Import all the Dependencies

In [35]:
```
import tensorflow as tf
from tensorflow.keras import models, layers
import matplotlib.pyplot as plt
from IPython.display import HTML
```

Set all the Constants

In [2]:
```
BATCH_SIZE = 32
IMAGE_SIZE = 256
CHANNELS = 3
EPOCHS = 5
```

Import data into tensorflow dataset object

In [3]:
```
dataset = tf.keras.preprocessing.image_dataset_from_directory(
    "../Dataset/CancerDetection",
    seed = 123,
    shuffle = True,
    image_size = (IMAGE_SIZE,IMAGE_SIZE),
    batch_size = BATCH_SIZE
)
```

```
Found 3297 files belonging to 2 classes.
```

In [4]:
```
class_names = dataset.class_names
class_names
```

Out[4]:
```
['benign', 'malignant']
```

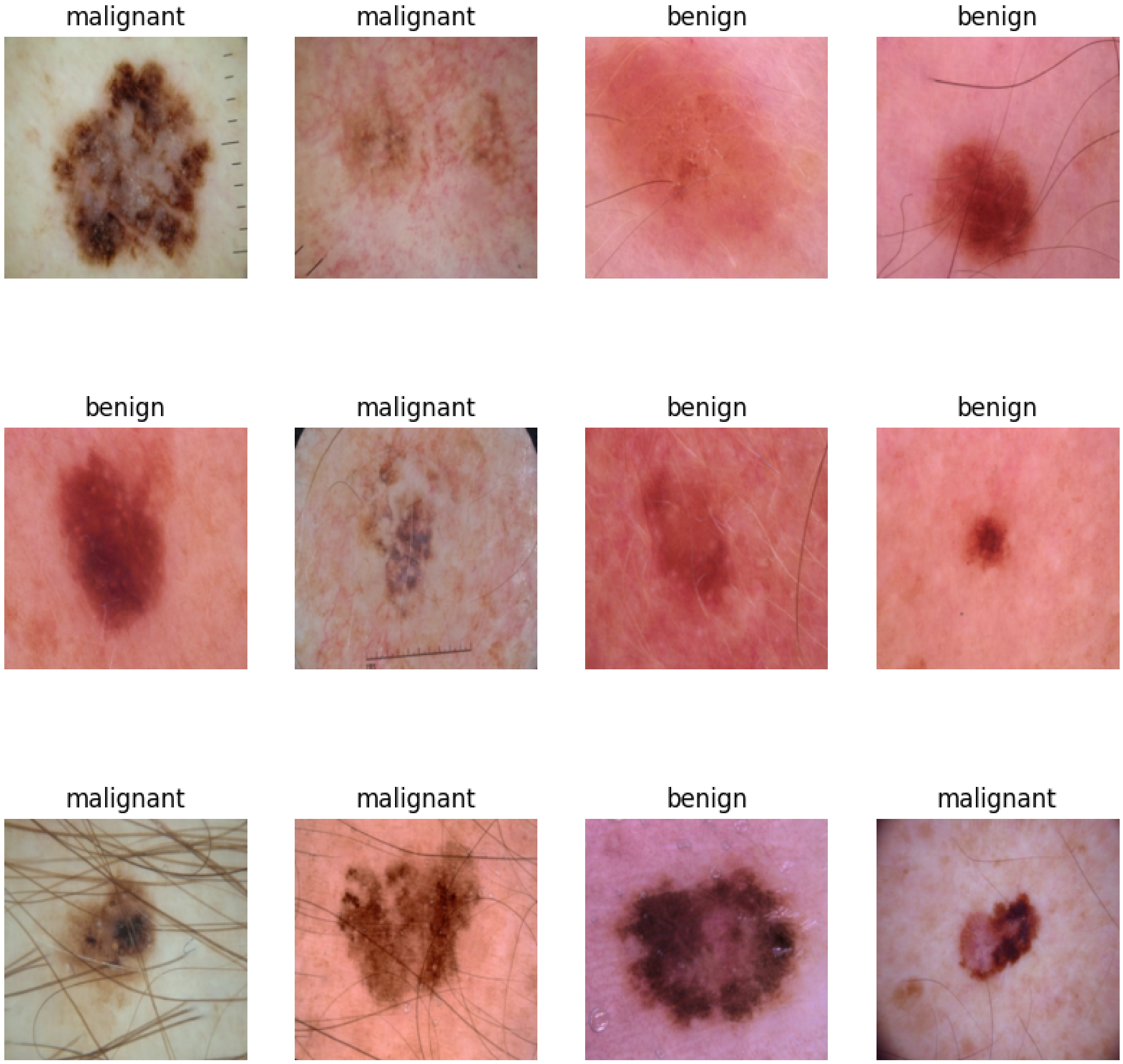
In [5]:
```
for image_batch, labels_batch in dataset.take(1):
    print(image_batch.shape)
    print(labels_batch.numpy())
```

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

Visualize some of the images from our dataset

In [6]:
```
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

In [7]:
```
len(dataset)
```

Out[7]: 104

In [8]:
```
train_size = 0.8
len(dataset)*train_size
```

Out[8]: 83.2

In [9]:
```
train_ds = dataset.take(54)
len(train_ds)
```

Out[9]: 54

In [10]:
```
test_ds = dataset.skip(54)
len(test_ds)
```

Out[10]: 50

In [11]:
```
val_size=0.1
len(dataset)*val_size
```

Out[11]: 10.4

In [12]:
```
val_ds = test_ds.take(6)
len(val_ds)
```

Out[12]: 6

In [13]:
```
test_ds = test_ds.skip(6)
len(test_ds)
```

Out[13]: 44

In [14]:
```
def get_dataset_partitions_tf(ds, train_split=0.8, val_split=0.1, test_split=0.1, s
    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
```

In [15]:
```
train_ds, val_ds, test_ds = get_dataset_partitions_tf(dataset)
```

In [16]:
```
len(train_ds)
```

Out[16]: 83

In [17]:
```
len(val_ds)
```
Out[17]:
```
10
```
In [18]:
```
len(test_ds)
```
Out[18]:
```
11
```

Cache, Shuffle, and Prefetch the Dataset

In [19]:
```
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

In [20]:
```
resize_and_rescale = tf.keras.Sequential([
  layers.experimental.preprocessing.Resizing(IMAGE_SIZE, IMAGE_SIZE),
  layers.experimental.preprocessing.Rescaling(1./255),
])
```
In [21]:
```
# Data Augmentation
# Data Augmentation is needed when we have less data, this boosts the accuracy of o

# 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)
```

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 [22]:
```
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_sha
    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 [23]:
```
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 [30]:
```
model.compile(
    optimizer='adam',
    loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=False),
    metrics=['accuracy']
)
```

In [31]:
```
history = model.fit(
    train_ds,
    batch_size = BATCH_SIZE,
    validation_data = val_ds,
    verbose = 1,
    epochs = 30,
)
```

```
Epoch 1/30
83/83 [==============================] - 90s 1s/step - loss: 0.3223 - accuracy: 0.
8587 - val_loss: 0.3234 - val_accuracy: 0.8438
Epoch 2/30
83/83 [==============================] - 96s 1s/step - loss: 0.2994 - accuracy: 0.
8587 - val_loss: 0.3101 - val_accuracy: 0.8781
Epoch 3/30
83/83 [==============================] - 96s 1s/step - loss: 0.2932 - accuracy: 0.
8678 - val_loss: 0.3326 - val_accuracy: 0.8531
Epoch 4/30
83/83 [==============================] - 96s 1s/step - loss: 0.2885 - accuracy: 0.
8690 - val_loss: 0.3524 - val_accuracy: 0.8500
Epoch 5/30
83/83 [==============================] - 92s 1s/step - loss: 0.2754 - accuracy: 0.
8819 - val_loss: 0.3092 - val_accuracy: 0.8750
Epoch 6/30
83/83 [==============================] - 89s 1s/step - loss: 0.2572 - accuracy: 0.
8846 - val_loss: 0.2639 - val_accuracy: 0.8938
Epoch 7/30
83/83 [==============================] - 89s 1s/step - loss: 0.2390 - accuracy: 0.
8941 - val_loss: 0.3316 - val_accuracy: 0.8719
Epoch 8/30
83/83 [==============================] - 89s 1s/step - loss: 0.2534 - accuracy: 0.
8884 - val_loss: 0.2632 - val_accuracy: 0.8844
Epoch 9/30
83/83 [==============================] - 89s 1s/step - loss: 0.2114 - accuracy: 0.
9082 - val_loss: 0.3196 - val_accuracy: 0.8781
Epoch 10/30
83/83 [==============================] - 89s 1s/step - loss: 0.2112 - accuracy: 0.
9147 - val_loss: 0.3019 - val_accuracy: 0.8781
Epoch 11/30
83/83 [==============================] - 88s 1s/step - loss: 0.1907 - accuracy: 0.
9211 - val_loss: 0.3064 - val_accuracy: 0.8906
Epoch 12/30
83/83 [==============================] - 89s 1s/step - loss: 0.1579 - accuracy: 0.
9375 - val_loss: 0.2775 - val_accuracy: 0.9062
Epoch 13/30
83/83 [==============================] - 87s 1s/step - loss: 0.1555 - accuracy: 0.
9345 - val_loss: 0.3725 - val_accuracy: 0.9031
Epoch 14/30
83/83 [==============================] - 90s 1s/step - loss: 0.1811 - accuracy: 0.
9246 - val_loss: 0.1959 - val_accuracy: 0.9250
Epoch 15/30
83/83 [==============================] - 87s 1s/step - loss: 0.1535 - accuracy: 0.
9390 - val_loss: 0.2035 - val_accuracy: 0.9187
Epoch 16/30
83/83 [==============================] - 88s 1s/step - loss: 0.1095 - accuracy: 0.
9566 - val_loss: 0.2149 - val_accuracy: 0.9469
Epoch 17/30
83/83 [==============================] - 88s 1s/step - loss: 0.1073 - accuracy: 0.
9554 - val_loss: 0.2990 - val_accuracy: 0.9156
Epoch 18/30
83/83 [==============================] - 88s 1s/step - loss: 0.1262 - accuracy: 0.
9501 - val_loss: 0.1919 - val_accuracy: 0.9469
Epoch 19/30
83/83 [==============================] - 88s 1s/step - loss: 0.1158 - accuracy: 0.
```

```
9558 - val_loss: 0.2957 - val_accuracy: 0.9219
Epoch 20/30
83/83 [==============================] - 88s 1s/step - loss: 0.0731 - accuracy: 0.
9691 - val_loss: 0.2134 - val_accuracy: 0.9438
Epoch 21/30
83/83 [==============================] - 192s 2s/step - loss: 0.0796 - accuracy:
0.9707 - val_loss: 0.2540 - val_accuracy: 0.9406
Epoch 22/30
83/83 [==============================] - 87s 1s/step - loss: 0.1301 - accuracy: 0.
9535 - val_loss: 0.2818 - val_accuracy: 0.9281
Epoch 23/30
83/83 [==============================] - 85s 1s/step - loss: 0.0884 - accuracy: 0.
9695 - val_loss: 0.2191 - val_accuracy: 0.9500
Epoch 24/30
83/83 [==============================] - 87s 1s/step - loss: 0.0564 - accuracy: 0.
9813 - val_loss: 0.2445 - val_accuracy: 0.9500
Epoch 25/30
83/83 [==============================] - 89s 1s/step - loss: 0.0583 - accuracy: 0.
9810 - val_loss: 0.2961 - val_accuracy: 0.9469
Epoch 26/30
83/83 [==============================] - 88s 1s/step - loss: 0.0449 - accuracy: 0.
9829 - val_loss: 0.2439 - val_accuracy: 0.9531
Epoch 27/30
83/83 [==============================] - 88s 1s/step - loss: 0.0152 - accuracy: 0.
9962 - val_loss: 0.2837 - val_accuracy: 0.9594
Epoch 28/30
83/83 [==============================] - 89s 1s/step - loss: 0.0162 - accuracy: 0.
9935 - val_loss: 0.3777 - val_accuracy: 0.9438
Epoch 29/30
83/83 [==============================] - 87s 1s/step - loss: 0.1858 - accuracy: 0.
9352 - val_loss: 0.2161 - val_accuracy: 0.9344
Epoch 30/30
83/83 [==============================] - 87s 1s/step - loss: 0.0412 - accuracy: 0.
9867 - val_loss: 0.2631 - val_accuracy: 0.9500
```

In [32]:

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

```
11/11 [==============================] - 3s 274ms/step - loss: 0.1538 - accuracy:
0.9631
```

In [27]:

```python
scores
```

Out[27]:

```
[0.31988516449928284, 0.8409090638160706]
```

Plotting History

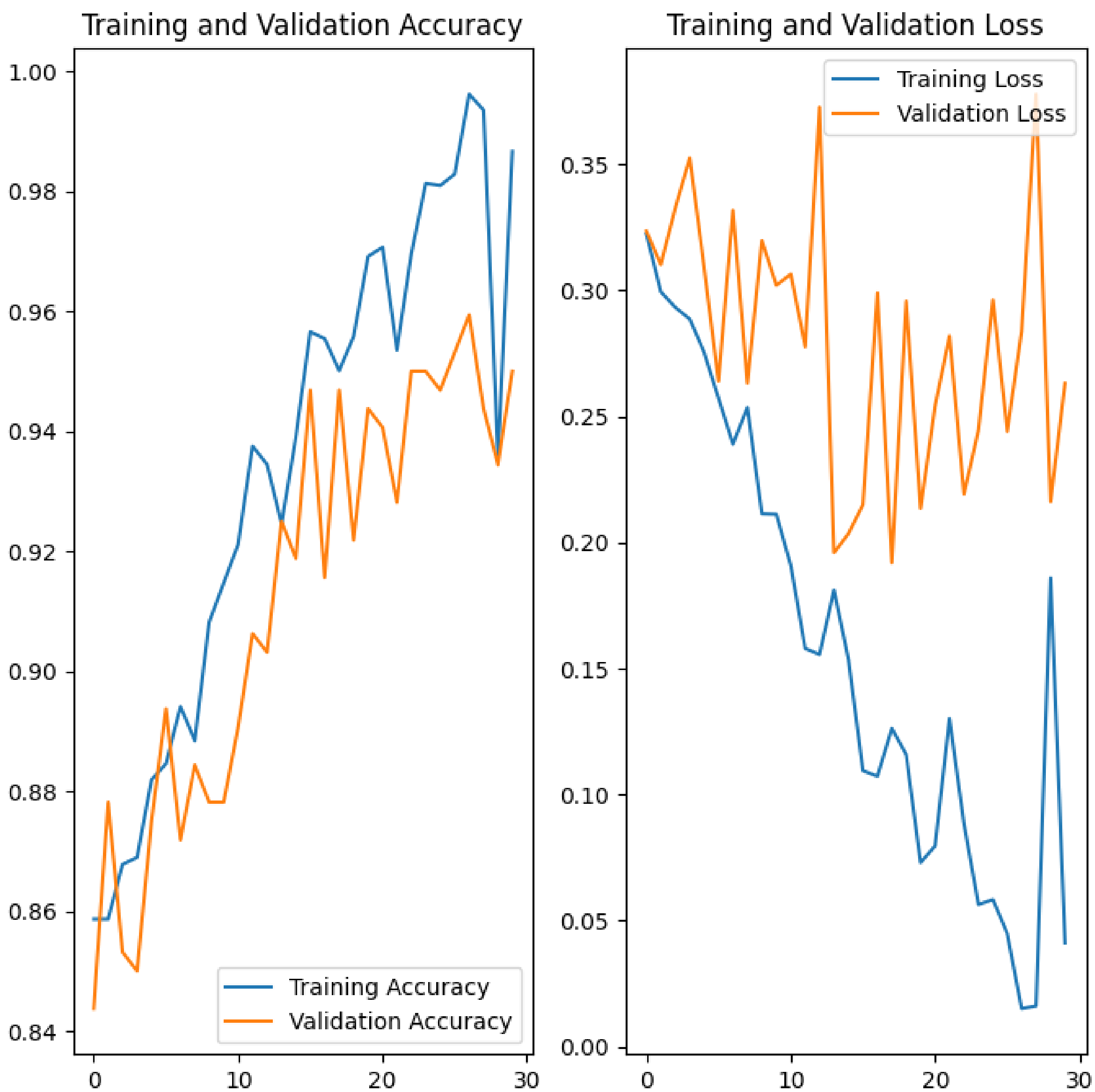
In [36]:

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

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

In [38]:

```python
plt.figure(figsize=(8, 8))
plt.subplot(1, 2, 1)
```

```
plt.plot(range(30), acc, label='Training Accuracy')
plt.plot(range(30), val_acc, label='Validation Accuracy')
plt.legend(loc='lower right')
plt.title('Training and Validation Accuracy')

plt.subplot(1, 2, 2)
plt.plot(range(30), loss, label='Training Loss')
plt.plot(range(30), val_loss, label='Validation Loss')
plt.legend(loc='upper right')
plt.title('Training and Validation Loss')
plt.show()
```



In [ ]: