Import all the Dependencies

In [62]:
```
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 [63]:
```
BATCH_SIZE = 32
IMAGE_SIZE = 224
CHANNELS = 3
EPOCHS = 50
```

Import data into tensorflow dataset object

In [64]:
```
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 [65]:
```
class_names = dataset.class_names
class_names
```

Out[65]:
```
['benign', 'malignant']
```

In [66]:
```
for image_batch, labels_batch in dataset.take(1):
    print(image_batch.shape)
    print(labels_batch.numpy())
```

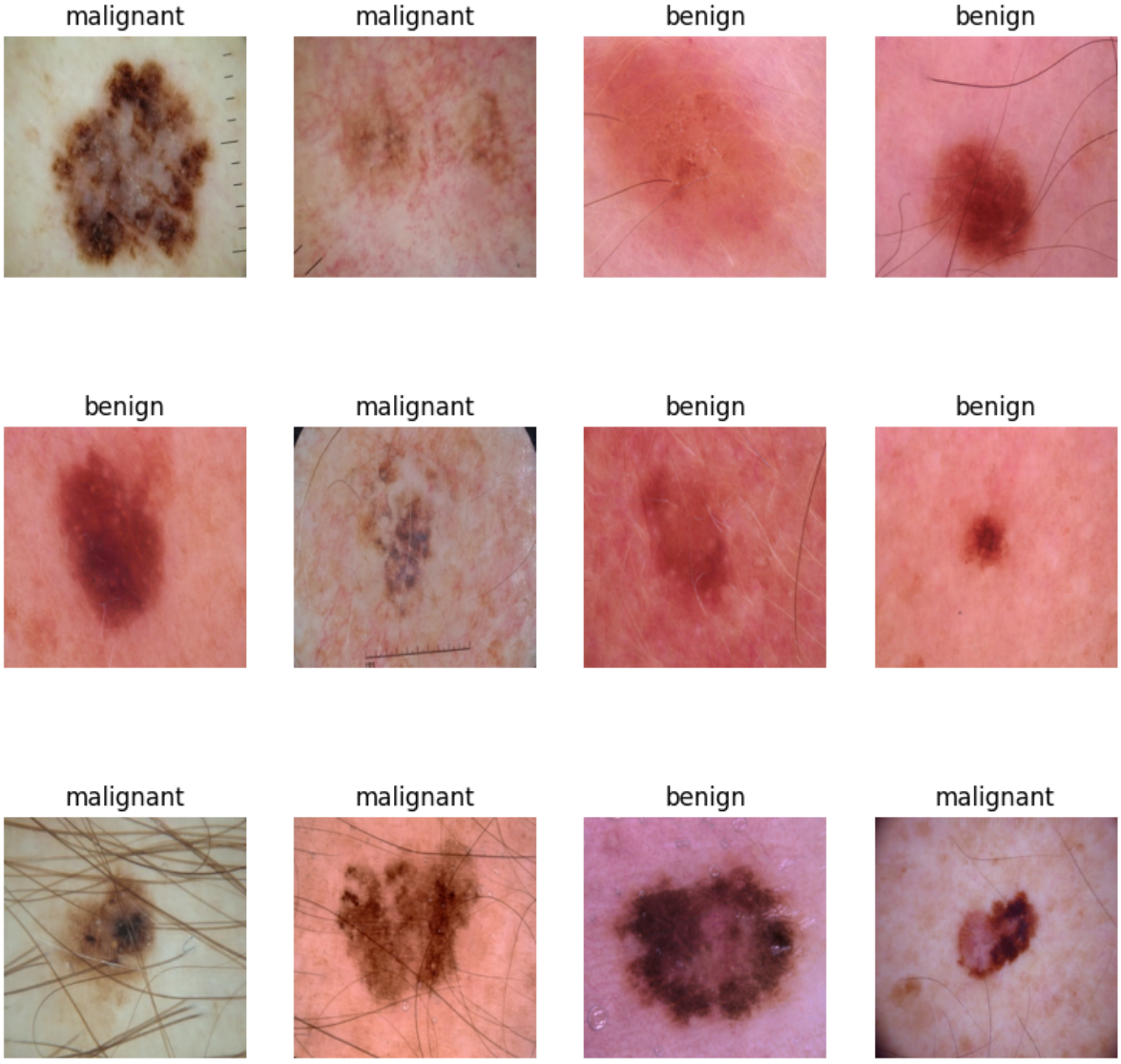
```
(32, 224, 224, 3)
[1 0 1 0 1 1 0 1 0 0 0 0 0 1 0 1 0 1 0 0 1 1 1 0 1 0 1 0 0 1 1 0]
```

Visualize some of the images from our dataset

In [67]:
```
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 [68]:
```
len(dataset)
```

Out[68]: 104

In [69]:
```
train_size = 0.8
len(dataset)*train_size
```

Out[69]: 83.2

In [70]:
```
train_ds = dataset.take(54)
len(train_ds)
```

Out[70]: 54

In [71]:
```
test_ds = dataset.skip(54)
len(test_ds)
```

Out[71]: 50

In [72]:
```
val_size=0.1
len(dataset)*val_size
```

Out[72]: 10.4

In [73]:
```
val_ds = test_ds.take(6)
len(val_ds)
```

Out[73]: 6

In [74]:
```
test_ds = test_ds.skip(6)
len(test_ds)
```

Out[74]: 44

In [75]:
```
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 [76]:
```
train_ds, val_ds, test_ds = get_dataset_partitions_tf(dataset)
```

In [77]:
```
len(train_ds)
```

Out[77]: 83

In [78]:
```
len(val_ds)
```

Out[78]: 10

In [79]:
```
len(test_ds)
```

Out[79]: 11

Cache, Shuffle, and Prefetch the Dataset

In [80]:
```
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 [81]:
```
resize_and_rescale = tf.keras.Sequential([
  layers.experimental.preprocessing.Resizing(IMAGE_SIZE, IMAGE_SIZE),
  layers.experimental.preprocessing.Rescaling(1./255),
])
```

In [82]:
```
# 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 [83]:
```
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 [84]:
```
model.summary()
```

```
Model: "sequential_5"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 sequential_4 (Sequential)   (32, 224, 224, 3)         0

 conv2d_12 (Conv2D)          (32, 222, 222, 32)        896

 max_pooling2d_12 (MaxPoolin  (32, 111, 111, 32)       0
 g2D)

 conv2d_13 (Conv2D)          (32, 109, 109, 64)        18496

 max_pooling2d_13 (MaxPoolin  (32, 54, 54, 64)         0
 g2D)

 conv2d_14 (Conv2D)          (32, 52, 52, 64)          36928

 max_pooling2d_14 (MaxPoolin  (32, 26, 26, 64)         0
 g2D)

 conv2d_15 (Conv2D)          (32, 24, 24, 64)          36928

 max_pooling2d_15 (MaxPoolin  (32, 12, 12, 64)         0
 g2D)

 conv2d_16 (Conv2D)          (32, 10, 10, 64)          36928

 max_pooling2d_16 (MaxPoolin  (32, 5, 5, 64)           0
 g2D)

 conv2d_17 (Conv2D)          (32, 3, 3, 64)            36928

 max_pooling2d_17 (MaxPoolin  (32, 1, 1, 64)           0
 g2D)

 flatten_2 (Flatten)         (32, 64)                  0

 dense_4 (Dense)             (32, 64)                  4160

 dense_5 (Dense)             (32, 3)                   195

=================================================================
Total params: 171,459
Trainable params: 171,459
Non-trainable params: 0
_________________________________________________________________
```

Compiling the Model

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

In [85]:
```
model.compile(
    optimizer='adam',
    loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=False),
    metrics=['accuracy']
)
```

In [86]:
```
history = model.fit(
    train_ds,
    batch_size = BATCH_SIZE,
    validation_data = val_ds,
    verbose = 1,
    epochs = EPOCHS,
)
```

```
Epoch 1/50
83/83 [==============================] - 74s 867ms/step - loss: 0.7160 - accuracy:
0.5295 - val_loss: 0.6717 - val_accuracy: 0.5813
Epoch 2/50
83/83 [==============================] - 65s 788ms/step - loss: 0.6192 - accuracy:
0.6568 - val_loss: 0.5113 - val_accuracy: 0.7563
Epoch 3/50
83/83 [==============================] - 68s 816ms/step - loss: 0.5315 - accuracy:
0.7512 - val_loss: 0.4893 - val_accuracy: 0.7594
Epoch 4/50
83/83 [==============================] - 64s 775ms/step - loss: 0.4663 - accuracy:
0.7699 - val_loss: 0.4754 - val_accuracy: 0.7750
Epoch 5/50
83/83 [==============================] - 64s 773ms/step - loss: 0.4559 - accuracy:
0.7745 - val_loss: 0.4408 - val_accuracy: 0.7844
Epoch 6/50
83/83 [==============================] - 64s 772ms/step - loss: 0.4447 - accuracy:
0.7787 - val_loss: 0.4707 - val_accuracy: 0.7531
Epoch 7/50
83/83 [==============================] - 68s 821ms/step - loss: 0.4166 - accuracy:
0.7886 - val_loss: 0.4199 - val_accuracy: 0.7812
Epoch 8/50
83/83 [==============================] - 65s 779ms/step - loss: 0.4222 - accuracy:
0.7890 - val_loss: 0.4570 - val_accuracy: 0.7688
Epoch 9/50
83/83 [==============================] - 65s 777ms/step - loss: 0.4363 - accuracy:
0.7771 - val_loss: 0.4099 - val_accuracy: 0.8094
Epoch 10/50
83/83 [==============================] - 64s 772ms/step - loss: 0.3905 - accuracy:
0.8118 - val_loss: 0.4639 - val_accuracy: 0.7781
Epoch 11/50
83/83 [==============================] - 64s 772ms/step - loss: 0.4004 - accuracy:
0.8015 - val_loss: 0.3884 - val_accuracy: 0.8094
Epoch 12/50
83/83 [==============================] - 65s 778ms/step - loss: 0.3781 - accuracy:
0.8259 - val_loss: 0.4139 - val_accuracy: 0.8031
Epoch 13/50
83/83 [==============================] - 64s 774ms/step - loss: 0.3595 - accuracy:
0.8339 - val_loss: 0.4750 - val_accuracy: 0.7688
Epoch 14/50
83/83 [==============================] - 64s 772ms/step - loss: 0.3557 - accuracy:
0.8339 - val_loss: 0.4263 - val_accuracy: 0.8062
Epoch 15/50
83/83 [==============================] - 64s 772ms/step - loss: 0.3500 - accuracy:
0.8301 - val_loss: 0.3483 - val_accuracy: 0.8500
Epoch 16/50
83/83 [==============================] - 64s 771ms/step - loss: 0.3249 - accuracy:
0.8488 - val_loss: 0.3978 - val_accuracy: 0.7906
Epoch 17/50
83/83 [==============================] - 64s 768ms/step - loss: 0.3493 - accuracy:
0.8320 - val_loss: 0.3710 - val_accuracy: 0.8406
Epoch 18/50
83/83 [==============================] - 64s 771ms/step - loss: 0.3284 - accuracy:
0.8484 - val_loss: 0.3562 - val_accuracy: 0.8188
Epoch 19/50
83/83 [==============================] - 980s 12s/step - loss: 0.3232 - accuracy:
```

```
0.8411 - val_loss: 0.3327 - val_accuracy: 0.8562
Epoch 20/50
83/83 [==============================] - 63s 763ms/step - loss: 0.2992 - accuracy:
0.8568 - val_loss: 0.3916 - val_accuracy: 0.8313
Epoch 21/50
83/83 [==============================] - 63s 763ms/step - loss: 0.2906 - accuracy:
0.8697 - val_loss: 0.3851 - val_accuracy: 0.8188
Epoch 22/50
83/83 [==============================] - 64s 766ms/step - loss: 0.2787 - accuracy:
0.8754 - val_loss: 0.3473 - val_accuracy: 0.8500
Epoch 23/50
83/83 [==============================] - 64s 767ms/step - loss: 0.2734 - accuracy:
0.8777 - val_loss: 0.2979 - val_accuracy: 0.8781
Epoch 24/50
83/83 [==============================] - 63s 765ms/step - loss: 0.2892 - accuracy:
0.8697 - val_loss: 0.4404 - val_accuracy: 0.7875
Epoch 25/50
83/83 [==============================] - 63s 763ms/step - loss: 0.2631 - accuracy:
0.8842 - val_loss: 0.2964 - val_accuracy: 0.8719
Epoch 26/50
83/83 [==============================] - 64s 766ms/step - loss: 0.2766 - accuracy:
0.8815 - val_loss: 0.3328 - val_accuracy: 0.8875
Epoch 27/50
83/83 [==============================] - 63s 763ms/step - loss: 0.2203 - accuracy:
0.9044 - val_loss: 0.2987 - val_accuracy: 0.8719
Epoch 28/50
83/83 [==============================] - 63s 764ms/step - loss: 0.2193 - accuracy:
0.9017 - val_loss: 0.3112 - val_accuracy: 0.8469
Epoch 29/50
83/83 [==============================] - 620s 8s/step - loss: 0.2093 - accuracy:
0.9093 - val_loss: 0.3342 - val_accuracy: 0.8687
Epoch 30/50
83/83 [==============================] - 63s 764ms/step - loss: 0.1703 - accuracy:
0.9291 - val_loss: 0.2999 - val_accuracy: 0.8813
Epoch 31/50
83/83 [==============================] - 63s 762ms/step - loss: 0.2079 - accuracy:
0.9147 - val_loss: 0.2536 - val_accuracy: 0.9031
Epoch 32/50
83/83 [==============================] - 65s 780ms/step - loss: 0.1628 - accuracy:
0.9333 - val_loss: 0.2725 - val_accuracy: 0.9000
Epoch 33/50
83/83 [==============================] - 74s 895ms/step - loss: 0.1525 - accuracy:
0.9436 - val_loss: 0.2483 - val_accuracy: 0.9187
Epoch 34/50
83/83 [==============================] - 70s 849ms/step - loss: 0.1400 - accuracy:
0.9436 - val_loss: 0.2916 - val_accuracy: 0.9031
Epoch 35/50
83/83 [==============================] - 69s 829ms/step - loss: 0.1301 - accuracy:
0.9501 - val_loss: 0.2916 - val_accuracy: 0.9031
Epoch 36/50
83/83 [==============================] - 66s 799ms/step - loss: 0.1571 - accuracy:
0.9360 - val_loss: 0.5141 - val_accuracy: 0.8594
Epoch 37/50
83/83 [==============================] - 65s 779ms/step - loss: 0.1094 - accuracy:
0.9554 - val_loss: 0.3518 - val_accuracy: 0.9094
Epoch 38/50
```

```
83/83 [==============================] - 65s 787ms/step - loss: 0.1231 - accuracy:
0.9490 - val_loss: 0.2415 - val_accuracy: 0.9219
Epoch 39/50
83/83 [==============================] - 66s 796ms/step - loss: 0.1322 - accuracy:
0.9493 - val_loss: 0.2804 - val_accuracy: 0.9094
Epoch 40/50
83/83 [==============================] - 68s 817ms/step - loss: 0.0840 - accuracy:
0.9680 - val_loss: 0.3192 - val_accuracy: 0.9250
Epoch 41/50
83/83 [==============================] - 68s 820ms/step - loss: 0.0799 - accuracy:
0.9672 - val_loss: 0.3489 - val_accuracy: 0.8906
Epoch 42/50
83/83 [==============================] - 70s 844ms/step - loss: 0.0604 - accuracy:
0.9775 - val_loss: 0.4007 - val_accuracy: 0.9344
Epoch 43/50
83/83 [==============================] - 69s 830ms/step - loss: 0.0410 - accuracy:
0.9870 - val_loss: 0.3645 - val_accuracy: 0.9500
Epoch 44/50
83/83 [==============================] - 65s 781ms/step - loss: 0.0934 - accuracy:
0.9661 - val_loss: 0.2939 - val_accuracy: 0.9312
Epoch 45/50
83/83 [==============================] - 65s 787ms/step - loss: 0.0407 - accuracy:
0.9874 - val_loss: 0.3951 - val_accuracy: 0.9438
Epoch 46/50
83/83 [==============================] - 65s 783ms/step - loss: 0.0887 - accuracy:
0.9714 - val_loss: 0.4440 - val_accuracy: 0.8750
Epoch 47/50
83/83 [==============================] - 65s 786ms/step - loss: 0.1385 - accuracy:
0.9467 - val_loss: 0.3571 - val_accuracy: 0.9281
Epoch 48/50
83/83 [==============================] - 65s 789ms/step - loss: 0.0350 - accuracy:
0.9893 - val_loss: 0.3179 - val_accuracy: 0.9469
Epoch 49/50
83/83 [==============================] - 65s 783ms/step - loss: 0.0249 - accuracy:
0.9916 - val_loss: 0.3432 - val_accuracy: 0.9594
Epoch 50/50
83/83 [==============================] - 65s 781ms/step - loss: 0.0889 - accuracy:
0.9695 - val_loss: 0.3531 - val_accuracy: 0.9281
```

In [87]:
```
scores = model.evaluate(test_ds)
```

```
11/11 [==============================] - 3s 145ms/step - loss: 0.1877 - accuracy:
0.9205
```

In [88]:
```
scores
```

Out[88]:
```
[0.18770451843738556, 0.9204545617103577]
```

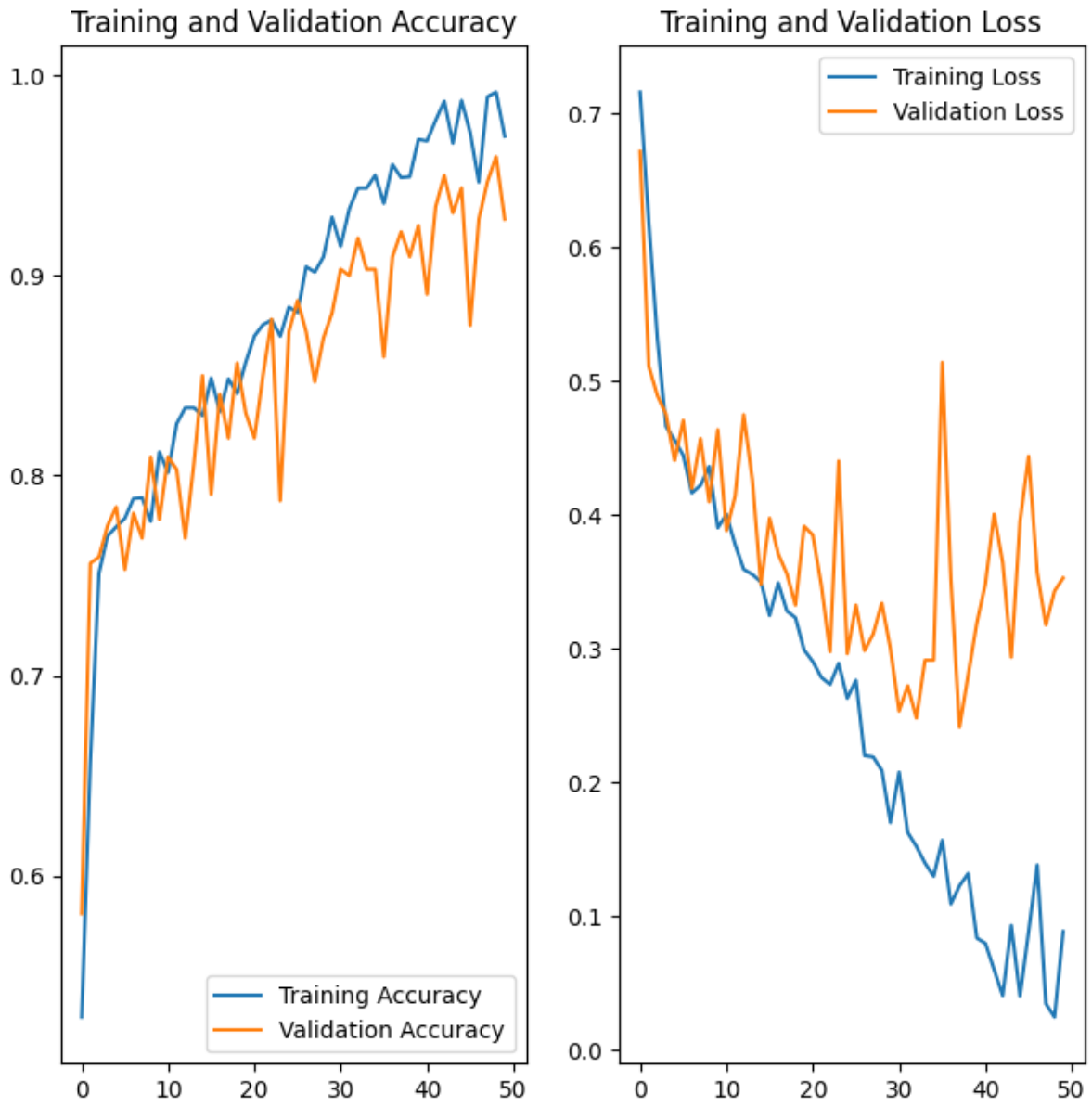
Plotting History

In [89]:
```
acc = history.history['accuracy']
val_acc = history.history['val_accuracy']
```

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

In [ ]:
```
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()
```



In [ ]:
```

```