Predicting Brain Tumors from MRI Scans

The complete model can be summarized in following steps :

1) **Importing Libraries** : Libraries like imutils , numpy , pandas matplotlib ,tensorflow, train_test_split ,model from tensorflow.keras.model .

2) **Data loading** : Images are loaded in image_dir .

3) **Augmentation of data** : This technique is very helpful especially when the size of datasets is small . I have applied classical augmentation to generate new images from the existing ones with some varieties .

```
def augment_data(file_dir, n_generated_samples, save_to_dir):
    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'
                                 )
```

4) **Making directories for augmented images :** Separate directories are made for storing augmented images with 'yes' and 'no' labels .

```
os.makedirs('/kaggle/working/augmented-images')
os.makedirs('/kaggle/working/augmented-images/yes')
os.makedirs('/kaggle/working/augmented-images/no')
```

5) **Brain Contour Cropping :** This method is helpful for varieties of tasks like Eliminating Unnecessary Background , enhancing model performance by improving computational efficiency .

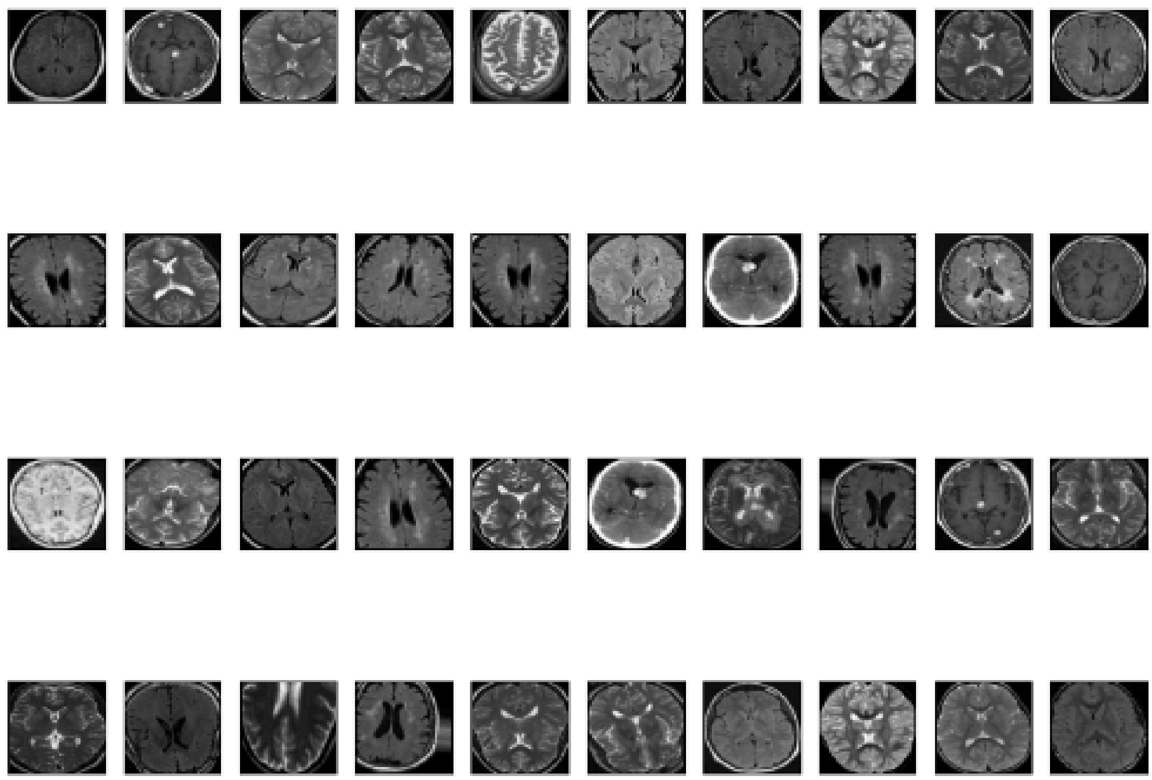
```python
def crop_brain_contour(image, plot=False):

    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    gray = cv2.GaussianBlur(gray, (5, 5), 0)

    thresh = cv2.threshold(gray, 45, 255, cv2.THRESH_BINARY)[1]
    thresh = cv2.erode(thresh, None, iterations=2)
    thresh = cv2.dilate(thresh, None, iterations=2)

    cnts = cv2.findContours(thresh.copy(), cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
    cnts = imutils.grab_contours(cnts)
    c = max(cnts, key=cv2.contourArea)

    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])

    new_image = image[extTop[1]:extBot[1], extLeft[0]:extRight[0]]
```

6) **Loading and Processing Data :** This function loads and preprocesses the images, resizing them and normalizing the pixel values.

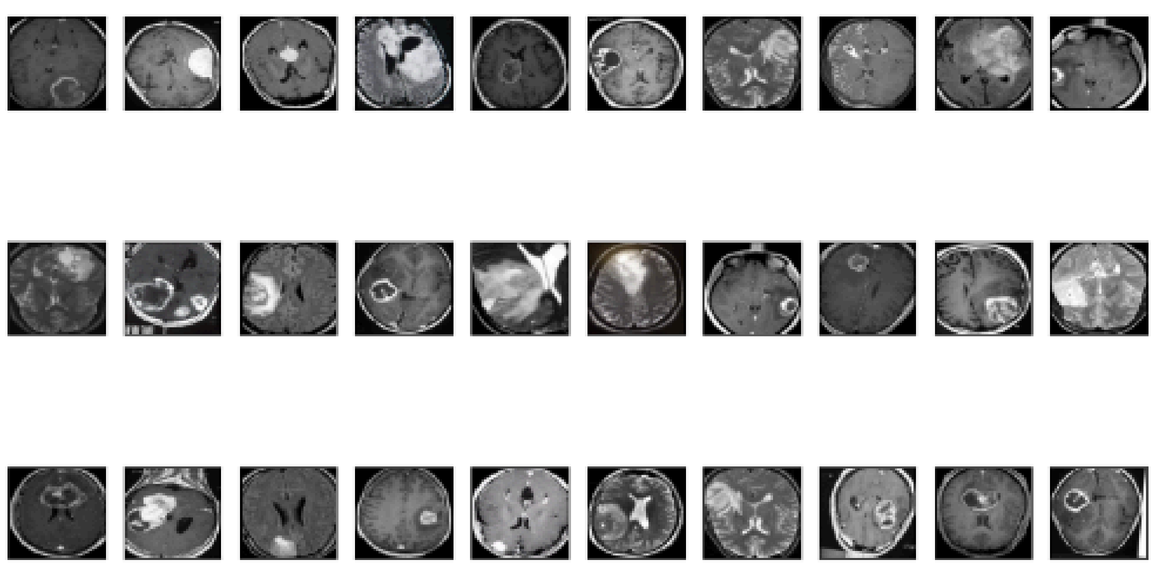
```python
def load_data(dir_list, image_size):

    X = []
    y = []
    image_width, image_height = image_size

    for directory in dir_list:
        for filename in listdir(directory):
            image = cv2.imread(directory+'/'+filename)
            image = crop_brain_contour(image, plot=False)
            image = cv2.resize(image, dsize=(image_width, image_height), interpolation=cv2.INTER_CUBIC)
            image = image / 255.
            X.append(image)
            if directory[-3:] == 'yes':
                y.append([1])
            else:
                y.append([0])
```

7) **Plotting of images** for better visualization .

Brain Tumor: No



Brain Tumor: Yes



8) **Train_test_split :** The complete datasets are divided into X_train, y_train, X_val, y_val, X_test, y_test .

```
def split_data(X, y, test_size=0.2 ,  random_state=42):

    X_train, X_test_val, y_train, y_test_val = train_test_split(X, y, test_size=test_size , random_state=ra
    X_test, X_val, y_test, y_val = train_test_split(X_test_val, y_test_val, test_size=0.5 , random_state=ra

    return X_train, y_train, X_val, y_val, X_test, y_test
```

9) **Building the CNN Model :** The building block of CNN is neurons . These neurons form input layers , hidden layers and output layers . The neurons in the layers consist of activation functions like Relu , tanh , sigmoid , softmax etc. These activation functions bring non-linearity .
Some of the key terms in CNN are :

- **Filter/Kernel**: A small matrix used to detect specific features in the input image.
- **Stride**: The step size with which the filter moves across the image.
- **Padding**: Adding extra pixels around the border of the input image to control the output size.
- **Pooling:** Reduces the spatial dimensions (height and width) of the feature maps while retaining the important information.
- **Dropout :** A regularization technique where a fraction of the neurons is randomly set to zero during training to prevent overfitting.
- **Batch Normalization :** Normalizes the output of a previous activation layer by subtracting the batch mean and dividing by the batch standard deviation. It stabilizes and speeds up training.

- **Loss Function :**

  Binary Cross-Entropy: Used for binary classification tasks.

  Categorical Cross-Entropy: Used for multi-class classification task

- **Backpropagation :** A method used in artificial neural networks to calculate the gradient of the loss function with respect to the network's weights.

```python
def build_model(input_shape):
    X_input = Input(input_shape)
    X = ZeroPadding2D((2, 2))(X_input)

    X = Conv2D(64, (3, 3), strides=(1, 1), padding='same')(X)
    X = BatchNormalization(axis=3)(X)
    X = Activation('relu')(X)
    X = MaxPooling2D((2, 2))(X)

    X = Conv2D(128, (3, 3), strides=(1, 1), padding='same')(X)
    X = BatchNormalization(axis=3)(X)
    X = Activation('relu')(X)
    X = MaxPooling2D((2, 2))(X)

    X = Conv2D(256, (3, 3), strides=(1, 1), padding='same')(X)
    X = BatchNormalization(axis=3)(X)
    X = Activation('relu')(X)
    X = MaxPooling2D((2, 2))(X)

    X = Flatten()(X)
    X = Dense(256, activation='relu')(X)
    X = Dropout(0.5)(X)
    X = Dense(1, activation='sigmoid')(X)
    model = Model(inputs=X_input, outputs=X)

    return model
```

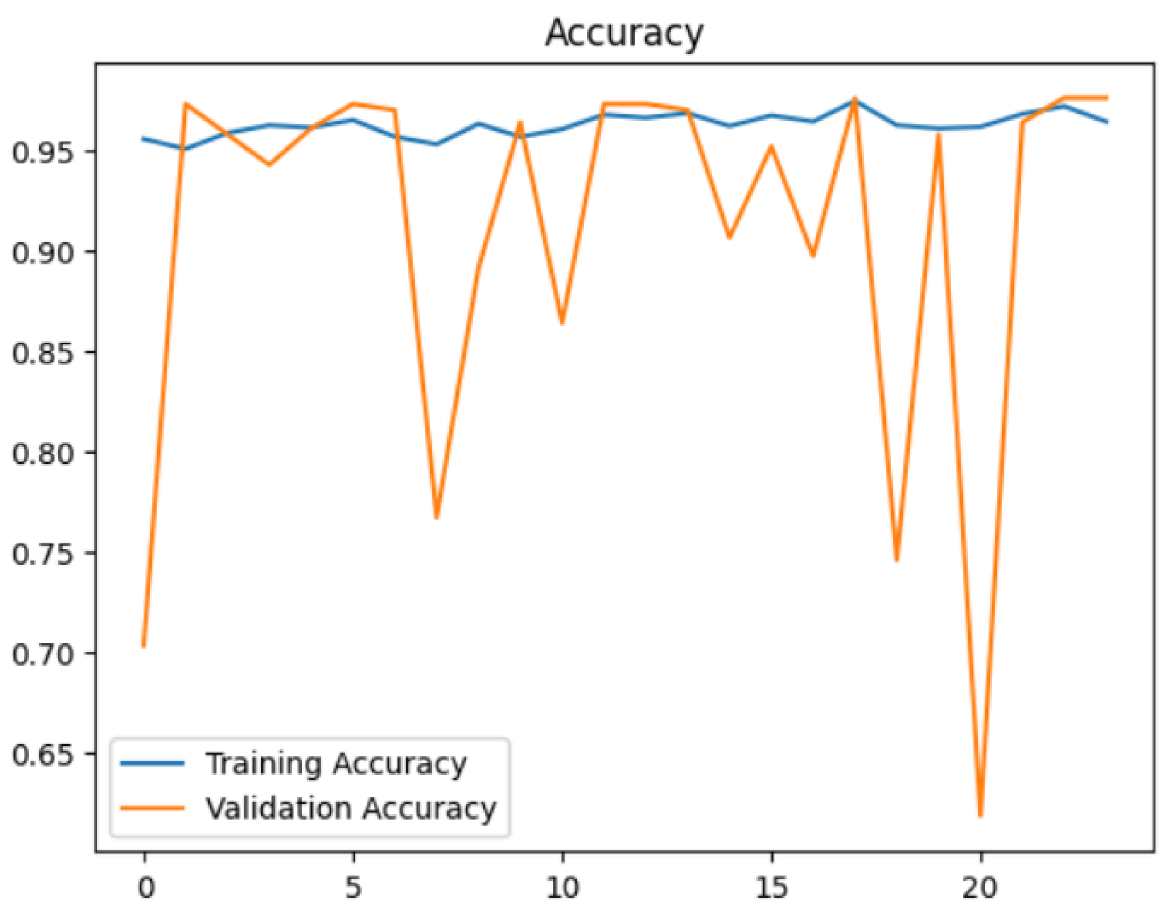
10) **Optimizer** : Algorithms or methods used to change the attributes of the neural network, such as weights and learning rate, to reduce the losses. Common optimizers include:

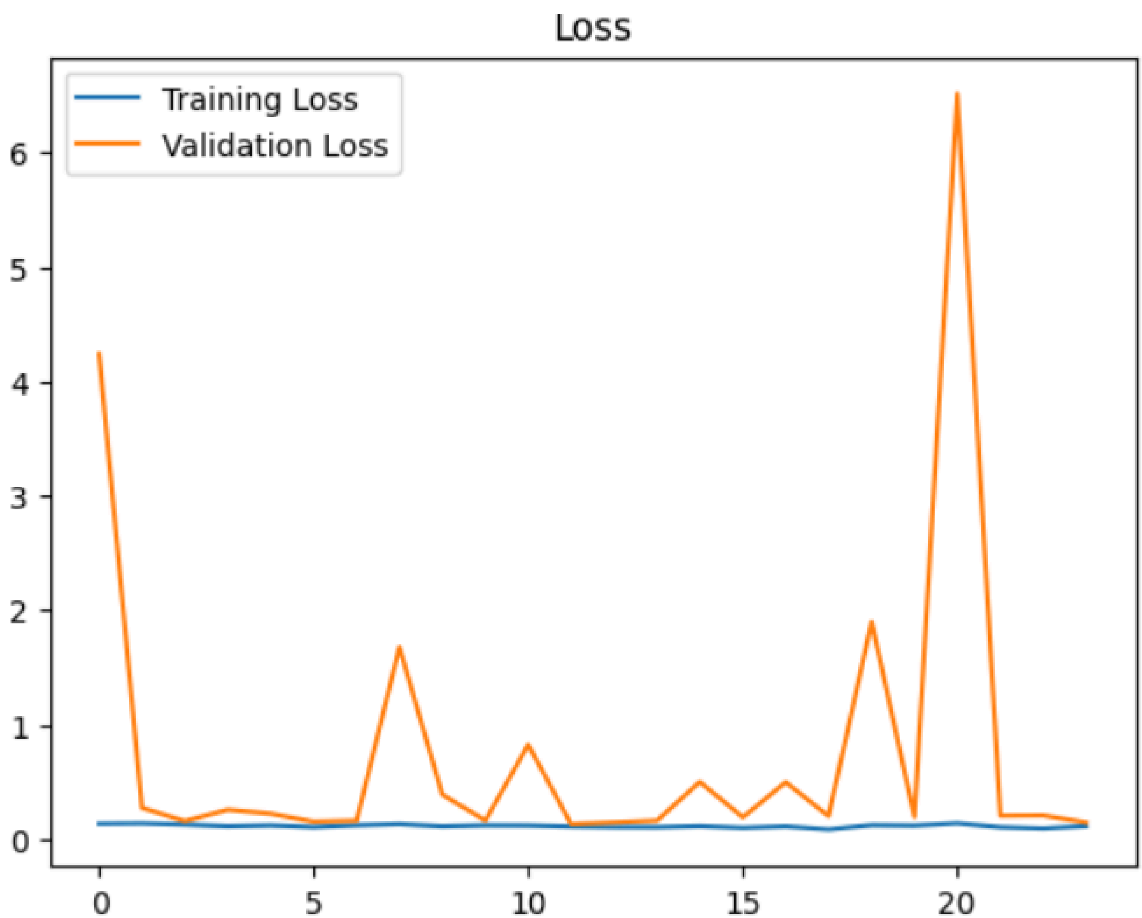
- **SGD (Stochastic Gradient Descent)**
- **Adam (Adaptive Moment Estimation)**

```
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
model.fit(x=X_train, y=y_train, batch_size=32, epochs=24, validation_data=(X_val, y_val))
```

```
Epoch 1/24
83/83 ━━━━━━━━━━━━━━━━━━━━ 18s 149ms/step - accuracy: 0.9592 - loss: 0.1225 - val_accuracy: 0.7030 -
Epoch 2/24
83/83 ━━━━━━━━━━━━━━━━━━━━ 8s 93ms/step - accuracy: 0.9514 - loss: 0.1279 - val_accuracy: 0.9727 - v
Epoch 3/24
83/83 ━━━━━━━━━━━━━━━━━━━━ 8s 93ms/step - accuracy: 0.9522 - loss: 0.1377 - val_accuracy: 0.9576 - v
```

11) Visualization






```
83/83 ━━━━━━━━━━━━━━━━━━━━ 1s 18ms/step
11/11 ━━━━━━━━━━━━━━━━━━━━ 0s 18ms/step
11/11 ━━━━━━━━━━━━━━━━━━━━ 0s 18ms/step
F1 Score: 97.00%
Recall Score: 98.00%
Accuracy: 97.00%
Precision score: 95.72%
```

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

$$\text{Precision} = \frac{TP}{TP + FP}$$

$$F1 = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

$$\text{Recall} = \frac{TP}{TP + FN}$$