Import all the Dependencies

In [1]:
```
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 = 224
CHANNELS = 3
EPOCHS = 50
```

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())
```

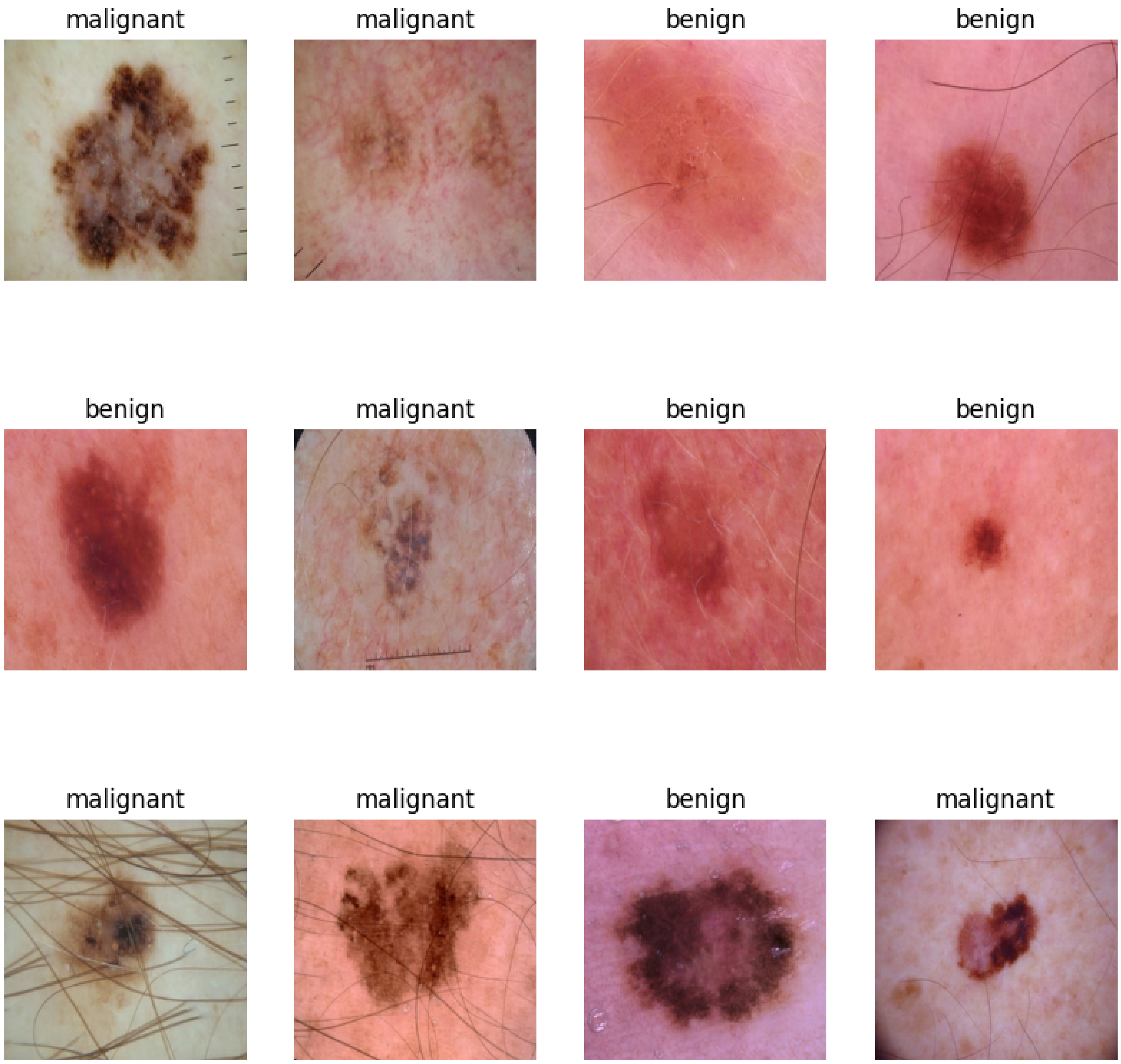
```
(32, 224, 224, 3)
[1 0 1 0 1 1 0 1 0 0 0 0 0 1 0 1 0 1 0 0 1 1 1 0 1 0 1 0 0 1 1 0]
```

Visualize some of the images from our dataset

In [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

In [19]:
```
actual_label_test = []

for image_batch, labels_batch in test_ds:
    temp = labels_batch.numpy()
    for j in temp:
        actual_label_test.append(j)

# print(len(actual_label_test))
# print(actual_label_test)
```

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]:
```
import time
t0 = time.time()
```

In [26]:
```
model.compile(
    optimizer='adam',
    loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=False),
    metrics=['accuracy']
)
```

In [27]:
```
history = model.fit(
    train_ds,
    batch_size = BATCH_SIZE,
    validation_data = val_ds,
    verbose = 1,
    epochs = EPOCHS,
)
```

```
Epoch 1/50
83/83 [==============================] - 70s 814ms/step - loss: 0.6229 - accuracy:
0.6408 - val_loss: 0.5331 - val_accuracy: 0.7563
Epoch 2/50
83/83 [==============================] - 65s 787ms/step - loss: 0.5029 - accuracy:
0.7470 - val_loss: 0.4719 - val_accuracy: 0.7906
Epoch 3/50
83/83 [==============================] - 67s 810ms/step - loss: 0.5033 - accuracy:
0.7553 - val_loss: 0.4614 - val_accuracy: 0.7812
Epoch 4/50
83/83 [==============================] - 67s 803ms/step - loss: 0.4406 - accuracy:
0.7756 - val_loss: 0.4557 - val_accuracy: 0.7937
Epoch 5/50
83/83 [==============================] - 67s 803ms/step - loss: 0.4227 - accuracy:
0.8001 - val_loss: 0.4529 - val_accuracy: 0.7844
Epoch 6/50
83/83 [==============================] - 67s 802ms/step - loss: 0.4185 - accuracy:
0.7922 - val_loss: 0.4567 - val_accuracy: 0.7781
Epoch 7/50
83/83 [==============================] - 67s 805ms/step - loss: 0.3846 - accuracy:
0.8159 - val_loss: 0.3859 - val_accuracy: 0.8094
Epoch 8/50
83/83 [==============================] - 67s 806ms/step - loss: 0.3852 - accuracy:
0.8117 - val_loss: 0.3742 - val_accuracy: 0.8188
Epoch 9/50
83/83 [==============================] - 67s 807ms/step - loss: 0.3661 - accuracy:
0.8242 - val_loss: 0.3501 - val_accuracy: 0.8375
Epoch 10/50
83/83 [==============================] - 68s 815ms/step - loss: 0.3576 - accuracy:
0.8279 - val_loss: 0.3509 - val_accuracy: 0.8406
Epoch 11/50
83/83 [==============================] - 72s 870ms/step - loss: 0.3457 - accuracy:
0.8279 - val_loss: 0.3545 - val_accuracy: 0.8344
Epoch 12/50
83/83 [==============================] - 69s 830ms/step - loss: 0.3427 - accuracy:
0.8355 - val_loss: 0.3676 - val_accuracy: 0.8188
Epoch 13/50
83/83 [==============================] - 68s 815ms/step - loss: 0.3383 - accuracy:
0.8434 - val_loss: 0.3191 - val_accuracy: 0.8562
Epoch 14/50
83/83 [==============================] - 68s 817ms/step - loss: 0.3390 - accuracy:
0.8373 - val_loss: 0.4017 - val_accuracy: 0.8094
Epoch 15/50
83/83 [==============================] - 66s 794ms/step - loss: 0.3199 - accuracy:
0.8539 - val_loss: 0.3142 - val_accuracy: 0.8844
Epoch 16/50
83/83 [==============================] - 67s 810ms/step - loss: 0.2937 - accuracy:
0.8607 - val_loss: 0.2836 - val_accuracy: 0.8750
Epoch 17/50
83/83 [==============================] - 78s 944ms/step - loss: 0.2937 - accuracy:
0.8720 - val_loss: 0.3220 - val_accuracy: 0.8750
Epoch 18/50
83/83 [==============================] - 66s 799ms/step - loss: 0.2773 - accuracy:
0.8746 - val_loss: 0.2505 - val_accuracy: 0.8813
Epoch 19/50
83/83 [==============================] - 67s 803ms/step - loss: 0.2722 - accuracy:
```

```
0.8697 - val_loss: 0.2423 - val_accuracy: 0.8938
Epoch 20/50
83/83 [==============================] - 67s 807ms/step - loss: 0.2517 - accuracy:
0.8855 - val_loss: 0.3039 - val_accuracy: 0.8656
Epoch 21/50
83/83 [==============================] - 67s 807ms/step - loss: 0.2420 - accuracy:
0.8934 - val_loss: 0.2330 - val_accuracy: 0.9031
Epoch 22/50
83/83 [==============================] - 66s 799ms/step - loss: 0.2355 - accuracy:
0.8965 - val_loss: 0.2238 - val_accuracy: 0.9156
Epoch 23/50
83/83 [==============================] - 66s 801ms/step - loss: 0.2188 - accuracy:
0.9014 - val_loss: 0.2489 - val_accuracy: 0.9031
Epoch 24/50
83/83 [==============================] - 66s 794ms/step - loss: 0.1960 - accuracy:
0.9119 - val_loss: 0.2141 - val_accuracy: 0.9281
Epoch 25/50
83/83 [==============================] - 66s 791ms/step - loss: 0.2294 - accuracy:
0.9040 - val_loss: 0.2148 - val_accuracy: 0.9250
Epoch 26/50
83/83 [==============================] - 66s 799ms/step - loss: 0.1671 - accuracy:
0.9266 - val_loss: 0.2653 - val_accuracy: 0.9156
Epoch 27/50
83/83 [==============================] - 66s 791ms/step - loss: 0.1719 - accuracy:
0.9330 - val_loss: 0.2072 - val_accuracy: 0.9281
Epoch 28/50
83/83 [==============================] - 66s 791ms/step - loss: 0.1391 - accuracy:
0.9431 - val_loss: 0.1866 - val_accuracy: 0.9250
Epoch 29/50
83/83 [==============================] - 66s 789ms/step - loss: 0.1738 - accuracy:
0.9285 - val_loss: 0.2418 - val_accuracy: 0.9250
Epoch 30/50
83/83 [==============================] - 67s 803ms/step - loss: 0.1362 - accuracy:
0.9458 - val_loss: 0.1647 - val_accuracy: 0.9594
Epoch 31/50
83/83 [==============================] - 66s 793ms/step - loss: 0.1145 - accuracy:
0.9537 - val_loss: 0.1998 - val_accuracy: 0.9344
Epoch 32/50
83/83 [==============================] - 66s 797ms/step - loss: 0.1113 - accuracy:
0.9544 - val_loss: 0.1934 - val_accuracy: 0.9531
Epoch 33/50
83/83 [==============================] - 65s 787ms/step - loss: 0.0726 - accuracy:
0.9721 - val_loss: 0.2280 - val_accuracy: 0.9625
Epoch 34/50
83/83 [==============================] - 66s 790ms/step - loss: 0.0659 - accuracy:
0.9721 - val_loss: 0.1954 - val_accuracy: 0.9563
Epoch 35/50
83/83 [==============================] - 66s 792ms/step - loss: 0.0859 - accuracy:
0.9665 - val_loss: 0.2145 - val_accuracy: 0.9469
Epoch 36/50
83/83 [==============================] - 66s 794ms/step - loss: 0.0812 - accuracy:
0.9736 - val_loss: 0.2344 - val_accuracy: 0.9375
Epoch 37/50
83/83 [==============================] - 66s 796ms/step - loss: 0.0826 - accuracy:
0.9695 - val_loss: 0.5614 - val_accuracy: 0.8625
Epoch 38/50
```

```
83/83 [==============================] - 65s 786ms/step - loss: 0.1109 - accuracy:
0.9612 - val_loss: 0.2305 - val_accuracy: 0.9563
Epoch 39/50
83/83 [==============================] - 67s 806ms/step - loss: 0.0465 - accuracy:
0.9838 - val_loss: 0.2568 - val_accuracy: 0.9563
Epoch 40/50
83/83 [==============================] - 69s 830ms/step - loss: 0.0901 - accuracy:
0.9631 - val_loss: 0.2245 - val_accuracy: 0.9625
Epoch 41/50
83/83 [==============================] - 66s 796ms/step - loss: 0.0300 - accuracy:
0.9910 - val_loss: 0.2671 - val_accuracy: 0.9688
Epoch 42/50
83/83 [==============================] - 65s 782ms/step - loss: 0.0118 - accuracy:
0.9970 - val_loss: 0.2785 - val_accuracy: 0.9500
Epoch 43/50
83/83 [==============================] - 65s 783ms/step - loss: 0.1412 - accuracy:
0.9499 - val_loss: 0.1459 - val_accuracy: 0.9719
Epoch 44/50
83/83 [==============================] - 67s 803ms/step - loss: 0.0371 - accuracy:
0.9868 - val_loss: 0.1493 - val_accuracy: 0.9688
Epoch 45/50
83/83 [==============================] - 66s 791ms/step - loss: 0.1044 - accuracy:
0.9646 - val_loss: 0.1707 - val_accuracy: 0.9594
Epoch 46/50
83/83 [==============================] - 66s 789ms/step - loss: 0.0242 - accuracy:
0.9932 - val_loss: 0.2261 - val_accuracy: 0.9750
Epoch 47/50
83/83 [==============================] - 66s 791ms/step - loss: 0.0072 - accuracy:
0.9992 - val_loss: 0.2408 - val_accuracy: 0.9719
Epoch 48/50
83/83 [==============================] - 66s 792ms/step - loss: 0.0027 - accuracy:
0.9996 - val_loss: 0.2775 - val_accuracy: 0.9719
Epoch 49/50
83/83 [==============================] - 66s 795ms/step - loss: 0.0010 - accuracy:
1.0000 - val_loss: 0.2859 - val_accuracy: 0.9719
Epoch 50/50
83/83 [==============================] - 67s 801ms/step - loss: 5.9172e-04 - accur
acy: 1.0000 - val_loss: 0.3017 - val_accuracy: 0.9719
```

In [28]:
```
t1 = time.time()
```

Training Speed

In [29]:
```
print("CNN Model Training time:  ", (t1-t0)/60 , "minutes")
```

```
CNN Model Training time:   55.648754107952115 minutes
```

Evaluation

In [30]:
```
scores = model.evaluate(test_ds)
```

```
11/11 [==============================] - 3s 151ms/step - loss: 0.3618 - accuracy:
0.9631
```

In [31]:
```
scores
```

Out[31]:
```
[0.36177319288253784, 0.9630681872367859]
```

Predictions

In [32]:
```
predicted = model.predict(test_ds)
```

```
11/11 [==============================] - 2s 141ms/step
```

In [33]:
```
import numpy as np

confidence = np.max(predicted, axis=1)
predictions = np.argmax(predicted, axis=1)
```

In [34]:
```
# predicted

# print(predicted)
print(len(predicted))
print(len(test_ds))

# print(predictions)
print(len(predictions))
```

```
352
11
352
```

In [ ]:

Plotting History

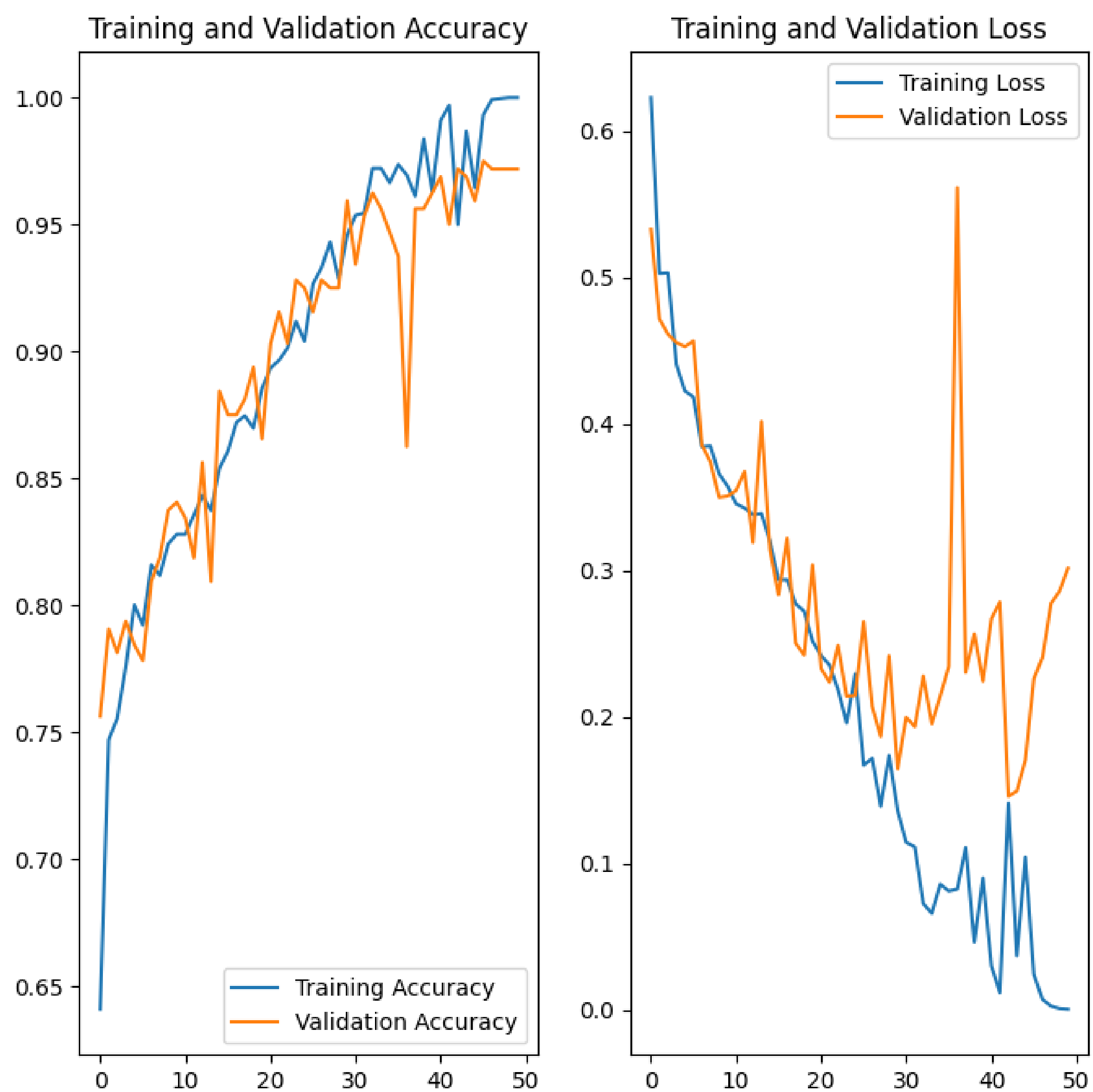
In [35]:
```
acc = history.history['accuracy']
val_acc = history.history['val_accuracy']

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

In [36]:
```
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()
```



Confusion Matrix

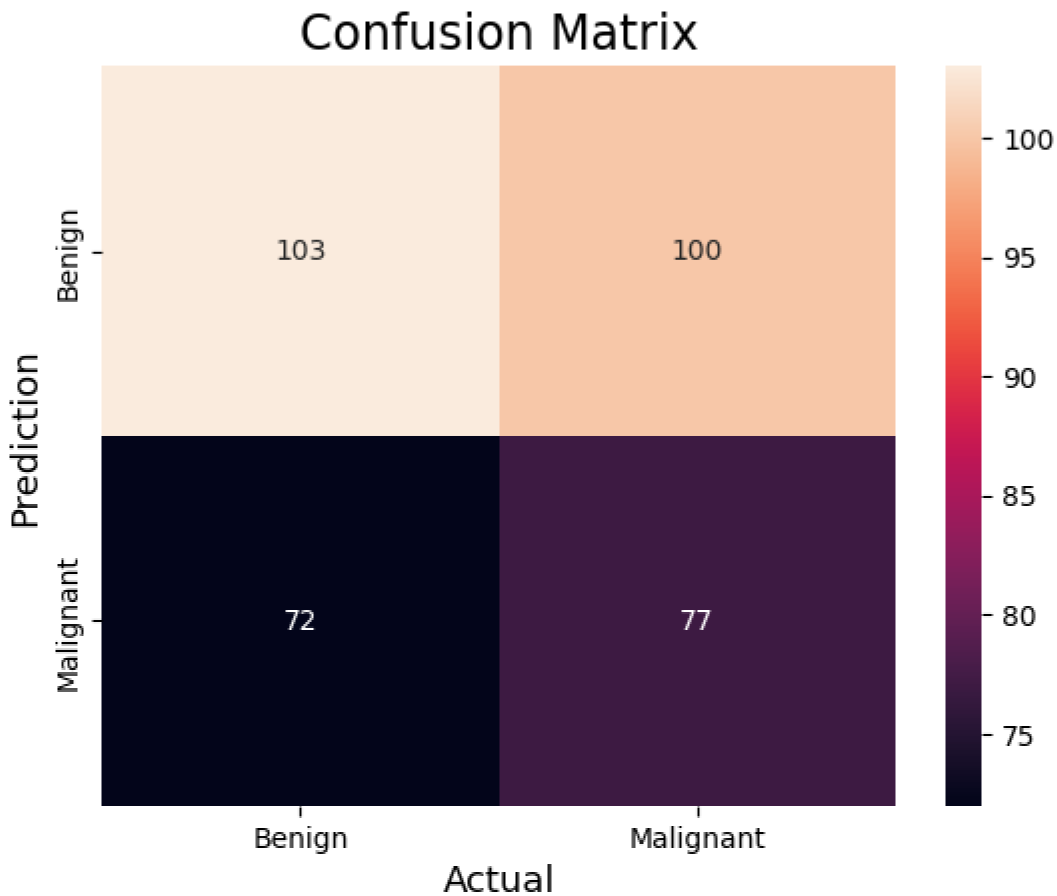
In [37]:
```
from sklearn.metrics import confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt
```

In [38]:
```
cm = confusion_matrix(actual_label_test, predictions)


sns.heatmap(
    cm,
    annot=True,
    fmt='g',
    xticklabels=['Benign','Malignant'],
    yticklabels=['Benign','Malignant']
```

```
)

plt.ylabel('Prediction',fontsize=13)
plt.xlabel('Actual',fontsize=13)
plt.title('Confusion Matrix',fontsize=17)
plt.show()
```



In [39]:
```
from sklearn.metrics import classification_report
print(classification_report(actual_label_test, predictions))
```

```
              precision    recall  f1-score   support

           0       0.59      0.51      0.54       203
           1       0.44      0.52      0.47       149

    accuracy                           0.51       352
   macro avg       0.51      0.51      0.51       352
weighted avg       0.52      0.51      0.51       352
```

Saving the Model

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

In [43]:
```
import os
# model_version=max([int(i) for i in os.listdir("../MODELS") + [0]])+1
```

```
model.save(f"../MODELS/final.h5")
```

In [ ]:

```
```