

# **Brain Tumor Classification**

**Author:** Suchana Biswas Cheity  
Department of CSE  
North Western University

## **Objective:**

The goal of this project is to build a model that can classify Brain Tumor using MRI images. So, the project focuses on applying Convolutional Neural Networks (CNNs) combined with Transfer Learning and advanced Deep Learning techniques like Multi-Head Attention for classifying Brain efficiently and accurately, providing practical AI application development experience.

## **Dataset**

The dataset (Kaggle - Brain Tumor MRI Dataset) used for this project is well-suited for machine learning and deep learning applications due to its labeled structure, making it ideal for supervised learning tasks. It is organized into separate folders for **Training** and **Testing** data, with subfolders for each tumor category. Each folder contains MRI scan images in JPG format.

**Dataset Link:** <https://www.kaggle.com/datasets/masoudnickparvar/brain-tumor-mri-dataset>

## **Dataset Description:**

The dataset consists of MRI images labeled into four categories:

1. **Glioma:** A type of tumor that occurs in the brain and spinal cord.
2. **Meningioma:** Tumors that form in the meninges, the membranes surrounding the brain and spinal cord.
3. **Pituitary Tumor:** Tumors that develop in the pituitary gland.
4. **No Tumor:** MRI images that show no signs of tumors.

## 1. Importing the Libraries

```
[ ] import os
    import numpy as np
    import pandas as pd
```

**os:** Helps in handling the file paths.

**numpy:** Supports numerical computations.

**pandas:** Manages structured data.

**numpy** and **pandas** are mainly used for Data manipulation and analysis.

```
[ ] import seaborn as sns
    import matplotlib.pyplot as plt
```

**seaborn** and **matplotlib** is used for Data visualization.

```
[ ] from sklearn.preprocessing import LabelEncoder
```

**Label Encoder** converts tumor categories into numeric labels such as Glioma = 0, Meningioma = 1, No tumor = 2, and Pituitary = 3.

```
[ ] from imblearn.over_sampling import RandomOverSampler
```

It imports the class used for oversampling to balance the dataset.

```
[ ] import time
import shutil
import pathlib
import itertools
from PIL import Image

import cv2
import seaborn as sns
sns.set_style('darkgrid')
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, classification_report

import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.models import Sequential
from tensorflow.keras.optimizers import Adam, Adamax
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Activation, Dropout, BatchNormalization
from tensorflow.keras import regularizers

import warnings
```

## System and Utility Libraries:

- **import time:** Provides functions to measure and manipulate time. Often used for tracking execution time of code segments.
- **import shutil:** Used for file operations like copying, moving, and removing files or directories.
- **import pathlib:** Helps in handling file paths across different operating systems, making file operations more manageable and cross-platform compatible.
- **import itertools:** Provides tools for iterating over data efficiently, such as generating combinations or permutations, useful in data preprocessing and transformations.

## Image Processing Libraries:

- **from PIL import Image:** Part of the Python Imaging Library (Pillow), this module provides powerful functions for opening, manipulating, and saving different image file formats.
- **import cv2:** Part of the OpenCV library, widely used for computer vision tasks such as image and video processing. It allows transformations like resizing, color conversion, and applying filters.

## Data Visualization Libraries:

- **import seaborn as sns:** A visualization library built on top of Matplotlib, specifically for statistical graphics. Here, `sns.set_style('darkgrid')` sets the overall theme of plots to a dark grid.
- **import matplotlib.pyplot as plt:** A core plotting library in Python, providing comprehensive functions for plotting and customizing visualizations, often used alongside Seaborn.

## Machine Learning and Evaluation Libraries:

- **from sklearn.model\_selection import train\_test\_split:** Part of Scikit-Learn, this function is used to split the data into training and testing subsets, useful for evaluating model performance.
- **from sklearn.metrics import confusion\_matrix, classification\_report:** Also from Scikit-Learn, these functions calculate performance metrics:
  - **confusion\_matrix** gives a summary of prediction outcomes (true positives, false positives, etc.),
  - **classification\_report** provides detailed metrics (precision, recall, F1-score) for each class.

## Deep Learning Libraries (TensorFlow and Keras)

- **import tensorflow as tf:** TensorFlow is a popular deep learning library that provides end-to-end machine learning and deep learning capabilities.
- **from tensorflow import keras:** Keras, now integrated within TensorFlow, offers a user-friendly API for building and training neural networks.

### Keras-specific modules:

- **from tensorflow.keras.models import Sequential:** The Sequential API allows you to build a model layer by layer, suitable for simple feedforward neural networks.
- **from tensorflow.keras.optimizers import Adam, Adamax:** Optimizers for adjusting learning rates during training:

- Adam is a widely used optimizer that combines the benefits of two popular optimizers, AdaGrad and RMSprop.
- Adamax is a variant of Adam optimized for models with sparse gradients.
- **from tensorflow.keras.preprocessing.image import ImageDataGenerator:** Used for real-time data augmentation, it creates batches of image data with random transformations, which improves generalization by increasing the diversity of the training data.

### **Keras Layers:**

- **Conv2D:** Convolution layer, essential for capturing spatial hierarchies in image data.
- **MaxPooling2D:** Pooling layer that reduces spatial dimensions to lessen computational load.
- **Flatten:** Converts 2D arrays to 1D arrays, usually at the end of convolutional layers to prepare data for dense layers.
- **Dense:** Fully connected layer; connects each neuron to every neuron in the previous layer.
- **Activation:** Applies activation functions like ReLU or sigmoid to introduce non-linearities.
- **Dropout:** Regularization technique that randomly drops units in the layer during training to prevent overfitting.
- **BatchNormalization:** Normalizes layer inputs, stabilizing and speeding up the training process.
- **from tensorflow.keras import regularizers:** Provides regularization techniques to reduce overfitting, like L1 and L2 regularization, which penalize large weights.

### **Warnings Library:**

- **import warnings:** Manages and filters warnings that may appear due to deprecated functions or potential issues in the code.

```
[ ] import tensorflow as tf
    from tensorflow.keras import layers, models
    from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint
```

- **EarlyStopping:**
  - This callback stops training when a monitored metric (e.g., validation loss) has stopped improving for a set number of epochs.
  - It helps avoid overfitting and reduces the training time by stopping once the model's performance stagnates.
- **ModelCheckpoint:**
  - This callback saves the model at specific checkpoints during training. You can configure it to save only the best-performing model based on a metric, such as validation accuracy.
  - Useful for model recovery and for loading the best model after training completes.

## 2. Data Handling and Preprocessing

### Mounting Google Drive:

```
[ ] from google.colab import drive
    drive.mount('/content/drive')
```

Mounted at /content/drive

This project is run on Google Colab, and the dataset is stored in Google Drive.

### Loading Data and Directory Setup:

```
[ ] path = '/content/drive/My Drive/Brain_Tumor_Classification'
    train = os.path.join(path, 'Training')
    test = os.path.join(path, 'Testing')
    categories = ["glioma", "meningioma", "notumor", "pituitary"]
```

The training and testing image directories are defined here, and the images are read and categorized.

## Collecting Image Paths, Labels, and Creating a DataFrame:

```
[ ] image_paths = []
    labels = []

    # Iterate through the categories in the Training and Testing folders
    for data_type in [train, test]:
        for category in categories:
            category_path = os.path.join(data_type, category)
            if os.path.exists(category_path): # Check if the category path exists
                for image_name in os.listdir(category_path):
                    image_path = os.path.join(category_path, image_name)
                    image_paths.append(image_path)
                    labels.append(category)
            else:
                print(f"Directory not found: {category_path}")

    #Creating a DataFrame
    df = pd.DataFrame({
        "image_path": image_paths,
        "label": labels
    })
```

It collects the file paths and labels for each image and checks if folder exists. If not, it prints a message. It also creates a DataFrame where each row has an image's path and its label.

```
[ ] df['label'].unique()
array(['glioma', 'meningioma', 'notumor', 'pituitary'], dtype=object)

[ ] df['label'].value_counts()

      count
label
notumor    2000
pituitary   1757
meningioma  1645
glioma      1621

dtype: int64
```

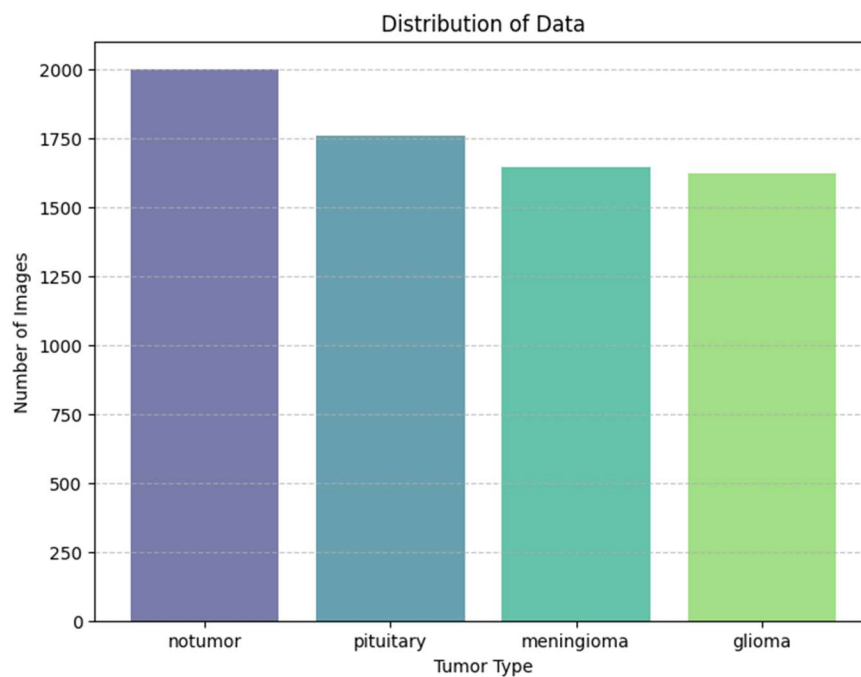
Here are the counts of how many images are in each category in the dataset.

## Visualizing Dataset Distribution:

```
[ ] import seaborn as sns
import matplotlib.pyplot as plt

# Count the number of images per category
label_counts = df["label"].value_counts()

# Plot the bar chart
plt.figure(figsize=(8, 6))
plt.bar(label_counts.index, label_counts.values, color=sns.color_palette("viridis", len(label_counts)), alpha=0.7)
plt.title("Distribution of Data")
plt.xlabel("Tumor Type")
plt.ylabel("Number of Images")
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.show()
```

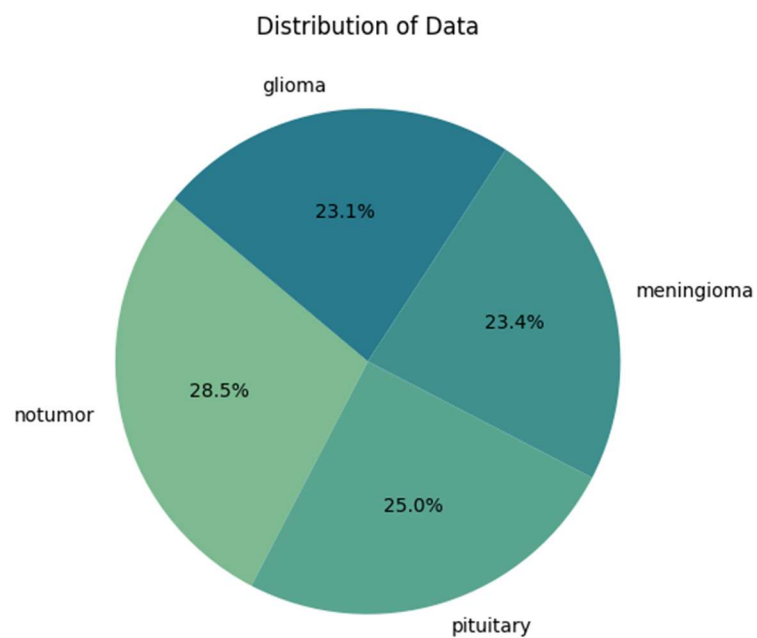


This one creates a bar plot to visualize the data distribution where the x-axis shows the Tumor Type and the y-axis shows the No of Images.

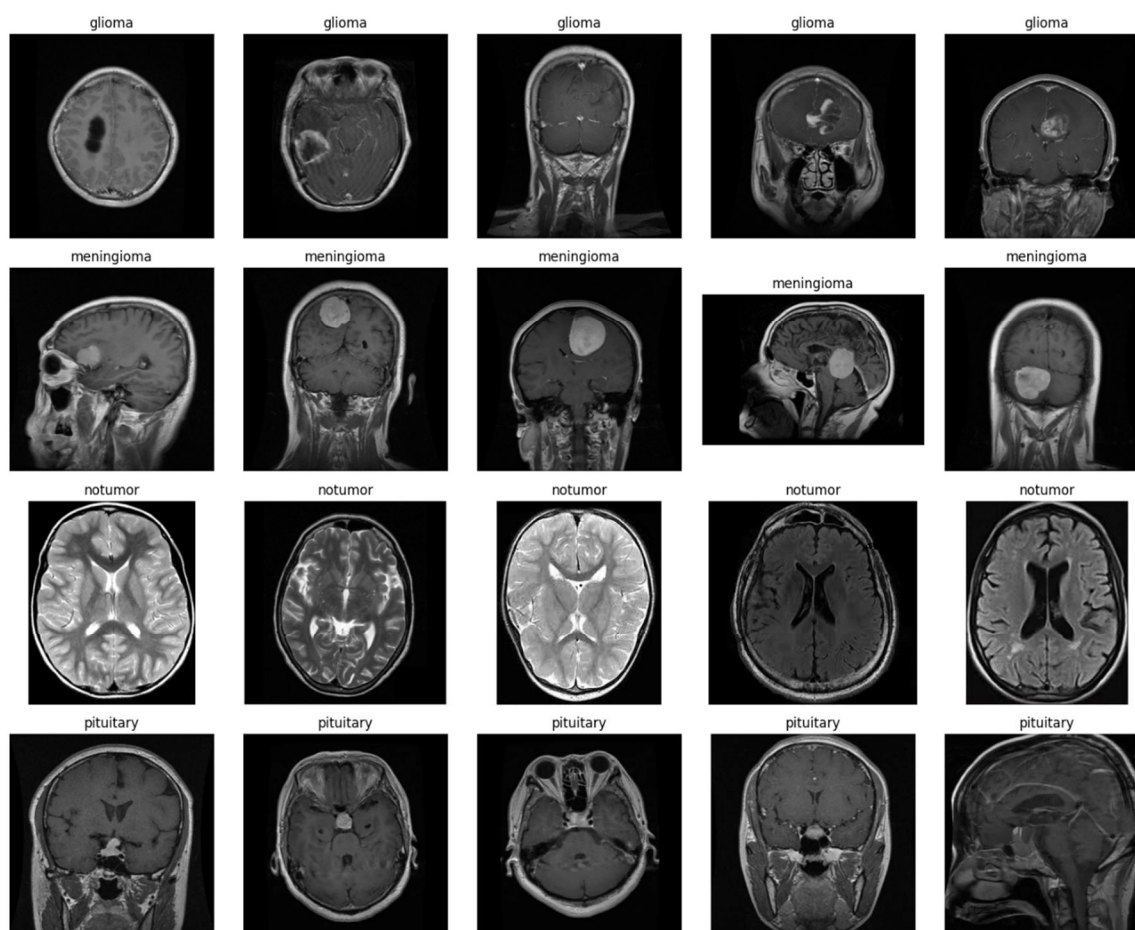
```
[ ] label_counts = df['label'].value_counts()
plt.figure(figsize=(8, 6))
plt.pie(label_counts, labels=label_counts.index, autopct='%1.1f%%', startangle=140, colors=sns.color_palette("crest"))
plt.title("Distribution of Data")
plt.show()
```

It creates a pie plot to show the data distribution ratio in the dataset.





Next, I've shown the converted images (BGR to RGB) from different categories.



## Category Encoding:

```
[ ] from sklearn.preprocessing import LabelEncoder

label_encoder = LabelEncoder()

df['category_encoded'] = label_encoder.fit_transform(df['label'])

[ ] df = df[['image_path', 'category_encoded']]

[ ] # Display the mapping of categories to encoded labels
category_mapping = dict(zip(label_encoder.classes_, label_encoder.transform(label_encoder.classes_)))
print("Category Encoding Mapping:")
for category, encoded_value in category_mapping.items():
    print(f"{category}: {encoded_value}")
```

Category Encoding Mapping:  
glioma: 0  
meningioma: 1  
notumor: 2  
pituitary: 3

Here I used Label Encoder to encode the target labels into numeric values. Encoding the categorical variables is a necessary step to make it more compatible with Machine Learning or Deep Learning models.

## Oversampling the Data:

```
[ ] from imblearn.over_sampling import RandomOverSampler

[ ] ros = RandomOverSampler(random_state=42)
X_resampled, y_resampled = ros.fit_resample(df[['image_path']], df['category_encoded'])

[ ] df_resampled = pd.DataFrame(X_resampled, columns=['image_path'])
df_resampled['category_encoded'] = y_resampled

print("\nClass distribution after oversampling:")
df_resampled['category_encoded'].value_counts()
```

Class distribution after oversampling:

category_encoded	count
0	2000
1	2000
2	2000
3	2000

dtype: int64

From the previous bar plots and pie charts, we can see that the ratio of the four categories is imbalanced. There are some categories that have more images than

the other which makes them imbalanced. To solve this problem, I've used oversampling where minority classes are duplicated to balance with the majority classes.

### Train, Validation, and Test Splits:

```
[ ] train_df_new, temp_df_new = train_test_split(
    df_resampled,
    train_size=0.8,
    shuffle=True,
    random_state=42,
    stratify=df_resampled['category_encoded']
)

valid_df_new, test_df_new = train_test_split(
    temp_df_new,
    test_size=0.5,
    shuffle=True,
    random_state=42,
    stratify=temp_df_new['category_encoded']
)
```

I've used stratified splitting method to ensure that the proportion of each class in the training, validation, and test sets remains consistent. Here **(80%)** of the data is used to train the model, **(10%)** of it is used to tune hyperparameters and evaluate performance during training, and **(10%)** of it is used for final evaluation to assess model performance.

### Image Generators:

```
[ ] from tensorflow.keras.preprocessing.image import ImageDataGenerator

batch_size = 16
img_size = (224, 224)
channels = 3
img_shape = (img_size[0], img_size[1], channels)

tr_gen = ImageDataGenerator(rescale=1./255)
ts_gen = ImageDataGenerator(rescale=1./255)


train_gen_new = tr_gen.flow_from_dataframe(
    train_df_new,
    x_col='image_path',
    y_col='category_encoded',
    target_size=img_size,
    class_mode='sparse',
    color_mode='rgb',
    shuffle=True,
    batch_size=batch_size
)
```

```

valid_gen_new = ts_gen.flow_from_dataframe(
    valid_df_new,
    x_col='image_path',
    y_col='category_encoded',
    target_size=img_size,
    class_mode='sparse',
    color_mode='rgb',
    shuffle=True,
    batch_size=batch_size
)

test_gen_new = ts_gen.flow_from_dataframe(
    test_df_new,
    x_col='image_path',
    y_col='category_encoded',
    target_size=img_size,
    class_mode='sparse',
    color_mode='rgb',
    shuffle=False,
    batch_size=batch_size
)

```

 Found 6400 validated image filenames belonging to 4 classes.  
 Found 800 validated image filenames belonging to 4 classes.  
 Found 800 validated image filenames belonging to 4 classes.

**Training Generator (train\_gen\_new)** feeds augmented data with shuffling, designed to improve model generalization. **Validation Generator (valid\_gen\_new)** provides non-augmented data, used to validate the model's performance during training. **Test Generator (test\_gen\_new)** supplies the model with non-augmented, non-shuffled data for final evaluation, allowing performance comparison across fixed test images. Overall, these generators make it easy to load, preprocess, and feed large datasets into a model in a memory-efficient way.

### 3. Model Architecture:

In this project, the brain tumor classification model is built using **VGG16**, a popular **convolutional neural network (CNN)** architecture. A **Convolutional Neural Network (CNN)** is a type of neural network designed for processing structured data like images. It is widely used in tasks like image classification, object detection, and more. **VGG16** is pre-trained on millions of images on ImageNet, so it can extract powerful features. We use the pre-trained weights and modify only the last layers, saving computation time and improving accuracy. **VGG16** has already achieved high accuracy in many image classification benchmarks. So, we chose this model.

```

def create_vgg16_model(input_shape):

    inputs = Input(shape=input_shape)

    base_model = VGG16(weights='imagenet', input_tensor=inputs, include_top=False)

    for layer in base_model.layers:
        layer.trainable = False

    x = base_model.output

    height, width, channels = 7, 7, 512
    x = Reshape((height * width, channels))(x)

    attention_output = MultiHeadAttention(num_heads=8, key_dim=channels)(x, x)
    attention_output = Reshape((height, width, channels))(attention_output)

    x = GaussianNoise(0.25)(attention_output)

    x = GlobalAveragePooling2D()(x)

    x = Dense(512, activation='relu')(x)
    x = BatchNormalization()(x)
    x = GaussianNoise(0.25)(x)
    x = Dropout(0.25)(x)

    outputs = Dense(4, activation='softmax')(x)

    model = Model(inputs=inputs, outputs=outputs)

    return model

input_shape = (224, 224, 3)

cnn_model = create_vgg16_model(input_shape)

```

Multi-Head Attention is a mechanism that enables the model to focus on multiple parts of an input simultaneously. Multi-head attention focuses on different parts of the feature map. It enhances the model's understanding of complex spatial relationships. Tumors often have subtle, non-obvious patterns. Attention helps the model capture long-range dependencies in images that CNNs alone might miss. I also added random noise to prevent overfitting by making the model more robust.

Adam (**Adaptive Moment Estimation**) is a popular optimization algorithm for training deep learning models. It optimizes learning speed and stability, allowing the model to converge faster than traditional optimizers like **Stochastic Gradient Descent (SGD)**. In CNNs, many weights may not be updated frequently. Adam ensures that learning rates are adapted, making the updates more effective. It adjusts learning rates automatically, reducing the need for extensive hyperparameter tuning.

## Training the Model:

```
[ ] history = cnn_model.fit(  
    train_gen_new,  
    validation_data=valid_gen_new,  
    epochs=5,  
    callbacks=[early_stopping],  
    verbose=1  
)
```

```
↵ Epoch 1/5  
400/400 ————— 2174s 5s/step - accuracy: 0.7419 - loss: 0.6714 - val_accuracy: 0.6888 - val_loss: 0.8618  
Epoch 2/5  
400/400 ————— 43s 105ms/step - accuracy: 0.8741 - loss: 0.3561 - val_accuracy: 0.7125 - val_loss: 1.2080  
Epoch 3/5  
400/400 ————— 81s 104ms/step - accuracy: 0.8996 - loss: 0.2759 - val_accuracy: 0.8763 - val_loss: 0.3401  
Epoch 4/5  
400/400 ————— 82s 103ms/step - accuracy: 0.9246 - loss: 0.2113 - val_accuracy: 0.8525 - val_loss: 0.4540  
Epoch 5/5  
400/400 ————— 43s 105ms/step - accuracy: 0.9423 - loss: 0.1770 - val_accuracy: 0.7688 - val_loss: 0.7828
```

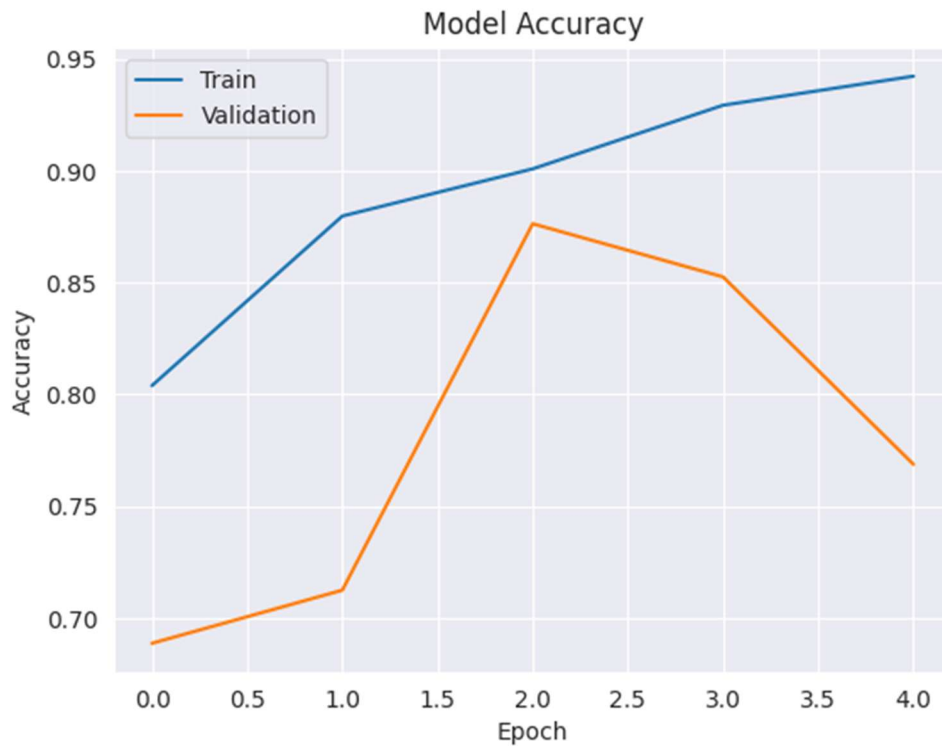
Here, **(80%)** of the data is used to train the model, **(10%)** of it is used to tune hyperparameters and evaluate performance during training. I've set the epochs to 5 which trains the model for 5 iterations over the dataset. If the validation performance stops improving, **early\_stopping** stops the training to avoid overfitting.

## 4. Evaluation Metrics:

### Accuracy and Loss Plots:

```
[ ] plt.plot(history.history['accuracy'])  
    plt.plot(history.history['val_accuracy'])  
    plt.title('Model Accuracy')  
    plt.ylabel('Accuracy')  
    plt.xlabel('Epoch')  
    plt.legend(['Train', 'Validation'], loc='upper left')  
    plt.show()
```

This plot is to visualize the model accuracy over the epochs.



```
[ ] plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='upper left')
plt.show()
```

This plot is to visualize the model loss over the epochs.





## Classification Report:

```
[ ] report = classification_report(test_labels, predicted_classes, target_names=list(test_gen_new.class_indices.keys()))
print(report)
```

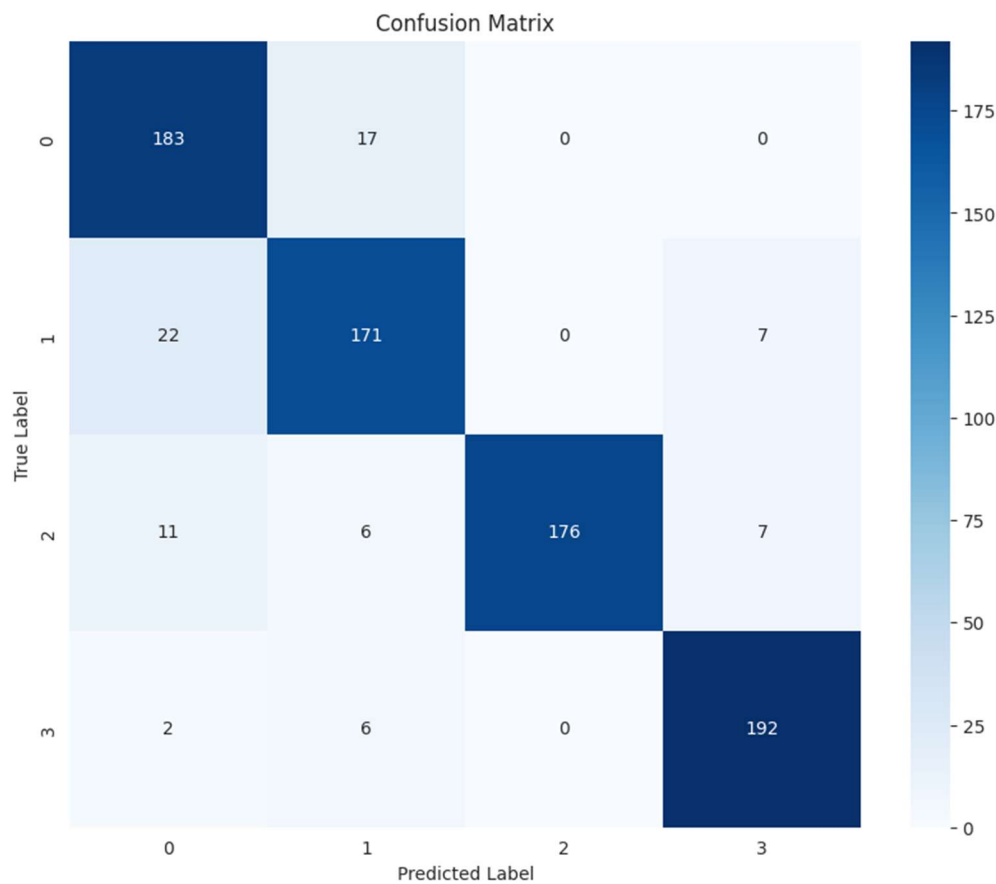
	precision	recall	f1-score	support
0	0.84	0.92	0.88	200
1	0.85	0.85	0.85	200
2	1.00	0.88	0.94	200
3	0.93	0.96	0.95	200
accuracy			0.90	800
macro avg	0.91	0.90	0.90	800
weighted avg	0.91	0.90	0.90	800

This classification report shows precision, recall, F1-score of the **VGG-16** model. **Class-0 (glioma)** has high recall which suggests effective detection. **Class-1 (meningioma)** has balanced precision and recall which indicates consistent classification. **Class-2 (no tumor)** has perfect precision which shows no false positives but the recall is slightly lower which indicates some false negatives. **Class-3 (pituitary)** has excellent performance with high precision and recall which shows minimal misclassifications.

Overall, the model achieves high accuracy of **90%** and robust performance across all categories.



Confusion Matrix Visualization:



Here is the confusion matrix for the model. **Class-3** has the highest accuracy with 192 correct predictions and minimal misclassifications. **Class 0** also shows strong performance but has 17 misclassifications into Class 1. **Class 1** exhibits notable confusion with Class 0 and some misclassifications into Class 3. **Class 2** has the highest diversity in misclassifications, particularly misclassified as Class 0 and Class 1.