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.

Import Necessary Modules

In [1]:
```
import tensorflow as tf
from tensorflow.keras.layers import Conv2D, Input, ZeroPadding2D, BatchNormalizati
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

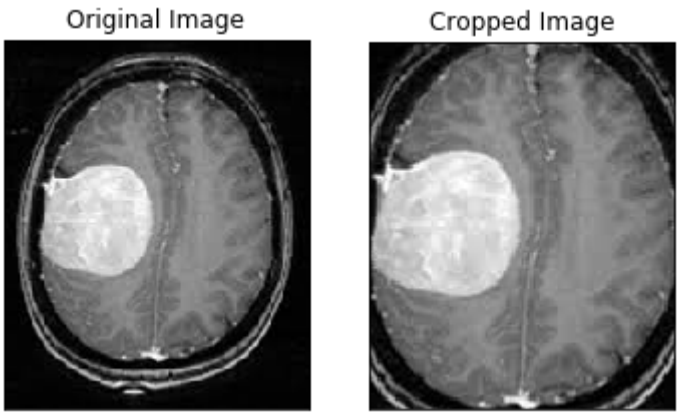
In order to crop the part that contains only the brain of the image, I used a cropping technique to find the extreme top, bottom, left and right points of the brain. You can read more about it here Finding extreme points in contours with OpenCV.

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_SIMI
    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,
    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, labelr

        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, labelr

        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, #_cha
        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), interpola
            # 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 exam
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, #_cha
        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, labe

            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, #_cha
        y: A numpy array with shape = (#_examples, 1)
    Returns:
        X_train: A numpy array with shape = (#_train_examples, image_width, image_h
        y_train: A numpy array with shape = (#_train_examples, 1)
        X_val: A numpy array with shape = (#_val_examples, image_width, image_heigh
        y_val: A numpy array with shape = (#_val_examples, 1)
        X_test: A numpy array with shape = (#_test_examples, image_width, image_hei
        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=te
    X_test, X_val, y_test, y_val = train_test_split(X_test_val, y_test_val, test_s

    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. 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. shap
    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 instanc
    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) acc
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',
```

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

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

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

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

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

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)
```

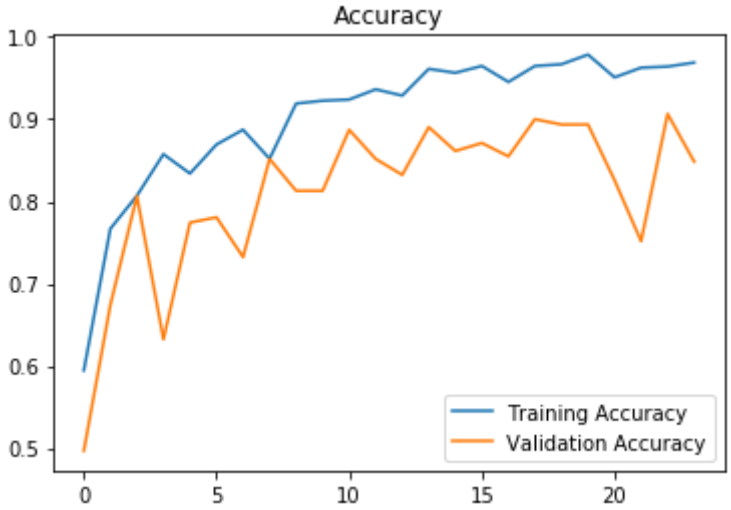
```
val_loss
val_acc
loss
acc
```

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: 
    print(f"Percentage of negative examples: {neg_prec}%, number of neg examples: 
```

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 [ ]: