Potato Disease Classification

Import all the Libraries

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

In [2]:
```
BATCH_SIZE = 32
IMAGE_SIZE = 256
CHANNELS=3
EPOCHS=30
```

Import data into tensorflow dataset object

In [3]:
```
dataset = tf.keras.preprocessing.image_dataset_from_directory(
    "mrdata",
    seed=1,
    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
```

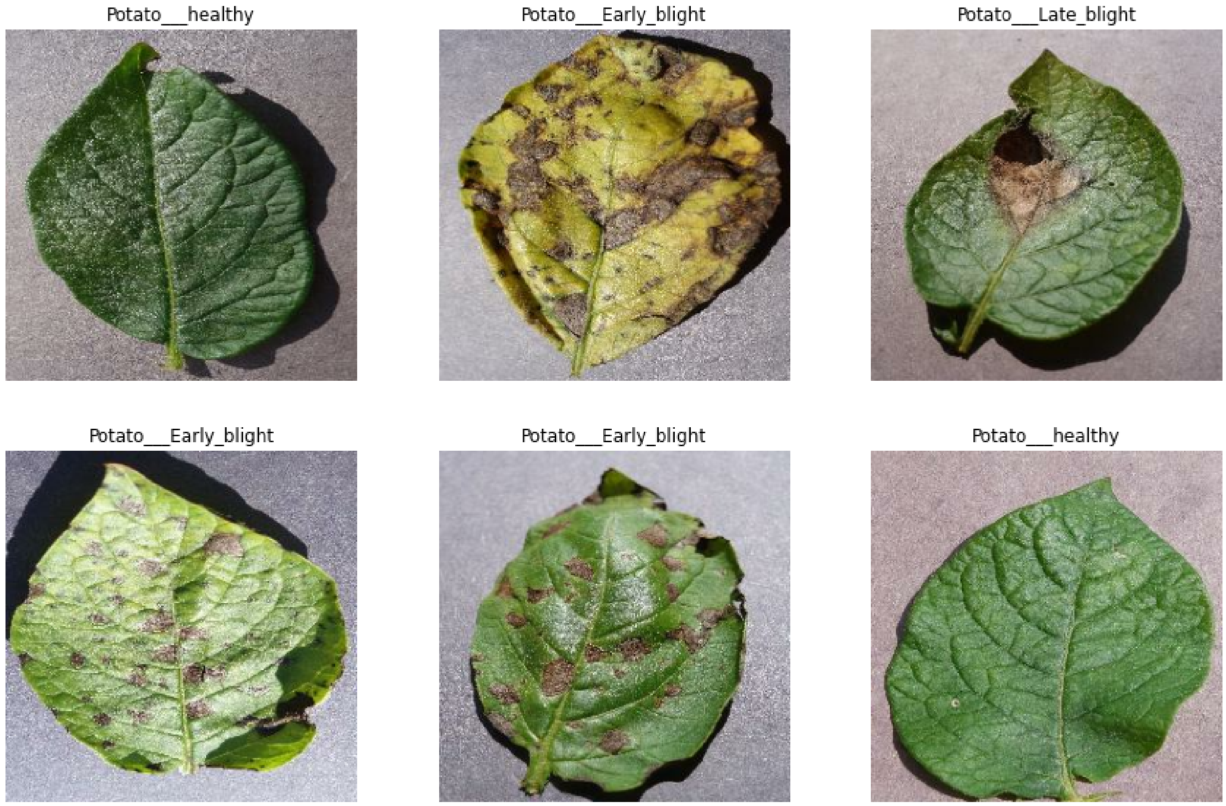
Out[4]:
```
['Potato___Early_blight', 'Potato___Late_blight', 'Potato___healthy']
```

In [5]:
```
for image_batch, labels_batch in dataset.take(1):
    print(image_batch.shape)
    print(labels_batch.numpy())
```

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

displaying some random pictures

In [6]:
```
plt.figure(figsize=(15, 15))
for image_batch, labels_batch in dataset.take(1):
    for i in range(6):
        ax = plt.subplot(3, 3, i + 1)
        plt.imshow(image_batch[i].numpy().astype("uint8"))
        plt.title(class_names[labels_batch[i]])
        plt.axis("off")
```



spiliting data_set

In [7]:
```
train_size = 0.8
len(dataset)*train_size
```

Out[7]:
```
54.400000000000006
```

In [8]:
```
train_dataset = dataset.take(54)
len(train_dataset)
```

Out[8]:
```
54
```

In [9]:
```
test_dataset = dataset.skip(54)
len(test_dataset)
```

Out[9]:
```
14
```

In [10]:
```
val_size=0.1
len(dataset)*val_size
```

Out[10]:
```
6.800000000000001
```

In [11]:
```
val_dataset = test_dataset.take(6)
len(val_dataset)
```

Out[11]:
```
6
```

In [12]:
```
test_dataset = test_dataset.skip(6)
len(test_dataset)
```

Out[12]: 8

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.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 [13]:

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

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.

In [14]:

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

model.build(input_shape=input_shape)
```

In [15]:

```
model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 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     
                                                                 
 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 [16]:
```
model.compile(
    optimizer='adam',
    loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=False),
    metrics=['accuracy']
)
```

In [19]:
```
history = model.fit(
    train_dataset,
```

```
    batch_size=BATCH_SIZE,
    validation_data=val_dataset,
    verbose=1,
    epochs=13,
)
```

```
Epoch 1/13
54/54 [==============================] - 144s 3s/step - loss: 0.1134 - accuracy: 0.95
54 - val_loss: 0.0705 - val_accuracy: 0.9740
Epoch 2/13
54/54 [==============================] - 146s 3s/step - loss: 0.0690 - accuracy: 0.97
69 - val_loss: 0.2391 - val_accuracy: 0.9219
Epoch 3/13
54/54 [==============================] - 145s 3s/step - loss: 0.0736 - accuracy: 0.97
11 - val_loss: 0.0680 - val_accuracy: 0.9844
Epoch 4/13
54/54 [==============================] - 145s 3s/step - loss: 0.0528 - accuracy: 0.98
32 - val_loss: 0.0582 - val_accuracy: 0.9844
Epoch 5/13
54/54 [==============================] - 144s 3s/step - loss: 0.1112 - accuracy: 0.95
72 - val_loss: 0.0707 - val_accuracy: 0.9844
Epoch 6/13
54/54 [==============================] - 145s 3s/step - loss: 0.0620 - accuracy: 0.97
63 - val_loss: 0.1083 - val_accuracy: 0.9583
Epoch 7/13
54/54 [==============================] - 148s 3s/step - loss: 0.0358 - accuracy: 0.99
07 - val_loss: 0.0324 - val_accuracy: 0.9844
Epoch 8/13
54/54 [==============================] - 147s 3s/step - loss: 0.0175 - accuracy: 0.99
48 - val_loss: 0.0252 - val_accuracy: 0.9844
Epoch 9/13
54/54 [==============================] - 148s 3s/step - loss: 0.0891 - accuracy: 0.96
59 - val_loss: 0.1009 - val_accuracy: 0.9583
Epoch 10/13
54/54 [==============================] - 143s 3s/step - loss: 0.0650 - accuracy: 0.97
22 - val_loss: 0.1069 - val_accuracy: 0.9583
Epoch 11/13
54/54 [==============================] - 146s 3s/step - loss: 0.0268 - accuracy: 0.98
84 - val_loss: 0.0288 - val_accuracy: 0.9844
Epoch 12/13
54/54 [==============================] - 146s 3s/step - loss: 0.0530 - accuracy: 0.98
03 - val_loss: 0.0598 - val_accuracy: 0.9635
Epoch 13/13
54/54 [==============================] - 147s 3s/step - loss: 0.0473 - accuracy: 0.98
21 - val_loss: 0.0189 - val_accuracy: 0.9948
```

In [20]:

```
scores = model.evaluate(test_dataset)
scores
```

```
8/8 [==============================] - 7s 625ms/step - loss: 0.0556 - accuracy: 0.982
8
```

Out[20]:

```
[0.05564812943339348, 0.982758641242981]
```

In [ ]:

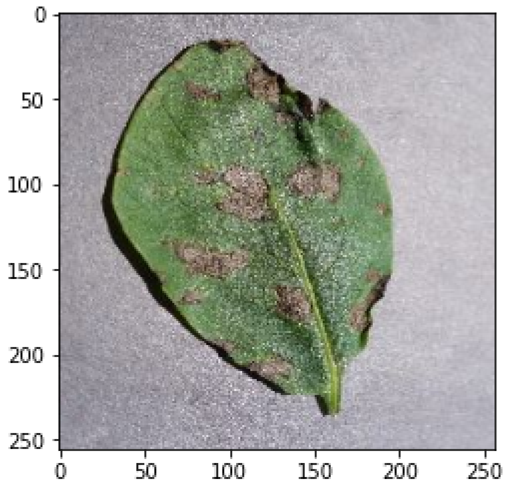
```
We can see above that we get 98.00% accuracy for our test dataset. This is considered
```

In [26]:

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

    first_image = images_batch[3].numpy().astype('uint8')
```

```
    first_label = labels_batch[3].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[3])])
```

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



In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]: