Brain Tumor Detection Using Deep Learning

By Abhijeet Gupta and Yash Yadav

# Introduction

Brain tumors are a major cause of mortality worldwide. Early detection of brain tumors is critical for improved prognosis and quality of life for patients. Magnetic Resonance Imaging (MRI) is a commonly used medical imaging technique for brain tumor detection. However, manual interpretation of MRI scans by radiologists can be time-consuming and subjective. Deep learning techniques have shown promise in detecting brain tumors from medical images. In this project, we explore the development of a deep-learning model for brain tumor detection using MRI scans.

# Problem Statement

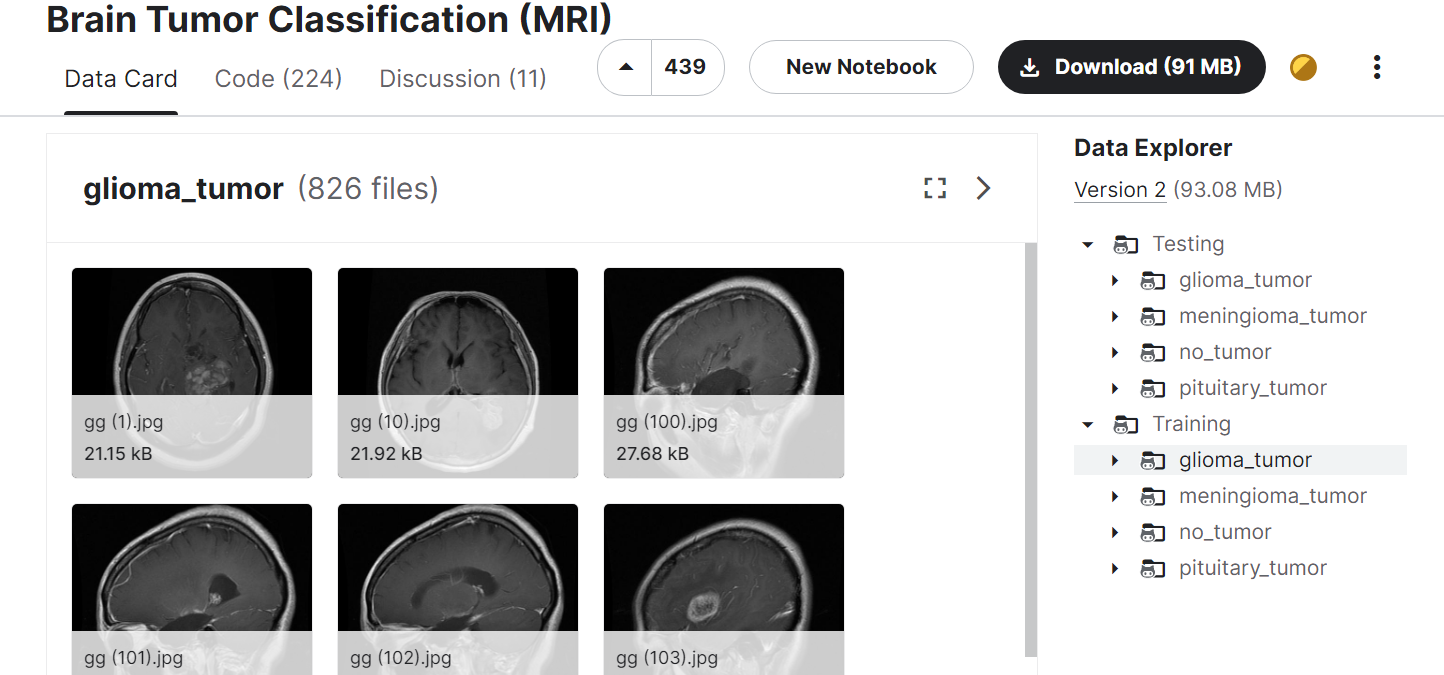
# The objective of our project is to develop a deep-learning model that can accurately detect brain tumors from MRI scans. We aim to improve the accuracy and efficiency of brain tumor detection to aid in early diagnosis and treatment.

# Methodology

## Data collection

For this project, we took an MRI dataset from Kaggle - <https://www.kaggle.com/datasets/sartajbhuvaji/brain-tumor-classification-mri>

This dataset contains preassigned training and testing data of 2870 and 394 images respectively. Within training and testing folders, we have 4 different types of image folders based on tumor type – glioma, meningioma, pituitary, and no tumor.

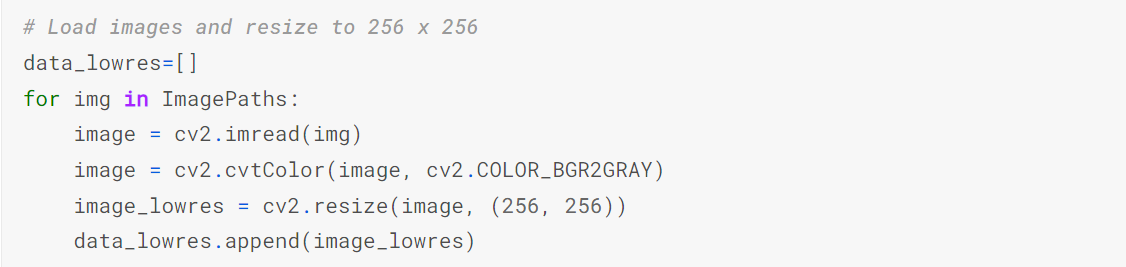


## Data pre-processing

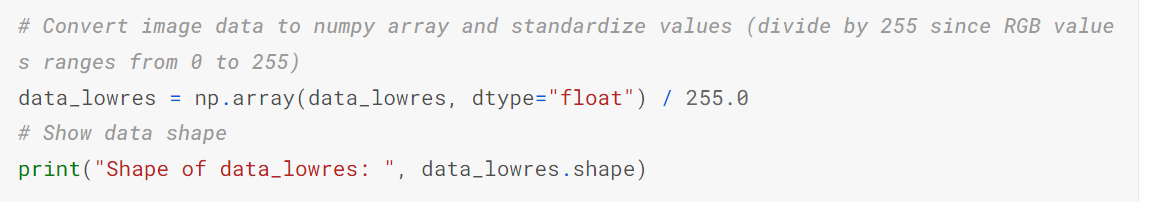
First of all, we convert the target variables into simple “Yes” (glioma, meningioma, pituitary) and “No” (no tumor), as the scope of the project, for now, is to simply predict whether a patient has cancer or not.



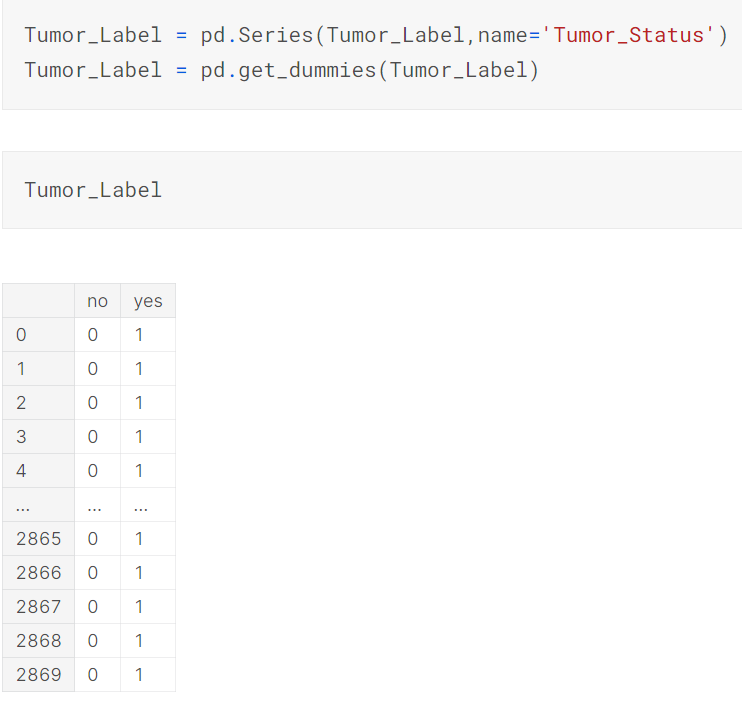
Next, we must scale our images to a standard size of 256 x 256



Next, we normalize the image



Finally, we convert target label from text to numeric format as well as dimension reshape for the Deep Learning Network to understand inputs





The above was done for training data – we repeat the same steps for testing image data

Model creation: deep learning using Convolutional Neural Network

We create a Convolutional Neural Network (CNN) from scratch using Keras, which is a Python interface to access and use TensorFlow deep learning libraries to create Neural Networks in Python.

In this project, we start with an input layer with an input shape as per our scaled image dimensions of (256,256,1).

Next, we apply 5 hidden layers, each consisting of a convolutional, max pooling, dropout, and batch normalization layer. Below is the function of each layer within hidden layers:

* Convolutional layers: Called Conv2D in Keras, this layer applies a set of filters to the input data and produces a set of feature maps as output.

In this project, all filters are 3x3 in size, and the number of filters increases from 1st hidden layer (32) to the last layer in multiples of 2 (the last/5th layer has 512 filters).

This layer also includes an activation function, which introduces nonlinearity into the network and allows it to model complex relationships in the input data. In this project, we use ReLU, which sets negative values to zero and leaves positive values unchanged.

Finally, the "padding" parameter is set to "same", as it means that the output of the convolutional layer will have the same spatial dimensions as the input data, and thus it preserves spatial information in the input data.

* Max Pooling layers: Called MaxPool2D in Keras, after each convolutional layer, a pooling layer is typically added to reduce the spatial size of the output feature maps. Pooling layers downsample the input data, reducing the sensitivity of the network to small shifts in the input and helping to prevent overfitting. The most common type of pooling layer is the max pooling layer, which selects the maximum value in each local region of the input.
* Dropout: Dropout is a regularization technique that is often used in CNNs to prevent overfitting. It involves randomly dropping out some of the neurons in the network during training, forcing the remaining neurons to learn more robust features. We set the dropout rate as 0.1 i.e. 10% of the previous layer neurons are not considered. This rate of 0.1 is not increased further, as it can result in underfitting
* Batch normalization: Batch normalization is another regularization technique that is used to improve the stability and speed of training. It involves normalizing the input to each layer to have zero mean and unit variance, which helps to prevent the distribution of activations from shifting too much during training.

Finally, we add a classifier head as the output layer. In this, we flatten the 2D output from hidden layers into a 1D vector and this is passed through a fully connected dense layer of 512 neuron units. This layer performs a linear transformation of the input data. After dropout and batch normalization, another dense layer is added (2 neurons) which uses a Sigmoid activation function to obtain the probability value of the 2 output classes (“Yes” or “No”).

In the case of testing data, one of the 2 output neurons with a higher probability value (between 0 and 1) is considered the predicted output label.

Code is as below

from tensorflow.keras import layers

model = keras.Sequential([

layers.InputLayer(input\_shape=(256, 256,1)),

*# First Convolutional Block*

layers.Conv2D(filters=32, kernel\_size=3, activation="relu", padding='same'),

layers.MaxPool2D(),

layers.Dropout(rate=0.1),

layers.BatchNormalization(),

*# Second Convolutional Block*

layers.Conv2D(filters=64, kernel\_size=3, activation="relu", padding='same'),

layers.MaxPool2D(),

layers.Dropout(rate=0.1),

layers.BatchNormalization(),

*# Third Convolutional Block*

layers.Conv2D(filters=128, kernel\_size=3, activation="relu", padding='same'),

layers.MaxPool2D(),

layers.Dropout(rate=0.1),

layers.BatchNormalization(),

*# Fourth Convolutional Block*

layers.Conv2D(filters=256, kernel\_size=3, activation="relu", padding='same'),

layers.MaxPool2D(),

layers.Dropout(rate=0.1),

layers.BatchNormalization(),

*# Fifth Convolutional Block*

layers.Conv2D(filters=512, kernel\_size=3, activation="relu", padding='same'),

layers.MaxPool2D(),

layers.Dropout(rate=0.1),

layers.BatchNormalization(),

*# Classifier Head*

layers.Flatten(),

layers.Dense(units=512, activation="relu"),

layers.Dropout(rate=0.1),

layers.BatchNormalization(),

layers.Dense(units=2, activation="sigmoid"),

])

model.summary()

# Model execution

Once the CNN model is created, we compile it using below parameters :

*# compile the model*

model.compile(optimizer='adam',

loss='binary\_crossentropy',

metrics=['binary\_accuracy'])

We then define early stopping object to prevent overfitting of the model during training. It involves monitoring the performance of the model on a validation dataset during training and stopping the training process when the performance on the validation dataset stops improving.

early\_stopping = EarlyStopping(

min\_delta=0.001, *# minimium amount of change to count as an improvement*

patience=20, *# how many epochs to wait before stopping*

restore\_best\_weights=True,

)

Finally, we fit the model to training data using early\_stopping defined above (considering validation data as the predefined test data), running up to a maximum of 1000 epochs

history = model.fit(

data\_lowres,

Tumor\_Label,

validation\_data = (data\_lowres\_test,Tumor\_Label\_test),

callbacks=[early\_stopping],

epochs=1000,

)

The CNN run is as below :

Epoch 1/1000

90/90 [==============================] - 14s 65ms/step - loss: 0.4704 - binary\_accuracy: 0.8643 - val\_loss: 2.0566 - val\_binary\_accuracy: 0.2665

Epoch 2/1000

90/90 [==============================] - 4s 49ms/step - loss: 0.2245 - binary\_accuracy: 0.9340 - val\_loss: 6.3312 - val\_binary\_accuracy: 0.2665

Epoch 3/1000

90/90 [==============================] - 4s 50ms/step - loss: 0.1550 - binary\_accuracy: 0.9429 - val\_loss: 6.3498 - val\_binary\_accuracy: 0.2665

Epoch 4/1000

90/90 [==============================] - 4s 49ms/step - loss: 0.1003 - binary\_accuracy: 0.9657 - val\_loss: 3.0912 - val\_binary\_accuracy: 0.2665

Epoch 5/1000

90/90 [==============================] - 4s 49ms/step - loss: 0.1146 - binary\_accuracy: 0.9606 - val\_loss: 2.3558 - val\_binary\_accuracy: 0.3173

Epoch 6/1000

90/90 [==============================] - 4s 50ms/step - loss: 0.1013 - binary\_accuracy: 0.9603 - val\_loss: 3.8395 - val\_binary\_accuracy: 0.3731

Epoch 7/1000

90/90 [==============================] - 4s 50ms/step - loss: 0.0985 - binary\_accuracy: 0.9634 - val\_loss: 2.5895 - val\_binary\_accuracy: 0.5203

Epoch 8/1000

90/90 [==============================] - 5s 51ms/step - loss: 0.0953 - binary\_accuracy: 0.9645 - val\_loss: 1.7004 - val\_binary\_accuracy: 0.5381

Epoch 9/1000

90/90 [==============================] - 4s 49ms/step - loss: 0.1260 - binary\_accuracy: 0.9530 - val\_loss: 10.3150 - val\_binary\_accuracy: 0.3109

Epoch 10/1000

90/90 [==============================] - 5s 50ms/step - loss: 0.0810 - binary\_accuracy: 0.9695 - val\_loss: 0.4229 - val\_binary\_accuracy: 0.8807

Epoch 11/1000

90/90 [==============================] - 5s 50ms/step - loss: 0.0430 - binary\_accuracy: 0.9857 - val\_loss: 0.3580 - val\_binary\_accuracy: 0.8921

Epoch 12/1000

90/90 [==============================] - 4s 50ms/step - loss: 0.0299 - binary\_accuracy: 0.9904 - val\_loss: 0.3904 - val\_binary\_accuracy: 0.8985

Epoch 13/1000

90/90 [==============================] - 5s 51ms/step - loss: 0.0254 - binary\_accuracy: 0.9920 - val\_loss: 0.3897 - val\_binary\_accuracy: 0.9048

Epoch 14/1000

90/90 [==============================] - 4s 50ms/step - loss: 0.0234 - binary\_accuracy: 0.9930 - val\_loss: 0.3953 - val\_binary\_accuracy: 0.9036

Epoch 15/1000

90/90 [==============================] - 5s 52ms/step - loss: 0.0408 - binary\_accuracy: 0.9866 - val\_loss: 0.6031 - val\_binary\_accuracy: 0.8744

Epoch 16/1000

90/90 [==============================] - 5s 50ms/step - loss: 0.0812 - binary\_accuracy: 0.9720 - val\_loss: 0.6539 - val\_binary\_accuracy: 0.8096

Epoch 17/1000

90/90 [==============================] - 4s 49ms/step - loss: 0.0379 - binary\_accuracy: 0.9871 - val\_loss: 0.3984 - val\_binary\_accuracy: 0.8756

Epoch 18/1000

90/90 [==============================] - 4s 49ms/step - loss: 0.0388 - binary\_accuracy: 0.9854 - val\_loss: 0.5624 - val\_binary\_accuracy: 0.8591

Epoch 19/1000

90/90 [==============================] - 4s 49ms/step - loss: 0.0180 - binary\_accuracy: 0.9941 - val\_loss: 0.5382 - val\_binary\_accuracy: 0.8845

Epoch 20/1000

90/90 [==============================] - 4s 50ms/step - loss: 0.0111 - binary\_accuracy: 0.9960 - val\_loss: 0.5194 - val\_binary\_accuracy: 0.8883

Epoch 21/1000

90/90 [==============================] - 4s 50ms/step - loss: 0.0078 - binary\_accuracy: 0.9984 - val\_loss: 0.4652 - val\_binary\_accuracy: 0.9277

Epoch 22/1000

90/90 [==============================] - 5s 51ms/step - loss: 0.0066 - binary\_accuracy: 0.9981 - val\_loss: 0.4703 - val\_binary\_accuracy: 0.9162

Epoch 23/1000

90/90 [==============================] - 4s 49ms/step - loss: 0.0090 - binary\_accuracy: 0.9983 - val\_loss: 0.4990 - val\_binary\_accuracy: 0.9048

Epoch 24/1000

90/90 [==============================] - 4s 50ms/step - loss: 0.0066 - binary\_accuracy: 0.9976 - val\_loss: 0.4307 - val\_binary\_accuracy: 0.9137

Epoch 25/1000

90/90 [==============================] - 4s 50ms/step - loss: 0.0035 - binary\_accuracy: 0.9997 - val\_loss: 0.5033 - val\_binary\_accuracy: 0.9137

Epoch 26/1000

90/90 [==============================] - 4s 50ms/step - loss: 0.0026 - binary\_accuracy: 0.9997 - val\_loss: 0.6141 - val\_binary\_accuracy: 0.8997

Epoch 27/1000

90/90 [==============================] - 4s 50ms/step - loss: 0.0108 - binary\_accuracy: 0.9972 - val\_loss: 0.4824 - val\_binary\_accuracy: 0.9239

Epoch 28/1000

90/90 [==============================] - 4s 50ms/step - loss: 0.0044 - binary\_accuracy: 0.9991 - val\_loss: 0.4828 - val\_binary\_accuracy: 0.9162

Epoch 29/1000

90/90 [==============================] - 5s 51ms/step - loss: 0.0040 - binary\_accuracy: 0.9990 - val\_loss: 0.7773 - val\_binary\_accuracy: 0.8693

Epoch 30/1000

90/90 [==============================] - 5s 50ms/step - loss: 0.0039 - binary\_accuracy: 0.9990 - val\_loss: 0.5383 - val\_binary\_accuracy: 0.9150

Epoch 31/1000

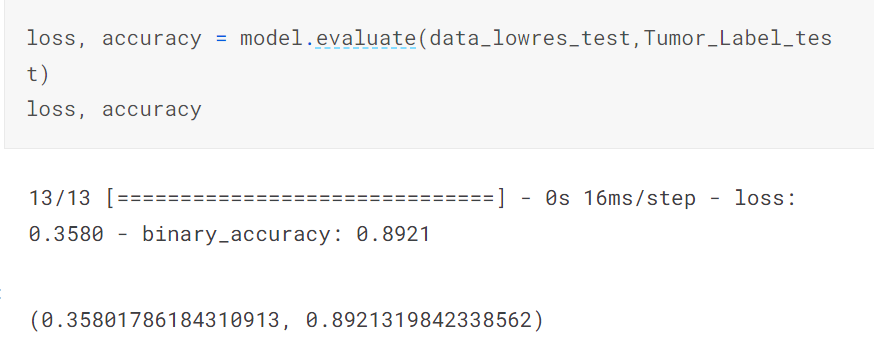
90/90 [==============================] - 4s 50ms/step - loss: 0.0031 - binary\_accuracy: 0.9995 - val\_loss: 0.7238 - val\_binary\_accuracy: 0.8807

Thanks to early stopping, the model training stops at 31st epoch, much earlier than the maximum of 1000 epochs, since validation loss has only increased since the 11th epoch. This means that the model is unlikely to be less lossy in further epochs and thus it’s better to stop it and save a lot of time.

# Evaluation Metrics We evaluated the performance of our model using the standard metrics of loss (binary cross entropy) and binary accuracy.

However, we have a class imbalance (only 395 out of 2870 are “No” Tumor i.e. 14% are “No” and 86% are “Yes”), and thus accuracy isn’t the best measure of model performance. Thus, we must consider Precision, Recall and F1-Score of “Yes” Tumor class.

# Results



We find testing / validation dataset loss to be 0.36 and accuracy as 89%.

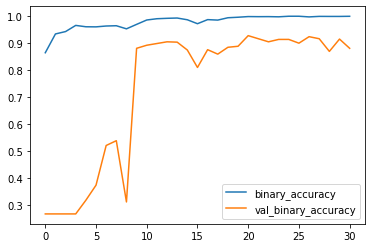
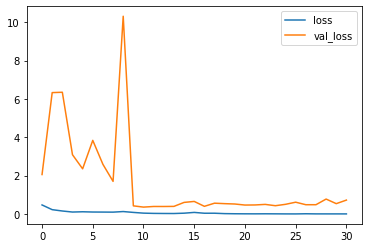
We can plot the training loss vs validation loss and training accuracy vs validation accuracy per epoch using below code :

history\_df = pd.DataFrame(history.history)

*# extract the accuracy values from the history object*

history\_df.loc[:, ['loss', 'val\_loss']].plot()

history\_df.loc[:, ['binary\_accuracy', 'val\_binary\_accuracy']].plot()



We see that validation loss and binary accuracy decrease and increase respectively as each epoch is executed. This is a good sign that our model performance is improving and ends up converging with each epoch.

However, since we have a class imbalance, we must consider Precision, Recall and F1-Score of “Yes” Tumor.

Before we calculate Precision, Recall and thus the F1 Score, we must convert the probability of the 2 output classes into a single label prediction, as only discrete labels can be used for confusion matrix calculation. Thus, we execute the below code:

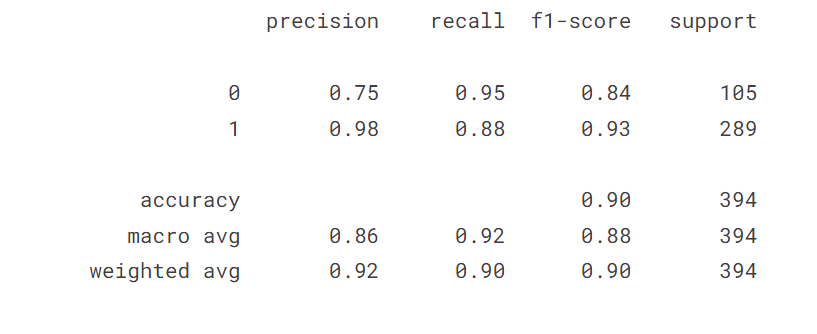
y\_pred = np.argmax(model.predict(data\_lowres\_test), axis=1)

This considers the class with a higher probability score as the predicted target label class.

Now, we calculate the confusion matrix to obtain precision, recall and F1 Score

from sklearn.metrics import classification\_report

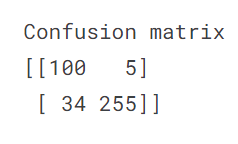
print(classification\_report(Tumor\_Label\_test['yes'], y\_pred))



from sklearn.metrics import confusion\_matrix

conf = confusion\_matrix(Tumor\_Label\_test['yes'], y\_pred)

print("**\n**Confusion matrix**\n**"+str(conf))



We have an F1 of 93% on Positive (“Yes”) Class.

# Conclusion

## What worked

Our project demonstrates the potential of deep learning for brain tumor detection from MRI scans. Our model achieved good performance on the MRI dataset, with an accuracy of 89% and an F1-Score on Tumor Positive of 93%.

## What did not work and why not

However, we could improve our model further by reducing False Negatives (34 out of 394 i.e., ~9%) - they can be dangerous as this means cancer patients are classified as non-cancerous.

Further, more research is needed to address the challenges of model interpretation, which is always a challenge with Deep Neural Networks.

Future work can explore the use of larger datasets obtained from a medical institute, and other deep learning techniques like Transfer Learning, which makes use of prebuilt deep learning models from platforms like HuggingFace to improve the accuracy and efficiency of brain tumor detection.

# Reproducibility

The working model with pre-uploaded dataset can be found on the given link - <https://www.kaggle.com/code/abhijeetgupta23/brain-tumor-detection-using-cnn-larger-dataset>

The advantage of a Cloud hosted platform like Kaggle is that there is no need to download image dataset and install python libraries – all these pre-exist in Kaggle and thus users don’t need to consume their personal hardware to make the code work. Kaggle also provides GPU to accelerate code run time significantly.

While you can see the code execution result by simply clicking the above given link, if someone wants to run this code by themselves, all they need to create a Kaggle account first. While they can run it right after signing in, phone verification is recommended so that Kaggle grants access to use GPU (P100) to run this code much faster.

Alternatively, if you do not want to create a Kaggle account, you can use the provided zip file to run the. ipynb file containing code including CNN. All you need to do is open this file in Jupyter Notebook and run it. However, you may need to pip install certain libraries like Tensorflow, Keras, numpy, pandas, matplotlib, etc. if it’s not already present.

# References

* <https://www.kaggle.com/datasets/sartajbhuvaji/brain-tumor-classification-mri/code>
* <https://iopscience.iop.org/article/10.1088/1757-899X/1055/1/012115>
* <https://bmcmedinformdecismak.biomedcentral.com/articles/10.1186/s12911-023-02114-6>
* <https://ieeexplore.ieee.org/document/9445185>
* <https://towardsdatascience.com/building-a-convolutional-neural-network-cnn-in-keras-329fbbadc5f5>
* <https://docs.opencv.org/3.4/d3/df2/tutorial_py_basic_ops.html>
* <https://betterprogramming.pub/5-useful-image-manipulation-techniques-using-python-opencv-505492d077ef?gi=eb9682af4471>
* <https://towardsdatascience.com/the-f1-score-bec2bbc38aa6>