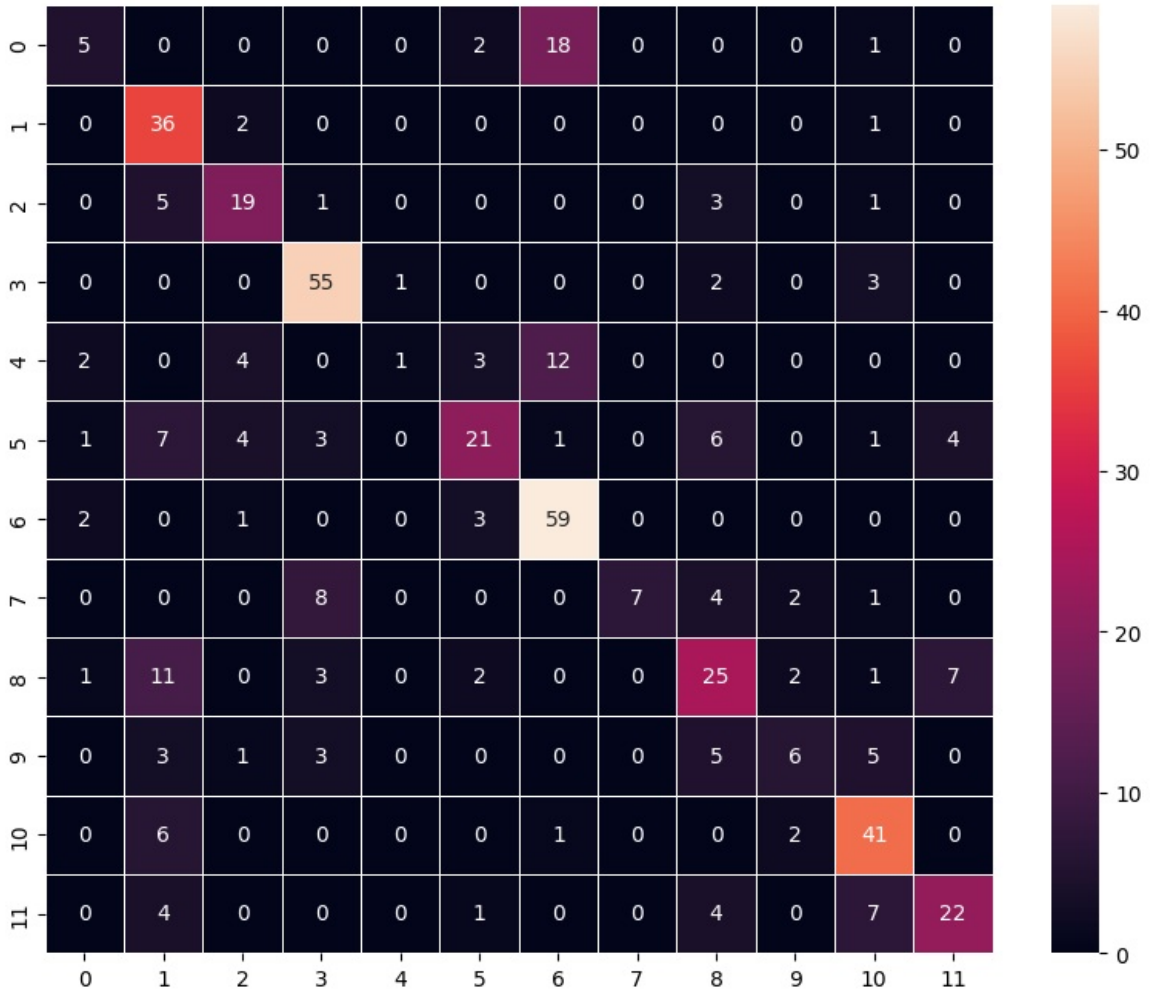
```
15/15 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step
```

In [71]:
```
# Showcasing the results
y_pred
```

Out[71]:
```
array([[9.91447848e-08, 7.24846404e-06, 2.37511273e-08, ...,
        2.72402223e-02, 6.68619899e-03, 2.06663390e-04],
       [1.50886115e-06, 4.53624278e-01, 2.12680306e-02, ...,
        1.35455310e-01, 2.13239938e-01, 9.80687514e-03],
       [1.20298018e-05, 4.33197059e-02, 1.62098987e-03, ...,
        2.22216368e-01, 3.22958559e-01, 1.45552969e-02],
       ...,
       [3.25565040e-01, 6.91932155e-06, 3.52285220e-04, ...,
        3.38106511e-05, 1.36095900e-04, 1.47102901e-03],
       [1.15147101e-04, 2.03371510e-01, 6.47721067e-02, ...,
        1.20214514e-01, 5.73326945e-02, 3.19215581e-02],
       [2.08291051e-04, 2.53269047e-01, 1.30244084e-02, ...,
        1.18003294e-01, 3.09163928e-01, 4.94714007e-02]], dtype=float32)
```

Confusion Matrix

In [72]:
```
# Obtaining the categorical values from y_test_encoded and y_pred
y_pred_arg=np.argmax(y_pred,axis=1)
y_test_arg=np.argmax(y_test_encoded,axis=1)

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



- In class 6, there still seems to be a big misclassification compared to the rest of the classes, especially on (0,6), as well as (4,6).
- Furthermore, in classes 1, 3, and 11, contain misclassifications.
- Some classes such as 0, 7, and 9 contain very minimal errors. The previous and the current models contain the same confusion matrix layout. However, model 2 appears to have more amount of misclassifications.

Final Model

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

In [75]:
```
pd.DataFrame({'Models':['Base CNN Model','CNN Model with Data Augmentation',],'Train Accuracy':['53%','51%'],'V
```

Out[75]:

	Models	Train Accuracy	Validation Accuracy	Test Accuracy
0	Base CNN Model	53%	62%	62%
1	CNN Model with Data Augmentation	51%	61%	62%

As seen from the above table, Model 1 (Base CNN Model) turns out to be slightly better due to the validation and test accuracy being more generalized. Even though both models have very similar accuracy scores across the training, validation, and tests, the data augmentation did not really improve from the first to the second model too much. Furthermore, the confusion matrix in Model 2 appeared to have many more misclassifications than in Model 1, although the classes that contained the most misclassifications where the same for both models.

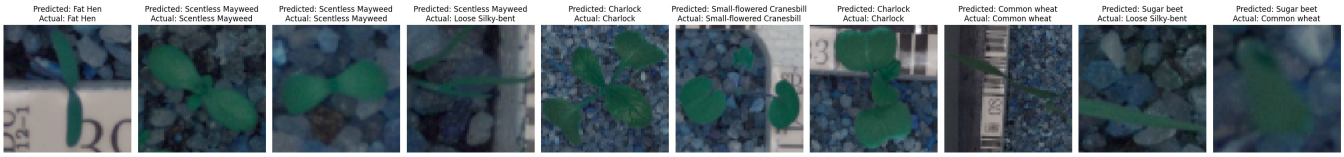
- Train Accuracy for both models remains consistent at 53%, indicating that neither model overfits the training data. This low training accuracy suggests that the models may not be complex enough to capture the underlying patterns in the data.
- Validation Accuracy and Test Accuracy show only a slight improvement in the Base CNN Model compared to the augmented model. However, both models converge to approximately 63-64% accuracy on unseen data, showing that the models are learning but not reaching high generalization levels.
- The confusion matrix indicates that misclassifications are prominent in both models, but Model 2 (CNN with Data Augmentation) suffers from slightly more misclassifications. Data augmentation typically improves model performance, but here it does not seem to have helped.

Visualizing the prediction

Visualizing Test Images in Model 1

In [79]:
```
import matplotlib.pyplot as plt
import numpy as np

# Assuming you have a dictionary called 'class_names' mapping labels to species names
# For example: class_names = {0: 'Rose', 1: 'Tulip', 2: 'Sunflower', ...}
class_names = {0: "Black-grass", 1: "Charlock", 2: "Cleavers", 3: "Common Chickweed", 4: "Common wheat", 5: "Fa
               7: "Maize", 8: "Scentless Mayweed", 9: "Shepherds Purse", 10: "Small-flowered Cranesbill", 11: "S


def visualize_predictions(model, X_test, y_test, num_images=5):
    # Select a random subset of images to visualize
    indices = np.random.choice(len(X_test), num_images, replace=False)
    sample_images = X_test[indices]
    # Access the label values using .iloc to index by row number
    sample_labels = y_test.iloc[indices]
    # Extract the numerical label values from the DataFrame
    # Convert labels to numerical values using the class_names dictionary
    true_classes = [list(class_names.keys())[list(class_names.values()).index(label)] for label in sample_label
    #sample_labels = sample_labels.values

    # Get model predictions
    predictions = model.predict(sample_images)
    predicted_classes = np.argmax(predictions, axis=1)

    # Map predicted and true labels to species names
    predicted_species = [class_names[label] for label in predicted_classes]
    true_species = [class_names[label] for label in true_classes] # Use true_classes instead of sample_labels

    # Plot the images with predicted and actual labels
    plt.figure(figsize=(30, 25))
    for i in range(num_images):
        plt.subplot(1, num_images, i + 1)
        plt.imshow(sample_images[i])
        plt.title(f"Predicted: {predicted_species[i]}\nActual: {true_species[i]}")
        plt.axis('off')
    plt.tight_layout()
    plt.show()

# Call the function to visualize predictions
visualize_predictions(model, X_test, y_test, num_images=10)
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 16ms/step
```



Out of the 10 images, 7/10 plant species were correctly predicted as their rightful name from the base model. This is a 70% accuracy which somewhat aligns better with what resulted from the test results from the base model, but it is only 10 out of the 4000+ images in the dataset.

Actionable Insights and Business Recommendations

Key Findings

- Despite applying data augmentation techniques, the model accuracy did not improve as expected. This suggests that the data may not benefit much from augmentation or that the augmentation techniques applied (e.g., rotation, flipping, zooming) did not introduce meaningful diversity to help the model generalize better.
- The confusion matrix analysis shows that certain classes are consistently misclassified across both models. This points to specific challenging classes where the features extracted by the CNN models might be insufficient to distinguish between similar categories of images.

Next Steps for Improvement

- Try using more sophisticated augmentation techniques such as color jittering, contrast adjustment, cutout, or mixup. These can provide more variety and diversity in training samples, helping the model generalize better.
- Since both models performed similar and struggled to capture sufficient training accuracy scores, some things that can be improved to perhaps make the accuracy better are:
  - Adding more layers to the CNN model, increasing the capacity to learn more complex features.
  - Increasing the number of filters in the Conv2D layers.
  - Introducing batch normalization after convolution layers, which helps stabilize training and potentially improve model performance.
  - Using dropout more effectively to prevent overfitting, and experimenting with different dropout rates.
- Perform hyperparameter tuning to optimize parameters such as learning rate, optimizer choice (e.g., Adam, SGD with momentum), and batch size. This may improve how well the model learns from data.
- Perhaps implementing a pre-trained CNN model (e.g., ResNet, VGG16) and fine-tuning it on your dataset could provide better

feature extraction, leading to improved accuracy on both training and test sets.

Business Insights

- Given that both models achieve roughly around 60-65% accuracy, neither is ready for production deployment without further optimization. However, Model 1 (Base CNN Model) performs slightly better, indicating that data augmentation alone isn't enough, and more aggressive techniques are needed to boost model performance.
- The low accuracy across both models suggests potential issues with the dataset, such as noisy, low-quality images or insufficient diversity across classes. Improving the quality of the training data may result in better model performance.

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js