Potato Disease Classification

Dataset credits: https://www.kaggle.com/arjuntejaswi/plant-village

Importing dependencies

In [1]:
```
import tensorflow as tf
from tensorflow.keras import models, layers
import matplotlib.pyplot as plt
```

Set all the Constants

In [2]:
```
IMAGE_SIZE = 256
BATCH_SIZE = 32
CHANNELS =3
EPOCHS = 10
```

Import data into tensorflow dataset object

We will use image_dataset_from_directory api to load all images in tensorflow dataset: https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/image_dataset_from_directory

In [3]:
```
dataset = tf.keras.preprocessing.image_dataset_from_directory(
    "/content/drive/MyDrive/PlantVillage",
    shuffle = True,
    image_size = (IMAGE_SIZE, IMAGE_SIZE),
    batch_size = BATCH_SIZE
)
```

```
Found 2152 files belonging to 3 classes.
```

In [4]:

```
class_names = dataset.class_names
class_names
```

Out[4]:

```
['Potato___Early_blight', 'Potato___Late_blight', 'Potato___healthy']
```

In [7]:

```
for image_batch, labels_batch in dataset.take(1):
    print(image_batch.shape)
    print(labels_batch.numpy())
```

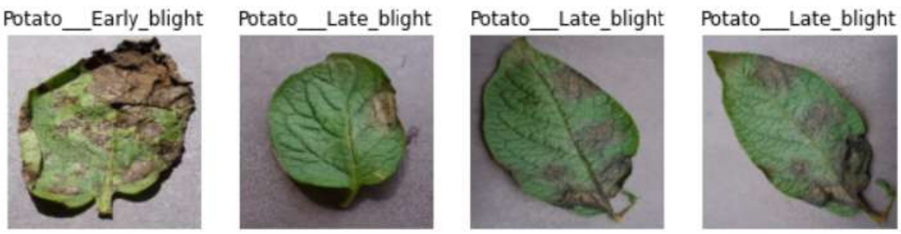
```
(32, 256, 256, 3)
[2 0 1 1 1 0 1 0 1 1 1 0 1 0 0 0 0 1 0 0 1 1 1 1 1 1 1 0 1 1 1 0]
```

As you can see above, each element in the dataset is a tuple. First element is a batch of 32 elements of images. Second element is a batch of 32 elements of class labels

Visualize some of the images from our dataset

In [8]:

```
plt.figure(figsize=(10,10))
for image_batch, label_batch in dataset.take(1):
  for i in range(12):
    ax = plt.subplot(3,4,1+i)
    plt.imshow(image_batch[i].numpy().astype("uint8"))
    plt.title(class_names[label_batch[i]])
    plt.axis("off")
```



Function to Split Dataset

Dataset should be bifurcated into 3 subsets, namely:

1. Training: Dataset to be used while training.
2. Validation: Dataset to be tested against while training
3. Test: Dataset to be tested against after we trained a model

In [9]:
```
len(dataset)
```

Out[9]: 68

In [10]:
```
train_size = 0.8
len(dataset)*train_size
```
Out[10]: 54.400000000000006

In [11]:
```
train_ds = dataset.take(54)
len(train_ds)
```
Out[11]: 54

In [12]:
```
test_ds = dataset.skip(54)
len(test_ds)
```
Out[12]: 14

In [13]:
```
val_size = 0.1
len(dataset)*val_size
```
Out[13]: 6.800000000000001

In [14]:
```
val_ds = test_ds.take(6)
len(val_ds)
```
Out[14]: 6

In [15]:
```
test_ds = test_ds.skip(6)
len(test_ds)
```
Out[15]: 8

In [16]:

```
def get_dataset_partitions_tf(ds, train_split=0.8, val_split=0.1, test_split=0.1, shuffle=True, shuffle_size=10000):
  assert  (train_split + test_split + val_split) == 1

  ds_size = len(ds)

  if shuffle:
    ds = ds.shuffle(shuffle_size, seed=12)

  train_size = int(train_split * ds_size)
  val_size = int(val_split * ds_size)

  val_size = int(val_split * ds_size)

  train_ds = ds.take(train_size)
  val_ds = ds.skip(train_size).take(val_size)
  test_ds = ds.skip(train_size).skip(val_size)

  return train_ds, val_ds, test_ds
```

In [17]:

```
train_ds, val_ds, test_ds = get_dataset_partitions_tf(dataset)
```

In [18]:

```
len(train_ds)
```

Out[18]: 54

In [19]:

```
len(val_ds)
```

Out[19]: 6

In [20]:

```
len(test_ds)
```

Out[20]: 8

Cache, Shuffle, and Prefetch the Dataset

In [21]:
```
train_ds = train_ds.cache().shuffle(1000).prefetch(buffer_size= tf.data.AUTOTUNE)
val_ds = val_ds.cache().shuffle(1000).prefetch(buffer_size= tf.data.AUTOTUNE)
test_ds = test_ds.cache().shuffle(1000).prefetch(buffer_size= tf.data.AUTOTUNE)
```

Building the Model

Creating a Layer for Resizing and Normalization

Before we feed our images to network, we should be resizing it to the desired size. Moreover, to improve model performance, we should normalize the image pixel value (keeping them in range 0 and 1 by dividing by 256). This should happen while training as well as inference. Hence we can add that as a layer in our Sequential Model.

In [22]:
```
resize_and_rescale = tf.keras.Sequential([
  layers.experimental.preprocessing.Resizing(IMAGE_SIZE, IMAGE_SIZE),
  layers.experimental.preprocessing.Rescaling(1.0/255),
])
```

Data Augmentation

Data Augmentation is needed when we have less data, this boosts the accuracy of our model by augmenting the data.

In [23]:
```
data_augmentation = tf.keras.Sequential([
 layers.experimental.preprocessing.RandomFlip("horizontal_and_vertical"),
 layers.experimental.preprocessing.RandomRotation(0.2),
])
```

Applying Data Augmentation to Train Dataset

In [24]:
```
train_ds = train_ds.map(
    lambda x, y: (data_augmentation(x, training=True), y)
).prefetch(buffer_size=tf.data.AUTOTUNE)
```

Model Architecture

We use a CNN coupled with a Softmax activation in the output layer. We also add the initial layers for resizing, normalization and Data Augmentation.

We are going to use convolutional neural network (CNN) here. CNN is popular for image classification tasks. Watch below video to understand fundamentals of CNN

In [25]:
```
input_shape = (BATCH_SIZE, IMAGE_SIZE, IMAGE_SIZE, CHANNELS)
n_classes = 3

model = models.Sequential([
    resize_and_rescale,
    layers.Conv2D(32, kernel_size = (3,3), activation='relu', input_shape=input_shape),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64,  kernel_size = (3,3), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64,  kernel_size = (3,3), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    layers.Flatten(),
```

```python
    layers.Dense(64, activation='relu'),
    layers.Dense(n_classes, activation='softmax'),
])

model.build(input_shape=input_shape)
```

In [26]:

```python
model.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 sequential (Sequential)     (32, 256, 256, 3)         0         
                                                                 
 conv2d (Conv2D)             (32, 254, 254, 32)        896       
                                                                 
 max_pooling2d (MaxPooling2D  (32, 127, 127, 32)       0         
 )                                                               
                                                                 
 conv2d_1 (Conv2D)           (32, 125, 125, 64)        18496     
                                                                 
 max_pooling2d_1 (MaxPooling  (32, 62, 62, 64)         0         
 2D)                                                             
                                                                 
 conv2d_2 (Conv2D)           (32, 60, 60, 64)          36928     
                                                                 
 max_pooling2d_2 (MaxPooling  (32, 30, 30, 64)         0         
 2D)                                                             
                                                                 
 conv2d_3 (Conv2D)           (32, 28, 28, 64)          36928     
                                                                 
 max_pooling2d_3 (MaxPooling  (32, 14, 14, 64)         0         
 2D)                                                             
                                                                 
 conv2d_4 (Conv2D)           (32, 12, 12, 64)          36928     
```

```
 conv2d_4 (Conv2D)           (32, 12, 12, 64)          36928

 max_pooling2d_4 (MaxPooling  (32, 6, 6, 64)           0
 2D)

 conv2d_5 (Conv2D)           (32, 4, 4, 64)            36928

 max_pooling2d_5 (MaxPooling  (32, 2, 2, 64)           0
 2D)

 flatten (Flatten)           (32, 256)                 0

 dense (Dense)               (32, 64)                  16448

 dense_1 (Dense)             (32, 3)                   195

=================================================================
Total params: 183,747
Trainable params: 183,747
Non-trainable params: 0
_________________________________________________________________
```

Compiling the Model

We use adam Optimizer, SparseCategoricalCrossentropy for losses, accuracy as a metric

In [27]:

```
model.compile(
    optimizer = 'adam',
    loss = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=False),
    metrics = ['accuracy']
)
```

In [28]:

```
history = model.fit(
    train_ds,
    epochs = EPOCHS,
    batch_size = BATCH_SIZE,
    verbose = 1,
    validation_data = val_ds
)
```

```
Epoch 1/10
54/54 [==============================] - 397s 3s/step - loss: 0.9037 - accuracy: 0.5023 - val_loss: 0.8763 - val_accuracy: 0.
6406
Epoch 2/10
54/54 [==============================] - 175s 3s/step - loss: 0.6836 - accuracy: 0.7274 - val_loss: 1.1239 - val_accuracy: 0.
5365
Epoch 3/10
54/54 [==============================] - 174s 3s/step - loss: 0.4899 - accuracy: 0.7998 - val_loss: 0.5737 - val_accuracy: 0.
7448
Epoch 4/10
54/54 [==============================] - 174s 3s/step - loss: 0.3376 - accuracy: 0.8611 - val_loss: 0.2391 - val_accuracy: 0.
9062
Epoch 5/10
54/54 [==============================] - 173s 3s/step - loss: 0.2831 - accuracy: 0.8866 - val_loss: 0.4325 - val_accuracy: 0.
7917
Epoch 6/10
54/54 [==============================] - 173s 3s/step - loss: 0.2391 - accuracy: 0.9057 - val_loss: 0.1906 - val_accuracy: 0.
9115
Epoch 7/10
54/54 [==============================] - 173s 3s/step - loss: 0.2075 - accuracy: 0.9201 - val_loss: 0.2772 - val_accuracy: 0.
9062
Epoch 8/10
54/54 [==============================] - 173s 3s/step - loss: 0.2129 - accuracy: 0.9161 - val_loss: 0.1976 - val_accuracy: 0.
9271
Epoch 9/10
54/54 [==============================] - 174s 3s/step - loss: 0.1979 - accuracy: 0.9144 - val_loss: 0.2772 - val_accuracy: 0.
8802
Epoch 10/10
54/54 [==============================] - 173s 3s/step - loss: 0.2155 - accuracy: 0.9109 - val_loss: 0.1297 - val_accuracy: 0.
9427
```

In [30]:
```
scores = model.evaluate(test_ds)
```

```
8/8 [==============================] - 11s 762ms/step - loss: 0.1287 - accuracy: 0.9570
```

You can see above that we get 95.70% accuracy for our test dataset. This is considered to be a pretty good accuracy

In [31]:
```
scores
```

Out[31]:
```
[0.1286834180355072, 0.95703125]
```

Scores is just a list containing loss and accuracy value

In [32]:
```
history.params
```

Out[32]:
```
{'epochs': 10, 'steps': 54, 'verbose': 1}
```

In [33]:
```
history.history.keys()
```

Out[33]:
```
dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])
```

loss, accuracy, val loss etc are a python list containing values of loss, accuracy etc at the end of each epoch

In [34]:
```
type(history.history['loss'])
```

Out[34]:
```
list
```

In [35]:
```
len(history.history['loss'])
```

Out[35]:
```
10
```

In [36]:

```
history.history['loss'][:5]
```

Out[36]:

```
[0.9037485718727112,
 0.683612048625946,
 0.48990094661712646,
 0.33762332797050476,
 0.28306978940963745]
```

In [37]:

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

loss = history.history['loss']
val_loss = history.history['val_loss']
```

Run prediction on a sample image

In [38]:

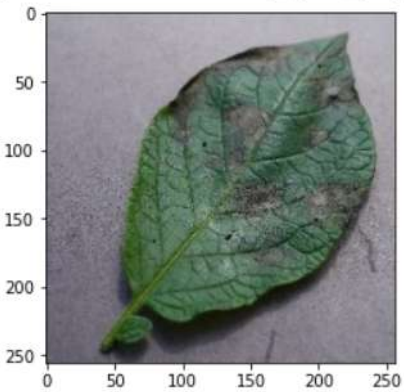
```
plt.figure(figsize=(8,8))
plt.subplot(1, 2, 1)
plt.plot(range(EPOCHS), acc, label = 'Training Accuracy')
plt.plot(range(EPOCHS), val_acc, label = 'Validation Loss')
plt.legend(loc = 'lower right')
plt.title('Training and Validation Accuracy')

plt.subplot(1, 2, 2)
plt.plot(range(EPOCHS), loss, label='Training Loss')
plt.plot(range(EPOCHS), val_loss, label='Validation Loss')
plt.legend(loc='upper right')
plt.title('Training and Validation Loss')
plt.show()
```

In [39]:

```
import numpy as np
for images_batch, labels_batch in test_ds.take(1):

  first_image = images_batch[0].numpy().astype('uint8')
  first_label = labels_batch[0].numpy()
```

```
first_image = images_batch[0].numpy().astype('uint8')
first_label = labels_batch[0].numpy()

print("first image to predict")
plt.imshow(first_image)
print("actual label:", class_names[first_label])

batch_prediction = model.predict(images_batch)
print("predicted label:", class_names[np.argmax(batch_prediction[0])])
```

```
first image to predict
actual label: Potato___Late_blight
predicted label: Potato___Late_blight
```

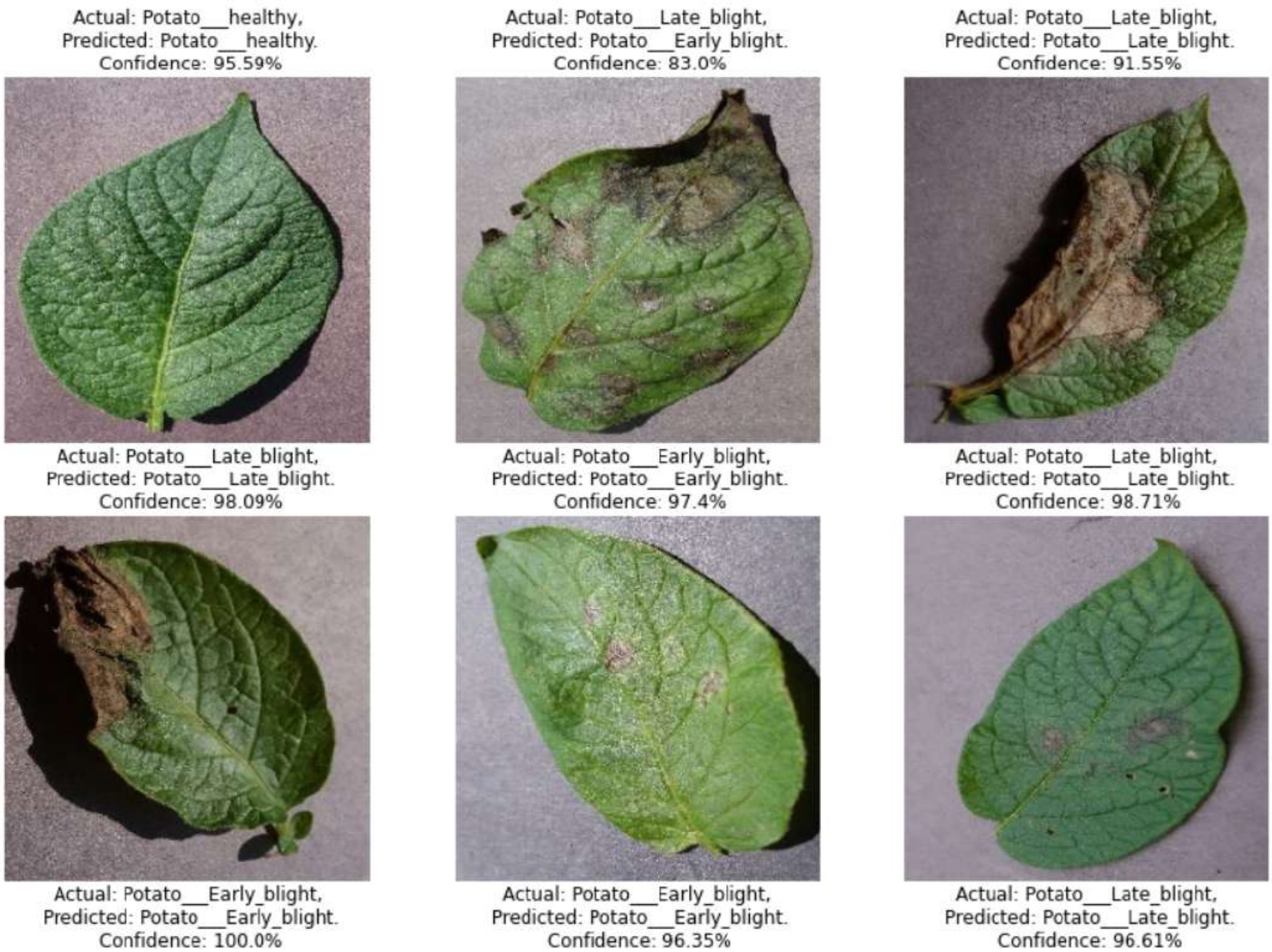


Write a function for inference

In [40]:
```python
def predict(model, img):
    img_array = tf.keras.preprocessing.image.img_to_array(images[i].numpy())
    img_array = tf.expand_dims(img_array, 0)

    predictions = model.predict(img_array)

    predicted_class = class_names[np.argmax(predictions[0])]
    confidence = round(100 * (np.max(predictions[0])), 2)
    return predicted_class, confidence
```

Now run inference on few sample images

In [41]:
```python
plt.figure(figsize=(15, 15))
for images, labels in test_ds.take(1):
    for i in range(9):
        ax = plt.subplot(3, 3, i + 1)
        plt.imshow(images[i].numpy().astype("uint8"))

        predicted_class, confidence = predict(model, images[i].numpy())
        actual_class = class_names[labels[i]]

        plt.title(f"Actual: {actual_class},\n Predicted: {predicted_class}.\n Confidence: {confidence}%")

        plt.axis("off")
```


Actual: Potato___healthy,
Predicted: Potato___healthy.
Confidence: 95.59%
Actual: Potato___Late_blight,
Predicted: Potato___Early_blight.
Confidence: 83.0%
Actual: Potato___Late_blight,
Predicted: Potato___Late_blight.
Confidence: 91.55%
Actual: Potato___Late_blight,
Predicted: Potato___Late_blight.
Confidence: 98.09%
Actual: Potato___Early_blight,
Predicted: Potato___Early_blight.
Confidence: 97.4%
Actual: Potato___Late_blight,
Predicted: Potato___Late_blight.
Confidence: 98.71%
Actual: Potato___Early_blight,
Predicted: Potato___Early_blight.
Confidence: 100.0%
Actual: Potato___Early_blight,
Predicted: Potato___Early_blight.
Confidence: 96.35%
Actual: Potato___Late_blight,
Predicted: Potato___Late_blight.
Confidence: 96.61%