Potato Disease Classification

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

Import all the Dependencies

```
import tensorflow as tf
from tensorflow.keras import models, layers
import matplotlib.pyplot as plt
from IPython.display import HTML
```

Set all the Constants

```
BATCH_SIZE = 32
IMAGE_SIZE = 256
CHANNELS=3
EPOCHS=50
```

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

```
dataset = tf.keras.preprocessing.image_dataset_from_directory(
    "PlantVillage",
    seed=123,
    shuffle=True,
    image_size=(IMAGE_SIZE,IMAGE_SIZE),
    batch_size=BATCH_SIZE
)
```

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

Watch below video on tensorflow input pipeline first if you don't know about tensorflow datasets

```
HTML("""
<iframe width="560" height="315" src="https://www.youtube.com/embed/VFEOskzhhbc"
""")
```

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

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

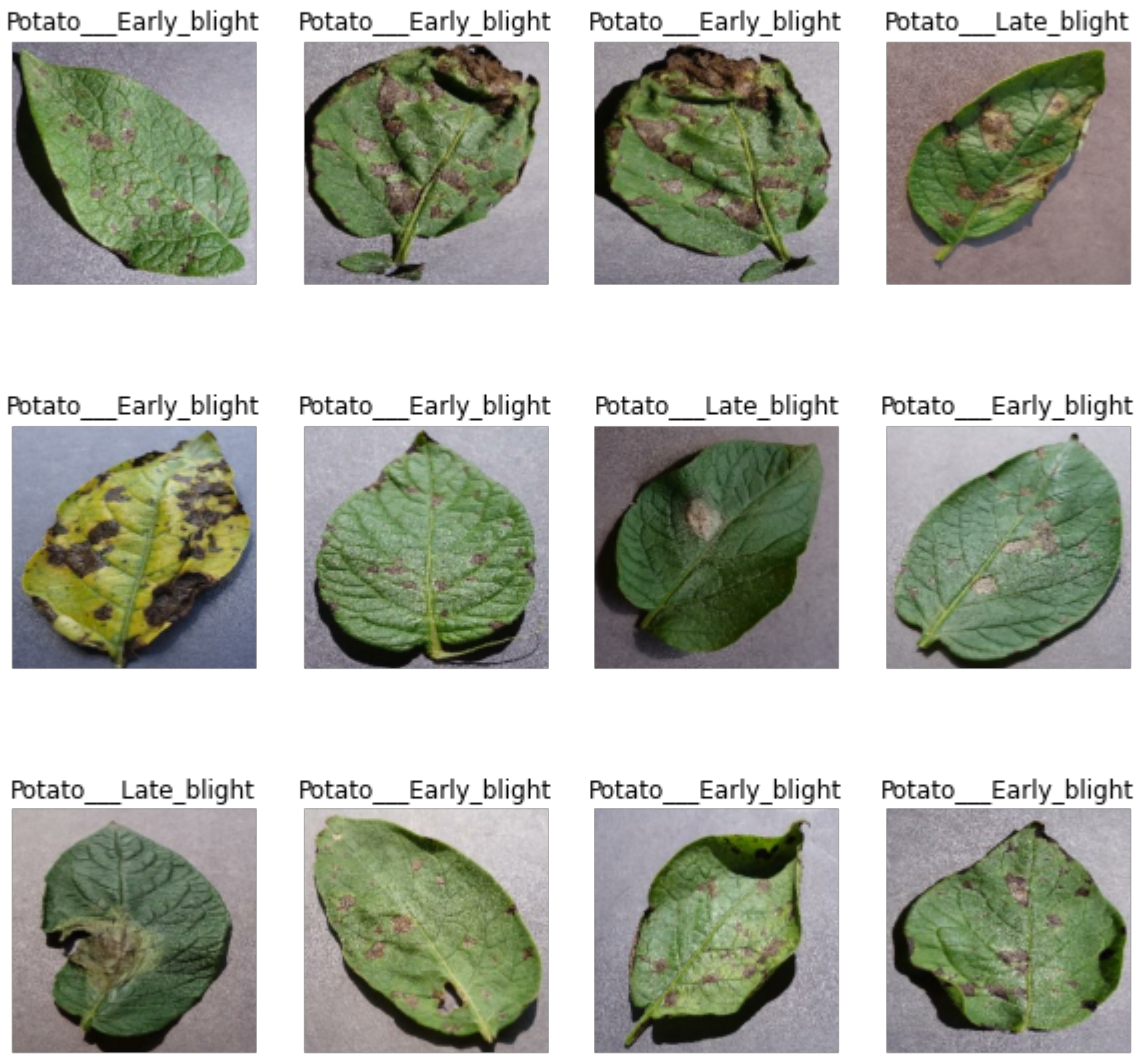
```
for image_batch, labels_batch in dataset.take(1):
    print(image_batch.shape)
    print(labels_batch.numpy())
```

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

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

```
len(dataset)
```

```
68
```

```
train_size = 0.8
len(dataset)*train_size
```

```
54.400000000000006
```

```
train_ds = dataset.take(54)
len(train_ds)
```

```
54
```

```
test_ds = dataset.skip(54)
len(test_ds)
```

```
14
```

```
val_size=0.1
len(dataset)*val_size
```

```
6.800000000000001
```

```
val_ds = test_ds.take(6)
len(val_ds)
```

```
6
```

```
test_ds = test_ds.skip(6)
len(test_ds)
```

```
8
```

```
def get_dataset_partitions_tf(ds, train_split=0.8, val_split=0.1, test_split=0.1,
    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)

    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
```

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

```
len(train_ds)
```

54

```
len(val_ds)
```

6

```
len(test_ds)
```

8

Cache, Shuffle, and Prefetch the Dataset

```
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.

You might be thinking why do we need to resize (256,256) image to again (256,256). You are right we don't need to but this will be useful when we are done with the training and start using the model for predictions. At that time somone can supply an image that is not (256,256) and this layer will resize it

```
resize_and_rescale = tf.keras.Sequential([
  layers.experimental.preprocessing.Resizing(IMAGE_SIZE, IMAGE_SIZE),
  layers.experimental.preprocessing.Rescaling(1./255),
])
```

Data Augmentation

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

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

Applying Data Augmentation to Train Dataset

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

Watch below video if you are not familiar with data augmentation

```
HTML("""
<iframe width="560" height="315" src="https://www.youtube.com/embed/mTVf7BN7S8w"
""")
```

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

```
HTML("""
<iframe width="560" height="315" src="https://www.youtube.com/embed/zfiSAzpy9NM"
""")
```

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

model.build(input_shape=input_shape)
```

```
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 (MaxPooling2 (32, 62, 62, 64)          0
_________________________________________________________________
conv2d_2 (Conv2D)            (32, 60, 60, 64)          36928
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (32, 30, 30, 64)          0
_________________________________________________________________
conv2d_3 (Conv2D)            (32, 28, 28, 64)          36928
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (32, 14, 14, 64)          0
_________________________________________________________________
conv2d_4 (Conv2D)            (32, 12, 12, 64)          36928
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (32, 6, 6, 64)            0
_________________________________________________________________
conv2d_5 (Conv2D)            (32, 4, 4, 64)            36928
```

```
_________________________________________________________________
max_pooling2d_5 (MaxPooling2 (32, 2, 2, 64)            0
_________________________________________________________________
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

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

```
history = model.fit(
    train_ds,
    batch_size=BATCH_SIZE,
    validation_data=val_ds,
    verbose=1,
    epochs=50,
)
```

```
Epoch 1/50
54/54 [==============================] - 20s 255ms/step - loss: 0.8802 - accurac
y: 0.5341 - val_loss: 0.8462 - val_accuracy: 0.5938
Epoch 2/50
54/54 [==============================] - 11s 196ms/step - loss: 0.6033 - accurac
y: 0.7396 - val_loss: 0.6225 - val_accuracy: 0.6979
Epoch 3/50
54/54 [==============================] - 9s 172ms/step - loss: 0.3647 - accuracy:
0.8403 - val_loss: 0.3065 - val_accuracy: 0.8802
Epoch 4/50
54/54 [==============================] - 10s 176ms/step - loss: 0.2776 - accurac
y: 0.8999 - val_loss: 0.2702 - val_accuracy: 0.8750
Epoch 5/50
54/54 [==============================] - 10s 179ms/step - loss: 0.2448 - accurac
y: 0.8953 - val_loss: 0.1857 - val_accuracy: 0.9062
Epoch 6/50
54/54 [==============================] - 9s 174ms/step - loss: 0.2020 - accuracy:
0.9144 - val_loss: 0.2987 - val_accuracy: 0.9115
Epoch 7/50
54/54 [==============================] - 10s 185ms/step - loss: 0.1751 - accurac
y: 0.9288 - val_loss: 0.1854 - val_accuracy: 0.9375
Epoch 8/50
54/54 [==============================] - 10s 180ms/step - loss: 0.1436 - accurac
y: 0.9444 - val_loss: 0.2273 - val_accuracy: 0.9167
Epoch 9/50
54/54 [==============================] - 10s 175ms/step - loss: 0.1128 - accurac
y: 0.9583 - val_loss: 0.1425 - val_accuracy: 0.9479
```

```
Epoch 10/50
54/54 [==============================] - 10s 179ms/step - loss: 0.1218 - accurac
y: 0.9549 - val_loss: 0.2310 - val_accuracy: 0.9115
Epoch 11/50
54/54 [==============================] - 10s 179ms/step - loss: 0.1524 - accurac
y: 0.9398 - val_loss: 0.0774 - val_accuracy: 0.9688
Epoch 12/50
54/54 [==============================] - 10s 186ms/step - loss: 0.1062 - accurac
y: 0.9578 - val_loss: 0.1787 - val_accuracy: 0.9427
Epoch 13/50
54/54 [==============================] - 9s 172ms/step - loss: 0.1299 - accuracy:
0.9549 - val_loss: 0.0929 - val_accuracy: 0.9531
Epoch 14/50
54/54 [==============================] - 9s 169ms/step - loss: 0.0971 - accuracy:
0.9601 - val_loss: 0.1230 - val_accuracy: 0.9531
Epoch 15/50
54/54 [==============================] - 9s 171ms/step - loss: 0.0967 - accuracy:
0.9659 - val_loss: 0.0804 - val_accuracy: 0.9635
Epoch 16/50
54/54 [==============================] - 9s 172ms/step - loss: 0.0764 - accuracy:
0.9676 - val_loss: 0.1225 - val_accuracy: 0.9531
Epoch 17/50
54/54 [==============================] - 9s 174ms/step - loss: 0.1157 - accuracy:
0.9543 - val_loss: 0.2200 - val_accuracy: 0.9219
Epoch 18/50
54/54 [==============================] - 10s 175ms/step - loss: 0.0947 - accurac
y: 0.9659 - val_loss: 0.1852 - val_accuracy: 0.9271
Epoch 19/50
54/54 [==============================] - 9s 174ms/step - loss: 0.0737 - accuracy:
0.9711 - val_loss: 0.0923 - val_accuracy: 0.9583
Epoch 20/50
54/54 [==============================] - 9s 173ms/step - loss: 0.0518 - accuracy:
0.9815 - val_loss: 0.0678 - val_accuracy: 0.9688
Epoch 21/50
54/54 [==============================] - 9s 172ms/step - loss: 0.0473 - accuracy:
0.9826 - val_loss: 0.0516 - val_accuracy: 0.9740
Epoch 22/50
54/54 [==============================] - 9s 173ms/step - loss: 0.0510 - accuracy:
0.9803 - val_loss: 0.3043 - val_accuracy: 0.8958
Epoch 23/50
54/54 [==============================] - 9s 175ms/step - loss: 0.0510 - accuracy:
0.9792 - val_loss: 0.2573 - val_accuracy: 0.9062
Epoch 24/50
54/54 [==============================] - 10s 176ms/step - loss: 0.0820 - accurac
y: 0.9670 - val_loss: 0.0828 - val_accuracy: 0.9635
Epoch 25/50
54/54 [==============================] - 9s 175ms/step - loss: 0.0459 - accuracy:
0.9844 - val_loss: 0.0912 - val_accuracy: 0.9740
Epoch 26/50
54/54 [==============================] - 10s 176ms/step - loss: 0.0361 - accurac
y: 0.9867 - val_loss: 0.0354 - val_accuracy: 0.9844
Epoch 27/50
54/54 [==============================] - 9s 170ms/step - loss: 0.0461 - accuracy:
0.9838 - val_loss: 0.0364 - val_accuracy: 0.9844
Epoch 28/50
54/54 [==============================] - 9s 170ms/step - loss: 0.0414 - accuracy:
0.9838 - val_loss: 0.1192 - val_accuracy: 0.9479
Epoch 29/50
54/54 [==============================] - 9s 168ms/step - loss: 0.0424 - accuracy:
0.9861 - val_loss: 0.0509 - val_accuracy: 0.9844
Epoch 30/50
54/54 [==============================] - 9s 167ms/step - loss: 0.0348 - accuracy:
0.9873 - val_loss: 0.1987 - val_accuracy: 0.9531
```

```
Epoch 31/50
54/54 [==============================] - 10s 178ms/step - loss: 0.0437 - accurac
y: 0.9821 - val_loss: 0.0371 - val_accuracy: 0.9948
Epoch 32/50
54/54 [==============================] - 10s 193ms/step - loss: 0.0439 - accurac
y: 0.9855 - val_loss: 0.1708 - val_accuracy: 0.9375
Epoch 33/50
54/54 [==============================] - 10s 187ms/step - loss: 0.0558 - accurac
y: 0.9774 - val_loss: 0.1559 - val_accuracy: 0.9531
Epoch 34/50
54/54 [==============================] - 9s 171ms/step - loss: 0.0412 - accuracy:
0.9821 - val_loss: 0.1024 - val_accuracy: 0.9583
Epoch 35/50
54/54 [==============================] - 9s 170ms/step - loss: 0.0312 - accuracy:
0.9902 - val_loss: 0.0919 - val_accuracy: 0.9583
Epoch 36/50
54/54 [==============================] - 10s 178ms/step - loss: 0.0431 - accurac
y: 0.9844 - val_loss: 0.0217 - val_accuracy: 0.9948
Epoch 37/50
54/54 [==============================] - 10s 178ms/step - loss: 0.0353 - accurac
y: 0.9896 - val_loss: 0.0092 - val_accuracy: 1.0000
Epoch 38/50
54/54 [==============================] - 9s 173ms/step - loss: 0.0206 - accuracy:
0.9936 - val_loss: 0.0079 - val_accuracy: 1.0000
Epoch 39/50
54/54 [==============================] - 9s 171ms/step - loss: 0.0307 - accuracy:
0.9913 - val_loss: 0.0209 - val_accuracy: 0.9896
Epoch 40/50
54/54 [==============================] - 9s 175ms/step - loss: 0.0143 - accuracy:
0.9948 - val_loss: 0.0240 - val_accuracy: 0.9896
Epoch 41/50
54/54 [==============================] - 9s 170ms/step - loss: 0.0196 - accuracy:
0.9936 - val_loss: 0.0441 - val_accuracy: 0.9844
Epoch 42/50
54/54 [==============================] - 9s 173ms/step - loss: 0.0382 - accuracy:
0.9832 - val_loss: 0.2912 - val_accuracy: 0.9271
Epoch 43/50
54/54 [==============================] - 9s 172ms/step - loss: 0.0416 - accuracy:
0.9832 - val_loss: 0.0425 - val_accuracy: 0.9896
Epoch 44/50
54/54 [==============================] - 9s 171ms/step - loss: 0.0162 - accuracy:
0.9948 - val_loss: 0.0567 - val_accuracy: 0.9792
Epoch 45/50
54/54 [==============================] - 9s 170ms/step - loss: 0.0990 - accuracy:
0.9653 - val_loss: 0.0892 - val_accuracy: 0.9688
Epoch 46/50
54/54 [==============================] - 9s 171ms/step - loss: 0.0243 - accuracy:
0.9919 - val_loss: 0.0174 - val_accuracy: 0.9948
Epoch 47/50
54/54 [==============================] - 9s 170ms/step - loss: 0.0476 - accuracy:
0.9844 - val_loss: 0.0217 - val_accuracy: 0.9896
Epoch 48/50
54/54 [==============================] - 9s 170ms/step - loss: 0.0184 - accuracy:
0.9931 - val_loss: 0.1227 - val_accuracy: 0.9635
Epoch 49/50
54/54 [==============================] - 10s 184ms/step - loss: 0.0298 - accurac
y: 0.9884 - val_loss: 0.0528 - val_accuracy: 0.9844
Epoch 50/50
54/54 [==============================] - 11s 196ms/step - loss: 0.0189 - accurac
y: 0.9948 - val_loss: 0.0064 - val_accuracy: 1.0000
```

```python
scores = model.evaluate(test_ds)
```

```
8/8 [==============================] - 1s 14ms/step - loss: 0.0063 - accuracy: 1.
0000
```

You can see above that we get 100.00% accuracy for our test dataset. This is considered to be a pretty good accuracy

```
scores
```

```
[0.006251859944313765, 1.0]
```

Scores is just a list containing loss and accuracy value

Plotting the Accuracy and Loss Curves

```
history
```

```
<tensorflow.python.keras.callbacks.History at 0x7f3d98437e50>
```

You can read documentation on history object here: https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/History

```
history.params
```

```
{'verbose': 1, 'epochs': 50, 'steps': 54}
```

```
history.history.keys()
```

```
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

```
type(history.history['loss'])
```

```
list
```

```
len(history.history['loss'])
```

```
50
```

```
history.history['loss'][:5] # show loss for first 5 epochs
```

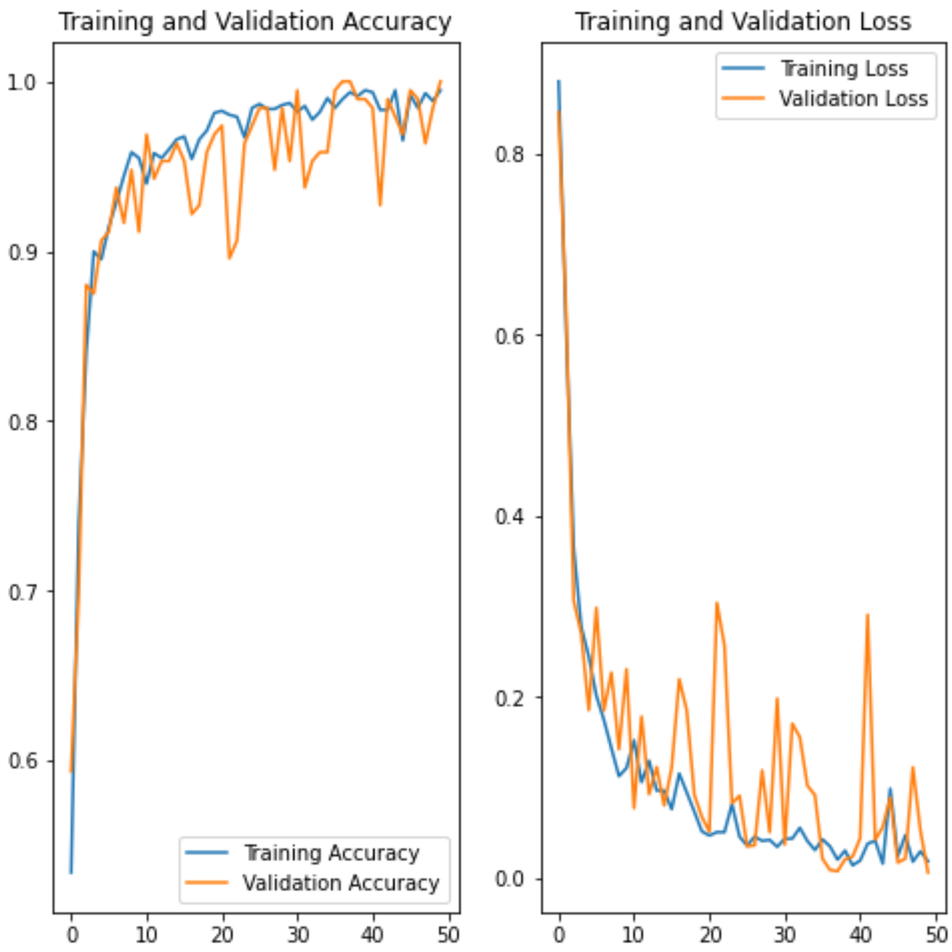
```
[0.8801848292350769,
 0.6033139228820801,
 0.3646925389766693,
 0.2776017189025879,
 0.24480397999286652]
```

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

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

```
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 Accuracy')
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()
```



Run prediction on a sample image

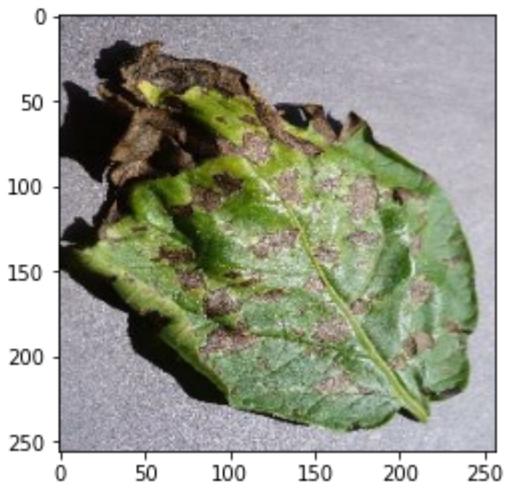
```
import numpy as np
for images_batch, labels_batch in test_ds.take(1):

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

    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___Early_blight
predicted label: Potato___Early_blight
```



Write a function for inference

```
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
```

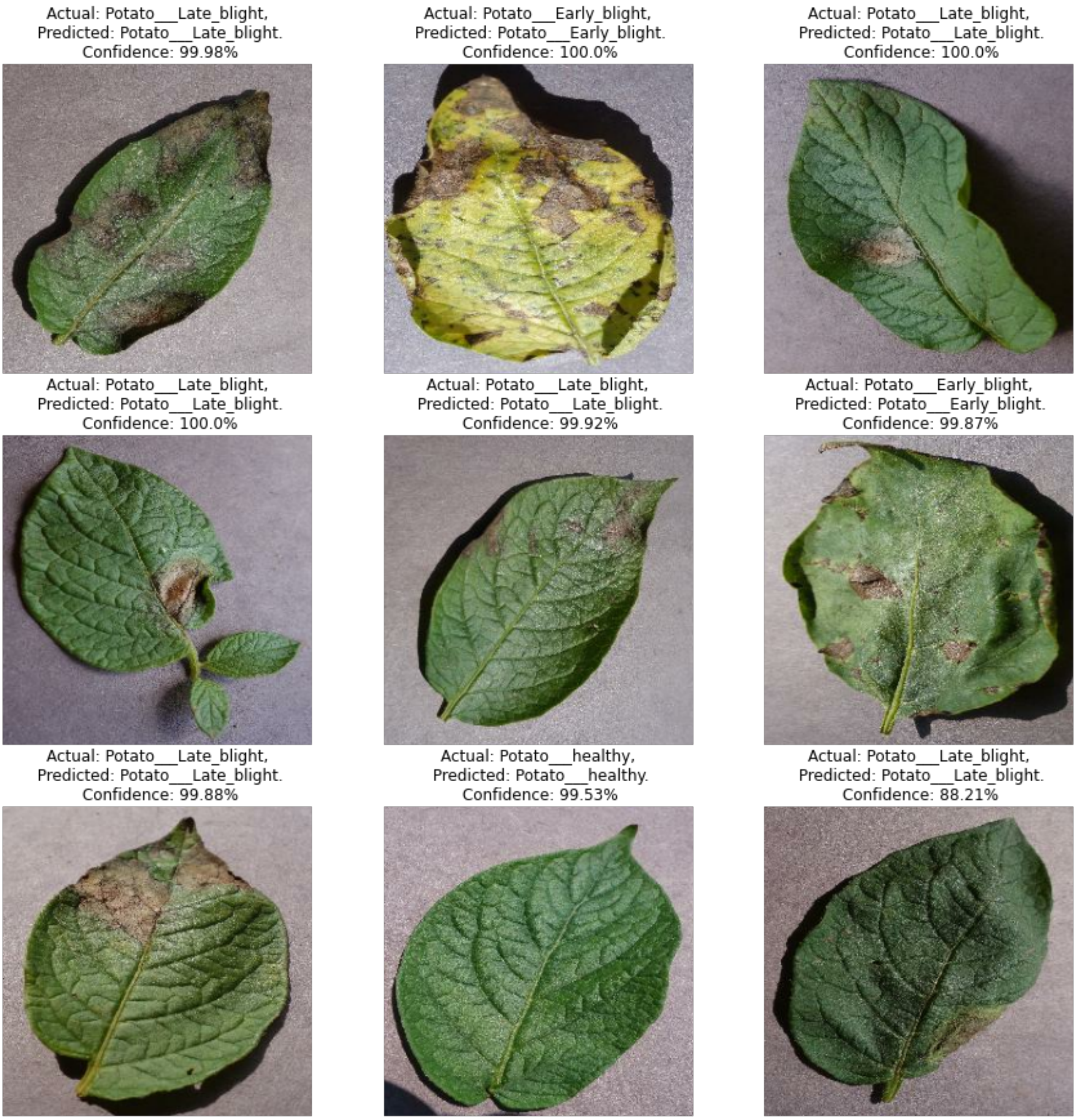
Now run inference on few sample images

```
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 Con

        plt.axis("off")
```



Saving the Model

We append the model to the list of models as a new version

```
import os
model_version=max([int(i) for i in os.listdir("../models") + [0]])+1
model.save(f"../models/{model_version}")
```

```
INFO:tensorflow:Assets written to: ../models/3/assets
```

```
model.save("../potatoes.h5")
```