Brain Tumor Detection Dataset

1. Introduction to Brain Tumor Detection Using CNN

Imagine a world where a clinician, armed with the keen eye of AI, can quickly and accurately identify tumors from intricate neuroimaging data. This is not science fiction, but the promise of transformative technology. Automated brain tumor detection holds the potential to revolutionize healthcare, offering:

- **Earlier Diagnosis:** Timely identification unlocks the door to early intervention, improving treatment outcomes and survival rates.
- **Enhanced Accuracy:** AI can analyze subtle patterns invisible to the human eye, reducing misdiagnosis and leading to more efficient resource allocation.
- **Streamlined Workflow:** Automated tumor detection can alleviate the burden on radiologists, enabling faster diagnosis and quicker response.

2. 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.

3.Libraries we used:

- **TensorFlow (tf):** The primary framework for building and training deep learning models.
- **Keras:** A high-level API within TensorFlow, providing user-friendly tools for model creation and training.
- **NumPy (np):** A fundamental library for numerical computations and array manipulation, essential for handling image data.

- **OpenCV (cv2):** A powerful library for real-time computer vision tasks, used here for image processing and manipulation.
- **Scikit-learn:** Provides tools for machine learning tasks, as data splitting and evaluation.

3.Specific Module Imports:

- **Conv2D, Input, ZeroPadding2D, BatchNormalization, Activation, MaxPooling2D, Flatten, Dense:** Key layers for building convolutional neural networks (CNNs) in Keras, used to construct your model architecture.
- **Model, load_model:** Classes for creating and loading Keras models.
- **TensorBoard, ModelCheckpoint:** Callbacks for monitoring training progress and saving the best model checkpoints.
- **train_test_split, f1_score, shuffle:** Functions for splitting data, evaluating model performance, and shuffling data for randomization.
- **imutils:** A library for convenient image processing operations.
- **matplotlib.pyplot (plt):** Used for creating visualizations, such as plotting images or results.
- **time:** For time-related tasks, potentially used for tracking execution time or timestamps.
- **os:** For interacting with the operating system, likely used for file and directory operations.

Data Augmentation: Creating More From Less

Data augmentation is a powerful technique in machine learning, particularly for tasks like image and audio recognition. It essentially involves artificially **manipulating existing data points to create new ones with the same label**. This "augmented" data helps to:

1. Increase the size and diversity of your dataset: With limited data, your model might overfit, memorizing specific patterns instead of learning generalizations. Data augmentation provides more training examples, diversifying the dataset and ensuring the model learns broader features.

2. Improve model generalizability: The real world presents images and sounds at different angles, with varying brightness, noise, and other distortions. Data augmentation mimics these variations, preparing the model to recognize patterns even under different conditions.

3. Reduce overfitting: Overfitting occurs when your model memorizes specific training examples, failing to generalize to unseen data. By introducing a wider range of examples, data augmentation prevents the model from overfitting to idiosyncrasies in the original dataset.

Here are some common data augmentation techniques

- Image Augmentation:
    - **Geometric transformations:** Rotating, scaling, shifting, and shearing images.
    - **Color alterations:** Changing brightness, contrast, hue, and saturation.
    - **Adding noise:** Simulating real-world image imperfections like blurring or pixelation.
    - **Cropping and flipping:** Extracting smaller regions or flipping images horizontally or vertically.

Jump into code:

We have two different code files , one for augmentation , and for the main code starting with the augmentation one

```
import tensorflow as tf
from keras.preprocessing.image import ImageDataGenerator
import cv2
import imutils
import matplotlib.pyplot as plt
from os import listdir
```

Here the main library we're using is keras where we import form it the imgeDataGenerator which going to be used to augment data

Starting of the data augmentation

```
def augment_data(file_dir, n_generated_samples, save_to_dir):
    """
    Arguments:
        file_dir: A string representing the directory where images that we want to augment are found.
        n_generated_samples: A string representing the number of generated samples using the given image.
        save_to_dir: A string representing the directory in which the generated images will be saved.
    """
    data_gen = ImageDataGenerator(rotation_range=10,
                                  width_shift_range=0.1,
                                  height_shift_range=0.1,
                                  shear_range=0.1,
                                  brightness_range=(0.3, 1.0),
                                  horizontal_flip=True,
                                  vertical_flip=True,
                                  fill_mode='nearest'
                                 )


    for filename in listdir(file_dir):
        # load the image
        image = cv2.imread(file_dir + '\\' + filename)
        # reshape the image
        image = image.reshape((1,)+image.shape)
        # prefix of the names for the generated sampels.
        save_prefix = 'aug_' + filename[:-4]
        # generate 'n_generated_samples' sample images
        i=0
        for batch in data_gen.flow(x=image, batch_size=1, save_to_dir=save_to_dir,
                                   save_prefix=save_prefix, save_format='jpg'):
            i += 1
            if i > n_generated_samples:
                break
```

1. The function takes three arguments:
    - file_dir: The directory path where the original images are located.
    - n_generated_samples: The number of augmented samples to generate for each original image.
    - save_to_dir: The directory path where the augmented images will be saved.
2. The function initializes an instance of the ImageDataGenerator class from the Keras library. This class provides various image transformation methods for data augmentation.
3. The ImageDataGenerator object is configured with several transformation parameters, including rotation range, width and height shift range, shear range, brightness range, and flipping options. These parameters control the types and degrees of transformations applied to the images during augmentation.

4. The function then iterates through the files in the file_dir directory using the listdir function. For each file, it performs the following steps:

    - Loads the image using the OpenCV library (cv2.imread).
    - Reshapes the image to add an extra dimension to match the input shape expected by the data_gen.flow method.
    - Sets the save prefix for the generated samples based on the original filename.
    - Utilizes a loop that generates n_generated_samples augmented images by calling the data_gen.flow method. The augmented images are saved in the save_to_dir directory with the save prefix and in JPEG format.
    - The loop breaks once the desired number of generated samples has been reached.

```
augmented_data_path = 'augmented data/'

# augment data for the examples with label equal to 'yes' representing tumurous examples
augment_data(file_dir=yes_path, n_generated_samples=6, save_to_dir=augmented_data_path+'yes')
# augment data for the examples with label equal to 'no' representing non-tumurous examples
augment_data(file_dir=no_path, n_generated_samples=9, save_to_dir=augmented_data_path+'no')
```

5. providing the directory paths for the original images (yes_path and no_path) and specifying the number of generated samples to create for each class. The augmented images are saved in the augmented_data_path directory.
6. Remember that 61% of the data (155 images) are tumorous. And, 39% of the data (98 images) are non-tumorous. So, in order to balance the data we can generate 9 new images for every image that belongs to 'no' class and 6 images for every image that belongs the 'yes' class.

Result of the Data Augmentation

We made a function to calculate the percentage of the positive percentage and the negative images

```
# Let's see how many tumorous and non-tumorous examples after performing data augmentation:
def data_summary(main_path):

    yes_path = main_path+'yes'
    no_path = main_path+'no'

    # number of files (images) that are in the the folder named 'yes' that represent tumorous (positive) examples
    m_pos = len(listdir(yes_path))
    # number of files (images) that are in the the folder named 'no' that represent non-tumorous (negative) examples
    m_neg = len(listdir(no_path))
    # number of all examples
    m = (m_pos+m_neg)

    pos_prec = (m_pos* 100.0)/ m
    neg_prec = (m_neg* 100.0)/ m

    print(f"Number of examples: {m}")
    print(f"Percentage of positive examples: {pos_prec}%, number of pos examples: {m_pos}")
    print(f"Percentage of negative examples: {neg_prec}%, number of neg examples: {m_neg}")
data_summary(augmented_data_path)
```

```
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
```

The main code for Brain Tumor Detection

Importing libraries

```
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

First of all we need to crop the part that contains only the brain since any processing on black pixels that have nothing to do with the brain will only result in bias and low accuracy

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.

The next code utilizes image processing techniques to isolate the brain region from the background and remove unnecessary information.

Also the visualization feature of the function, which allows for a clear comparison between the original image and the cropped image. This can aid in understanding the effectiveness of the cropping process and visually showcase the region of interest.

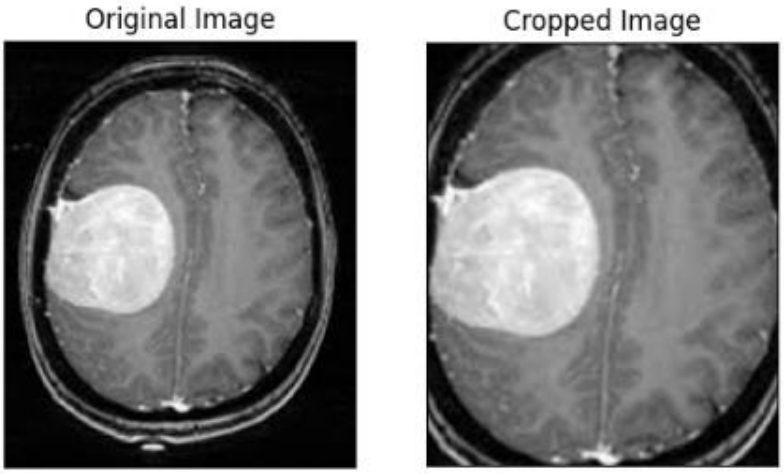
```python
def crop_brain_contour(image, plot=False):

    # 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
```

1. The function crop_brain_contour takes two arguments:
    - image: The input image in BGR format.
    - plot (optional): A boolean flag indicating whether to display a plot of the original and cropped images. By default, it is set to False.
2. The code begins by converting the input image to grayscale using cv2.cvtColor and applying a Gaussian blur using cv2.GaussianBlur. This helps reduce noise and smoothens the image.
3. A thresholding operation is performed on the grayscale image using cv2.threshold. This converts the image into a binary format, where pixel values above a certain threshold are set to white, and pixel values below the threshold are set to black.
4. A series of morphological operations, specifically erosion and dilation, are applied to remove any small regions of noise. These operations help to refine the binary image and improve the quality of the brain contour.
5. Using cv2.findContours, the function identifies the contours in the thresholded image. The largest contour is selected as the brain contour using max(cnts, key=cv2.contourArea).
6. The extreme points of the brain contour, i.e., the leftmost, rightmost, topmost, and bottommost points, are found using NumPy indexing and stored in extLeft, extRight, extTop, and extBot variables, respectively.
7. A new image is cropped out of the original image using the four extreme points. This is achieved by indexing the original image with the appropriate coordinates: image[extTop[1]:extBot[1], extLeft[0]:extRight[0]].
8. If the plot flag is set to True, the function displays a plot with two subplots:
    - The first subplot shows the original image using plt.imshow.
    - The second subplot shows the cropped image using plt.imshow.
    - Various formatting options are applied to remove axes labels and ticks.

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:

```
ex_img = cv2.imread('yes/Y1.jpg')
ex_new_img = crop_brain_contour(ex_img, True)
```

✓ 0.3s



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

```
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

```
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))
```

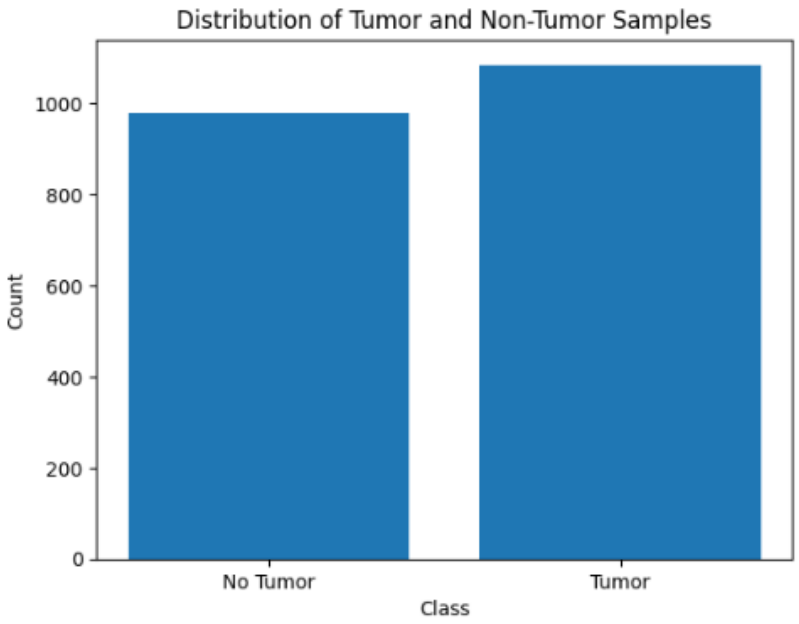
As we see, we have 2065 images. Each images has a shape of **(240, 240, 3)=(image_width, image_height, number_of_channels)**

Plot sample images:

1. Distribution of Tumor and Non-Tumor Samples

```
# Distribution of Tumor and Non-Tumor Samples
def plot_class_distribution(X, y):
    unique, counts = np.unique(y, return_counts=True)
    labels = ['No Tumor', 'Tumor']

    plt.bar(labels, counts)
    plt.xlabel('Class')
    plt.ylabel('Count')
    plt.title('Distribution of Tumor and Non-Tumor Samples')
    plt.show()


# Call the plot_class_distribution function with your loaded data (X and y)
plot_class_distribution(X, Y)
```

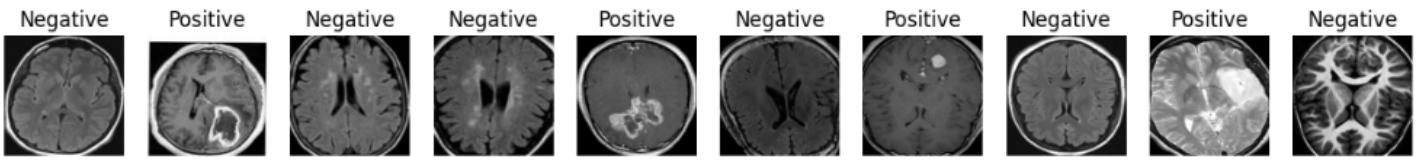
✓ 0.1s



2. showing some negative to positive images

```
def visualize_data_samples(X, Y, num_samples=10):

    classes = ['Negative', 'Positive']  # Assuming 0 represents negative and 1 represents positive

    fig, axes = plt.subplots(1, num_samples, figsize=(15, 3))

    for i in range(num_samples):
        # Retrieve image and label
        image = X[i]
        label = Y[i]

        # Plot the image
        axes[i].imshow(image)
        axes[i].axis('off')
        axes[i].set_title(classes[label[0]])

    plt.show()


# Call the visualize_data_samples function with your loaded data
visualize_data_samples(X, Y)
```

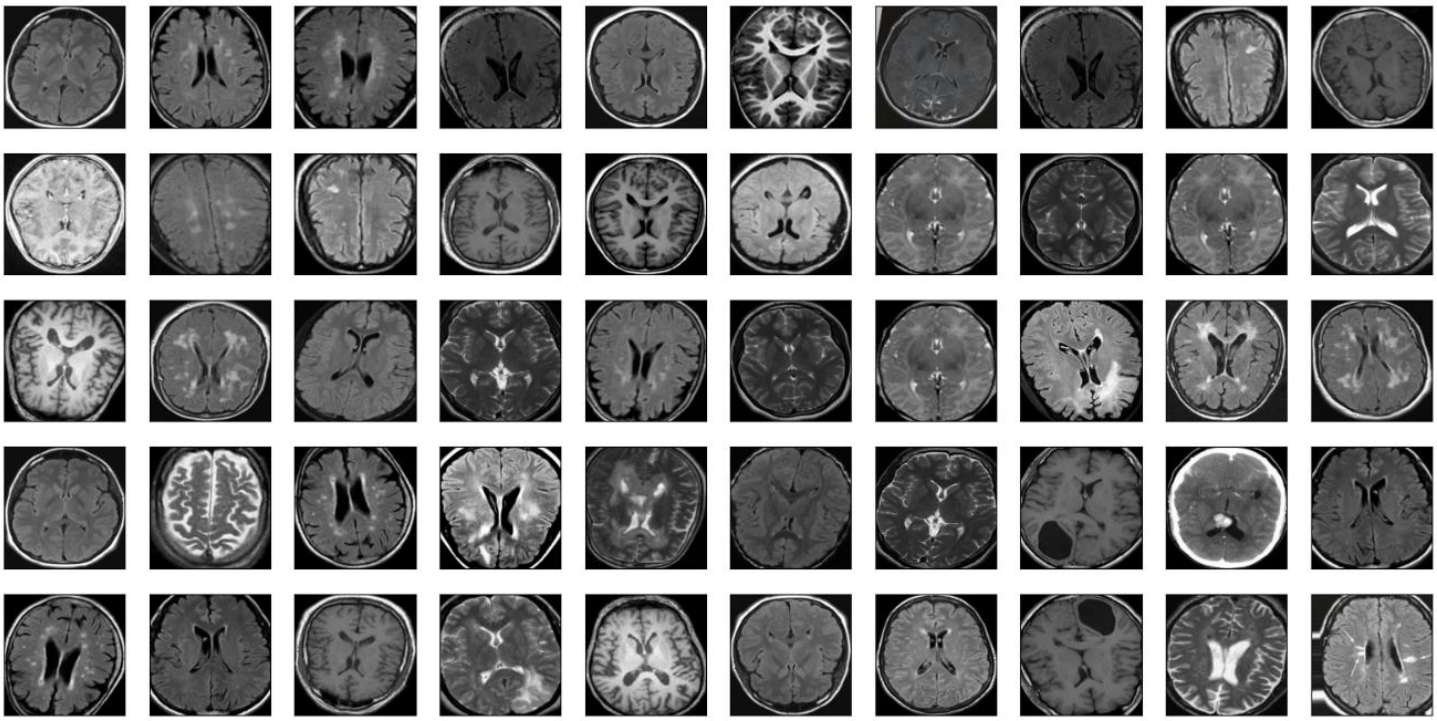
✓ 0.5s



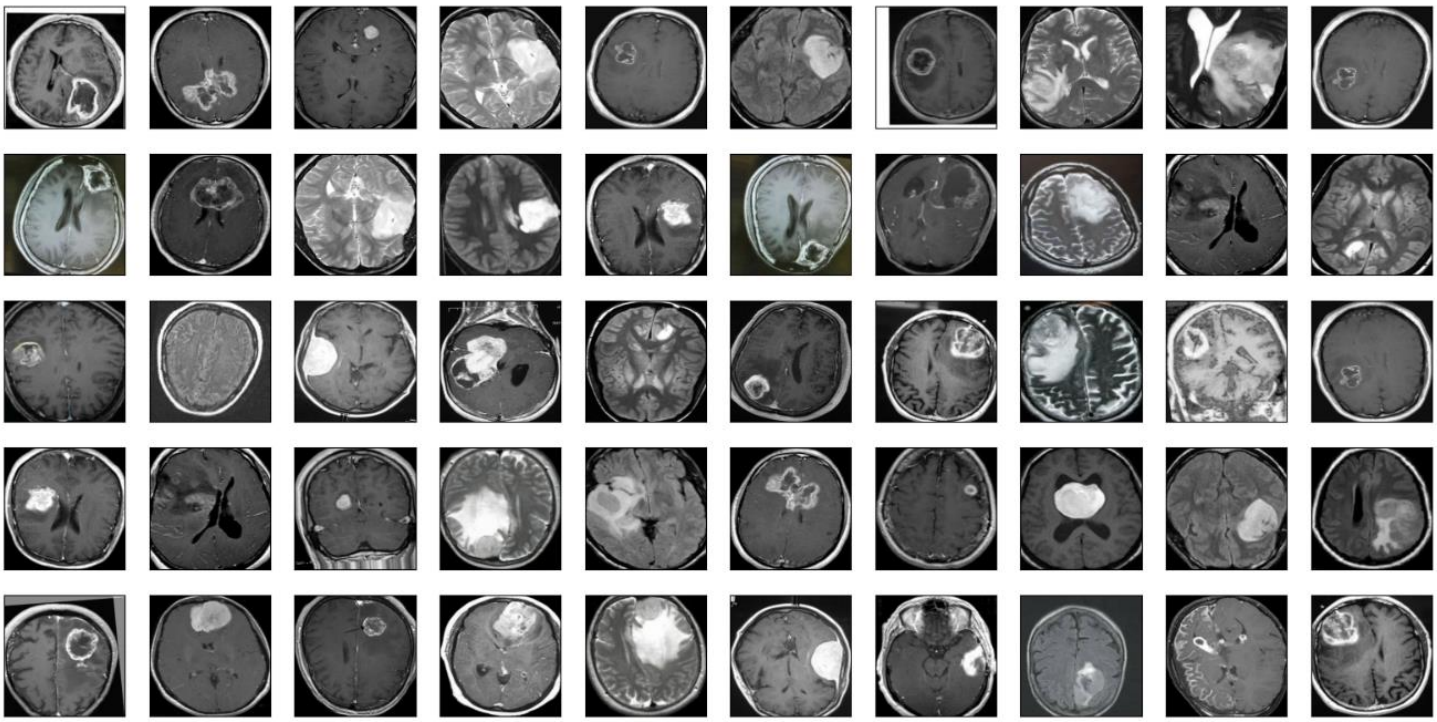
3. deeper look into the data

```
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()
plot_sample_images(X, Y)
```

Brain Tumor: No



Brain Tumor: Yes

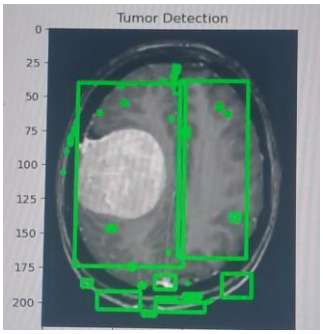


Trying visually detected the brain tumor

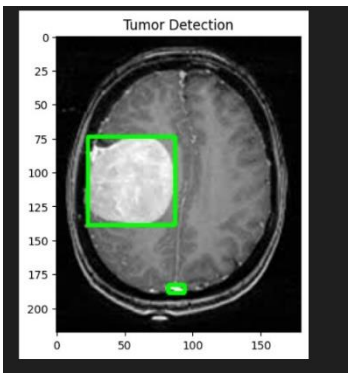
```python
def detect_tumor(image, plot=False):
    # Check if the image is already in grayscale
    if len(image.shape) == 2:
        gray = image
    else:
        # Convert the image to grayscale
        gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

    # Apply Gaussian blur
    blurred = cv2.GaussianBlur(gray, (5, 5), 0)

    # Threshold the image to create a binary mask
    _, thresh = cv2.threshold(blurred, 190, 255, cv2.THRESH_BINARY)

    # Find contours in the binary mask
    contours, _ = cv2.findContours(thresh, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

    # Draw bounding boxes around contours
    image_with_boxes = image.copy()
    for contour in contours:
        x, y, w, h = cv2.boundingRect(contour)
        cv2.rectangle(image_with_boxes, (x, y), (x + w, y + h), (0, 255, 0), 2)

    if plot:
        # Display the result
        plt.imshow(cv2.cvtColor(image_with_boxes, cv2.COLOR_BGR2RGB))
        plt.title("Tumor Detection")
        plt.show()

    return image_with_boxes
```

1. The function detect_tumor takes two arguments:
    - image: The input image, which can be in either grayscale or BGR format.
    - plot (optional): A boolean flag indicating whether to display a plot of the image with the detected tumor regions. By default, it is set to False.
2. The code first checks if the image is already in grayscale by examining the shape of the image. If the shape has only two dimensions, it is assumed to be grayscale. Otherwise, the image is converted to grayscale using cv2.cvtColor with the conversion code cv2.COLOR_BGR2GRAY.
3. Gaussian blur is applied to the grayscale image using cv2.GaussianBlur to reduce noise and smoothen the image.
4. A binary mask is created by thresholding the blurred image using cv2.threshold. Pixels with intensity values above a certain threshold (190 in this case) are set to white, while pixels below the threshold are set to black.
5. Contours are extracted from the binary mask using cv2.findContours. The contours represent the boundaries of potential tumor regions.
6. Bounding boxes are drawn around each contour using cv2.rectangle. The coordinates of the bounding box are determined using cv2.boundingRect.
7. If the plot flag is set to True, the function displays the image with the detected tumor regions highlighted. The image is converted from BGR to RGB format using cv2.cvtColor, and then displayed using plt.imshow. A title is added to the plot using plt.title.

We tried to do a rectangle shape around the tumor if the image MRI contains a tumor, here is the first result we got using threshold 128,255



As shown the detected areas were not just the tumor position so we changed the threshold and this was the output



Split the data

Using a function called `split_data` that takes input data `X` and labels `Y`.

This function splits them into:

- Training
- Validation
- Test sets
  using the `train_test_split` function.

The used split was:

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

Note:

The choice of how to split your data into training, validation, and test sets depends on various factors, (70% training, 15% validation, 15% testing) is a common and reasonable choice in many cases.

For example, the majority of the data is used for training the model which is important because it allows the model to learn patterns and relationships within the data.

```
def split_data(X,Y, test_size=0.2):
    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. 15% of the data for testing.

```
X_train, y_train, X_val, y_val, X_test, y_test = split_data(X, Y, test_size=0.3)
```

Data information

```
print ("number of training examples = " + str(X_train.shape[0]))
print ("number of Validation 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 Validation 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)
```

Building a convolutional neural network model:

```python
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
```

`build_model` function ➔ defines a simple convolutional neural network (CNN) for brain detection using Keras.

Each input x (image) has a shape of (240, 240, 3) and is fed into the neural network.

It goes through the following layers:

1. A Zero Padding layer with a pool size of (2, 2).
2. A convolutional layer with 32 filters, with a filter size of (7, 7) and a stride equals to 1.
3. A batch normalization layer to normalize pixel values to speed up computation.
4. A ReLU activation layer.
5. A Max Pooling layer with f=4 and s=4.
6. A Max Pooling layer with f=4 and s=4, same as before.
7. A flatten layer in order to flatten the 3-dimensional matrix into a one-dimensional vector.
8. A Dense (output unit) fully connected layer with one neuron with a sigmoid activation (since this is a binary classification task).

This architecture is suitable for binary classification tasks like brain detection.

Model compilation:

It is the step that prepares the model to learn from a given dataset and adjust its internal parameters (weights and biases) to optimize its performance by specifying the optimizer, loss function, and evaluation metrics.

Loss Function: The loss function quantifies the difference between the predicted output of the model and the true output. It measures the model's performance during training

Optimizer: determines how the model's weights are updated during training.

Compile the model:

```
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
```

We used the Adam optimizer, which is a popular optimization algorithm that combines the benefits of the AdaGrad and RMSProp optimizers.

The loss function is set to 'binary_crossentropy', which is commonly used for binary classification problems. It measures the difference between the predicted probabilities and the true labels, with the goal of minimizing this difference during training.

```
# tensorboard
log_file_name = f'brain_tumor_detection_cnn_{int(time.time())}'
tensorboard = TensorBoard(log_dir=f'logs/{log_file_name}')
```

```
# checkpoint
# unique file name that will include the epoch and the validation (development) accuracy
filepath="cnn-parameters-improvement-{epoch:02d}-{val_accuracy:.2f}"
# save the model with the best validation (development) accuracy till now
checkpoint = ModelCheckpoint("models/{}.h5".format(filepath, monitor='val_accuracy', verbose=1, save_best_only=True, mode='max'))
```

`log_file_name`: It is a string variable generated based on the current timestamp, providing a unique name for the TensorBoard log directory.

`TensorBoard`: It is a Keras callback,is used for visualizing metrics such as loss and accuracy during the training process.

The **`log_dir`** parameter specifies the directory where TensorBoard will write its log files.

filepath: It is a string specifying the format for the saved model file names. **{epoch:02d}** is a placeholder for the epoch number

{val_accuracy:.2f} is a placeholder for the validation accuracy formatted as a floating-point number with two decimal places.

ModelCheckpoint: It is a Keras callback that saves the model after every epoch. The **monitor** parameter specifies the metric to monitor for saving the best model, and **mode** determines whether to maximize or minimize the monitored metric.

So ModelCheckpoint is used to save the model's parameters during training. It helps in creating checkpoints at regular intervals and only saves the model if the monitored metric (in this case, validation accuracy) improves.
Used later in loading the best model based on validation performance rather than the last epoch. It is especially useful for preventing overfitting.

Train the model:

```
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)}")
```

```
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)}")
```

```
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)}")
```

```
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)}")
```

Training our neural network model multiple times with different numbers of epochs, with measuring the elapsed time for each training session. This is a common practice for monitoring the time it takes to train a model under different configurations.

Another way we could use:

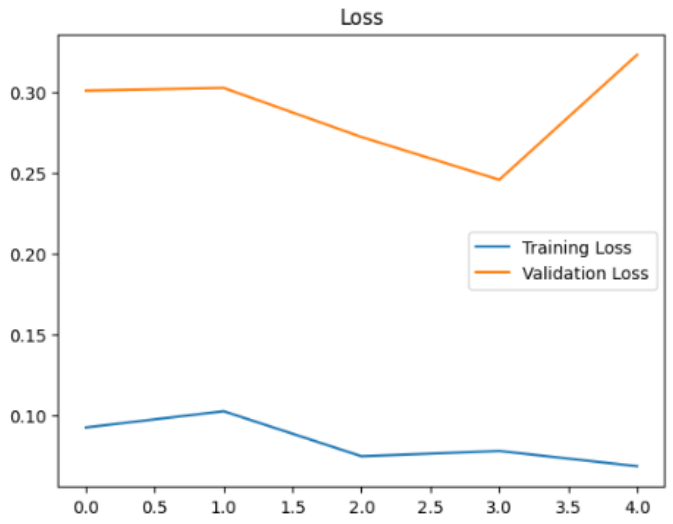
```
def train_model(epochs):
    start_time = time.time()

    model.fit(x=X_train, y=y_train, batch_size=32, epochs=epochs, validation_data=(X_val, y_val), callbacks=[tensorboard, checkpo

    end_time = time.time()
    execution_time = (end_time - start_time)

    print(f"Elapsed time for {epochs} epochs: {hms_string(execution_time)}")

# Train the model with different numbers of epochs
train_model(10)
train_model(3)
train_model(3)
train_model(3)
train_model(5)
```

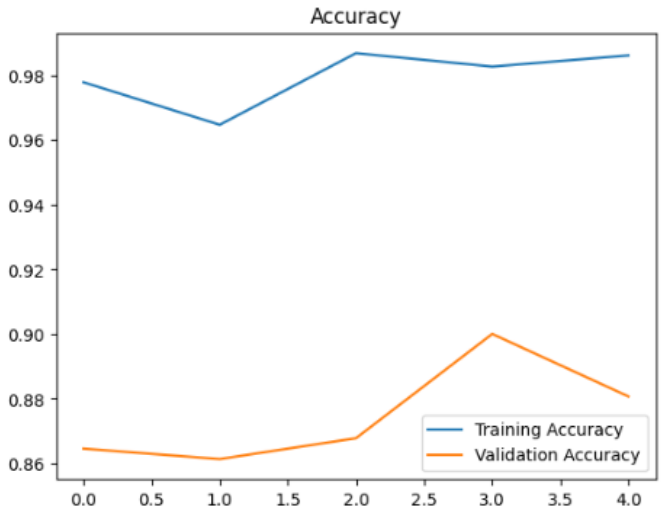
This is a more efficient way to structure the previous code by avoiding the repetitions.

Plot Loss & Accuracy

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.

```
def plot_metrics(history):

    train_loss = history['loss']
    val_loss = history['val_loss']
    train_acc = history['accuracy']
    val_acc = history['val_accuracy']

    # 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()
```





Evaluating 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

```
best_model = load_model(filepath='models/cnn-parameters-improvement-04-0.90.h5')
```

Evaluate the best model on the testing data:

```
loss, acc = best_model.evaluate(x=X_test, y=y_test)
```

```
10/10 [==============================] - 1s 119ms/step - loss: 0.2091 - accuracy: 0.9129
```

Accuracy of the best model on the testing data:

```
# Accuracy of the best model on the testing data:
print (f"Test Loss = {loss}")
print (f"Test Accuracy = {acc}")
```

✓ 0.0s

```
Test Loss = 0.10166341066360474
Test Accuracy = 0.9677419066429138
```

Confusion matric and F1 score

A confusion matrix is typically a square matrix with rows and columns representing the different classes in your classification problem. It is divided into four sections:

True Positives (TP): This represents the cases where the model correctly predicted the positive class (e.g., tumor) when it was actually positive.

True Negatives (TN): This represents the cases where the model correctly predicted the negative class (e.g., non-tumor) when it was actually negative.

False Positives (FP): This represents the cases where the model incorrectly predicted the positive class (e.g., tumor) when it was actually negative.

False Negatives (FN): This represents the cases where the model incorrectly predicted the negative class (e.g., non-tumor) when it was actually positive.

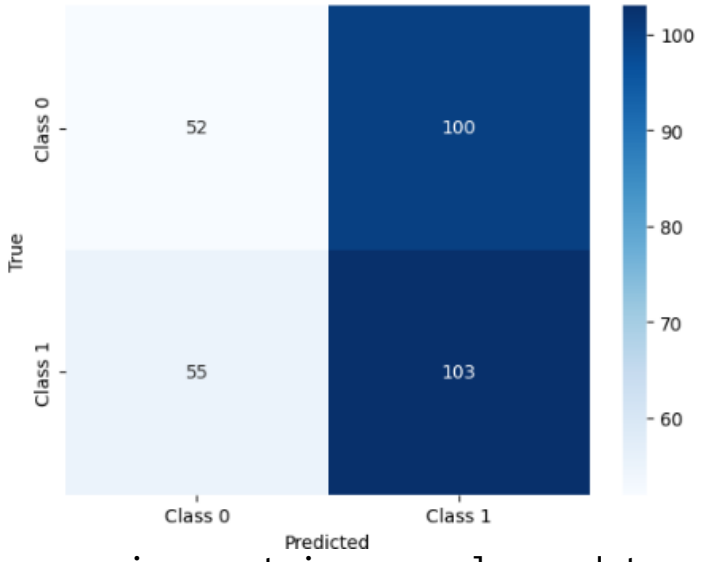
The confusion matrix allows you to assess the model's performance in terms of these four categories. It helps you understand the types of errors the model is making and provides insights into its accuracy, precision, recall, and other evaluation metrics.

confusion matrix

```
from sklearn.metrics import confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sns
```

```
y_test_prob = best_model.predict(X_test)
y_pred = np.where(y_test_prob > 0.5, 1, 0)
cm = confusion_matrix(y_test, y_pred)
# Plot the confusion matrix using seaborn
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=['Class 0', 'Class 1'], yticklabels=['Class 0', 'Class 1'])
plt.xlabel('Predicted')
plt.ylabel('True')
plt.show()
```

```
10/10 [==============================] - 1s 108ms/step
```



F1 score is a metric commonly used to evaluate the performance of a classification model, particularly in imbalanced datasets where the distribution of classes is uneven. It combines precision and recall into a single value that represents the model's overall accuracy.

The F1 score is calculated using the following formula:

F1 score = 2 * (precision * recall) / (precision + recall)

Precision is the ratio of true positives (TP) to the sum of true positives and false positives (FP). It measures the proportion of correctly predicted positive instances among all instances predicted as positive. Precision indicates how well the model predicts positive instances accurately.
Recall, also known as sensitivity or true positive rate, is the ratio of true positives (TP) to the sum of true positives and false negatives (FN). It measures the proportion of correctly predicted positive instances among all actual positive instances. Recall indicates how well the model captures positive instances.

The F1 score combines precision and recall by taking their harmonic mean. It provides a balanced measure of the model's ability to identify both positive and negative instances correctly. The harmonic mean emphasizes lower values, so the
F1 score is lower when either precision or recall is low.

The F1 score ranges from 0 to 1, with 1 representing the best possible value (perfect precision and recall), and 0 representing the worst value (poor precision or recall).

F1 score for the best model on the testing data:

```
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
```

```
y_test_prob = best_model.predict(X_test)
```

```
10/10 [==============================] - 1s 108ms/step
```

```
f1score = compute_f1_score(y_test, y_test_prob)
print(f"F1 score: {f1score}")
```

```
F1 score: 0.5706371191135734
```

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

```
y_val_prob = best_model.predict(X_val)
```

```
10/10 [==============================] - 1s 104ms/step
```

```
f1score_val = compute_f1_score(y_val, y_val_prob)
print(f"F1 score: {f1score_val}")
```

```
F1 score: 0.5653333333333334
```

Results Interpretation

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

```
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}")
```

✓ 0.0s

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

```
print("Training Data:")
data_percentage(y_train)
print("Validation Data:")
data_percentage(y_val)
print("Testing Data:")
data_percentage(y_test)
```

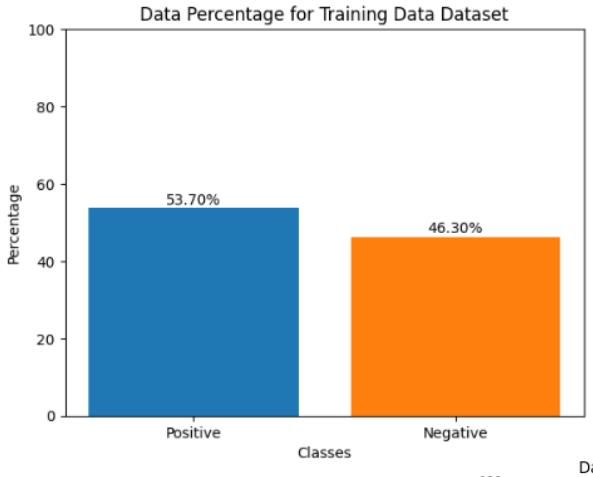
```
Training Data:
Number of examples: 1445
Percentage of positive examples: 53.70242214532872%, number of pos examples: 776
Percentage of negative examples: 46.29757785467128%, number of neg examples: 669
Validation Data:
Number of examples: 310
Percentage of positive examples: 48.70967741935484%, number of pos examples: 151
Percentage of negative examples: 51.29032258064516%, number of neg examples: 159
Testing Data:
Number of examples: 310
Percentage of positive examples: 50.96774193548387%, number of pos examples: 158
Percentage of negative examples: 49.03225806451613%, number of neg examples: 152
```

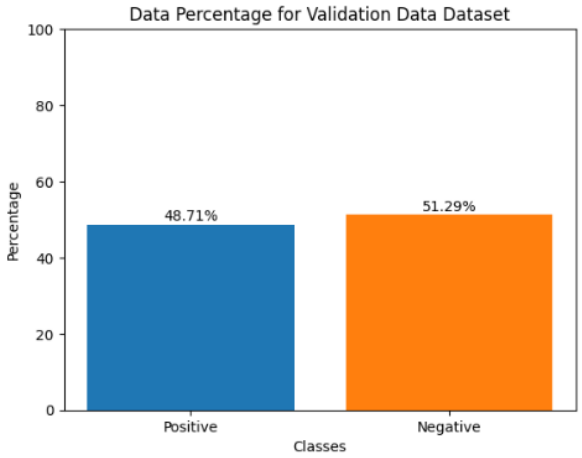
As expectred, the percentage of positive examples are around 50%.

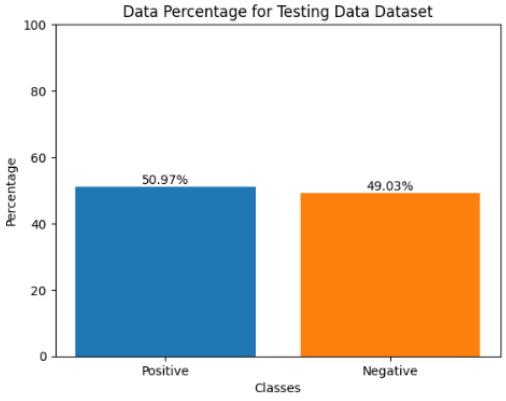
better way to visualize percentage of positive and negative examples

```python
def visualize_data_percentage(y, dataset_name):
    m=len(y)
    n_positive = np.sum(y)
    n_negative = m - n_positive

    positive_prec = (n_positive* 100.0)/ m
    negative_prec = (n_negative* 100.0)/ m
    labels = ['Positive', 'Negative']
    percentages = [positive_prec, negative_prec]
    colors = ['#1f77b4', '#ff7f0e']

    plt.bar(labels, percentages, color=colors)
    plt.title(f'Data Percentage for {dataset_name} Dataset')
    plt.xlabel('Classes')
    plt.ylabel('Percentage')
    plt.ylim([0, 100])

    for i, val in enumerate(percentages):
        plt.text(i, val + 1, f'{val:.2f}%', ha='center')

    plt.show()
```

```python
# Call the visualize_data_percentage function for each dataset
visualize_data_percentage(y_train, 'Training Data')
visualize_data_percentage(y_val, 'Validation Data')
visualize_data_percentage(y_test, 'Testing Data')
```







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