**Brain Tumor Detection Using a Convolutional Neural Network**

**About the Brain MRI Images dataset:**  
The dataset contains 2 folders: yes and no which contains 253 Brain MRI Images. The folder yes contains 155 Brain MRI Images that are tumorous and the folder no contains 98 Brain MRI Images that are non-tumorous. You can find it [here](https://www.kaggle.com/navoneel/brain-mri-images-for-brain-tumor-detection).

**Import Necessary Modules**

In [1]:

**import** tensorflow **as** tf

**from** tensorflow.keras.layers **import** Conv2D, Input, ZeroPadding2D, BatchNormalization, Activation, MaxPooling2D, Flatten, Dense

**from** tensorflow.keras.models **import** Model, load\_model

**from** tensorflow.keras.callbacks **import** TensorBoard, ModelCheckpoint

**from** sklearn.model\_selection **import** train\_test\_split

**from** sklearn.metrics **import** f1\_score

**from** sklearn.utils **import** shuffle

**import** cv2

**import** imutils

**import** numpy **as** np

**import** matplotlib.pyplot **as** plt

**import** time

**from** os **import** listdir

**%matplotlib** inline

**Data Preparation & Preprocessing**

In order to crop the part that contains only the brain of the image, I used a cropping technique to find the extreme top, bottom, left and right points of the brain. You can read more about it here [Finding extreme points in contours with OpenCV](https://www.pyimagesearch.com/2016/04/11/finding-extreme-points-in-contours-with-opencv/).

In [2]:

**def** crop\_brain\_contour(image, plot**=False**):

*#import imutils*

*#import cv2*

*#from matplotlib import pyplot as plt*

*# Convert the image to grayscale, and blur it slightly*

gray **=** cv2**.**cvtColor(image, cv2**.**COLOR\_BGR2GRAY)

gray **=** cv2**.**GaussianBlur(gray, (5, 5), 0)

*# Threshold the image, then perform a series of erosions +*

*# dilations to remove any small regions of noise*

thresh **=** cv2**.**threshold(gray, 45, 255, cv2**.**THRESH\_BINARY)[1]

thresh **=** cv2**.**erode(thresh, **None**, iterations**=**2)

thresh **=** cv2**.**dilate(thresh, **None**, iterations**=**2)

*# Find contours in thresholded image, then grab the largest one*

cnts **=** cv2**.**findContours(thresh**.**copy(), cv2**.**RETR\_EXTERNAL, cv2**.**CHAIN\_APPROX\_SIMPLE)

cnts **=** imutils**.**grab\_contours(cnts)

c **=** max(cnts, key**=**cv2**.**contourArea)

*# Find the extreme points*

extLeft **=** tuple(c[c[:, :, 0]**.**argmin()][0])

extRight **=** tuple(c[c[:, :, 0]**.**argmax()][0])

extTop **=** tuple(c[c[:, :, 1]**.**argmin()][0])

extBot **=** tuple(c[c[:, :, 1]**.**argmax()][0])

*# crop new image out of the original image using the four extreme points (left, right, top, bottom)*

new\_image **=** image[extTop[1]:extBot[1], extLeft[0]:extRight[0]]

**if** plot:

plt**.**figure()

plt**.**subplot(1, 2, 1)

plt**.**imshow(image)

plt**.**tick\_params(axis**=**'both', which**=**'both',

top**=False**, bottom**=False**, left**=False**, right**=False**,

labelbottom**=False**, labeltop**=False**, labelleft**=False**, labelright**=False**)

plt**.**title('Original Image')

plt**.**subplot(1, 2, 2)

plt**.**imshow(new\_image)

plt**.**tick\_params(axis**=**'both', which**=**'both',

top**=False**, bottom**=False**, left**=False**, right**=False**,

labelbottom**=False**, labeltop**=False**, labelleft**=False**, labelright**=False**)

plt**.**title('Cropped Image')

plt**.**show()

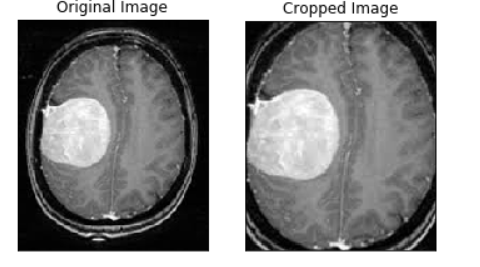
**return** new\_image

In order to better understand what it's doing, let's grab an image from the dataset and apply this cropping function to see the result:

In [3]:

ex\_img **=** cv2**.**imread('yes/Y1.jpg')

ex\_new\_img **=** crop\_brain\_contour(ex\_img, **True**)



**Load up the data:**

The following function takes two arguments, the first one is a list of directory paths for the folders 'yes' and 'no' that contain the image data and the second argument is the image size, and for every image in both directories and does the following:

1. Read the image.
2. Crop the part of the image representing only the brain.
3. Resize the image (because the images in the dataset come in different sizes (meaning width, height and # of channels). So, we want all of our images to be (240, 240, 3) to feed it as an input to the neural network.
4. Apply normalization because we want pixel values to be scaled to the range 0-1.
5. Append the image to *X* and its label to *y*.

After that, Shuffle *X* and *y*, because the data is ordered (meaning the arrays contains the first part belonging to one class and the second part belonging to the other class, and we don't want that).  
Finally, Return *X* and *y*.

In [5]:

**def** load\_data(dir\_list, image\_size):

"""

Read images, resize and normalize them.

Arguments:

dir\_list: list of strings representing file directories.

Returns:

X: A numpy array with shape = (#\_examples, image\_width, image\_height, #\_channels)

y: A numpy array with shape = (#\_examples, 1)

"""

*# load all images in a directory*

X **=** []

y **=** []

image\_width, image\_height **=** image\_size

**for** directory **in** dir\_list:

**for** filename **in** listdir(directory):

*# load the image*

image **=** cv2**.**imread(directory **+** '\\' **+** filename)

*# crop the brain and ignore the unnecessary rest part of the image*

image **=** crop\_brain\_contour(image, plot**=False**)

*# resize image*

image **=** cv2**.**resize(image, dsize**=**(image\_width, image\_height), interpolation**=**cv2**.**INTER\_CUBIC)

*# normalize values*

image **=** image **/** 255.

*# convert image to numpy array and append it to X*

X**.**append(image)

*# append a value of 1 to the target array if the image*

*# is in the folder named 'yes', otherwise append 0.*

**if** directory[**-**3:] **==** 'yes':

y**.**append([1])

**else**:

y**.**append([0])

X **=** np**.**array(X)

y **=** np**.**array(y)

*# Shuffle the data*

X, y **=** shuffle(X, y)

print(f'Number of examples is: {len(X)}')

print(f'X shape is: {X**.**shape}')

print(f'y shape is: {y**.**shape}')

**return** X, y

Load up the data that we augmented earlier in the Data Augmentation notebook.  
**Note:** the augmented data directory contains not only the new generated images but also the original images.

In [6]:

augmented\_path **=** 'augmented data/'

*# augmented data (yes and no) contains both the original and the new generated examples*

augmented\_yes **=** augmented\_path **+** 'yes'

augmented\_no **=** augmented\_path **+** 'no'

IMG\_WIDTH, IMG\_HEIGHT **=** (240, 240)

X, y **=** load\_data([augmented\_yes, augmented\_no], (IMG\_WIDTH, IMG\_HEIGHT))

Number of examples is: 2065

X shape is: (2065, 240, 240, 3)

y shape is: (2065, 1)

As we see, we have 2065 images. Each images has a shape of **(240, 240, 3)=(image\_width, image\_height, number\_of\_channels)**

**Plot sample images:**

In [6]:

**def** plot\_sample\_images(X, y, n**=**50):

"""

Plots n sample images for both values of y (labels).

Arguments:

X: A numpy array with shape = (#\_examples, image\_width, image\_height, #\_channels)

y: A numpy array with shape = (#\_examples, 1)

"""

**for** label **in** [0,1]:

*# grab the first n images with the corresponding y values equal to label*

images **=** X[np**.**argwhere(y **==** label)]

n\_images **=** images[:n]

columns\_n **=** 10

rows\_n **=** int(n**/** columns\_n)

plt**.**figure(figsize**=**(20, 10))

i **=** 1 *# current plot*

**for** image **in** n\_images:

plt**.**subplot(rows\_n, columns\_n, i)

plt**.**imshow(image[0])

*# remove ticks*

plt**.**tick\_params(axis**=**'both', which**=**'both',

top**=False**, bottom**=False**, left**=False**, right**=False**,

labelbottom**=False**, labeltop**=False**, labelleft**=False**, labelright**=False**)

i **+=** 1

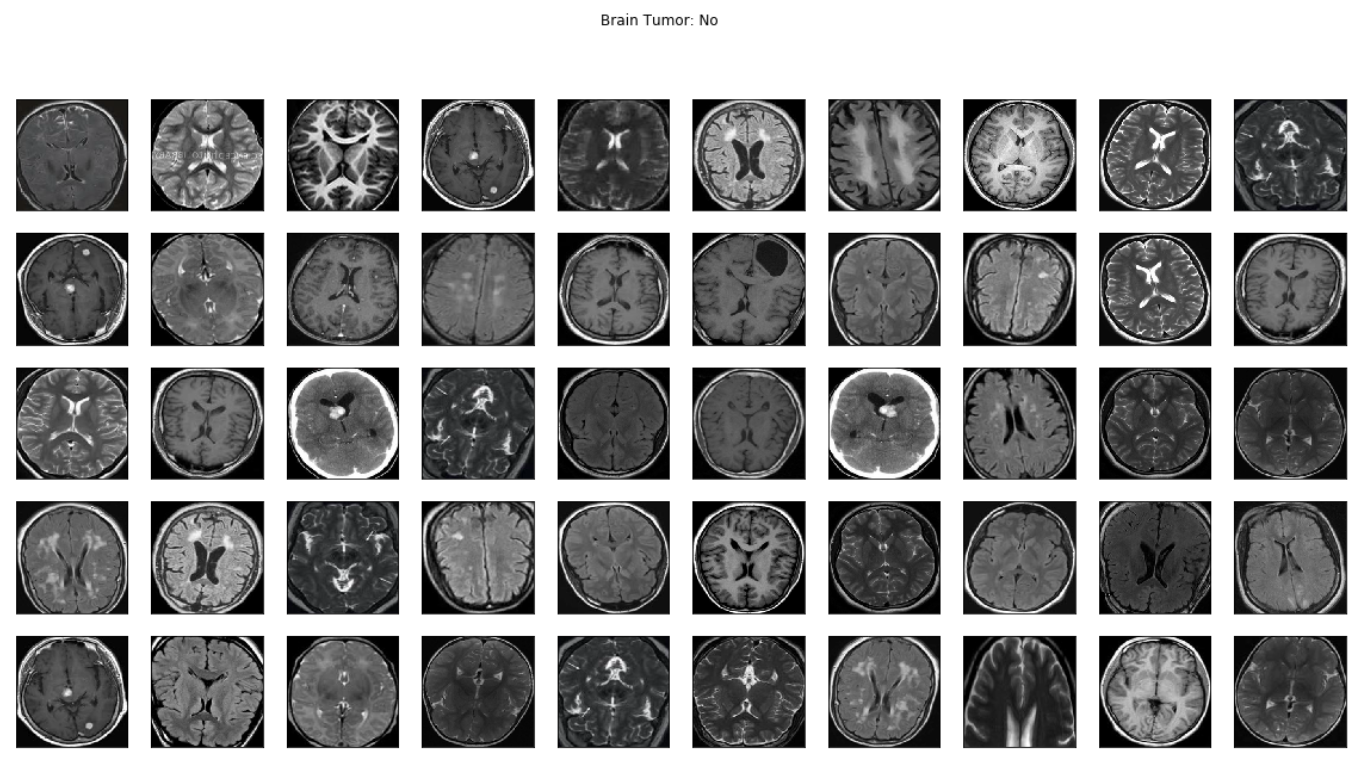
label\_to\_str **=** **lambda** label: "Yes" **if** label **==** 1 **else** "No"

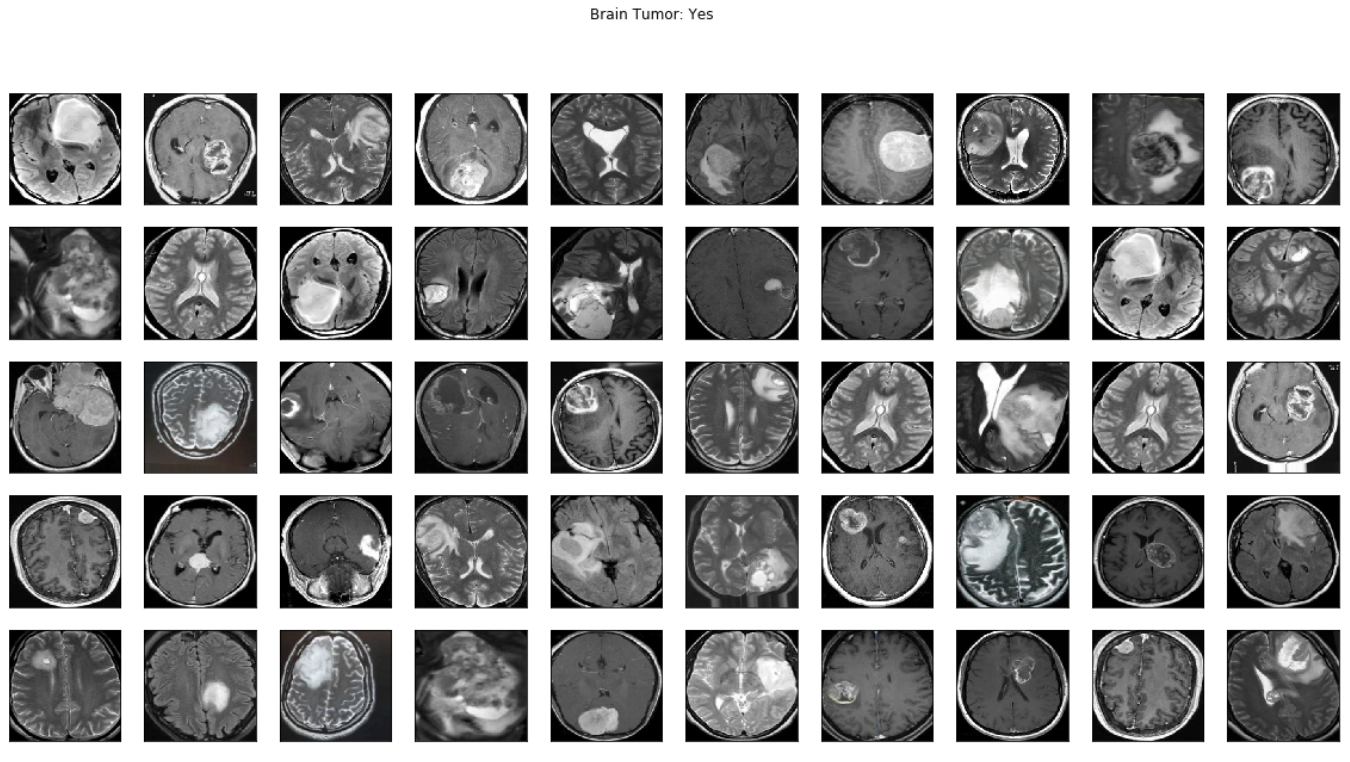
plt**.**suptitle(f"Brain Tumor: {label\_to\_str(label)}")

plt**.**show()

In [7]:

plot\_sample\_images(X, y)





**Split the data:**

Split *X* and *y* into training, validation (development) and validation sets.

In [8]:

**def** split\_data(X, y, test\_size**=**0.2):

"""

Splits data into training, development and test sets.

Arguments:

X: A numpy array with shape = (#\_examples, image\_width, image\_height, #\_channels)

y: A numpy array with shape = (#\_examples, 1)

Returns:

X\_train: A numpy array with shape = (#\_train\_examples, image\_width, image\_height, #\_channels)

y\_train: A numpy array with shape = (#\_train\_examples, 1)

X\_val: A numpy array with shape = (#\_val\_examples, image\_width, image\_height, #\_channels)

y\_val: A numpy array with shape = (#\_val\_examples, 1)

X\_test: A numpy array with shape = (#\_test\_examples, image\_width, image\_height, #\_channels)

y\_test: A numpy array with shape = (#\_test\_examples, 1)

"""

X\_train, X\_test\_val, y\_train, y\_test\_val **=** train\_test\_split(X, y, test\_size**=**test\_size)

X\_test, X\_val, y\_test, y\_val **=** train\_test\_split(X\_test\_val, y\_test\_val, test\_size**=**0.5)

**return** X\_train, y\_train, X\_val, y\_val, X\_test, y\_test

Let's use the following way to split:

1. 70% of the data for training.
2. 15% of the data for validation
3. 3.15% of the data for testing.

In [9]:

X\_train, y\_train, X\_val, y\_val, X\_test, y\_test **=** split\_data(X, y, test\_size**=**0.3)

In [10]:

print ("number of training examples = " **+** str(X\_train**.**shape[0]))

print ("number of development examples = " **+** str(X\_val**.**shape[0]))

print ("number of test examples = " **+** str(X\_test**.**shape[0]))

print ("X\_train shape: " **+** str(X\_train**.**shape))

print ("Y\_train shape: " **+** str(y\_train**.**shape))

print ("X\_val (dev) shape: " **+** str(X\_val**.**shape))

print ("Y\_val (dev) shape: " **+** str(y\_val**.**shape))

print ("X\_test shape: " **+** str(X\_test**.**shape))

print ("Y\_test shape: " **+** str(y\_test**.**shape))

number of training examples = 1445

number of development examples = 310

number of test examples = 310

X\_train shape: (1445, 240, 240, 3)

Y\_train shape: (1445, 1)

X\_val (dev) shape: (310, 240, 240, 3)

Y\_val (dev) shape: (310, 1)

X\_test shape: (310, 240, 240, 3)

Y\_test shape: (310, 1)

Some helper functions:

In [11]:

*# Nicely formatted time string*

**def** hms\_string(sec\_elapsed):

h **=** int(sec\_elapsed **/** (60 **\*** 60))

m **=** int((sec\_elapsed **%** (60 **\*** 60)) **/** 60)

s **=** sec\_elapsed **%** 60

**return** f"{h}:{m}:{round(s,1)}"

In [12]:

**def** compute\_f1\_score(y\_true, prob):

*# convert the vector of probabilities to a target vector*

y\_pred **=** np**.**where(prob **>** 0.5, 1, 0)

score **=** f1\_score(y\_true, y\_pred)

**return** score

**Build the model**

Let's build a convolutional neural network model:

In [13]

**def** build\_model(input\_shape):

"""

Arugments:

input\_shape: A tuple representing the shape of the input of the model. shape=(image\_width, image\_height, #\_channels)

Returns:

model: A Model object.

"""

*# Define the input placeholder as a tensor with shape input\_shape.*

X\_input **=** Input(input\_shape) *# shape=(?, 240, 240, 3)*

*# Zero-Padding: pads the border of X\_input with zeroes*

X **=** ZeroPadding2D((2, 2))(X\_input) *# shape=(?, 244, 244, 3)*

*# CONV -> BN -> RELU Block applied to X*

X **=** Conv2D(32, (7, 7), strides **=** (1, 1), name **=** 'conv0')(X)

X **=** BatchNormalization(axis **=** 3, name **=** 'bn0')(X)

X **=** Activation('relu')(X) *# shape=(?, 238, 238, 32)*

*# MAXPOOL*

X **=** MaxPooling2D((4, 4), name**=**'max\_pool0')(X) *# shape=(?, 59, 59, 32)*

*# MAXPOOL*

X **=** MaxPooling2D((4, 4), name**=**'max\_pool1')(X) *# shape=(?, 14, 14, 32)*

*# FLATTEN X*

X **=** Flatten()(X) *# shape=(?, 6272)*

*# FULLYCONNECTED*

X **=** Dense(1, activation**=**'sigmoid', name**=**'fc')(X) *# shape=(?, 1)*

*# Create model. This creates your Keras model instance, you'll use this instance to train/test the model.*

model **=** Model(inputs **=** X\_input, outputs **=** X, name**=**'BrainDetectionModel')

**return** model

Define the image shape:

In [14]:

IMG\_SHAPE **=** (IMG\_WIDTH, IMG\_HEIGHT, 3)

In [15]:

model **=** build\_model(IMG\_SHAPE)

In [16]:

model**.**summary()

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

input\_1 (InputLayer) (None, 240, 240, 3) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

zero\_padding2d (ZeroPadding2 (None, 244, 244, 3) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

conv0 (Conv2D) (None, 238, 238, 32) 4736

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

bn0 (BatchNormalization) (None, 238, 238, 32) 128

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

activation (Activation) (None, 238, 238, 32) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

max\_pool0 (MaxPooling2D) (None, 59, 59, 32) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

max\_pool1 (MaxPooling2D) (None, 14, 14, 32) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

flatten (Flatten) (None, 6272) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

fc (Dense) (None, 1) 6273

=================================================================

Total params: 11,137

Trainable params: 11,073

Non-trainable params: 64

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Compile the model:

In [17]:

model**.**compile(optimizer**=**'adam', loss**=**'binary\_crossentropy', metrics**=**['accuracy'])

In [18]:

*# tensorboard*

log\_file\_name **=** f'brain\_tumor\_detection\_cnn\_{int(time**.**time())}'

tensorboard **=** TensorBoard(log\_dir**=**f'logs/{log\_file\_name}')

In [19]:

*# checkpoint*

*# unique file name that will include the epoch and the validation (development) accuracy*

filepath**=**"cnn-parameters-improvement-{epoch:02d}-{val\_acc:.2f}"

*# save the model with the best validation (development) accuracy till now*

checkpoint **=** ModelCheckpoint("models/{}.model"**.**format(filepath, monitor**=**'val\_acc', verbose**=**1, save\_best\_only**=True**, mode**=**'max'))

**Train the model**

In [20]:

start\_time **=** time**.**time()

model**.**fit(x**=**X\_train, y**=**y\_train, batch\_size**=**32, epochs**=**10, validation\_data**=**(X\_val, y\_val), callbacks**=**[tensorboard, checkpoint])

end\_time **=** time**.**time()

execution\_time **=** (end\_time **-** start\_time)

print(f"Elapsed time: {hms\_string(execution\_time)}")

Train on 1445 samples, validate on 310 samples

Epoch 1/10

1445/1445 [==============================] - 434s 300ms/step - loss: 0.8331 - acc: 0.5945 - val\_loss: 0.6829 - val\_acc: 0.4968

Epoch 2/10

1445/1445 [==============================] - 463s 320ms/step - loss: 0.4817 - acc: 0.7668 - val\_loss: 0.6342 - val\_acc: 0.6742

Epoch 3/10

1445/1445 [==============================] - 471s 326ms/step - loss: 0.4361 - acc: 0.8069 - val\_loss: 0.5294 - val\_acc: 0.8065

Epoch 4/10

1445/1445 [==============================] - 465s 322ms/step - loss: 0.3641 - acc: 0.8574 - val\_loss: 0.6092 - val\_acc: 0.6323

Epoch 5/10

1445/1445 [==============================] - 457s 316ms/step - loss: 0.3940 - acc: 0.8339 - val\_loss: 0.4689 - val\_acc: 0.7742

Epoch 6/10

1445/1445 [==============================] - 452s 313ms/step - loss: 0.3154 - acc: 0.8692 - val\_loss: 0.4448 - val\_acc: 0.7806

Epoch 7/10

1445/1445 [==============================] - 465s 322ms/step - loss: 0.2776 - acc: 0.8872 - val\_loss: 0.4747 - val\_acc: 0.7323

Epoch 8/10

1445/1445 [==============================] - 439s 304ms/step - loss: 0.3271 - acc: 0.8519 - val\_loss: 0.3655 - val\_acc: 0.8516

Epoch 9/10

1445/1445 [==============================] - 435s 301ms/step - loss: 0.2182 - acc: 0.9190 - val\_loss: 0.4557 - val\_acc: 0.8129

Epoch 10/10

1445/1445 [==============================] - 438s 303ms/step - loss: 0.2054 - acc: 0.9225 - val\_loss: 0.4038 - val\_acc: 0.8129

Elapsed time: 1:15:23.8

Let's train for a few more epochs:

In [36]:

start\_time **=** time**.**time()

model**.**fit(x**=**X\_train, y**=**y\_train, batch\_size**=**32, epochs**=**3, validation\_data**=**(X\_val, y\_val), callbacks**=**[tensorboard, checkpoint])

end\_time **=** time**.**time()

execution\_time **=** (end\_time **-** start\_time)

print(f"Elapsed time: {hms\_string(execution\_time)}")

Train on 1445 samples, validate on 310 samples

Epoch 1/3

1445/1445 [==============================] - 431s 299ms/step - loss: 0.2065 - acc: 0.9239 - val\_loss: 0.3357 - val\_acc: 0.8871

Epoch 2/3

1445/1445 [==============================] - 432s 299ms/step - loss: 0.1811 - acc: 0.9363 - val\_loss: 0.3529 - val\_acc: 0.8516

Epoch 3/3

1445/1445 [==============================] - 425s 294ms/step - loss: 0.1827 - acc: 0.9287 - val\_loss: 0.4038 - val\_acc: 0.8323

Elapsed time: 0:21:29.4

In [37]:

start\_time **=** time**.**time()

model**.**fit(x**=**X\_train, y**=**y\_train, batch\_size**=**32, epochs**=**3, validation\_data**=**(X\_val, y\_val), callbacks**=**[tensorboard, checkpoint])

end\_time **=** time**.**time()

execution\_time **=** (end\_time **-** start\_time)

print(f"Elapsed time: {hms\_string(execution\_time)}")

Train on 1445 samples, validate on 310 samples

Epoch 1/3

1445/1445 [==============================] - 438s 303ms/step - loss: 0.1471 - acc: 0.9612 - val\_loss: 0.3190 - val\_acc: 0.8903

Epoch 2/3

1445/1445 [==============================] - 432s 299ms/step - loss: 0.1384 - acc: 0.9564 - val\_loss: 0.3509 - val\_acc: 0.8613

Epoch 3/3

1445/1445 [==============================] - 429s 297ms/step - loss: 0.1240 - acc: 0.9647 - val\_loss: 0.3358 - val\_acc: 0.8710

Elapsed time: 0:21:38.5

In [38]:

start\_time **=** time**.**time()

model**.**fit(x**=**X\_train, y**=**y\_train, batch\_size**=**32, epochs**=**3, validation\_data**=**(X\_val, y\_val), callbacks**=**[tensorboard, checkpoint])

end\_time **=** time**.**time()

execution\_time **=** (end\_time **-** start\_time)

print(f"Elapsed time: {hms\_string(execution\_time)}")

Train on 1445 samples, validate on 310 samples

Epoch 1/3

1445/1445 [==============================] - 536s 371ms/step - loss: 0.1586 - acc: 0.9453 - val\_loss: 0.4005 - val\_acc: 0.8548

Epoch 2/3

1445/1445 [==============================] - 427s 296ms/step - loss: 0.1244 - acc: 0.9647 - val\_loss: 0.3149 - val\_acc: 0.9000

Epoch 3/3

1445/1445 [==============================] - 429s 297ms/step - loss: 0.1074 - acc: 0.9668 - val\_loss: 0.3118 - val\_acc: 0.8935

Elapsed time: 0:23:11.9

In [39]:

start\_time **=** time**.**time()

model**.**fit(x**=**X\_train, y**=**y\_train, batch\_size**=**32, epochs**=**5, validation\_data**=**(X\_val, y\_val), callbacks**=**[tensorboard, checkpoint])

end\_time **=** time**.**time()

execution\_time **=** (end\_time **-** start\_time)

print(f"Elapsed time: {hms\_string(execution\_time)}")

Train on 1445 samples, validate on 310 samples

Epoch 1/5

1445/1445 [==============================] - 427s 296ms/step - loss: 0.0899 - acc: 0.9785 - val\_loss: 0.3310 - val\_acc: 0.8935

Epoch 2/5

1445/1445 [==============================] - 426s 295ms/step - loss: 0.1343 - acc: 0.9509 - val\_loss: 0.5169 - val\_acc: 0.8258

Epoch 3/5

1445/1445 [==============================] - 425s 294ms/step - loss: 0.1137 - acc: 0.9626 - val\_loss: 0.6945 - val\_acc: 0.7516

Epoch 4/5

1445/1445 [==============================] - 430s 298ms/step - loss: 0.1018 - acc: 0.9640 - val\_loss: 0.3210 - val\_acc: 0.9065

Epoch 5/5

1445/1445 [==============================] - 434s 300ms/step - loss: 0.0949 - acc: 0.9689 - val\_loss: 0.4250 - val\_acc: 0.8484

Elapsed time: 0:35:41.9

In [21]:

history **=** model**.**history**.**history

In [22]:

**for** key **in** history**.**keys():

print(key)

val\_loss

val\_acc

loss

acc

**Plot Loss & Accuracy**

In [23]:

**def** plot\_metrics(history):

train\_loss **=** history['loss']

val\_loss **=** history['val\_loss']

train\_acc **=** history['acc']

val\_acc **=** history['val\_acc']

*# Loss*

plt**.**figure()

plt**.**plot(train\_loss, label**=**'Training Loss')

plt**.**plot(val\_loss, label**=**'Validation Loss')

plt**.**title('Loss')

plt**.**legend()

plt**.**show()

*# Accuracy*

plt**.**figure()

plt**.**plot(train\_acc, label**=**'Training Accuracy')

plt**.**plot(val\_acc, label**=**'Validation Accuracy')

plt**.**title('Accuracy')

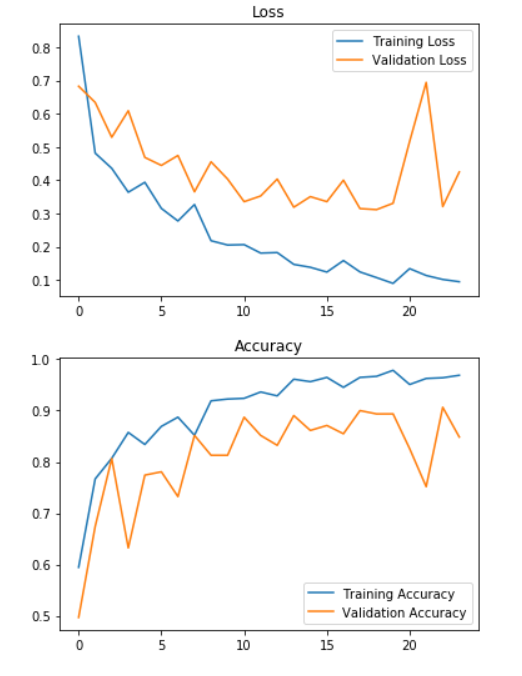
plt**.**legend()

plt**.**show()

**Note:** Since we trained the model using more than model.fit() function call, this made the history only contain the metric values of the epochs for the last call (which was for 5 epochs), so to plot the metric values across the whole process of trianing the model from the beginning, I had to grab the rest of the values.

In [68]:

plot\_metrics(history)



**Results**

Let's experiment with the best model (the one with the best validation accuracy):

Concretely, the model at the 23rd iteration with validation accuracy of 91%

**Load the best model**

In [71]:

best\_model **=** load\_model(filepath**=**'models/cnn-parameters-improvement-23-0.91.model')

In [72]:

best\_model**.**metrics\_names

Out[72]:

['loss', 'acc']

Evaluate the best model on the testing data:

In [73]:

loss, acc **=** best\_model**.**evaluate(x**=**X\_test, y**=**y\_test)

310/310 [==============================] - 18s 57ms/step

**Accuracy of the best model on the testing data:**

In [74]:

print (f"Test Loss = {loss}")

print (f"Test Accuracy = {acc}")

Test Loss = 0.33390871454631127

Test Accuracy = 0.8870967741935484

**F1 score for the best model on the testing data:**

In [75]:

y\_test\_prob **=** best\_model**.**predict(X\_test)

In [76]:

f1score **=** compute\_f1\_score(y\_test, y\_test\_prob)

print(f"F1 score: {f1score}")

F1 score: 0.8829431438127091

Let's also find the f1 score on the validation data:

In [83]:

y\_val\_prob **=** best\_model**.**predict(X\_val)

In [85]:

f1score\_val **=** compute\_f1\_score(y\_val, y\_val\_prob)

print(f"F1 score: {f1score\_val}")

F1 score: 0.9123867069486403

**Results Interpretation**

Let's remember the percentage of positive and negative examples:

In [77]:

**def** data\_percentage(y):

m**=**len(y)

n\_positive **=** np**.**sum(y)

n\_negative **=** m **-** n\_positive

pos\_prec **=** (n\_positive**\*** 100.0)**/** m

neg\_prec **=** (n\_negative**\*** 100.0)**/** m

print(f"Number of examples: {m}")

print(f"Percentage of positive examples: {pos\_prec}%, number of pos examples: {n\_positive}")

print(f"Percentage of negative examples: {neg\_prec}%, number of neg examples: {n\_negative}")

In [81]:

*# the whole data*

data\_percentage(y)

Number of examples: 2065

Percentage of positive examples: 52.54237288135593%, number of pos examples: 1085

Percentage of negative examples: 47.45762711864407%, number of neg examples: 980

In [79]:

print("Training Data:")

data\_percentage(y\_train)

print("Validation Data:")

data\_percentage(y\_val)

print("Testing Data:")

data\_percentage(y\_test)

Training Data:

Number of examples: 1445

Percentage of positive examples: 52.8719723183391%, number of pos examples: 764

Percentage of negative examples: 47.1280276816609%, number of neg examples: 681

Validation Data:

Number of examples: 310

Percentage of positive examples: 54.83870967741935%, number of pos examples: 170

Percentage of negative examples: 45.16129032258065%, number of neg examples: 140

Testing Data:

Number of examples: 310

Percentage of positive examples: 48.70967741935484%, number of pos examples: 151

Percentage of negative examples: 51.29032258064516%, number of neg examples: 159

As expectred, the percentage of positive examples are around 50%.

**Conclusion:**

**Now, the model detects brain tumor with:**

**88.7%** accuracy on the **test set**.  
**0.88** f1 score on the **test set**.  
These resutls are very good considering that the data is balanced.

**Performance Table:**

|  | **Validation set** | **Test set** |
| --- | --- | --- |
| Accuracy | 91% | 89% |
| F1 score | 0.91 | 0.88 |

Hooray!

In [ ]: