**BRAIN TUMOR:**

**A brain tumor** is an abnormal growth of cells within the brain or the central nervous system. These tumors can be benign (non-cancerous) or malignant (cancerous) and can originate from brain tissue itself or from cells in other parts of the body that have spread to the brain (metastatic tumors).

Here's a detailed explanation of brain tumors:

1. Classification:
   * Primary Brain Tumors: These tumors originate within the brain or its surrounding structures.
     + Gliomas: The most common primary brain tumors, originating from glial cells. They include astrocytomas, oligodendrogliomas, and glioblastomas.
     + Meningiomas: Arise from the meninges, the protective layers around the brain.
     + Pituitary Adenomas: Develop in the pituitary gland.
     + Medulloblastomas: Typically affect children and originate in the cerebellum.
   * Metastatic Brain Tumors: Secondary tumors that have spread to the brain from other parts of the body, commonly from the lungs, breasts, or skin.
2. Benign vs. Malignant:
   * Benign Brain Tumors: These tumors are non-cancerous and usually grow slowly. They don't invade surrounding tissues or spread to other parts of the body. However, they can cause problems if they press on vital areas of the brain.
   * Malignant Brain Tumors (Brain Cancer): These tumors are cancerous and can invade nearby tissues, grow rapidly, and spread to other parts of the brain or body. Glioblastoma is one of the most aggressive malignant brain tumors.
3. Causes:
   * The exact cause of most brain tumors is not well-understood, but certain risk factors, such as exposure to radiation, family history, and certain genetic disorders, may increase the risk of developing a brain tumor.
4. Symptoms:
   * Symptoms can vary widely depending on the location and size of the tumor. Common symptoms include headaches, seizures, nausea, vomiting, changes in vision, weakness, difficulty with coordination, and changes in personality or cognitive function.
5. Diagnosis:
   * Diagnosis typically involves a combination of imaging tests such as MRI and CT scans, followed by a biopsy or surgical removal of the tumor to confirm its type and grade (degree of malignancy).
6. Treatment:
   * Treatment options depend on the type, grade, and location of the tumor, as well as the patient's overall health. Common treatment options include:
     + Surgery: To remove the tumor.
     + Radiation Therapy: Using high-energy beams to target and destroy cancer cells.
     + Chemotherapy: Medications to kill cancer cells or stop their growth.
     + Targeted Therapy: Medications that specifically target cancer cells.
     + Immunotherapy: Boosting the body's immune system to fight cancer.
     + Supportive Care: Managing symptoms and improving quality of life, which may include physical therapy, speech therapy, and pain management.
7. Prognosis:
   * The prognosis for a brain tumor varies widely based on its type and grade. Some benign tumors can be successfully treated with surgery and have a good prognosis. Malignant brain tumors are more challenging to treat, and the prognosis is often less favorable, particularly for high-grade gliomas.
8. Recovery and Follow-up:
   * After treatment, patients may require ongoing follow-up care, including monitoring for recurrence, managing side effects of treatment, and addressing any long-term effects on physical and cognitive function.

Living with a brain tumor can be challenging, and it often involves a multidisciplinary approach with healthcare providers specializing in neurosurgery, neuro-oncology, radiation oncology, and supportive care to provide comprehensive care for the patient. The specific treatment plan and prognosis will be determined by a healthcare team based on the individual case.

some additional details about brain tumors:

1. Clinical Grading:
   * The World Health Organization (WHO) classifies brain tumors into different grades based on their characteristics:
     + Grade I: These are benign tumors with a low potential for spreading.
     + Grade II: These tumors are generally slow-growing, but they can become more aggressive over time.
     + Grade III: Anaplastic tumors that are malignant and grow more rapidly.
     + Grade IV: The most aggressive tumors, such as glioblastoma multiforme.
2. Complications:
   * Brain tumors can lead to various complications, including neurological deficits, cognitive changes, and seizures. Treatment-related complications can also occur, such as infection, bleeding, or side effects from radiation and chemotherapy.
3. Support and Coping:
   * Dealing with a brain tumor diagnosis can be emotionally and physically challenging. Support from family, friends, and support groups can be essential. Palliative care and hospice care may be needed for advanced cases.
4. Research and Advancements:
   * Ongoing research is dedicated to understanding the underlying causes of brain tumors, developing more effective treatments, and improving the quality of life for patients. Advances in targeted therapies and immunotherapies offer promising options for some individuals.
5. Prevention:
   * Since the causes of most brain tumors are not well-understood, there are no specific preventive measures. Reducing exposure to radiation (e.g., from excessive X-ray exposure) and adopting a healthy lifestyle, including not smoking and maintaining a balanced diet, can promote overall health, which may indirectly reduce the risk of cancer.
6. Clinical Trials:
   * Clinical trials offer experimental treatments and therapies for brain tumor patients. They are essential for advancing the field and may provide patients with access to cutting-edge treatments.
7. Pediatric Brain Tumors:
   * Brain tumors in children are different from those in adults and often require specialized care. Medulloblastomas, ependymomas, and brainstem gliomas are common pediatric brain tumors. Pediatric neuro-oncology specialists are involved in their diagnosis and treatment.
8. Legal and Financial Considerations:
   * Dealing with a brain tumor diagnosis can be expensive. It's important to understand your insurance coverage, seek financial assistance if needed, and be aware of legal rights and protections for individuals with disabilities or serious medical conditions.

It's crucial to consult with a healthcare professional if you or a loved one is experiencing symptoms suggestive of a brain tumor. Timely diagnosis and appropriate treatment can significantly impact the outcome and quality of life for those affected by brain tumors. Each case is unique, and treatment plans are tailored to the individual's specific condition, goals, and needs. Regular communication with healthcare providers and a strong support network can make a significant difference in the patient's journey.

=====================================================================

Deep learning is a subfield of machine learning that focuses on the development of artificial neural networks inspired by the human brain to process and understand data. It has gained immense popularity in recent years due to its remarkable success in various applications, such as image and speech recognition, natural language processing, and autonomous systems like self-driving cars. Here's an explanation of deep learning:

1. Neural Networks: Deep learning relies on artificial neural networks, which are composed of interconnected nodes or "neurons." These neurons are organized into layers, including input, hidden, and output layers. Information flows through the network, with each neuron performing a weighted computation and passing its result to the next layer.
2. Deep vs. Shallow Learning:
   * Shallow Learning: Shallow machine learning models typically have a small number of layers or none at all. They are less effective at capturing complex patterns and hierarchies in data.
   * Deep Learning: Deep neural networks have multiple hidden layers (hence the term "deep"), enabling them to learn intricate, hierarchical representations from raw data. This depth allows them to handle high-dimensional data efficiently.
3. Feature Learning:
   * Deep learning is particularly effective at feature learning. Instead of relying on human-engineered features, it learns relevant features directly from the data. This ability to automatically extract meaningful features makes deep learning models versatile and capable of handling a wide range of tasks.
4. Training and Backpropagation:
   * Training a deep learning model involves adjusting the weights and biases of the network to minimize the difference between the predicted output and the actual target. The most common algorithm for this is backpropagation, which calculates the gradients of the model's error with respect to its parameters and updates the weights accordingly.
5. Activation Functions:
   * Activation functions introduce non-linearity into the network, allowing it to learn complex relationships. Common activation functions include the sigmoid, ReLU (Rectified Linear Unit), and tanh.
6. Deep Learning Architectures:
   * Various deep learning architectures exist, each designed for specific tasks. Some popular architectures include:
     + Convolutional Neural Networks (CNNs): Suited for image and video analysis, they use convolutional layers to capture local patterns.
     + Recurrent Neural Networks (RNNs): Ideal for sequence data like text and speech, RNNs have connections that loop back on themselves to capture temporal dependencies.
     + Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU): Specialized RNN architectures for handling longer sequences and mitigating the vanishing gradient problem.
     + Transformers: Introduced in natural language processing, transformers have revolutionized tasks like machine translation and language understanding.
     + Autoencoders: Used for unsupervised learning and data compression, autoencoders have applications in anomaly detection and dimensionality reduction.
7. Training Data and Deep Learning:
   * Deep learning models require large amounts of labeled data for training. This is known as supervised learning. For some tasks, limited data can be augmented using techniques like data synthesis or transfer learning.
8. Challenges in Deep Learning:
   * Deep learning requires substantial computational resources, and overfitting (learning noise in the data rather than the actual pattern) can be a concern with small datasets. Regularization techniques and dropout are used to mitigate this issue.
9. Applications:
   * Deep learning has made significant advances in various domains, including computer vision (object detection, image classification), natural language processing (text generation, sentiment analysis, machine translation), speech recognition, recommendation systems, and reinforcement learning (used in gaming and autonomous systems).
10. Future and Research:
    * Research in deep learning is ongoing, with a focus on improving model interpretability, transfer learning, and enhancing the efficiency and scalability of deep neural networks.

Deep learning has revolutionized the field of artificial intelligence and has the potential to continue driving innovation in many areas, from healthcare to autonomous vehicles, as its capabilities and applications continue to expand.

======================================================================

A Convolutional Neural Network (CNN) is a specialized type of deep neural network designed primarily for processing and analyzing visual data. CNNs are at the core of many computer vision tasks, such as image classification, object detection, and facial recognition. They are designed to automatically and adaptively learn patterns and features from images, making them highly effective for tasks that involve understanding visual content. Here's an in-depth explanation of CNNs:

1. Convolutional Layers:
   * CNNs are composed of layers with a specific architecture. The core of a CNN is the convolutional layer. In this layer, small filters or kernels (typically 3x3 or 5x5) slide over the input image to detect patterns and features.
   * Each filter performs a convolution operation by computing a dot product between its values and a local region of the input image. The result of this operation is known as a feature map.
2. Feature Hierarchies:
   * CNNs use multiple convolutional layers to create hierarchies of features. In early layers, filters detect basic features like edges and corners, while deeper layers learn to recognize complex structures, textures, and object parts.
   * This hierarchy of features enables the network to progressively recognize more abstract and high-level concepts as it processes deeper layers.
3. Pooling Layers:
   * After convolutional layers, pooling layers are often used to reduce the spatial dimensions of the feature maps. Pooling involves downsampling the feature maps by taking the maximum or average value within small windows.
   * Pooling helps reduce the computational load, decrease the risk of overfitting, and make the network more translation-invariant.
4. Fully Connected Layers:
   * Following the convolutional and pooling layers, CNNs often include one or more fully connected layers. These layers are typical neural network layers in which all neurons are connected to every neuron in the previous layer.
   * Fully connected layers are responsible for making final decisions or predictions, such as classifying an object in an image.
5. Activation Functions:
   * Activation functions like ReLU (Rectified Linear Unit) are applied after each convolution and fully connected layer. ReLU introduces non-linearity and helps the network learn complex patterns.
6. Parameters and Training:
   * CNNs have a large number of learnable parameters, mainly the weights associated with filters in convolutional layers. These parameters are learned through a process called backpropagation, where the network is trained on labeled data.
   * The network learns to minimize the difference between its predictions and the actual targets using gradient descent and optimization algorithms.
7. Applications:
   * CNNs have a wide range of applications, including image classification, object detection, image segmentation, face recognition, medical image analysis, self-driving cars, and more. They have also been adapted for non-visual data like speech and text.
8. Transfer Learning:
   * Transfer learning is a technique where pre-trained CNN models, trained on a large dataset like ImageNet, are used as a starting point for new tasks. By fine-tuning these models with a smaller dataset, you can leverage the learned features and save training time and resources.
9. Challenges and Variations:
   * Challenges in CNNs include overfitting, especially with small datasets, and the need for substantial computational resources. Various CNN architectures have been developed, such as VGG, Inception, ResNet, and MobileNet, each with different design principles to address these challenges.

Convolutional Neural Networks have revolutionized the field of computer vision and significantly improved the accuracy of various visual recognition tasks. Their ability to automatically learn features from raw image data, along with advancements in model architectures and training techniques, has led to their wide adoption in practical applications across diverse domains.

======================================================================

Python is a versatile and widely-used high-level programming language known for its simplicity and readability. It is a general-purpose language with a wide range of applications, from web development and scientific computing to artificial intelligence and data analysis. Here's an in-depth explanation of Python:

1. History and Development:
   * Python was created by Guido van Rossum and first released in 1991. It was designed to be easy to read and write, with a clear and straightforward syntax. Python's development is managed by the Python Software Foundation (PSF), and it has a large and active community of developers.
2. Interpreted Language:
   * Python is an interpreted language, which means that code is executed line by line, rather than being compiled into machine code. This makes it more accessible for beginners and simplifies cross-platform development.
3. Syntax and Readability:
   * Python is known for its clean and readable syntax. Indentation (whitespace) is used to define code blocks instead of curly braces or keywords, making the code more readable and encouraging good coding practices.
4. Dynamically Typed:
   * Python is dynamically typed, meaning that you don't need to specify variable types explicitly. The interpreter determines the data type at runtime. This allows for greater flexibility but requires careful coding to avoid unexpected results.
5. Rich Standard Library:
   * Python comes with a comprehensive standard library that includes modules and packages for various tasks, from file handling to web development. This minimizes the need to write code from scratch and accelerates development.
6. Cross-Platform Compatibility:
   * Python is available on various platforms, including Windows, macOS, and Linux, making it suitable for cross-platform development.
7. Community and Ecosystem:
   * Python has a vibrant and active community of developers and users. This community contributes to a wealth of third-party libraries and frameworks, such as NumPy, Pandas, Django, Flask, and TensorFlow, which extend Python's capabilities for specific tasks and domains.
8. Web Development:
   * Python is widely used for web development. Frameworks like Django and Flask simplify the process of building web applications. Python's simplicity and readability also make it a popular choice for scripting and automation tasks on web servers.
9. Data Science and Scientific Computing:
   * Python is a top choice for data science and scientific computing. Libraries like NumPy, Pandas, SciPy, and Matplotlib provide tools for data analysis, visualization, and statistical modeling. Jupyter notebooks are often used for interactive data exploration.
10. Artificial Intelligence and Machine Learning:
    * Python is the language of choice for AI and machine learning. Libraries like TensorFlow, PyTorch, and scikit-learn provide robust frameworks for building and training machine learning models.
11. Scripting and Automation:
    * Python is frequently used for scripting tasks, system administration, and automation. It can interact with operating system functions, file handling, and network protocols.
12. Game Development:
    * While not as popular as other languages like C++ or C#, Python is used for game development. The Pygame library, for example, simplifies game development with Python.
13. Education and Learning:
    * Python is widely used in education due to its simplicity and readability. It is often recommended for beginners in programming.
14. Software Testing and Quality Assurance:
    * Python is used for testing and quality assurance in software development. Libraries like pytest make it easy to write and run tests.
15. Limitations:
    * Python's interpreted nature can make it slower than compiled languages like C++. It may not be the best choice for certain applications that require high-performance computing.

Python's combination of simplicity, readability, and a rich ecosystem of libraries and frameworks has made it a versatile and popular choice for a wide range of applications, from simple scripts to complex AI systems. Its active community and open-source nature ensure that Python continues to evolve and remain relevant in the rapidly changing world of technology and programming

======================================================================

**CODE EXPLANATION**

# General Imports import tensorflow as tf import pandas as pd import numpy as np import random import os # Visualization import matplotlib.pyplot as plt import seaborn as sns # Building Model from keras.utils import plot\_model from tensorflow.keras import models from tensorflow.keras.layers import BatchNormalization from tensorflow.keras.layers import MaxPooling2D from tensorflow.keras.layers import Conv2D from tensorflow.keras.layers import Dense from tensorflow.keras.layers import Dropout from tensorflow.keras.layers import Flatten from tensorflow.keras.optimizers import legacy # Training Model from tensorflow.keras.callbacks import EarlyStopping from tensorflow.keras.callbacks import ReduceLROnPlateau from tensorflow.keras.callbacks import ModelCheckpoint # Data Processing from tensorflow.keras.preprocessing.image import ImageDataGenerator from tensorflow.keras.preprocessing.image import img\_to\_array from tensorflow.keras.preprocessing.image import array\_to\_img from tensorflow.keras.preprocessing.image import load\_img

This initial code block consists of various imports that are necessary for building and training a deep learning model using TensorFlow and Keras. Let's break down each import:

1. tensorflow and tf: These are the primary libraries for working with deep learning models in Python, and tf is often used as an alias for tensorflow.
2. pandas and pd: These libraries are used for data manipulation and analysis. They are often used to read and preprocess data.
3. numpy and np: Numpy is a library for numerical computations in Python. It's commonly used to handle numerical data, such as arrays and matrices.
4. random: The random module provides functions for generating random numbers. It's often used for tasks like shuffling data.
5. os: The os module provides a way to work with the operating system, which is useful for tasks like file manipulation and directory management.
6. matplotlib.pyplot and plt: These are used for data visualization, particularly for creating plots and charts.
7. seaborn and sns: Seaborn is a data visualization library that builds on top of matplotlib. It is often used to create more aesthetically pleasing and informative statistical graphics.
8. keras.utils.plot\_model: This function is used to visualize the structure of a Keras model, showing how layers are connected.
9. tensorflow.keras.models: This module is used for defining and working with Keras models.
10. tensorflow.keras.layers: It provides various layer types, like Conv2D (convolutional layers), Dense (fully connected layers), BatchNormalization, MaxPooling2D, Dropout, and Flatten. These layers are building blocks for deep learning models.
11. tensorflow.keras.optimizers: This module contains various optimization algorithms that are used to update the model weights during training.
12. tensorflow.keras.callbacks: Callbacks are functions that can be applied at different stages of the training process. Common callbacks include EarlyStopping (to stop training when a certain condition is met), ReduceLROnPlateau (to reduce the learning rate on a plateau), and ModelCheckpoint (to save the model during training).
13. tensorflow.keras.preprocessing.image: This module is used for processing image data, including data augmentation and loading images as arrays. It includes functions like ImageDataGenerator, img\_to\_array, array\_to\_img, and load\_img.

These imports set up the environment for building and training deep learning models, and they include tools for data processing, visualization, and model development. If you have any questions about any specific part of the code or how to use these libraries, please feel free to ask.



# Global variables SAVE = False SEED = 111 # Setting seed for consistent results tf.keras.utils.set\_random\_seed(SEED) tf.random.set\_seed(SEED) np.random.seed(SEED) # Data Visualization updates %config InlineBackend.figure\_format = 'retina' plt.rcParams["figure.figsize"] = (16, 10) plt.rcParams.update({'font.size': 14})

This code block sets some global variables and configures settings for data visualization. Let's break down what each part of the code does:

1. SAVE = False: This is a global variable that is set to False. It appears to be a flag that might control whether to save some results or models. If set to True, it might trigger the saving of some outputs.
2. SEED = 111: Another global variable that sets the random seed to 111. Setting a random seed ensures that the randomness in the code (e.g., random initialization of neural networks) is reproducible. Using a fixed seed is essential for obtaining consistent results when running experiments.
3. Setting Random Seeds:
   * tf.keras.utils.set\_random\_seed(SEED): This sets the random seed for TensorFlow operations related to Keras. It ensures that the behavior of Keras functions that involve randomness remains consistent across runs.
   * tf.random.set\_seed(SEED): This sets the random seed for TensorFlow's random number generator.
   * np.random.seed(SEED): This sets the random seed for NumPy, ensuring that NumPy's random functions produce the same results on each run.
4. Data Visualization Updates:
   * %config InlineBackend.figure\_format = 'retina': This is a Jupyter Notebook magic command. It configures the rendering of figures to be of higher quality (e.g., for retina displays).
   * plt.rcParams["figure.figsize"] = (16, 10): This sets the default figure size for Matplotlib plots to be 16 inches wide and 10 inches high.
   * plt.rcParams.update({'font.size': 14}): This sets the font size for text in plots to 14.

These settings are essential for ensuring that your experiments are reproducible and for controlling the appearance of your data visualizations. If you have any further questions about these settings or their implications, please let me know.



# Function for inporting data def get\_data\_labels(directory, shuffle=True, random\_state=0): """ Function used for going into the main training directory whose directory has sub-class-types. """ from sklearn.utils import shuffle import os # Lists to store data and labels data\_path = [] data\_labels = [] for label in os.listdir(directory): label\_dir = os.path.join(directory, label) # Avoid MacOS storing path if not os.path.isdir(label\_dir): continue # Going into each folder and getting image path for image in os.listdir(label\_dir): image\_path = os.path.join(label\_dir, image) data\_path.append(image\_path) data\_labels.append(label) if shuffle: data\_path, data\_labels = shuffle(data\_path, data\_labels, random\_state=random\_state) return data\_path, data\_labels

This function, get\_data\_labels, is designed to retrieve data and their corresponding labels from a directory containing sub-class types. Let's break down what this function does:

1. It starts by importing necessary libraries:
   * sklearn.utils.shuffle: This function is used for shuffling data. It ensures that the data and labels are shuffled if shuffle is set to `True.
   * os: This library is used for various file and directory operations.
2. The function takes the following parameters:
   * directory: The main directory containing sub-class types.
   * shuffle (default is True): A boolean flag to indicate whether the data and labels should be shuffled.
   * random\_state (default is 0): An integer used to set the random seed for shuffling data. This ensures reproducibility when shuffling.
3. Inside the function, two empty lists, data\_path and data\_labels, are initialized. These lists will be used to store the image paths and corresponding labels.
4. The function iterates through the sub-directories within the main directory. Each sub-directory represents a label or class in the dataset.
5. Within each label's sub-directory, it iterates through the images and constructs the full path to each image. These paths are stored in the data\_path list, and the corresponding label (the sub-directory name) is stored in the data\_labels list.
6. If the shuffle parameter is set to True, the data\_path and data\_labels lists are shuffled. The shuffling is done using the sklearn.utils.shuffle function with the specified random\_state.
7. Finally, the function returns the data\_path list (containing image file paths) and the data\_labels list (containing corresponding class labels). These can be used for further processing, such as loading the images, preprocessing, and training a model.

This function is useful for preparing data for training deep learning models, especially when the data is organized into sub-directories by class, which is a common structure for image classification tasks.



# Setting up file paths for training and testing USER\_PATH = r"/kaggle/input/brain-tumors-dataset/Data" normal = USER\_PATH + r'/' tumor = USER\_PATH + r'/Tumor' # Getting data using above function normal\_paths, normal\_labels = get\_data\_labels(normal, shuffle=False) tumor\_paths, tumor\_labels = get\_data\_labels(tumor, shuffle=False) # Printing Normal and tumor sample sizes print('Normal Brain Scans') print(f'Number of Paths: {len(normal\_paths)}') print(f'Number of Labels: {len(normal\_labels)}') print('\nTumor Brain Scans') print(f'Number of Paths: {len(tumor\_paths)}') print(f'Number of Labels: {len(tumor\_labels)}')

This code block sets up file paths for training and testing data, gets the data using the get\_data\_labels function, and prints the sample sizes for normal and tumor brain scans. Let's break it down step by step:

1. USER\_PATH: This variable is set to the base path where the dataset is located. It appears to be a raw path to the dataset directory.
2. normal and tumor: These variables are created by appending the dataset directory path with specific subdirectories. normal points to the root directory, and tumor points to the subdirectory named 'Tumor' within the dataset.
3. normal\_paths and normal\_labels: These variables store the image file paths and labels (class names) for the normal brain scans. The get\_data\_labels function is used to retrieve these.
4. tumor\_paths and tumor\_labels: These variables store the image file paths and labels (class names) for the tumor brain scans. Again, the get\_data\_labels function is used to retrieve these.
5. The code then prints the sample sizes for the normal and tumor brain scans, showing the number of paths (images) and labels (class names) for each category. This information is useful for understanding the size and distribution of the dataset.

The code sets up the data paths and labels, but it doesn't load or preprocess the images at this point. This is typically the initial step when preparing a dataset for deep learning. The paths and labels will be used later for data loading and model training.



print(f'Some Tumor labels: {tumor\_labels[500:505]}') # Remove the word "tumor" from each element in the list tumor\_labels = [path.replace('\_tumor', '').title() for path in tumor\_labels] # Print the cleaned paths print(f'Some Tumor labels: {tumor\_labels[500:505]}')

This code block performs two main actions:

1. Printing a Subset of Tumor Labels:
   * It prints a subset of tumor labels to the console. Specifically, it prints the tumor labels from index 500 to 504 (a subset of the tumor\_labels list). This is done using a formatted string with an f-string.
2. Cleaning Tumor Labels:
   * It then proceeds to clean the tumor labels by removing the word "tumor" from each element in the list. This is achieved using a list comprehension and the replace and title string methods.
   * path.replace('\_tumor', '') replaces the substring "\_tumor" with an empty string, effectively removing it from each label.
   * .title() capitalizes the first letter of each word in the resulting label. This is typically done to ensure consistent capitalization style in class labels.
3. Printing the Cleaned Tumor Labels:
   * Finally, it prints a subset of the cleaned tumor labels (the same range as before) to the console, showing the labels with the word "tumor" removed and with proper capitalization.

This code is useful for data preprocessing, especially when dealing with class labels that need cleaning or formatting to match a specific naming convention or style.



from sklearn.model\_selection import train\_test\_split # Merging paths and labels MRI\_paths = normal\_paths + tumor\_paths MRI\_labels = normal\_labels + tumor\_labels # Split the data into training and testing sets train\_paths, test\_paths, train\_labels, test\_labels = train\_test\_split( MRI\_paths, MRI\_labels, test\_size=0.35, random\_state=SEED ) # Create DataFrames for training and testing data train\_df = pd.DataFrame({'path': train\_paths, 'label': train\_labels}) test\_df = pd.DataFrame({'path': test\_paths, 'label': test\_labels})

This code block performs the following operations:

1. Importing train\_test\_split:
   * It imports the train\_test\_split function from the sklearn.model\_selection module. This function is commonly used to split a dataset into training and testing subsets.
2. Merging Paths and Labels:
   * MRI\_paths and MRI\_labels are created by concatenating the paths and labels of normal brain scans (normal\_paths and normal\_labels) with the paths and labels of tumor brain scans (tumor\_paths and tumor\_labels). This creates a unified list of image file paths and their corresponding class labels for the entire dataset.
3. Splitting the Data:
   * The train\_test\_split function is used to split the data into training and testing sets. The data to be split are MRI\_paths and MRI\_labels.
   * test\_size=0.35 indicates that 35% of the data will be allocated to the testing set, while the remaining 65% will be used for training.
   * random\_state=SEED ensures that the random splitting is reproducible, as it uses the seed value specified earlier.
4. Creating DataFrames:
   * Two pandas DataFrames, train\_df and test\_df, are created to store the training and testing data.
   * The training DataFrame, train\_df, has two columns: 'path' and 'label'. 'path' contains the file paths to the training images, and 'label' contains the corresponding class labels.
   * The testing DataFrame, test\_df, is created in a similar manner, containing paths and labels for the testing images.

These DataFrames make it easier to manage and access the data during the training and testing phases of the machine learning or deep learning model development. They provide a structured way to organize and manipulate the data.



# 2 values in train\_df and 1 in test\_df have a un-defined tumor value train\_df = train\_df[train\_df['label'] != 'Tumor'] test\_df = test\_df[test\_df['label'] != 'Tumor']

In this code block, it appears that there are some entries in both the train\_df and test\_df DataFrames where the 'label' is set to 'Tumor' (possibly representing undefined values or missing labels). The code filters out these entries and keeps only the rows where the 'label' is not equal to 'Tumor'. This effectively removes any data points with undefined or missing class labels.

Here's what each line of code does:

1. train\_df = train\_df[train\_df['label'] != 'Tumor']:
   * It filters the train\_df DataFrame and keeps only the rows where the 'label' column is not equal to 'Tumor'. This line effectively removes any rows with the label 'Tumor' from the training data.
2. test\_df = test\_df[test\_df['label'] != 'Tumor']:
   * Similarly, it filters the test\_df DataFrame and keeps only the rows where the 'label' column is not equal to 'Tumor'. This line effectively removes any rows with the label 'Tumor' from the testing data.

This step is essential to ensure that the data used for training and testing the model contains only valid, defined class labels. Undefined or missing labels can cause issues when training and evaluating machine learning or deep learning models.



CLASS\_TYPES = pd.unique(train\_df['label']) N\_TYPES = len(CLASS\_TYPES) print(f'Classes: {CLASS\_TYPES} and length {N\_TYPES}')

In this code block, the unique class labels and the number of unique classes in the training dataset (train\_df) are determined and printed. Here's a breakdown of what this code does:

1. CLASS\_TYPES = pd.unique(train\_df['label']):
   * It extracts the unique class labels from the 'label' column of the train\_df DataFrame using the pd.unique function. This creates an array containing all the unique class labels present in the training data.
2. N\_TYPES = len(CLASS\_TYPES):
   * It calculates the number of unique classes by determining the length of the CLASS\_TYPES array. This count represents the total number of distinct classes or categories in the dataset.
3. print(f'Classes: {CLASS\_TYPES} and length {N\_TYPES}'):
   * It prints the unique class labels and the number of unique classes in the training data. This provides information about the diversity of classes in the dataset.

This information is crucial when working with classification tasks, as it helps you understand the different categories that the model is expected to classify data into. It also informs decisions related to model architecture and evaluation.



# Mapping dictionary tumor\_classes = {'Normal': 0, 'Glioma': 1, 'Meningioma': 2, 'Pituitary': 3 } # Apply mapping to both DataFrames train\_df['label\_encoded'] = train\_df['label'].map(tumor\_classes) test\_df['label\_encoded'] = test\_df['label'].map(tumor\_classes)

In this code block, a mapping dictionary called tumor\_classes is defined, and then this mapping is applied to both the train\_df and test\_df DataFrames to create a new column called 'label\_encoded'. Here's what each part of the code does:

1. tumor\_classes: This is a dictionary that maps class labels to numerical codes. In this case, it associates the following class labels with numerical codes:
   * 'Normal' is mapped to 0.
   * 'Glioma' is mapped to 1.
   * 'Meningioma' is mapped to 2.
   * 'Pituitary' is mapped to 3.
2. train\_df['label\_encoded'] = train\_df['label'].map(tumor\_classes):
   * This line creates a new column in the train\_df DataFrame called 'label\_encoded'. It's based on the 'label' column of the DataFrame, where each class label in the 'label' column is mapped to its corresponding numerical code using the map method. The result is a new column with the numerical representations of the class labels.
3. test\_df['label\_encoded'] = test\_df['label'].map(tumor\_classes):
   * Similarly, a 'label\_encoded' column is created in the test\_df DataFrame, applying the same mapping to the 'label' column of the testing data.

This mapping from class labels to numerical codes is common in machine learning and deep learning tasks, as it allows models to work with numerical data for classification while preserving the relationship between classes and their codes. It is often used when training models, as numerical representations are more convenient for mathematical operations.



\_, ax = plt.subplots(ncols=3, figsize=(20, 14)) # Plotting training data types class\_counts = [len([x for x in train\_labels if x == label]) for label in CLASS\_TYPES] print('Training Counts') print(dict(zip(CLASS\_TYPES, class\_counts))) ax[0].set\_title('Training Data') ax[0].pie( class\_counts, labels=CLASS\_TYPES, colors=['#FAC500','#0BFA00', '#0066FA','#FA0000'], autopct=lambda p: '{:.2f}%\n{:,.0f}'.format(p, p \* sum(class\_counts) / 100), explode=tuple(0.01 for i in range(N\_TYPES)), textprops={'fontsize': 20} ) # Plotting distribution of train test split ax[1].set\_title('Train Test Split') ax[1].pie( [len(train\_labels), len(test\_labels)], labels=['Train','Test'], colors=['darkcyan', 'orange'], autopct=lambda p: '{:.2f}%\n{:,.0f}'.format(p, p \* sum([len(train\_labels), len(test\_labels)]) / 100), explode=(0.1, 0), startangle=85, textprops={'fontsize': 20} ) # Plotting testing data types class\_counts = [len([x for x in test\_labels if x == label]) for label in CLASS\_TYPES] print('\nTesting Counts') print(dict(zip(CLASS\_TYPES, class\_counts))) ax[2].set\_title('Testing Data') ax[2].pie( class\_counts, labels=CLASS\_TYPES, colors=['#FAC500', '#0BFA00', '#0066FA', '#FA0000'], autopct=lambda p: '{:.2f}%\n{:,.0f}'.format(p, p \* sum(class\_counts) / 100), explode=tuple(0.01 for i in range(N\_TYPES)), # Explode the slices slightly for better visualization textprops={'fontsize': 20} # Set the font size for the text on the pie chart ) plt.show()

This code block is responsible for creating three pie charts to visualize the distribution of different classes in the training and testing datasets. It provides insights into the class balance and the split between training and testing data. Here's a breakdown of what each part of the code does:

1. \_ (Underscore): An underscore is commonly used as a placeholder for a variable that you don't intend to use. In this case, it might be used to indicate that the result of plt.subplots is not of interest.
2. ax = plt.subplots(ncols=3, figsize=(20, 14)): This line creates a figure with three subplots arranged in a single row (specified by ncols=3). The figsize argument sets the figure's size to 20 inches wide and 14 inches high. The individual subplots will be accessible through the ax variable.
3. Plotting Training Data Types:
   * class\_counts is a list of counts for each class label in the training data.
   * A pie chart is created on the first subplot (ax[0]) to visualize the distribution of classes in the training data.
   * labels=CLASS\_TYPES assigns the class names as labels.
   * colors assigns colors to each class.
   * autopct specifies the format for the percentage and count display on the chart.
   * explode determines the degree to which slices are separated for better visualization.
   * textprops specifies the font size for the text on the chart.
4. Plotting Distribution of Train Test Split:
   * A pie chart is created on the second subplot (ax[1]) to visualize the distribution between training and testing data.
   * This pie chart shows the proportion of data allocated to training and testing sets.
   * The labels are 'Train' and 'Test', and colors are assigned to each.
   * autopct, explode, and textprops work similarly to the first pie chart.
5. Plotting Testing Data Types:
   * class\_counts is redefined for the testing data to create a pie chart in the third subplot (ax[2]).
   * It visualizes the distribution of classes in the testing data, following a similar structure to the first pie chart.
6. plt.show(): This command displays the entire figure containing all three pie charts.

Overall, this code provides a clear visualization of the class distribution in the training and testing datasets, as well as the split between training and testing data. It's helpful for understanding the balance of classes and the distribution of data across different categories.



# Function to display a list of images based on the given index def show\_images(paths, label\_paths, index\_list=range(10), im\_size=250, figsize=(12, 8), save=False): """ Show images from a given path based on the inputted list indices related to the desired images one wishes to see. """ num\_images = len(index\_list) num\_rows = (num\_images + 3) // 4 \_, ax = plt.subplots(nrows=num\_rows, ncols=4, figsize=figsize) ax = ax.flatten() for i, index in enumerate(index\_list): if i >= num\_images: break image = load\_img(paths[index], target\_size=(im\_size, im\_size)) ax[i].imshow(image) ax[i].set\_title(f'{index}: {label\_paths[index]}') ax[i].axis('off') plt.tight\_layout() if save: plt.savefig('show\_image.pdf') else: plt.show()

This is a Python function named show\_images that is used to display a list of images based on the given indices. Here's an explanation of what this function does:

* paths: This parameter should be a list of file paths to the images that you want to display.
* label\_paths: This parameter should be a list of labels or names corresponding to the images. The labels are displayed below each image.
* index\_list: This parameter is a list of indices that specifies which images to display. By default, it's set to range(10), which would display the first 10 images. You can provide a custom list of indices to display specific images.
* im\_size: This parameter specifies the size (width and height) at which the images should be displayed. By default, it's set to 250 pixels.
* figsize: This parameter sets the size of the matplotlib figure in which the images will be displayed. By default, it's set to (12, 8).
* save: This is a boolean parameter. If set to True, the displayed images will be saved as a PDF file named 'show\_image.pdf'. If set to False, the images will be displayed in the matplotlib figure.

The function performs the following steps:

1. It calculates the number of images to display based on the length of index\_list.
2. It creates a matplotlib figure with subplots, arranging them in rows and columns based on the number of images to display.
3. It loads and displays the specified images from the paths list at the specified size (im\_size) and with their corresponding labels below each image.
4. If the save parameter is set to True, the figure is saved as a PDF file. Otherwise, it's displayed on the screen.

This function is a useful tool for quickly visualizing a set of images and their corresponding labels in a grid-like format. It's commonly used for visually inspecting and verifying data during data preprocessing and exploratory data analysis.



# Four different data classification images, from three different angles (images are independent) show\_images(train\_paths, train\_labels, index\_list=[0, 160, 6, 64, 197, 5, 21, 472, 8, 438, 3, 495], im\_size=350, figsize=(13,10), save=SAVE)

This code block uses the show\_images function to display a set of images for data classification. The function is called with the following parameters:

* train\_paths: A list of file paths to the training images.
* train\_labels: A list of labels corresponding to the training images.
* index\_list: A list of indices specifying which images to display. The indices [0, 160, 6, 64, 197, 5, 21, 472, 8, 438, 3, 495] are provided to display specific images.
* im\_size: The size at which the images should be displayed, set to 350 pixels.
* figsize: The size of the matplotlib figure, set to (13, 10).
* save: Whether to save the displayed images as a PDF file, based on the value of the global variable SAVE.

The images to display are selected by their corresponding indices from the train\_paths list. The labels from the train\_labels list are displayed below each image.

This code will generate a visual representation of a set of images from the training dataset, showing a variety of images for data classification. The specific indices chosen in index\_list will determine which images are displayed. If SAVE is set to True, the images will be saved as a PDF file; otherwise, they will be displayed on the screen.



def img\_preprocessing(image\_path, label\_encoded): # Read and decode the image img = tf.io.read\_file(image\_path) img = tf.image.decode\_jpeg(img, channels=3) # Convert to grayscale img = tf.image.rgb\_to\_grayscale(img) # Resize and normalize the image img = tf.image.resize(img, size=(150, 150)) img = img / 255.0 return img, tf.one\_hot(label\_encoded, depth=N\_TYPES) # Creating dataset loaders with label indices train\_loader = tf.data.Dataset.from\_tensor\_slices((train\_df['path'], train\_df['label\_encoded'])) train\_dataset = (train\_loader.map(img\_preprocessing ).batch(batch\_size).shuffle(train\_df['path'].shape[0]).prefetch(batch\_size)) test\_loader = tf.data.Dataset.from\_tensor\_slices((test\_df['path'], test\_df['label\_encoded'])) test\_dataset = (test\_loader.map(img\_preprocessing ).batch(batch\_size).shuffle(test\_df['path'].shape[0]).prefetch(batch\_size))

In this code block, a function named img\_preprocessing is defined, and two dataset loaders (train\_loader and test\_loader) are created for preparing image data for training and testing deep learning models. Let's break down what each part of the code does:

### img\_preprocessing Function:

* img\_preprocessing is a function that takes two parameters: image\_path and label\_encoded.
* It reads and decodes the image from the file specified by image\_path using TensorFlow functions.
* The image is then converted to grayscale using the tf.image.rgb\_to\_grayscale function.
* The image is resized to a target size of (150, 150) pixels.
* The image is normalized by dividing it by 255.0 to ensure pixel values are in the range [0, 1].
* The function returns the preprocessed image and the one-hot encoded label based on the provided label\_encoded and the global variable N\_TYPES, which is the number of unique classes.

### Dataset Loaders:

* train\_loader and test\_loader are created using tf.data.Dataset.from\_tensor\_slices. These datasets are constructed from the image file paths and their corresponding one-hot encoded labels.
* The .map method is used to apply the img\_preprocessing function to each element of the dataset. This function preprocesses the images and encodes the labels.
* The datasets are then batched with a specified batch\_size. Batching groups multiple images together for more efficient training.
* The shuffle method is used to shuffle the data within each dataset. The argument passed to shuffle is the size of the dataset, ensuring that the data is shuffled effectively.
* The .prefetch method is used to prefetch data during training, improving data loading performance.

These datasets (train\_dataset and test\_dataset) are now ready for training and testing deep learning models. The preprocessing steps, such as reading, decoding, converting to grayscale, resizing, and normalization, are applied to the images within the datasets, making them suitable for model input.



# Define the model architecture model = models.Sequential() # Convolutional layer 1 model.add(Conv2D(32, (4, 4), activation="relu", input\_shape=image\_shape)) model.add(MaxPooling2D(pool\_size=(3, 3))) # Convolutional layer 2 model.add(Conv2D(64, (3, 3), activation="relu")) model.add(MaxPooling2D(pool\_size=(3, 3))) # Convolutional layer 3 model.add(Conv2D(128, (3, 3), activation="relu")) model.add(MaxPooling2D(pool\_size=(3, 3))) # # Convolutional layer 4 model.add(Conv2D(128, (3, 3), activation="relu")) model.add(Flatten()) # Full connect layers model.add(Dense(512, activation="relu")) model.add(Dropout(0.5, seed=SEED)) model.add(Dense(N\_TYPES, activation="softmax")) model.summary() optimizer = legacy.Adam(learning\_rate=0.001, beta\_1=0.87, beta\_2=0.9995) model.compile(optimizer=optimizer, loss='categorical\_crossentropy', metrics= ['accuracy'])

In this code block, a Convolutional Neural Network (CNN) model is defined using the Keras library. Here's a breakdown of the model architecture and compilation:

### Model Architecture:

1. model = models.Sequential(): A Sequential model is created. This model allows you to build a neural network by stacking layers sequentially.
2. Convolutional Layer 1:
   * Conv2D(32, (4, 4), activation="relu", input\_shape=image\_shape): The first convolutional layer is added to the model with 32 filters of size (4, 4), ReLU activation function, and an input shape specified by image\_shape.
3. Max Pooling Layer 1:
   * MaxPooling2D(pool\_size=(3, 3)): A max-pooling layer is added with a pool size of (3, 3) to reduce the spatial dimensions of the feature maps.
4. Convolutional Layer 2:
   * Conv2D(64, (3, 3), activation="relu"): The second convolutional layer is added with 64 filters of size (3, 3) and ReLU activation.
5. Max Pooling Layer 2:
   * MaxPooling2D(pool\_size=(3, 3)): Another max-pooling layer is added with a pool size of (3, 3).
6. Convolutional Layer 3:
   * Conv2D(128, (3, 3), activation="relu"): The third convolutional layer is added with 128 filters of size (3, 3) and ReLU activation.
7. Max Pooling Layer 3:
   * MaxPooling2D(pool\_size=(3, 3)): The third max-pooling layer is added with a pool size of (3, 3).
8. Convolutional Layer 4:
   * Conv2D(128, (3, 3), activation="relu"): The fourth convolutional layer is added with 128 filters of size (3, 3) and ReLU activation.
9. Flattening Layer:
   * Flatten(): The feature maps are flattened into a one-dimensional vector.
10. Fully Connected Layers:
    * Dense(512, activation="relu"): A fully connected layer with 512 units and ReLU activation is added.
    * Dropout(0.5, seed=SEED): A dropout layer is added with a dropout rate of 0.5. Dropout is used to prevent overfitting.
    * Dense(N\_TYPES, activation="softmax"): The final fully connected layer with an output size equal to the number of unique classes. It uses the softmax activation function for multiclass classification.

### Model Summary:

The model.summary() function is called to print a summary of the model's architecture, showing the layers, output shapes, and the number of parameters.

### Model Compilation:

The model is compiled using the following settings:

* Optimizer: An Adam optimizer is used with a learning rate of 0.001, a beta\_1 value of 0.87, and a beta\_2 value of 0.9995.
* Loss Function: Categorical cross-entropy is used as the loss function, which is appropriate for multiclass classification tasks.
* Metrics: The model is evaluated based on accuracy.

This code defines the architecture of a CNN model suitable for the classification of images and prepares it for training.



from visualkeras import layered\_view # Visualize the model layered\_view(model, legend=True, max\_xy=300)

This code block uses the visualkeras library to create a layered view of the defined Keras model. The layered view provides a visual representation of the model's architecture. Here's what this code does:

1. from visualkeras import layered\_view: It imports the layered\_view function from the visualkeras library.
2. layered\_view(model, legend=True, max\_xy=300): This line generates a visual representation of the model architecture. Here are the parameters used:
   * model: The Keras model that you want to visualize.
   * legend=True: It displays a legend that explains the different elements in the visual representation.
   * max\_xy=300: This parameter sets the maximum x and y coordinate values for positioning the layers in the visual representation. This value defines the size of the plot.

The visual representation generated by visualkeras provides a clear and intuitive way to understand the structure of the neural network, showing the layers, their connections, and the flow of data through the model. It can be particularly useful for model architecture visualization and presentation.



# addition of callbacks model\_es = EarlyStopping(monitor='loss', min\_delta=1e-9, patience=6, verbose=True) model\_rlr = ReduceLROnPlateau(monitor='val\_loss', factor=0.5, patience=5, verbose=True) model\_cp = ModelCheckpoint('best\_weights.h5', save\_best\_only=True, monitor='val\_accuracy') # Training the model history = model.fit(train\_dataset, epochs=epochs, batch\_size=batch\_size, validation\_data=test\_dataset, callbacks=[model\_es, model\_rlr, model\_cp])

In this code block, callbacks are added to the model and the model is trained. Callbacks are functions that can be applied at different stages during the training process to monitor or modify the behavior of the training. Here's an explanation of the code:

### Callbacks:

Three types of callbacks are defined and added to the training process:

1. model\_es (EarlyStopping):
   * This callback is configured with the following settings:
     + monitor='loss': It monitors the training loss.
     + min\_delta=1e-9: It specifies the minimum change in the monitored quantity (loss) to be considered as an improvement.
     + patience=6: If there is no improvement for 6 consecutive epochs, training will stop early.
     + verbose=True: It prints a message when early stopping is triggered.
2. model\_rlr (ReduceLROnPlateau):
   * This callback is configured with the following settings:
     + monitor='val\_loss': It monitors the validation loss.
     + factor=0.5: It reduces the learning rate by a factor of 0.5 when triggered.
     + patience=5: If there is no improvement in validation loss for 5 consecutive epochs, the learning rate is reduced.
     + verbose=True: It prints a message when the learning rate is reduced.
3. model\_cp (ModelCheckpoint):
   * This callback is configured to save the best model weights during training.
     + 'best\_weights.h5': It specifies the name of the file where the best model weights will be saved.
     + save\_best\_only=True: It saves only the best weights, determined based on validation accuracy.
     + monitor='val\_accuracy': It monitors the validation accuracy to determine the best weights.

### Training the Model:

The model.fit method is used to train the model. Here are the key parameters used:

* train\_dataset: The training dataset created earlier, which includes preprocessed images and one-hot encoded labels.
* epochs: The number of training epochs.
* batch\_size: The batch size for training.
* validation\_data: The testing dataset, used for validation during training.
* callbacks: The list of callbacks to apply during training, including early stopping, learning rate reduction, and model checkpointing.

The history variable is used to store training history, which includes metrics and losses over training epochs.

Overall, this code configures and trains the model while monitoring key metrics and making adjustments using the defined callbacks. The training process may stop early if there is no improvement, and the best model weights are saved. Learning rate reduction is also applied to potentially improve convergence.



from tensorflow.keras.models import load\_model # Load the trained model model = load\_model('/kaggle/working/best\_weights.h5')

This code block uses the load\_model function from TensorFlow's Keras API to load a previously trained model from a saved model file. Here's what the code does:

1. from tensorflow.keras.models import load\_model: It imports the load\_model function from TensorFlow's Keras models module. This function is used to load a pre-trained Keras model from a saved model file.
2. model = load\_model('/kaggle/working/best\_weights.h5'): This line loads the trained model from the file located at the specified path, which is '/kaggle/working/best\_weights.h5'. The loaded model is assigned to the variable model.

The loaded model can be used for various purposes, such as making predictions on new data, evaluating its performance, or fine-tuning it for further training. This is particularly useful when you want to reuse a previously trained model without having to retrain it from scratch.



import numpy as np import matplotlib.pyplot as plt import seaborn as sns from sklearn.metrics import confusion\_matrix from tensorflow.keras.models import load\_model def plot\_confusion\_matrix(cm, class\_names): plt.figure(figsize=(9, 9)) sns.heatmap(cm, annot=True, fmt="d", cmap=plt.cm.Blues, cbar=False, xticklabels=class\_names, yticklabels=class\_names) plt.title("Confusion Matrix") plt.xlabel("Predicted Class") plt.ylabel("True Class") plt.show() def generate\_confusion\_matrix(model, test\_dataset, class\_names): from sklearn.metrics import classification\_report # Assuming test\_dataset is a TensorFlow dataset true\_labels = [] predicted\_labels = [] for batch in test\_dataset: images, labels = batch # Predict classes for the batch predictions = model.predict(images, verbose=False) predicted\_classes = np.argmax(predictions, axis=1) # Convert one-hot encoded labels to class indices true\_classes = np.argmax(labels, axis=1) true\_labels.extend(true\_classes) predicted\_labels.extend(predicted\_classes) # Calculate confusion matrix cm = confusion\_matrix(true\_labels, predicted\_labels) # Print accuracy metrics using classification\_report class\_report = classification\_report(true\_labels, predicted\_labels, target\_names=class\_names) print("Classification Report:\n", class\_report) return cm

This code defines two functions for working with confusion matrices and evaluating the performance of a model on a test dataset. Here's an explanation of each function:

### plot\_confusion\_matrix Function:

This function is responsible for visualizing a confusion matrix using a heatmap. It takes two parameters: cm (the confusion matrix) and class\_names (the names of the classes). Here's how it works:

* It creates a figure with a size of (9, 9) to display the confusion matrix heatmap.
* The sns.heatmap function from the Seaborn library is used to plot the heatmap. The cm parameter provides the confusion matrix, and annot=True displays the values in each cell. The heatmap is colored using the Blues colormap.
* The xticklabels and yticklabels are set to class\_names to label the x-axis and y-axis with class names.
* The title, x-label, and y-label of the plot are set.
* Finally, the plot is displayed using plt.show().

### generate\_confusion\_matrix Function:

This function generates a confusion matrix and a classification report for a model's performance on a test dataset. It takes three parameters: model (the trained model), test\_dataset (the test dataset containing images and labels), and class\_names (the names of the classes). Here's how it works:

* It initializes two empty lists, true\_labels and predicted\_labels, to store the true and predicted class labels.
* It iterates over batches of data in the test\_dataset. For each batch:
  + It predicts the classes for the batch using the model.
  + It extracts true class labels from the one-hot encoded labels in the batch.
  + It appends the true and predicted class labels to their respective lists.
* After processing all batches, it calculates the confusion matrix using confusion\_matrix from Scikit-Learn, which takes the true labels and predicted labels as input.
* It also prints the classification report using classification\_report from Scikit-Learn, providing precision, recall, F1-score, and support for each class.
* The function returns the confusion matrix cm.

These functions are helpful for assessing the model's performance, visualizing class-wise accuracy, and gaining insights into classification results. The plot\_confusion\_matrix function provides a visual representation of the confusion matrix, while the generate\_confusion\_matrix function calculates the confusion matrix and classification report based on the model's predictions.



cm = generate\_confusion\_matrix(model, test\_dataset, list(tumor\_classes.keys()))

In this code, the generate\_confusion\_matrix function is called to calculate the confusion matrix and print a classification report based on the predictions made by the model on the test\_dataset. The results are stored in the cm variable. Here's what the code does:

1. cm = generate\_confusion\_matrix(model, test\_dataset, list(tumor\_classes.keys())): This line calls the generate\_confusion\_matrix function with the following parameters:
   * model: The trained model that you want to evaluate.
   * test\_dataset: The test dataset containing images and labels.
   * list(tumor\_classes.keys()): A list of class names, obtained from the tumor\_classes dictionary, which is used as class\_names in the function to label the confusion matrix.
2. The function generate\_confusion\_matrix processes the predictions and ground truth labels in the test dataset using the model. It calculates the confusion matrix and classification report.
3. The calculated confusion matrix is stored in the variable cm.

At this point, you have both the confusion matrix (cm) and a classification report for evaluating the model's performance on the test dataset. The confusion matrix provides insights into the model's classification accuracy for each class, while the classification report gives more detailed metrics, including precision, recall, F1-score, and support for each class.



plot\_confusion\_matrix(cm, list(tumor\_classes.keys()))

This code block calls the plot\_confusion\_matrix function to visualize the confusion matrix generated earlier. Here's what the code does:

1. plot\_confusion\_matrix(cm, list(tumor\_classes.keys())): This line calls the plot\_confusion\_matrix function with the following parameters:
   * cm: The confusion matrix that was previously generated using the generate\_confusion\_matrix function.
   * list(tumor\_classes.keys()): A list of class names, obtained from the tumor\_classes dictionary, which is used as class\_names in the function to label the confusion matrix.
2. The function plot\_confusion\_matrix visualizes the confusion matrix as a heatmap. It displays the true positive, true negative, false positive, and false negative values for each class, allowing you to assess the model's performance in classifying different types of tumors.

By running this code, you will see a heatmap that provides a visual representation of the confusion matrix, making it easier to interpret and understand the model's performance on a per-class basis. It allows you to quickly identify areas where the model is performing well and where it may be making classification errors.



# channnel map plot def plot\_channel\_activation\_maps(model, image, images\_per\_row=16, N=8, save=False): """ Function to visualize how the first N layers of the model observe the input image. Parameters: model (tensorflow.keras.models.Model): The Keras model for which to visualize the activation maps. image (numpy.ndarray): The input image for which to generate activation maps. images\_per\_row (int): Number of activation maps to display per row in the grid. N (int): Number of layers to visualize. save (bool): If True, save the plots as PDF files. Returns: None """ from tensorflow.keras.models import Model # Create a sub-model that outputs activations for the first N layers activation\_model = Model(inputs=model.input, outputs=[layer.output for layer in model.layers[:N]]) activations = activation\_model.predict(image) # Get the names of the layers for labeling the plots layer\_names = [layer.name for layer in model.layers[:N]] # Visualize the feature maps for each layer for layer\_name, layer\_activation in zip(layer\_names, activations): # This is the number of features in the feature map n\_features = layer\_activation.shape[-1] # The feature map has shape (1, size, size, n\_features) size = layer\_activation.shape[1] # We will tile the activation channels in this matrix n\_cols = n\_features // images\_per\_row display\_grid = np.zeros((size \* n\_cols, images\_per\_row \* size)) # We'll tile each filter into this big horizontal grid for col in range(n\_cols): for row in range(images\_per\_row): channel\_image = layer\_activation[0, :, :, col \* images\_per\_row + row] # Post-process the feature to make it visually palatable channel\_image -= channel\_image.mean() epsilon = 1e-8 # A small epsilon value to avoid division by zero channel\_std = channel\_image.std() + epsilon channel\_image /= channel\_std channel\_image \*= 64 channel\_image += 128 channel\_image = np.clip(channel\_image, 0, 255).astype('uint8') display\_grid[col \* size: (col + 1) \* size, row \* size: (row + 1) \* size] = channel\_image # Display the grid scale = 1. / size plt.figure(figsize=(scale \* display\_grid.shape[1], scale \* display\_grid.shape[0])) plt.title(layer\_name) plt.grid(False) plt.axis('off') plt.imshow(display\_grid, aspect='auto', cmap='viridis') if save: plt.savefig(f'plot\_channel\_activation\_maps\_{layer\_name}.pdf') plt.show()

This code defines a function plot\_channel\_activation\_maps that visualizes how the first N layers of a neural network model observe the input image by displaying activation maps. Here's an explanation of the function:

### Parameters:

* model (tensorflow.keras.models.Model): The Keras model for which you want to visualize the activation maps.
* image (numpy.ndarray): The input image for which you want to generate activation maps.
* images\_per\_row (int): The number of activation maps to display per row in the grid.
* N (int): The number of layers to visualize.
* save (bool): If True, the plots will be saved as PDF files.

### Function Description:

1. The function creates a sub-model (activation\_model) that takes the input image and returns the activations for the first N layers of the model. This allows you to extract intermediate layer activations for visualization.
2. It predicts the activations using the activation\_model on the input image.
3. For each layer, the function visualizes the feature maps by displaying them in a grid. Each layer's feature maps are displayed as a single image.
4. The function scales the activation values to make them visually palatable, and then displays the grid with the feature maps for the layer.
5. If the save parameter is set to True, the function saves each layer's activation map as a PDF file with a name that includes the layer's name.
6. Finally, the function displays the grid for each layer with their corresponding feature maps.

This function is useful for understanding how an image is processed through the layers of a deep neural network and observing what kind of features are activated at different layers. It can help in visualizing the "learned" features of the model.



path, label = test\_df.iloc[0, 0], test\_df.iloc[0, 1] print(path, label)

This code block extracts the path and label of the first sample in the test\_df DataFrame. Here's what the code does:

1. test\_df.iloc[0, 0] retrieves the value in the first row (index 0) and first column (index 0) of the test\_df DataFrame. This corresponds to the path of the first image in the test dataset.
2. test\_df.iloc[0, 1] retrieves the value in the first row (index 0) and second column (index 1) of the test\_df DataFrame. This corresponds to the label of the first image in the test dataset.
3. The path variable is assigned the path of the first image, and the label variable is assigned the label of the first image.
4. The values of path and label are then printed to the console, showing the path to the image file and the associated label. This allows you to verify and inspect the data for the first test sample.



# getting image to test output im = load\_img(path, target\_size=(150, 150), color\_mode="grayscale") im = img\_to\_array(im) # Reshape it to (1, 150, 150, 3) image\_tensor = np.expand\_dims(im, axis=0) print(f'x reshaped: {image\_tensor.shape}') # normilzation tensor image\_tensor /= np.max(image\_tensor) # ~ np.max(img\_tensor) # Display the class name print(f"Class name of the first image: {label}") array\_to\_img(im)

In this code block, the image corresponding to the path obtained earlier is loaded, preprocessed, and displayed. Here's a breakdown of what each part of the code does:

1. im = load\_img(path, target\_size=(150, 150), color\_mode="grayscale"): This line loads the image from the specified path. It resizes the image to a target size of (150, 150) pixels and specifies that the image is in grayscale mode. The loaded image is stored in the im variable.
2. im = img\_to\_array(im): This line converts the loaded image to a NumPy array, which can be processed by the model.
3. image\_tensor = np.expand\_dims(im, axis=0): The image array is reshaped to have a batch dimension of 1. This is done by adding a new axis at the beginning. The resulting image\_tensor has a shape of (1, 150, 150, 1), where the last dimension represents the number of color channels (in this case, 1 for grayscale).
4. image\_tensor /= np.max(image\_tensor): The pixel values in image\_tensor are normalized by dividing by the maximum pixel value. This step ensures that the pixel values are in the range [0, 1].
5. print(f'x reshaped: {image\_tensor.shape}'): This line prints the shape of the reshaped image tensor to verify its dimensions.
6. print(f"Class name of the first image: {label}"): The class name (label) of the first image is printed. This label was obtained from the test\_df DataFrame and corresponds to the class to which the image belongs.
7. array\_to\_img(im): Finally, the image is displayed using the array\_to\_img function. This function converts the NumPy array back to an image format for visualization.

The code loads, preprocesses, normalizes, and displays the first test image along with its associated class label. It prepares the image for input to the model for inference or testing.



plot\_channel\_activation\_maps(model=model, image=image\_tensor, N=5, save=False)

This code calls the plot\_channel\_activation\_maps function to visualize the activation maps for the first 5 layers of the neural network model when provided with the input image tensor. Here's what the code does:

1. plot\_channel\_activation\_maps(model=model, image=image\_tensor, N=5, save=False): This line calls the plot\_channel\_activation\_maps function with the following parameters:
   * model: The trained neural network model for which you want to visualize the activation maps.
   * image\_tensor: The input image tensor that represents the image you want to use for generating activation maps.
   * N=5: The parameter specifies that you want to visualize the activation maps for the first 5 layers of the model.
   * save=False: The save parameter is set to False, so the plots will not be saved as PDF files.
2. The function plot\_channel\_activation\_maps processes the model and input image to visualize the activation maps for the specified layers (in this case, the first 5 layers).
3. For each layer, the function visualizes the feature maps and displays them in a grid format, as described earlier.
4. The generated plots for each layer's activation maps are displayed in the output.

This code provides insights into how the input image is processed through the initial layers of the neural network model, allowing you to see the activations and features that are observed at each layer. It can be a useful tool for understanding the behavior of the model and how it extracts features from images during inference.

=======================================================================

import tensorflow as tf

from tensorflow.keras.preprocessing.image import load\_img, img\_to\_array

import numpy as np

# Define the path to the image you want to classify

image\_path = 'path\_to\_your\_image.jpg' # Replace with the actual image path

# Load the image and preprocess it

img = load\_img(image\_path, target\_size=(150, 150), color\_mode="grayscale")

img = img\_to\_array(img)

image\_tensor = np.expand\_dims(img, axis=0)

image\_tensor /= 255.0 # Normalize the image (if it's not already normalized)

# Load the trained model

model = tf.keras.models.load\_model('best\_weights.h5') # Replace with the path to your model

# Make a prediction with the loaded model

predictions = model.predict(image\_tensor)

# Map the prediction to a class label

class\_labels = ['Normal', 'Glioma', 'Meningioma', 'Pituitary'] # Replace with your class labels

predicted\_class = class\_labels[np.argmax(predictions)]

# Print the predicted class

print(f'Predicted class: {predicted\_class}')

=======================================================================