**Deep learning based brain tumor diagnosis**

**INTRODUCTION :**

**A brain tumor is a mass or growth of abnormal cells in the brain. The brain is a complex organ that controls many of our body's functions, and when there is a problem with the cells in the brain, it can lead to a range of symptoms and health problems.**

**Brain tumors can be either noncancerous or cancerous. They can develop from different types of brain cells and can occur in any part of the brain or spinal cord. Brain tumors can be primary, which means they originate in the brain, or secondary, which means they spread to the brain from other parts of the body.**

**here are some key points regarding brain tumor statistics worldwide:**

* **In 2020, there were an estimated 314,000 new cases of brain and nervous system tumors worldwide.**
* **Brain tumors account for about 2% of all cancers globally.**
* **The incidence of brain tumors varies by geographic region and by age, with the highest incidence rates observed in North America, Europe, and Australia.**
* **Malignant brain tumors account for about 40-45% of all brain tumors**
* **Brain tumors can affect people of all ages, but certain age groups are more at risk, with the highest incidence rates observed in people over the age of 65.**
* **The prognosis for brain tumor patients varies depending on factors such as the type, location, and stage of the tumor, as well as the patient's overall health and age.**

**PROBLEM STATEMENT :**

**The diagnosis and classification of brain tumors is a critical task for medical professionals in determining the appropriate treatment plan for patients. However, the current manual methods for diagnosing and classifying brain tumors are time-consuming, subjective, and rely heavily on the expertise of the medical professionals.**

**The aim of this project is to develop a computer-based system that can accurately and efficiently classify brain tumors using medical images such as magnetic resonance imaging (MRI) scans. The system should be able to differentiate between benign and malignant tumors and classify the tumor type based on its location and characteristics.**

**The proposed system can help medical professionals in making faster and more accurate diagnoses, which can lead to more effective treatment plans for patients. Furthermore, the development of an automated system can reduce the subjectivity and variability of human interpretations, leading to more consistent and reliable diagnoses.**

**ABOUT PROJECT :**

**The Brain Tumor Classification using CNN and BigDl project aims to develop a deep learning model to classify brain MRI scans into FOUR categories: brain tumors or healthy brain tissue. The project uses a Convolutional Neural Network (CNN) architecture, which is a type of deep neural network commonly used for image classification tasks.**

**The model is trained on a dataset of brain MRI scans and their corresponding labels (tumor or healthy). The dataset is preprocessed to ensure that all images are of the same size and that the tumor and healthy images are evenly distributed.**

**The CNN architecture used in this project consists of several layers, including convolutional layers, activation functions, pooling layers, and fully connected layers. The model is trained using the BigDL distributed deep learning framework, which is designed to scale deep learning applications across large clusters of machines.**

**Once the model is trained, it is used to predict the class of new brain MRI scans as either brain tumors or healthy brain tissue. The final output is a classification report, which provides information on the accuracy and performance of the model.**

**The Brain Tumor Classification using CNN and BigDl project has the potential to improve the accuracy and efficiency of brain tumor detection, which can ultimately lead to earlier diagnoses and better treatment outcomes for patients.**

**Why use this project??**

There are several reasons why a deep-learning based brain tumor diagnosis project using a big data enabled convolution neural network (CNN) approach can be useful:

Improved accuracy: Deep learning algorithms, particularly CNNs, have been shown to achieve high levels of accuracy in diagnosing brain tumors. By using a big data enabled CNN approach, the algorithm can be trained on a large dataset, which can improve the accuracy of the diagnosis.

Speed and efficiency: Deep learning algorithms can analyze medical images much faster than humans can. This can lead to quicker and more efficient diagnoses, which can be particularly beneficial in emergency situations.

Consistency: Deep learning algorithms are consistent in their diagnoses and do not suffer from fatigue or variability in their interpretations of medical images. This can lead to more reliable diagnoses, which can be particularly important in cases where a misdiagnosis could lead to severe consequences.

Scalability: A big data enabled CNN approach can be scaled to handle large amounts of data. As the size of the dataset grows, the accuracy of the algorithm can improve, leading to even better diagnoses.

Potential cost savings: By enabling more accurate and efficient diagnoses, a deep-learning based brain tumor diagnosis project using a big data enabled CNN approach could potentially lead to cost savings in the healthcare system. For example, it could reduce the need for unnecessary tests or procedures and lead to more timely interventions.

Overall, a deep-learning based brain tumor diagnosis project using a big data enabled CNN approach has the potential to improve patient outcomes, reduce healthcare costs, and advance our understanding of brain tumors.

**Target Audience?**

The target audience for a deep-learning based brain tumor diagnosis project using a big data enabled convolution neural network (CNN) approach includes a wide range of stakeholders, such as:

Healthcare professionals: Neurologists, radiologists, and other healthcare professionals involved in brain tumor diagnosis can benefit from this project by utilizing the technology to improve the accuracy and efficiency of their diagnoses.

Patients: Patients who are seeking a brain tumor diagnosis can benefit from this project by receiving faster and more accurate diagnoses, which can lead to earlier treatment and improved outcomes.

Researchers: Researchers studying brain tumors can use the data generated by this project to gain insights into the disease and potentially develop new treatments.

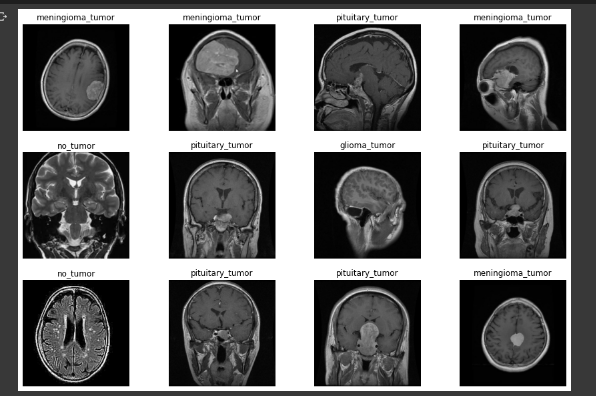
Healthcare organizations: Hospitals, clinics, and other healthcare organizations can benefit from this project by improving the quality of care they provide to patients, potentially reducing costs, and improving their reputation as leaders in the field of healthcare innovation.

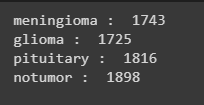
Technology companies: Technology companies developing deep learning algorithms, image analysis software, and other medical imaging technologies can use this project to test and refine their products, potentially leading to new advancements and market opportunities.

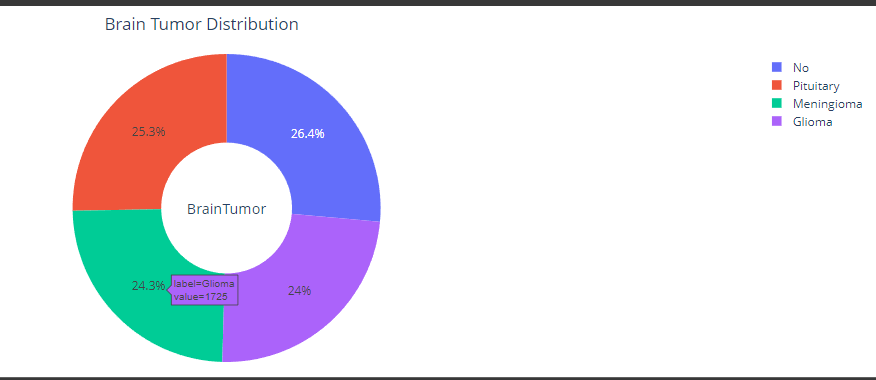
**About Dataset???**

****

****

****

****

****

**Balanced dataset.**

**Size 1.5 gb.**

**Resized to 150x150 RGB**

**Project Flow :**

**The flow of a deep-learning based brain tumor diagnosis project using a big data enabled convolution neural network (CNN) approach typically involves the following steps:**

1. **Data collection: The first step is to collect a large dataset of brain MRI images. The images should be labeled with the corresponding diagnosis, such as the type of tumor present or whether the image is of healthy brain tissue.**
2. **Data preprocessing: The dataset needs to be preprocessed before it can be used to train the CNN. This may involve tasks such as resizing the images, removing artifacts, and normalizing the pixel intensities.**
3. **CNN architecture selection: The next step is to select an appropriate CNN architecture. This may involve testing several different architectures to determine which one performs the best on the dataset.**
4. **CNN training: Once the CNN architecture has been selected, the model is trained on the labeled dataset. This process involves feeding the MRI images into the CNN and adjusting the model parameters to minimize the difference between the predicted diagnoses and the true diagnoses.**
5. **CNN testing: After the CNN is trained, it is tested on a separate dataset of brain MRI images to evaluate its performance.**
6. **Model optimization: The CNN model may be further optimized by adjusting hyperparameters such as the learning rate or number of layers.**
7. **Deployment: Once the CNN has been optimized and tested, it can be deployed in a clinical setting to analyze new brain MRI images and provide a diagnosis.**
8. **Monitoring and improvement: The performance of the CNN can be monitored over time to ensure that it continues to provide accurate diagnoses. The model can also be updated with new data to improve its performance over time.**

**CODE EXPLanation**

**This appears to be a code snippet for loading image data from two directories ("Training" and "Testing") and storing the images in arrays** X\_train **and** y\_train**.**

**The** os **module is used to navigate through the directories, and** cv2 **is used to read and resize the images. The** image\_size **variable is used to specify the size of the images after resizing.**

**The** X\_train **array is filled with the resized images, and the** y\_train **array is filled with the corresponding labels for each image (i.e., the type of tumor present or whether the image is of healthy brain tissue).**

**The final lines of code convert** X\_train **and** y\_train **to NumPy arrays for further processing.**

**The** plt.subplots **function creates a grid of four subplots, and the** fig **and** ax **variables are used to reference the figure and the individual subplots, respectively.**

**The loop iterates through each label and selects the first image in the training dataset that matches that label. The corresponding image is then displayed in the appropriate subplot using** imshow**, and the title of the subplot is set to the label of the image.**

**The** axis('off') **function is used to remove the axis labels from the subplots.**

**Overall, this code snippet is useful for visualizing the images in the training dataset and ensuring that they are correctly labeled.**

**This code snippet appears to be converting the categorical labels in** y\_train **and** y\_test **to one-hot encoded vectors using the** tf.keras.utils.to\_categorical() **function.**

**First, a new list called** y\_train\_new **is created to store the index of each label in the** labels **list. The** index() **method returns the index of the first occurrence of a given element in a list, so** labels.index(i) **returns the index of the current label** i **in the** labels **list.**

**Next,** y\_train\_new **is assigned to** y\_train**, effectively replacing the original labels with their corresponding indices.**

**The** to\_categorical() **function is then used to convert the integer-encoded labels to one-hot encoded vectors, which are required for training a neural network. The same process is repeated for** y\_test**.**

**The** weights **parameter is set to 'imagenet', which means that the model will be initialized with pre-trained weights on the ImageNet dataset. Using pre-trained weights can help improve the performance of the model, especially when the dataset is small.**

**The** include\_top **parameter is set to False, which means that the top layer(s) of the pre-trained model (i.e., the output layer(s) that generate class predictions) will be excluded. This allows us to add our own custom output layers for our specific classification task.**

**The** input\_shape **parameter is set to** (image\_size, image\_size, 3)**, which specifies the expected input size of the model. In this case,** image\_size **is a variable that represents the size of the images in the dataset (150x150), and** 3 **indicates that the images are in RGB format (i.e., they have 3 color channels).**

**This line of code is compiling the deep learning model for training using the** compile() **method.**

**The** loss **parameter is set to 'categorical\_crossentropy', which is a common loss function used for multi-class classification problems. It measures the difference between the predicted class probabilities and the true class probabilities, and aims to minimize this difference during training.**

**The** optimizer **parameter is set to 'Adam', which is an optimization algorithm commonly used for deep learning. It adapts the learning rate during training and can help the model converge faster and more reliably than other optimization algorithms.**

**The** metrics **parameter is set to 'accuracy', which is a metric used to evaluate the performance of the model during training and testing. It measures the proportion of correctly classified images out of all images in the dataset.**

**These lines of code are creating two callback functions for the** fit() **method in TensorFlow Keras.**

**The** TensorBoard **callback is used to write log data to a directory for visualization with TensorBoard. The** log\_dir **parameter specifies the directory to write the logs to.**

**The** ReduceLROnPlateau **callback is used to reduce the learning rate of the optimizer during training if the validation accuracy does not improve for a certain number of epochs (**patience **parameter). The** monitor **parameter specifies the metric to monitor, in this case 'val\_accuracy'. The** factor **parameter specifies the factor by which to reduce the learning rate when the callback is triggered, and the** min\_delta **parameter specifies the minimum change in the monitored quantity to qualify as an improvement. The** mode **parameter specifies whether to look for an increase or decrease in the monitored quantity, and the** verbose **parameter specifies the level of output.**

**This code block contains code for plotting the training and validation accuracy/loss over epochs. It uses the Matplotlib and Seaborn libraries for visualization.**

**The first line sets the variable** epochs **to a list of integers from 0 to 11, which represent the number of epochs (iterations) the model was trained for.**

**The** fig, ax = plt.subplots(1,2,figsize=(14,7)) **line initializes a Matplotlib figure with two subplots side by side, each with a size of 14x7 inches.**

**The next four lines extract the training and validation accuracy/loss data from the** history **object, which is the output of the** fit() **method used to train the model.**

**The** fig.text() **method is used to add a title to the figure.**

**The two** ax.plot() **lines plot the training and validation accuracy/loss data on their respective subplots. The** marker **and** markerfacecolor **arguments control the style of the markers used to plot the data points, while the** color **argument controls the color of the lines connecting the data points. The** label **argument specifies the label to be displayed in the legend.**

**The** ax.legend() **method is used to display the legend on each subplot.**

**The** ax.set\_xlabel() **and** ax.set\_ylabel() **methods are used to set the labels for the x and y axes on each subplot.**

**Finally, the** fig.show() **method displays the figure.**

**The** predict **method of the model is used to predict the class probabilities of the test data. The** argmax **function is used to get the class with the highest probability for each image.** y\_test\_new **is then created by using the** argmax **function to convert the one-hot encoded labels into their corresponding integer class labels.**

**STREAMLIT APP**

**python**

**Copy code**

**import streamlit as st st.title("Brain Tumor Classification") st.header("Convolutional Neural Network (CNN) for Brain Tumor Classification") st.text("Upload a brain MRI scan for image classification as Brain Tumor or Healthy")**

**This creates a Streamlit app with a title, header, and text description. The user is instructed to upload a brain MRI scan for classification.**

**This function loads a trained model from a given file path, resizes and preprocesses the input image to match the expected input size of the model, makes a prediction on the image, and returns the predicted label as an integer.**

**Here's a breakdown of the function:**

* img**: the input image that will be classified**
* weights\_file**: the file path of the trained model**

**Steps:**

1. **The trained model is loaded using** tf.keras.models.load\_model**.**
2. **The input image is converted from RGB to BGR format using** cv2.cvtColor**.**
3. **The image is then resized to the expected input size of the model (150x150) using** cv2.resize**.**
4. **The image is reshaped to a 4D tensor with dimensions (1,150,150,3) to match the expected input shape of the model.**
5. **The model's** predict **method is called on the input image to generate predictions.**
6. np.argmax **is used to find the index of the predicted class with the highest probability. This index is returned as the predicted label.**

**Layers:**

input\_1 **is the name of the input layer in the Keras model. It is an input layer that expects the input shape of** (None, 150, 150, 3)**, where** None **indicates that the batch size can be of any size, and** 150, 150, 3 **is the shape of the input image. This means that the model expects images of height and width 150 pixels with 3 channels (RGB). The input layer doesn't perform any computations, but it just specifies the input shape of the model.**

rescaling **is a preprocessing layer that normalizes the input data to a specified range. In this case, the layer rescales each pixel value in the input image to be in the range [0,1]. This is done by dividing each pixel value by 255, which is the maximum pixel value that can be represented by an 8-bit image. The output of the** rescaling **layer is an image tensor with pixel values in the range [0,1].**

Normalization **is a layer that performs featurewise normalization of the input data. It normalizes each input feature (channel in this case) separately by subtracting the mean and dividing by the standard deviation computed over the training dataset.**

**The arguments for the** Normalization **layer are** mean **and** variance**, which specify the means and variances used for featurewise normalization. In this case, the mean is** (0.485, 0.456, 0.406) **and the variance is** (0.229, 0.224, 0.225)**. These values are computed from the ImageNet dataset and used for transfer learning.**

**After the normalization layer, the input data is scaled to have zero mean and unit variance for each feature (channel), which can help the model converge faster during training.**

**It seems that the layer names have changed slightly from the previous example. Assuming that** rescaling **is the first layer and** normalization **is the second layer, the next layer would be the following:**

stem\_conv\_pad (ZeroPadding2D) (None, 152, 152, 3) 0 ['normalization[0][0]']

**This layer is a zero padding layer that pads the input tensor with zeros on both sides of the height and width dimensions, to ensure that the convolutional layer that follows preserves the spatial dimensions of the input tensor. The output tensor has shape** (None, 152, 152, 3)**, where** None **represents the batch size.**

**This layer is a convolutional layer with 32 filters, each of size 3x3, and a stride of 2. The layer takes the output from the previous layer and applies the convolution operation, resulting in 32 feature maps of size 75x75. The weights of this layer are learned during training.**

**BatchNormalization is a layer in a neural network that normalizes the inputs from the previous layer at each batch by subtracting the batch mean and dividing by the batch standard deviation. This helps to stabilize the learning process and reduce the dependence of the network on the scale of the input data. The stem\_bn layer in this case is normalizing the output from the stem\_conv layer before it is passed on to the next layer.**

**This is an activation layer in the EfficientNet model, which applies an activation function to the output of a batch normalization layer. Specifically, it applies the ReLU activation function to the output of the previous batch normalization layer** block2b\_bn**. The output of this layer has a shape of** (None, 38, 38, 144)**.**

**These lines of code define a new model by adding a few layers on top of the pre-trained EfficientNet model.**

GlobalAveragePooling2D **layer reduces the spatial dimensions of the previous layer to 1x1 and averages the values across each channel.**

Dropout **layer randomly sets a fraction of the input units to zero during training, which helps to prevent overfitting.**

Dense **layer is a fully connected layer that maps the output of the previous layer to the number of classes in the classification problem, which in this case is 4 (Glioma Tumor, No Tumor, Meningioma Tumor, Pituitary Tumor).**

**Finally, a new** Model **object is created with the input and output layers specified as the input and output of the EfficientNet model and the newly added layers, respectively.**

**Pooling is a downsampling operation that reduces the dimensionality of feature maps in a neural network. It is often used in conjunction with convolutional layers in convolutional neural networks (CNNs). Pooling works by dividing the input feature map into non-overlapping regions and taking the maximum, average, or sum of each region. This reduces the size of the feature map while retaining important information, making it computationally more efficient to process by subsequent layers. Common types of pooling include max pooling, average pooling, and min pooling.**

**Zero padding is a technique used in convolutional neural networks (CNNs) to preserve the spatial dimensions of the input volume after convolutional layers. It involves adding zeros to the border of the input volume to increase its size, so that the output volume has the same spatial dimensions as the input volume. This is important because as the convolutional layers reduce the spatial dimensions of the input volume, without padding, the spatial dimensions of the output volume will decrease with each layer. Zero padding helps avoid this problem and preserve the spatial information of the input volume throughout the network.**

**What is the difference between Spark session and Spark context?**

**In earlier versions of Spark or Pyspark, SparkContext was an entry point for programming with RDD and connecting to Spark Cluster. With the introduction of Spark 2.0 SparkSession, it became an entry point for programming with DataFrame and Dataset**

**This code defines a neural network model using the Keras API, which is a high-level neural network API that runs on top of TensorFlow.**

**The model architecture consists of three layers:**

1. **The first layer is an input layer with 13 nodes, which corresponds to the number of features in the input data.**
2. **The second layer is a fully connected layer with 14 nodes and ReLU activation function.**
3. **The third layer is another fully connected layer with 8 nodes and ReLU activation function.**
4. **The fourth and final layer is a fully connected layer with 2 nodes and no activation function.**

**The Model function in Keras is used to create a model object that takes the input and output layers as arguments. In this case, the input layer is x1 and the output layer is dense3. The resulting model object can be used to train and evaluate the model using methods such as fit() and evaluate().**

**This model is suitable for a binary classification task where the output should be one of two possible classes. The output layer has two nodes with no activation function, which means that the output will be a vector of two values representing the model's confidence for each class. The class with the highest confidence will be the predicted class for a given input.**

**This code creates an instance of a neural network classifier using a pre-defined Keras model object and sets various hyperparameters for the training process.**

* **NNClassifier is likely a custom class or function that takes the following arguments:**
  + model**: the Keras model object to be used for classification**
  + CrossEntropyCriterion()**: a loss function used to compute the difference between the predicted and actual output during training**
  + [13]**: a list specifying the input dimensions of the model**
* setOptimMethod(Adam())**: sets the optimization algorithm to Adam, which is a popular gradient descent optimization algorithm for neural networks**
* setBatchSize(32)**: sets the batch size to 32, meaning that the model will be trained on 32 input-output pairs at a time**
* setMaxEpoch(150)**: sets the maximum number of epochs (iterations over the entire training dataset) to 150, meaning that the model will train for up to 150 epochs or until early stopping criteria are met.**

**Overall, this code sets up a neural network classifier using a pre-defined Keras model object and sets various hyperparameters for the training process. The resulting classifier can be used to train on a dataset and make predictions on new data.**

**This code performs validation of the neural network classifier by making predictions on a validation dataset and computing the accuracy of the model.**

* nnModel **is likely an instance of the neural network classifier created previously.**
* transform(validationDF) **applies the trained model to the validation dataset** validationDF **to generate predictions. The resulting DataFrame** predictionDF **contains the predicted labels for each input in the validation dataset.**
* cache() **is used to cache the** predictionDF **DataFrame in memory to speed up subsequent operations that use it.**
* sample(False, 0.1) **selects a random sample of 10% of the rows in the** predictionDF **DataFrame and shows them in the console.**
* MulticlassClassificationEvaluator **is a class from the PySpark MLlib library used to evaluate the classification model's accuracy.**
* labelCol="label", predictionCol="prediction", metricName="accuracy" **sets up the evaluator to compute the accuracy of the model's predictions, using the** label **and** prediction **columns from the** predictionDF **DataFrame.**
* evaluate(predictionDF) **computes the accuracy of the model by comparing the predicted labels in** predictionDF **with the actual labels in** validationDF**.**
* print("Accuracy = %g " % (accuracy)) **displays the accuracy of the model in the console.**

**Overall, this code validates the performance of the neural network classifier on a validation dataset and reports the accuracy of the model.**