

# NeuroScan - Detecting Brain Tumors with CNN-powered MRI Analysis



## About the Dataset:

The dataset consists of 253 files containing Brain MRI (Magnetic Resonance Imaging) scans. MRI is a non-invasive medical imaging technique that provides detailed images of the brain's internal structures. These images are captured using strong magnetic fields and radio waves, allowing for the visualization of various tissues within the brain.

The dataset is labeled with two distinct classes:

- **No Tumor:** This class represents brain MRI scans from individuals who do not have any detectable brain tumors. These scans serve as the baseline for comparison against scans with tumors.
- **Tumor:** This class includes brain MRI scans from individuals who have been diagnosed with brain tumors. Tumors in the brain can vary in size, location, and type, which makes their accurate detection and classification a challenging task.

Each MRI scan file in the dataset likely contains a 3D array of pixel intensity values that represent different tissue types and structures within the brain. These arrays can be used as input data for various machine learning algorithms, such as Convolutional Neural Networks (CNNs), to learn patterns associated with the presence or absence of brain tumors.

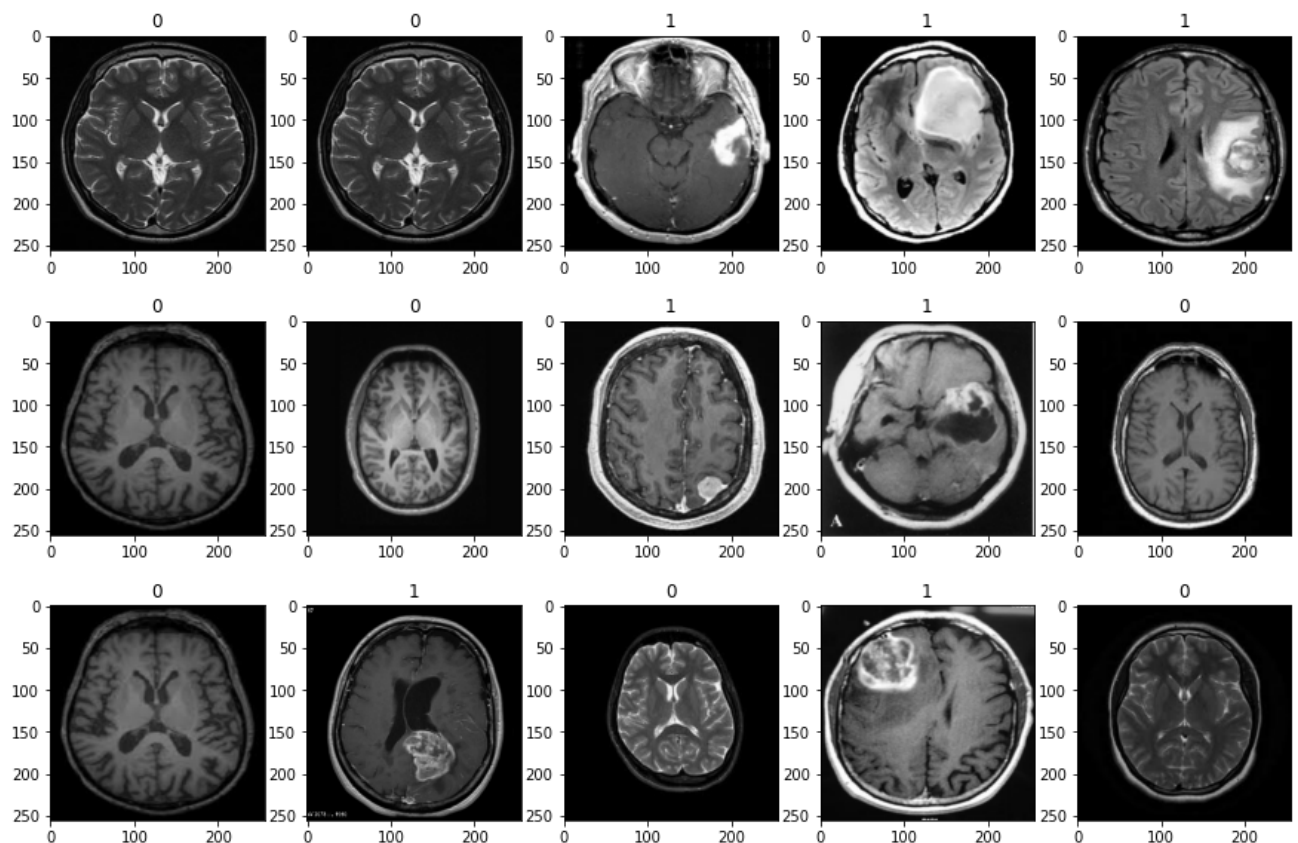
## Objective:

The primary objective of this project is to develop a machine learning model, specifically a Convolutional Neural Network (CNN) using the Keras framework, to accurately detect the presence of brain tumors in MRI scans. The CNN architecture is chosen due to its effectiveness in processing image data and capturing intricate patterns. The successful implementation of this project can contribute to medical diagnosis and treatment planning for patients with brain tumors.

## Importing Essential Libraries and Tools



```
In [318]: fig, ax = plt.subplots(3, 5, figsize=(15,10))
ax = ax.flatten()
for idx, img in enumerate(batch[0][:15]):
    ax[idx].imshow(img.astype(int))
    ax[idx].title.set_text(batch[1][idx])
```



## Data Scaling

Since our data consists of images and images consist of pixels, we divide all the pixel values by 255—each pixel can have a value in  $[0, 255]$ —so that all the pixel values are on the same scale i.e.  $[0, 1]$ .

```
In [319]: data = data.map(lambda x,y: (x/255, y))
```

```
In [320]: batch = data.as_numpy_iterator().next()
```

```
Cleanup called...
Cleanup called...
Cleanup called...
Cleanup called...
Cleanup called...
Cleanup called...
Cleanup called...
Cleanup called...
```

```
In [321]: print("Minimum value of the scaled data:", batch[0].min())
print("Maximum value of the scaled data:", batch[0].max())
```

```
Minimum value of the scaled data: 0.0
Maximum value of the scaled data: 1.0
```

# Train-Validation-Test Split

```
In [322]: print("There are", len(data), "batches in our data")
```

There are 8 batches in our data

Now, we have to divide the whole data into 3 separate sets: Train set for training the model, Validation set for adjusting the hyperparameters of our model and Test set for evaluating our model on the set that our model has not seen before. As it can be seen, we have 8 batches in our data. I preferred allocating 4 batches for Train set, 2 batches for Validation set and 2 batches for Test set.

```
In [323]: train_size = int(len(data)*0.6)
val_size = int(len(data)*0.2)+1
test_size = int(len(data)*0.2)+1
```

```
In [324]: print("Train Size:", train_size)
print("Validation Size:", val_size)
print("Test Size:", test_size)

print("Sum of Train, Validation and Test sizes is equal to:", train_size + val_size +
```

Test Size: 2

Sum of Train, Validation and Test sizes is equal to: 8

Train Size: 4

Validation Size: 2

Test Size: 2

```
In [325]: train = data.take(train_size)
val = data.skip(train_size).take(val_size)
test = data.skip(train_size + val_size).take(test_size)
```

## Data Augmentation

Because our Train set has relatively small number of images, we can apply data augmentation which is reproducing the images by applying some changes such as random rotating, random flipping, random zoom and random contrast. This may possibly increase the accuracy score of the model. Since we will be applying the data augmentation in the beginning of the neural network architecture, we should pass the input shape.

Note: Data augmentation will be inactive when testing the data. Input images will be augmented during calls to `model.fit` (not `model.evaluate` or `model.predict`). If you want to learn more about data augmentation in Tensorflow, you can check the official documentation.

```
In [326]: batch = data.as_numpy_iterator().next()
```

Cleanup called...

Cleanup called...

Cleanup called...

Cleanup called...

Cleanup called...

Cleanup called...

Cleanup called...

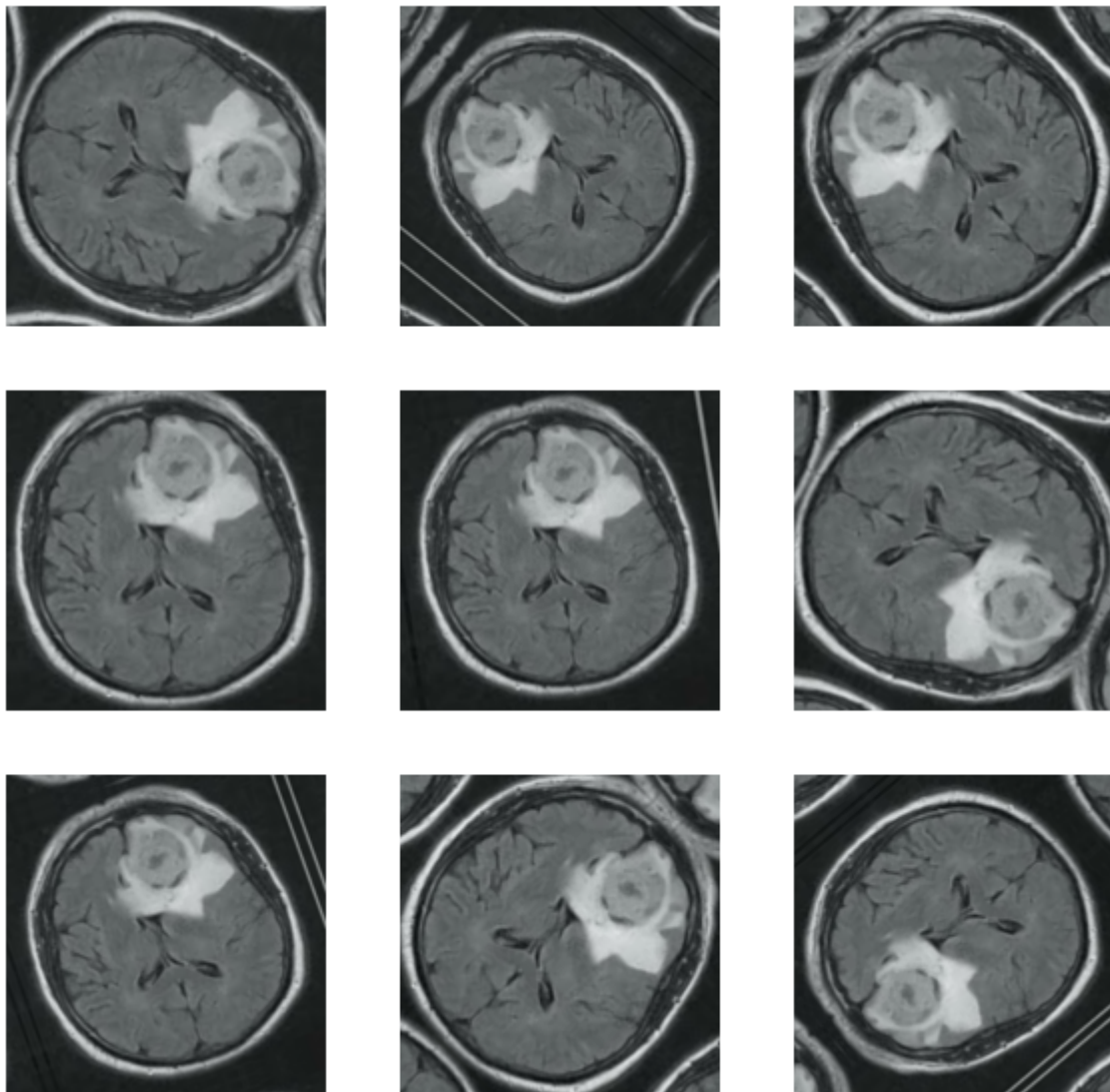
Cleanup called...

Cleanup called...

```
In [327]: data_augmentation = Sequential([
    layers.RandomFlip("horizontal_and_vertical", input_shape=(256,256,3)),
    layers.RandomZoom(0.1),
    layers.RandomContrast(0.1),
    layers.RandomRotation(0.2)
])

image = batch[0]

plt.figure(figsize=(10, 10))
for i in range(9):
    augmented_image = data_augmentation(image)
    ax = plt.subplot(3, 3, i + 1)
    plt.imshow(augmented_image[0])
    plt.axis("off")
```



## Building Deep Learning Model

```
In [328]: model = Sequential([
    data_augmentation,
    Conv2D(16, (3,3), 1, activation="relu", padding="same"),
    Conv2D(16, (3,3), 1, activation="relu", padding="same"),
    MaxPooling2D(),
    Conv2D(32, (5,5), 1, activation="relu", padding="same"),
    Conv2D(32, (5,5), 1, activation="relu", padding="same"),
    MaxPooling2D(),
    Conv2D(16, (3,3), 1, activation="relu", padding="same"),
    Conv2D(16, (3,3), 1, activation="relu", padding="same"),
    MaxPooling2D(),

    Flatten(),
    Dense(128, activation="relu"),
    Dense(1, activation="sigmoid")
])
```

```
In [329]: model.compile(optimizer="adam", loss=keras.losses.BinaryCrossentropy(), metrics=["acc
```

```
In [330]: model.summary()
```

Model: "sequential\_1"

Layer (type)	Output Shape	Param #
=====		
sequential (Sequential)	(None, 256, 256, 3)	0
<hr/>		
conv2d (Conv2D)	(None, 256, 256, 16)	448
<hr/>		
conv2d_1 (Conv2D)	(None, 256, 256, 16)	2320
<hr/>		
max_pooling2d (MaxPooling2D)	(None, 128, 128, 16)	0
<hr/>		
conv2d_2 (Conv2D)	(None, 128, 128, 32)	12832
<hr/>		
conv2d_3 (Conv2D)	(None, 128, 128, 32)	25632
<hr/>		
max_pooling2d_1 (MaxPooling2D)	(None, 64, 64, 32)	0
<hr/>		
conv2d_4 (Conv2D)	(None, 64, 64, 16)	4624
<hr/>		
conv2d_5 (Conv2D)	(None, 64, 64, 16)	2320
<hr/>		
max_pooling2d_2 (MaxPooling2D)	(None, 32, 32, 16)	0
<hr/>		
flatten (Flatten)	(None, 16384)	0
<hr/>		
dense (Dense)	(None, 128)	2097280
<hr/>		
dense_1 (Dense)	(None, 1)	129
<hr/>		
=====		
Total params: 2,145,585		
Trainable params: 2,145,585		
Non-trainable params: 0		
<hr/>		

```
In [331]: history = model.fit(train, epochs=15, validation_data=val)
```

Epoch 1/15

Cleanup called...  
Cleanup called...  
Cleanup called...  
Cleanup called...  
Cleanup called...  
Cleanup called...  
Cleanup called...  
Cleanup called...

4/4 [=====] - ETA: 0s - loss: 0.7056 - accuracy: 0.5469

Cleanup called...  
Cleanup called...  
Cleanup called...  
Cleanup called...  
Cleanup called...  
Cleanup called...  
Cleanup called...

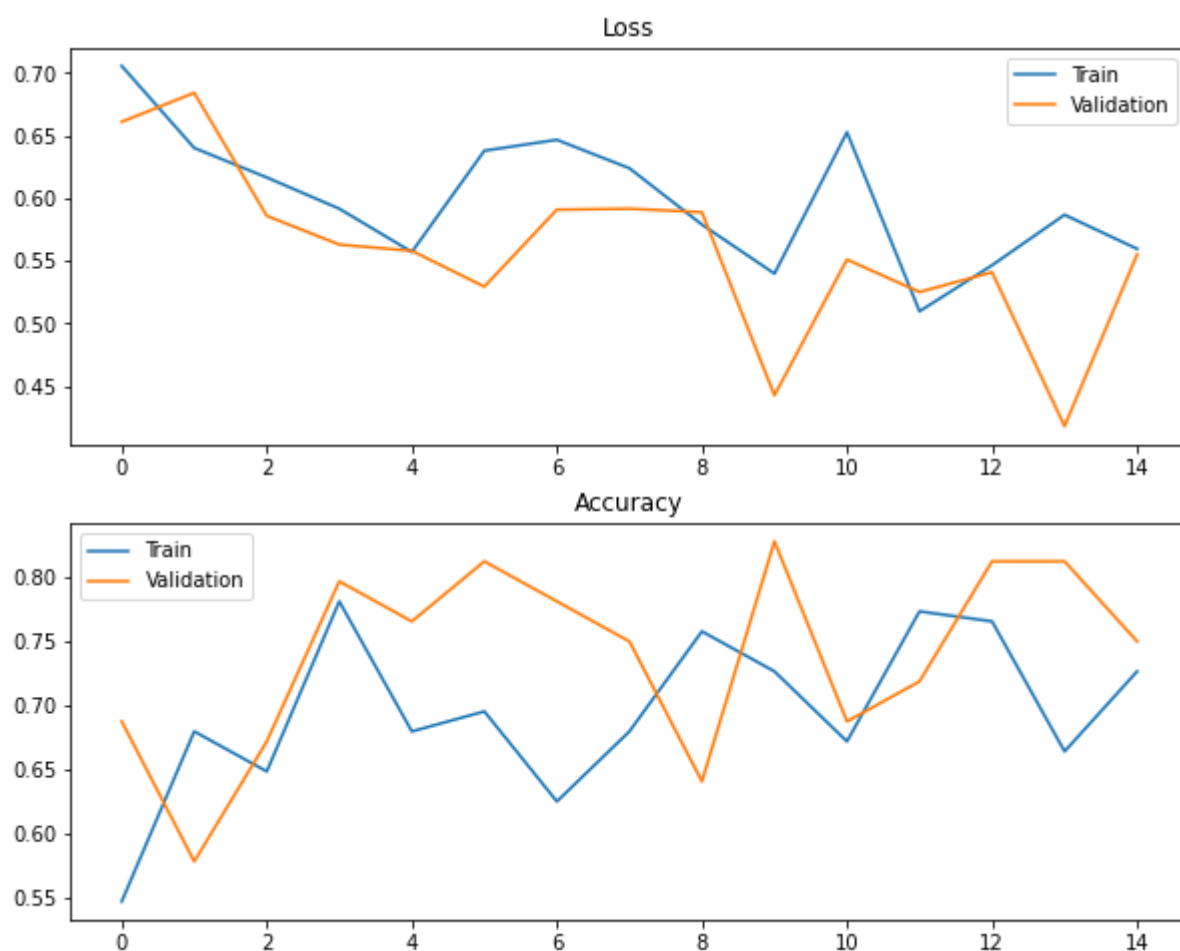
## Plotting the Performance

```
In [332]: fig, ax = plt.subplots(2, 1, figsize=(10,8))

ax[0].plot(history.history["loss"], label="Train")
ax[0].plot(history.history["val_loss"], label="Validation")
ax[0].title.set_text("Loss")
ax[0].legend()

ax[1].plot(history.history["accuracy"], label="Train")
ax[1].plot(history.history["val_accuracy"], label="Validation")
ax[1].title.set_text("Accuracy")
ax[1].legend()

plt.show()
```



## Results



```
In [333]: bin_acc = BinaryAccuracy()
recall = Recall()
precision = Precision()

for batch in test.as_numpy_iterator():
    X, y = batch
    yhat = model.predict(X)
    bin_acc.update_state(y, yhat)
    recall.update_state(y, yhat)
    precision.update_state(y, yhat)

print("Accuracy:", bin_acc.result().numpy(), "\nRecall:", recall.result().numpy(), "\nPrecision:", precision.result().numpy())
```

Cleanup called...

Cleanup called...

Cleanup called...

Cleanup called...

Cleanup called...

Cleanup called...

Cleanup called...

Cleanup called...

Accuracy: 0.78394395

Recall: 0.75757575

Precision: 0.8333333

## Manual Testing

We have already evaluated our model using various metrics and visualizations but it is always a good practice to test the model by hand to make sure everything is working well. In the code below, I randomly chose an image and plotted it with its true label on title so let's see if our model is going to classify this example correctly.

```
In [334]: batch = test.as_numpy_iterator().next()
```

Cleanup called...

Cleanup called...

Cleanup called...

Cleanup called...

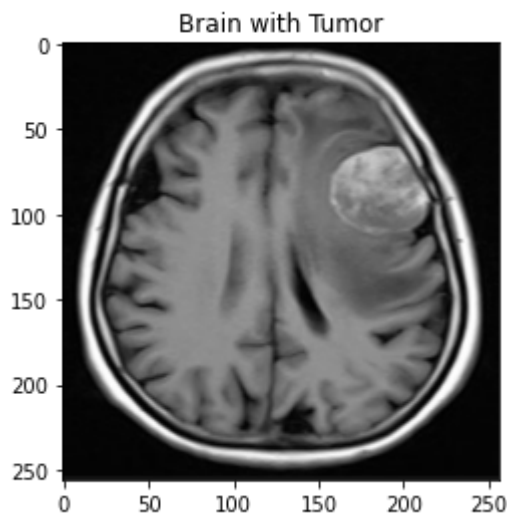
Cleanup called...

Cleanup called...

Cleanup called...

Cleanup called...

```
In [335]: img, label = batch[0][15], batch[1][15]
plt.imshow(img)
if label==1:
    plt.title("Brain with Tumor")
else:
    plt.title("Brain with No Tumor")
plt.show()
```



```
In [336]: y_hat = model.predict(np.expand_dims(img, 0))
```

We are able to see the probability of this brain's having tumor below. I opted to determine the classification threshold as 0.5. Meaning that, if it is below 0.5 this will be classified as Brain Having No Tumor otherwise it is going to be classified as Brain Having Tumor.

```
In [337]: y_hat
```

```
Out[337]: array([[0.86894506]], dtype=float32)
```

```
In [338]: if y_hat < 0.5:
            print("No tumor detected")
else:
            print("Tumor detected")
```

Tumor detected

Here we can see that our model predicted its class correctly.