Import all the Dependencies

In [2]:
```
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 [3]:
```
BATCH_SIZE = 32
IMAGE_SIZE = 224
CHANNELS = 3
EPOCHS = 50
```

Import data into tensorflow dataset object

In [4]:
```
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 [5]:
```
class_names = dataset.class_names
class_names
```

Out[5]:
```
['benign', 'malignant']
```

In [6]:
```
for image_batch, labels_batch in dataset.take(1):
    print(image_batch.shape)
    print(labels_batch.numpy())
```

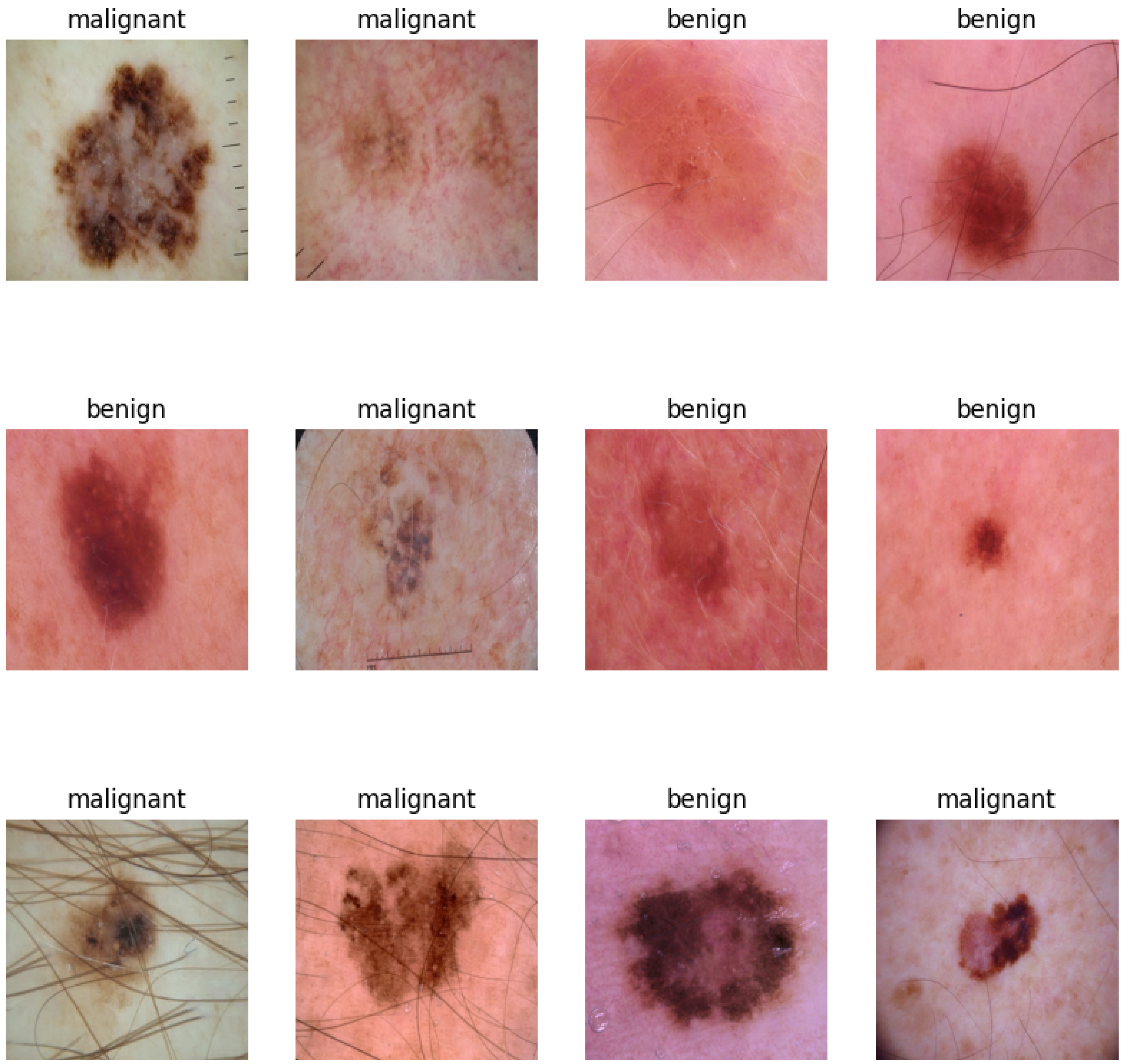
```
(32, 224, 224, 3)
[1 0 1 0 1 1 0 1 0 0 0 0 0 1 0 1 0 1 0 0 1 1 1 0 1 0 1 0 0 1 1 0]
```

Visualize some of the images from our dataset

In [7]:
```
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 [8]:
```
len(dataset)
```

Out[8]: 104

In [9]:
```
train_size = 0.8
len(dataset)*train_size
```

Out[9]: 83.2

In [10]:
```
train_ds = dataset.take(54)
len(train_ds)
```

Out[10]: 54

In [11]:
```
test_ds = dataset.skip(54)
len(test_ds)
```

Out[11]: 50

In [12]:
```
val_size=0.1
len(dataset)*val_size
```

Out[12]: 10.4

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

Out[13]: 6

In [14]:
```
test_ds = test_ds.skip(6)
len(test_ds)
```

Out[14]: 44

In [15]:
```
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 [16]:
```
train_ds, val_ds, test_ds = get_dataset_partitions_tf(dataset)
```

In [17]:
```
len(train_ds)
```

Out[17]: 83

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

In [19]:
```
len(test_ds)
```
Out[19]: 11

Cache, Shuffle, and Prefetch the Dataset

In [20]:
```
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 [21]:
```
resize_and_rescale = tf.keras.Sequential([
  layers.experimental.preprocessing.Resizing(IMAGE_SIZE, IMAGE_SIZE),
  layers.experimental.preprocessing.Rescaling(1./255),
])
```

In [22]:
```
# 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 [23]:
```
input_shape = (BATCH_SIZE, IMAGE_SIZE, IMAGE_SIZE, CHANNELS)
n_classes = 2

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 [24]:
```
model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 sequential (Sequential)     (32, 224, 224, 3)         0         
                                                                 
 conv2d (Conv2D)             (32, 222, 222, 32)        896       
                                                                 
 max_pooling2d (MaxPooling2D  (32, 111, 111, 32)       0         
 )                                                               
                                                                 
 conv2d_1 (Conv2D)           (32, 109, 109, 64)        18496     
                                                                 
 max_pooling2d_1 (MaxPooling  (32, 54, 54, 64)         0         
 2D)                                                             
                                                                 
 conv2d_2 (Conv2D)           (32, 52, 52, 64)          36928     
                                                                 
 max_pooling2d_2 (MaxPooling  (32, 26, 26, 64)         0         
 2D)                                                             
                                                                 
 conv2d_3 (Conv2D)           (32, 24, 24, 64)          36928     
                                                                 
 max_pooling2d_3 (MaxPooling  (32, 12, 12, 64)         0         
 2D)                                                             
                                                                 
 conv2d_4 (Conv2D)           (32, 10, 10, 64)          36928     
                                                                 
 max_pooling2d_4 (MaxPooling  (32, 5, 5, 64)           0         
 2D)                                                             
                                                                 
 conv2d_5 (Conv2D)           (32, 3, 3, 64)            36928     
                                                                 
 max_pooling2d_5 (MaxPooling  (32, 1, 1, 64)           0         
 2D)                                                             
                                                                 
 flatten (Flatten)           (32, 64)                  0         
                                                                 
 dense (Dense)               (32, 64)                  4160      
                                                                 
 dense_1 (Dense)             (32, 2)                   130       
                                                                 
=================================================================
Total params: 171,394
Trainable params: 171,394
Non-trainable params: 0
_________________________________________________________________
```

Compiling the Model

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

In [25]:
```
model.compile(
    optimizer='adam',
    loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=False),
    metrics=['accuracy']
)
```

In [26]:
```
history = model.fit(
    train_ds,
    batch_size = BATCH_SIZE,
    validation_data = val_ds,
    verbose = 1,
    epochs = EPOCHS,
)
```

```
Epoch 1/50
83/83 [==============================] - 72s 822ms/step - loss: 0.6194 - accuracy:
0.6171 - val_loss: 0.4817 - val_accuracy: 0.7594
Epoch 2/50
83/83 [==============================] - 67s 808ms/step - loss: 0.5245 - accuracy:
0.7516 - val_loss: 0.4742 - val_accuracy: 0.7812
Epoch 3/50
83/83 [==============================] - 71s 856ms/step - loss: 0.4729 - accuracy:
0.7650 - val_loss: 0.4939 - val_accuracy: 0.7437
Epoch 4/50
83/83 [==============================] - 70s 848ms/step - loss: 0.4530 - accuracy:
0.7726 - val_loss: 0.4584 - val_accuracy: 0.7656
Epoch 5/50
83/83 [==============================] - 69s 830ms/step - loss: 0.4630 - accuracy:
0.7634 - val_loss: 0.4764 - val_accuracy: 0.7656
Epoch 6/50
83/83 [==============================] - 68s 816ms/step - loss: 0.4415 - accuracy:
0.7859 - val_loss: 0.5565 - val_accuracy: 0.7563
Epoch 7/50
83/83 [==============================] - 68s 817ms/step - loss: 0.4127 - accuracy:
0.7931 - val_loss: 0.4492 - val_accuracy: 0.7688
Epoch 8/50
83/83 [==============================] - 195s 2s/step - loss: 0.4047 - accuracy:
0.7996 - val_loss: 0.4970 - val_accuracy: 0.7531
Epoch 9/50
83/83 [==============================] - 66s 797ms/step - loss: 0.4041 - accuracy:
0.8004 - val_loss: 0.4108 - val_accuracy: 0.8000
Epoch 10/50
83/83 [==============================] - 66s 799ms/step - loss: 0.3880 - accuracy:
0.8057 - val_loss: 0.3754 - val_accuracy: 0.8062
Epoch 11/50
83/83 [==============================] - 66s 797ms/step - loss: 0.3645 - accuracy:
0.8248 - val_loss: 0.4034 - val_accuracy: 0.8094
Epoch 12/50
83/83 [==============================] - 67s 813ms/step - loss: 0.3507 - accuracy:
0.8312 - val_loss: 0.3859 - val_accuracy: 0.8281
Epoch 13/50
83/83 [==============================] - 68s 824ms/step - loss: 0.3512 - accuracy:
0.8324 - val_loss: 0.4961 - val_accuracy: 0.7531
Epoch 14/50
83/83 [==============================] - 70s 845ms/step - loss: 0.3369 - accuracy:
0.8419 - val_loss: 0.4042 - val_accuracy: 0.8000
Epoch 15/50
83/83 [==============================] - 68s 819ms/step - loss: 0.3259 - accuracy:
0.8430 - val_loss: 0.3093 - val_accuracy: 0.8562
Epoch 16/50
83/83 [==============================] - 69s 828ms/step - loss: 0.3557 - accuracy:
0.8331 - val_loss: 0.3680 - val_accuracy: 0.8219
Epoch 17/50
83/83 [==============================] - 111s 1s/step - loss: 0.3150 - accuracy:
0.8453 - val_loss: 0.3507 - val_accuracy: 0.8219
Epoch 18/50
83/83 [==============================] - 68s 817ms/step - loss: 0.3107 - accuracy:
0.8610 - val_loss: 0.3108 - val_accuracy: 0.8625
Epoch 19/50
83/83 [==============================] - 68s 819ms/step - loss: 0.2812 - accuracy:
```

```
0.8682 - val_loss: 0.3533 - val_accuracy: 0.8500
Epoch 20/50
83/83 [==============================] - 92s 1s/step - loss: 0.2842 - accuracy: 0.
8678 - val_loss: 0.3068 - val_accuracy: 0.8719
Epoch 21/50
83/83 [==============================] - 68s 817ms/step - loss: 0.2560 - accuracy:
0.8811 - val_loss: 0.2780 - val_accuracy: 0.8687
Epoch 22/50
83/83 [==============================] - 124s 2s/step - loss: 0.2615 - accuracy:
0.8838 - val_loss: 0.2898 - val_accuracy: 0.8781
Epoch 23/50
83/83 [==============================] - 66s 799ms/step - loss: 0.2324 - accuracy:
0.8945 - val_loss: 0.2494 - val_accuracy: 0.9094
Epoch 24/50
83/83 [==============================] - 66s 795ms/step - loss: 0.2251 - accuracy:
0.9051 - val_loss: 0.2640 - val_accuracy: 0.8969
Epoch 25/50
83/83 [==============================] - 66s 795ms/step - loss: 0.2195 - accuracy:
0.9097 - val_loss: 0.3024 - val_accuracy: 0.8875
Epoch 26/50
83/83 [==============================] - 67s 802ms/step - loss: 0.1888 - accuracy:
0.9219 - val_loss: 0.2154 - val_accuracy: 0.9250
Epoch 27/50
83/83 [==============================] - 67s 812ms/step - loss: 0.2161 - accuracy:
0.9105 - val_loss: 0.2832 - val_accuracy: 0.8875
Epoch 28/50
83/83 [==============================] - 68s 821ms/step - loss: 0.1923 - accuracy:
0.9181 - val_loss: 0.1924 - val_accuracy: 0.9250
Epoch 29/50
83/83 [==============================] - 68s 823ms/step - loss: 0.1375 - accuracy:
0.9490 - val_loss: 0.2038 - val_accuracy: 0.9250
Epoch 30/50
83/83 [==============================] - 68s 817ms/step - loss: 0.1914 - accuracy:
0.9250 - val_loss: 0.2189 - val_accuracy: 0.9187
Epoch 31/50
83/83 [==============================] - 68s 821ms/step - loss: 0.1469 - accuracy:
0.9368 - val_loss: 0.2540 - val_accuracy: 0.9219
Epoch 32/50
83/83 [==============================] - 69s 830ms/step - loss: 0.1182 - accuracy:
0.9547 - val_loss: 0.1943 - val_accuracy: 0.9312
Epoch 33/50
83/83 [==============================] - 70s 846ms/step - loss: 0.1043 - accuracy:
0.9600 - val_loss: 0.2745 - val_accuracy: 0.9281
Epoch 34/50
83/83 [==============================] - 70s 841ms/step - loss: 0.1127 - accuracy:
0.9543 - val_loss: 0.2321 - val_accuracy: 0.9187
Epoch 35/50
83/83 [==============================] - 70s 848ms/step - loss: 0.0710 - accuracy:
0.9718 - val_loss: 0.2954 - val_accuracy: 0.9375
Epoch 36/50
83/83 [==============================] - 92s 1s/step - loss: 0.0834 - accuracy: 0.
9672 - val_loss: 0.2719 - val_accuracy: 0.9375
Epoch 37/50
83/83 [==============================] - 66s 794ms/step - loss: 0.0677 - accuracy:
0.9760 - val_loss: 0.3173 - val_accuracy: 0.9469
Epoch 38/50
```

```
83/83 [==============================] - 66s 794ms/step - loss: 0.0503 - accuracy:
0.9802 - val_loss: 0.2826 - val_accuracy: 0.9312
Epoch 39/50
83/83 [==============================] - 66s 799ms/step - loss: 0.0938 - accuracy:
0.9646 - val_loss: 0.1945 - val_accuracy: 0.9469
Epoch 40/50
83/83 [==============================] - 67s 805ms/step - loss: 0.1038 - accuracy:
0.9573 - val_loss: 0.2324 - val_accuracy: 0.9344
Epoch 41/50
83/83 [==============================] - 68s 819ms/step - loss: 0.1376 - accuracy:
0.9501 - val_loss: 0.1662 - val_accuracy: 0.9531
Epoch 42/50
83/83 [==============================] - 68s 824ms/step - loss: 0.0399 - accuracy:
0.9848 - val_loss: 0.1627 - val_accuracy: 0.9500
Epoch 43/50
83/83 [==============================] - 83s 1s/step - loss: 0.0181 - accuracy: 0.
9935 - val_loss: 0.2045 - val_accuracy: 0.9594
Epoch 44/50
83/83 [==============================] - 66s 796ms/step - loss: 0.0425 - accuracy:
0.9855 - val_loss: 0.3577 - val_accuracy: 0.9469
Epoch 45/50
83/83 [==============================] - 67s 811ms/step - loss: 0.0969 - accuracy:
0.9665 - val_loss: 0.2752 - val_accuracy: 0.9406
Epoch 46/50
83/83 [==============================] - 71s 854ms/step - loss: 0.0335 - accuracy:
0.9897 - val_loss: 0.2701 - val_accuracy: 0.9594
Epoch 47/50
83/83 [==============================] - 70s 849ms/step - loss: 0.0657 - accuracy:
0.9752 - val_loss: 0.2473 - val_accuracy: 0.9375
Epoch 48/50
83/83 [==============================] - 70s 841ms/step - loss: 0.0422 - accuracy:
0.9848 - val_loss: 0.2405 - val_accuracy: 0.9500
Epoch 49/50
83/83 [==============================] - 75s 902ms/step - loss: 0.0119 - accuracy:
0.9985 - val_loss: 0.2365 - val_accuracy: 0.9563
Epoch 50/50
83/83 [==============================] - 68s 816ms/step - loss: 0.0511 - accuracy:
0.9832 - val_loss: 0.1869 - val_accuracy: 0.9219
```

In [27]:
```python
scores = model.evaluate(test_ds)
```

```
11/11 [==============================] - 3s 159ms/step - loss: 0.1375 - accuracy:
0.9545
```

In [28]:
```python
scores
```

Out[28]:
```
[0.137520432472229, 0.9545454382896423]
```

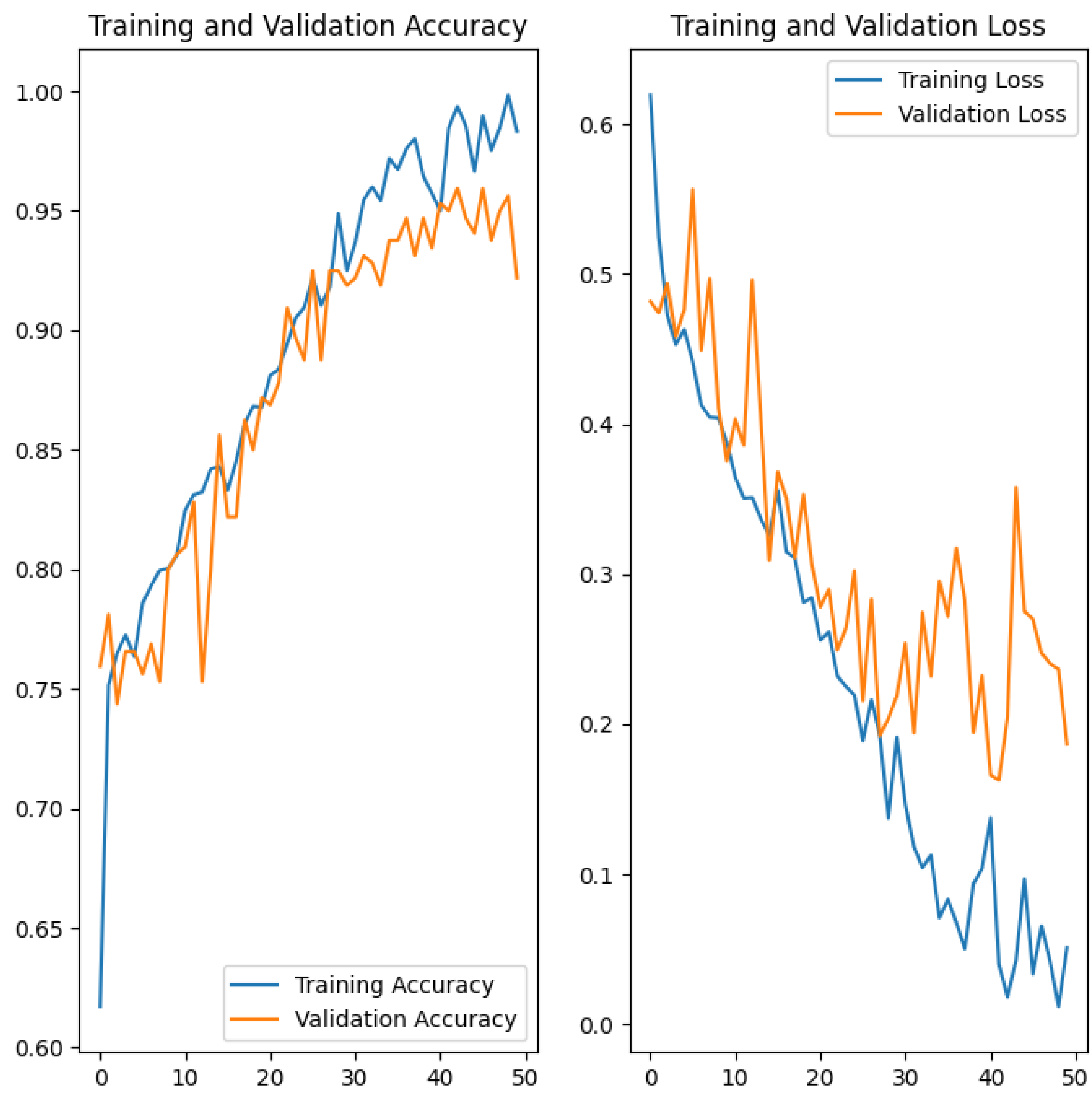
Plotting History

In [29]:
```python
acc = history.history['accuracy']
val_acc = history.history['val_accuracy']
```

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

In [30]:
```
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 [31]:
```
# model.predict
```

In [32]:
```
model.save('final.h5')
```

In [ ]: