SOURCE CODE PREPARATION

 **os**: This module provides a way to interact with the operating system. You can use it to read or write files, navigate directories, and more.

 **numpy**: This is a powerful library for numerical computations in Python. It allows you to work with arrays and matrices, and perform mathematical operations on them.

 **pandas**: This library is used for data manipulation and analysis. It provides data structures like DataFrames to store and manipulate large datasets easily.

 **seaborn**: This is a statistical data visualization library based on matplotlib. It provides a high-level interface for drawing attractive and informative statistical graphics.

 **matplotlib.pyplot**: This is a plotting library used for creating static, interactive, and animated visualizations in Python. It's often used together with numpy and pandas.

 **mplcyberpunk**: This is a style for matplotlib that gives your plots a "cyberpunk" aesthetic, with bright neon colors and a dark background.

 **cv2 (OpenCV)**: This is an open-source computer vision and machine learning software library. It contains tools for processing images and videos.

 **tensorflow as tf**: TensorFlow is an open-source machine learning framework developed by Google. It provides tools for building and training machine learning models.

 **sklearn.utils.shuffle**: This function is part of the scikit-learn library and is used to shuffle datasets.

 **sklearn.model\_selection.train\_test\_split**: This function splits your dataset into training and testing sets.

 **tensorflow.keras.models.Sequential**: This is a type of model in Keras (a high-level neural networks API), where you can stack layers sequentially to build a neural network.

 **tensorflow.keras.layers**:

* **GlobalMaxPooling2D**: This layer performs max pooling operation on spatial data.
* **GlobalAveragePooling2D**: This layer performs average pooling operation on spatial data.
* **Dropout**: This layer helps prevent overfitting by randomly setting a fraction of input units to 0 during training.
* **Dense**: This is a fully connected neural network layer.
* **BatchNormalization**: This layer helps normalize the inputs of a layer, speeding up training and reducing sensitivity to network initialization.

 **tensorflow.keras.callbacks**:

* **EarlyStopping**: Stops training when a monitored metric has stopped improving.
* **ReduceLROnPlateau**: Reduces learning rate when a metric has stopped improving.
* **TensorBoard**: Provides a visualization tool for TensorFlow.
* **ModelCheckpoint**: Saves the model after every epoch.

 **sklearn.metrics**:

* **classification\_report**: Generates a report of classification metrics.
* **confusion\_matrix**: Creates a confusion matrix to evaluate the accuracy of a classification.

 **tqdm**: This library is used to create progress bars, making it easy to see how long a loop or process will take.

 **random**: This module implements pseudo-random number generators for various distributions, useful for random sampling and random number generation.

IMPORTING DATASET AND PREPROCESSING IT :

 **Initialize Lists and Variables**:

* x\_train = []: An empty list to store training images.
* y\_train = []: An empty list to store labels for the training images.
* labels = ['glioma\_tumor', 'no\_tumor', 'meningioma\_tumor', 'pituitary\_tumor']: A list of labels for different types of brain tumors and the 'no tumor' category.
* image\_size = 150: The desired size (150x150 pixels) to which all images will be resized.

 **Loading and Processing Training Images**:

* For each label in the labels list:
  + Construct the folder path where training images for that label are stored.
  + For each image file in the folder:
    - Read the image using cv2.imread().
    - Resize the image to 150x150 pixels using cv2.resize().
    - Append the resized image to x\_train.
    - Append the corresponding label to y\_train.

 **Loading and Processing Testing Images**:

* Repeat the same process for testing images:
  + Construct the folder path where testing images for that label are stored.
  + For each image file in the folder:
    - Read the image using cv2.imread().
    - Resize the image to 150x150 pixels using cv2.resize().
    - Append the resized image to x\_train.
    - Append the corresponding label to y\_train.
* **x\_train = np.array(x\_train)**: This line converts the list of training images (x\_train) into a numpy array. Numpy arrays are more efficient and easier to work with for numerical and machine learning operations.
* **y\_train = np.array(y\_train)**: This line converts the list of labels (y\_train) into a numpy array. Like the images, using numpy arrays for labels makes them easier to manage and process during machine learning tasks.
* **a = len(x\_train)**: This line calculates the number of images in the x\_train array and stores it in the variable a.
* **print(a)**: This line prints the value of a, which is the total number of images in your training dataset.

 **b = random.randint(0, a)**: This line generates a random integer b between 0 and a-1 (where a is the total number of images in x\_train). This random number b will be used to select a random image from the dataset.

* **print(y\_train[b])**: This line prints the label of the randomly selected image. y\_train[b] gives you the label corresponding to the image at index b.
* **print(plt.imshow(x\_train[b]))**: This line displays the randomly selected image using matplotlib.pyplot. plt.imshow(x\_train[b]) shows the image at index b in the x\_train array.
* **label\_counts = {label: np.sum(y\_train == label) for label in labels}**: This line creates a dictionary called label\_counts. For each label in the labels list, it calculates the number of times that label appears in y\_train. It uses np.sum(y\_train == label) to count the occurrences of each label.
* **print(label\_counts)**: This line prints the label\_counts dictionary, which shows how many images there are for each label in the dataset.

DATA VISUALIZATION OF LABEL COUNTS:

1. **Set up the figure**:
   * plt.figure(figsize=(8,6)): This line creates a new figure for plotting with a specified size of 8 inches by 6 inches.
2. **Create the bar chart**:
   * colors=["red","purple","blue","green"]: This line defines a list of colors to use for the bars in the bar chart.
   * plt.subplot(2,1,1): This line sets up a subplot layout with 2 rows and 1 column, and activates the first subplot.
   * bars=plt.bar(label\_counts.keys(), label\_counts.values(), color=colors): This line creates a bar chart with the keys of label\_counts (tumor types) on the x-axis and their corresponding values (counts) on the y-axis. Each bar is colored using the colors list.
   * mplcyberpunk.add\_bar\_gradient(bars=bars): This line adds a gradient effect to the bars using the mplcyberpunk style.
   * plt.ylabel('Counts'): This line sets the label for the y-axis as 'Counts'.
   * plt.title('Distribution of Labels'): This line sets the title of the plot as 'Distribution of Labels'.
3. **Display sample images**:
   * k=0: This initializes a counter k to keep track of subplot positions.
   * For each label in labels:
     + j=0: This initializes a counter j to iterate through the images.
     + The while True loop:
       - if(y\_train[j]==i): This checks if the label of the current image matches the current label.
       - plt.subplot(2,4,k+5): This sets up a subplot in the second row (positioned from 5 to 8) to display the image.
       - plt.imshow(x\_train[j]): This displays the image corresponding to the label.
       - plt.axis('off'): This turns off the axis for the image.
       - k+=1: This increments the counter to move to the next subplot.
       - break: This breaks out of the loop once an image with the current label is found.
     + j+=1: This increments the counter to move to the next image.
4. **Adjust layout and show the plot**:
   * plt.tight\_layout(): This adjusts the subplot parameters to give some padding and prevent overlap.
   * plt.show(): This displays the entire figure with the bar chart and sample images.

SHUFFLING THE DATA:

 **x\_train**: The array of training images.

 **y\_train**: The array of labels corresponding to the training images.

 **random\_state=101**: This is a seed value that ensures the shuffling process is reproducible. If you use the same seed value again, you'll get the same shuffled order.

SPLITTING THE DATA:

* **x\_train, x\_test, y\_train, y\_test = train\_test\_split(x\_train, y\_train, test\_size=0.2, random\_state=101)**: This line splits your data into training and testing sets.
  + **x\_train**: The array of training images after the split.
  + **x\_test**: The array of testing images after the split.
  + **y\_train**: The array of training labels after the split.
  + **y\_test**: The array of testing labels after the split.
  + **test\_size=0.2**: This specifies that 20% of the data should be used for testing, and the remaining 80% will be used for training.
  + **random\_state=101**: This is a seed value that ensures the split is reproducible. If you use the same seed value again, you'll get the same split.
* **y\_train\_new = [labels.index(i) for i in y\_train]**: This line creates a new list, y\_train\_new, where each label in y\_train is replaced with its corresponding index from the labels list. Essentially, it converts the string labels into numerical indices.
* **y\_test\_new = [labels.index(i) for i in y\_test]**: Similarly, this line creates a new list, y\_test\_new, where each label in y\_test is replaced with its corresponding index from the labels list.
* **print(y\_train\_new[0])**: This line prints the first element of the y\_train\_new list, which is the numerical index of the first label in the original y\_train list.

ONE HOT ENCODING:

 **y\_train = tf.keras.utils.to\_categorical(y\_train\_new, num\_classes=len(labels))**: This line converts the list y\_train\_new of numerical labels into a one-hot encoded format. Each label is represented as a binary vector where only the index corresponding to the label is 1, and all other indices are 0. The num\_classes=len(labels) parameter specifies the number of different labels.

 **y\_test = tf.keras.utils.to\_categorical(y\_test\_new, num\_classes=len(labels))**: Similarly, this line converts the list y\_test\_new of numerical labels into a one-hot encoded format.

 **print(y\_train[0])**: This line prints the first one-hot encoded label in the y\_train array.

**print(np.argmax(y\_train[0]))**: This line finds the index of the maximum value in the first one-hot encoded label vector in y\_train and prints it.

**np.argmax(y\_train[0])**: This function returns the index of the highest value in the one-hot encoded vector y\_train[0]. Since the vector is one-hot encoded, the highest value (1) indicates the original label's index.

BUILDING THE DEEP LEARNING MODEL:

 **Load the EfficientNetB0 Model**:

* efficientnetB0 = tf.keras.applications.EfficientNetB0(weights='imagenet', include\_top=False, input\_shape=(image\_size, image\_size, 3)): This line loads the EfficientNetB0 model pretrained on the ImageNet dataset. It does not include the top layers (the classification layers), and the input shape is set to match your images (150x150 pixels with 3 color channels).

 **Build the Custom Model**:

* model = efficientnetB0.output: This gets the output of the EfficientNetB0 base model.
* model = tf.keras.layers.GlobalAveragePooling2D()(model): This adds a Global Average Pooling layer to reduce the spatial dimensions of the output from the EfficientNetB0 model.
* model = tf.keras.layers.Dense(1024, activation='relu')(model): This adds a Dense (fully connected) layer with 1024 units and ReLU activation, which introduces non-linearity.
* model = tf.keras.layers.Dropout(rate=0.4)(model): This adds a Dropout layer with a rate of 0.4, which helps prevent overfitting by randomly setting 40% of the input units to 0 during training.
* model = tf.keras.layers.Dense(4, activation='softmax')(model): This adds a final Dense layer with 4 units (one for each class) and softmax activation, which outputs a probability distribution over the 4 classes.
* model = tf.keras.models.Model(inputs=efficientnetB0.input, outputs=model): This creates a new model by specifying the input from the EfficientNetB0 base and the output from the custom layers added.

 **Compile the Model**:

* model.compile(optimizer='adam', loss='categorical\_crossentropy', metrics=['accuracy']): This compiles the model with the Adam optimizer, categorical cross-entropy loss (suitable for multi-class classification), and accuracy as a metric to evaluate the model's performance.

MODEL BUILDING:

 The layers in the model, their types, and names.

 The output shape of each layer.

 The number of parameters (weights) in each layer and the total number of parameters in the model.

* **reduce\_lr = ReduceLROnPlateau(monitor='val\_accuracy', factor=0.1, patience=2, min\_delta=0.0001, verbose=1)**: This line creates a callback that adjusts the learning rate during training.

Parameters explained:

* **monitor='val\_accuracy'**: This tells the callback to monitor the validation accuracy.
* **factor=0.1**: When triggered, this reduces the learning rate by a factor of 0.1 (i.e., it will be multiplied by 0.1).
* **patience=2**: This specifies that the learning rate will be reduced if the validation accuracy does not improve for 2 consecutive epochs.
* **min\_delta=0.0001**: This is the minimum change in validation accuracy that will be considered as an improvement.
* **verbose=1**: This enables the display of messages when the learning rate is reduced.

TRAINING THE MODEL:

* **history = model.fit(x\_train, y\_train, validation\_split=0.1, epochs=12, verbose=1, batch\_size=32, callbacks=[reduce\_lr])**: This line trains the model on the training data.

Parameters explained:

* **x\_train**: The training images.
* **y\_train**: The one-hot encoded labels for the training images.
* **validation\_split=0.1**: This reserves 10% of the training data for validation, which is used to evaluate the model's performance during training.
* **epochs=12**: This specifies that the model will be trained for 12 iterations over the entire training dataset.
* **verbose=1**: This enables the display of progress messages during training.
* **batch\_size=32**: This specifies that the training data will be divided into batches of 32 samples each. The model will be updated after each batch.
* **callbacks=[reduce\_lr]**: This includes the ReduceLROnPlateau callback to adjust the learning rate if the validation accuracy plateaus.

**\*model.save('efficientnetB0.h5')**: This line saves the trained model to a file named 'efficientnetB0.h5'.

PLOTTING THE GRAPH FOR LOSS AND ACCURACY:

* **plt.figure(figsize=(12, 5))**: This line creates a figure with a specified size of 12 inches wide and 5 inches tall.

1. **Plotting Training and Validation Loss**:
   * plt.subplot(1, 2, 1): This sets up a subplot grid with 1 row and 2 columns, and activates the first subplot.
   * plt.plot(history.history['loss'], label='Train Loss'): This plots the training loss values stored in history.history['loss'] and labels it as 'Train Loss'.
   * plt.plot(history.history['val\_loss'], label='Validation Loss'): This plots the validation loss values stored in history.history['val\_loss'] and labels it as 'Validation Loss'.
   * plt.title('Training and Validation Loss'): This sets the title of the plot.
   * plt.xlabel('Epoch'): This labels the x-axis as 'Epoch', representing the number of training iterations.
   * plt.ylabel('Loss'): This labels the y-axis as 'Loss', representing the amount of error in predictions.
   * plt.legend(): This displays a legend to differentiate between 'Train Loss' and 'Validation Loss'.
   * plt.grid(True): This adds a grid to the plot for better readability.
   * mplcyberpunk.make\_lines\_glow(): This adds a cyberpunk-style glow effect to the lines in the plot.
2. **Plotting Training and Validation Accuracy**:
   * plt.subplot(1, 2, 2): This activates the second subplot in the same grid setup.
   * plt.plot(history.history['accuracy'], label='Train Accuracy'): This plots the training accuracy values stored in history.history['accuracy'] and labels it as 'Train Accuracy'.
   * plt.plot(history.history['val\_accuracy'], label='Validation Accuracy'): This plots the validation accuracy values stored in history.history['val\_accuracy'] and labels it as 'Validation Accuracy'.
   * plt.title('Training and Validation Accuracy'): This sets the title of the plot.
   * plt.xlabel('Epoch'): This labels the x-axis as 'Epoch'.
   * plt.ylabel('Accuracy'): This labels the y-axis as 'Accuracy', representing the percentage of correct predictions.
   * plt.legend(): This displays a legend to differentiate between 'Train Accuracy' and 'Validation Accuracy'.
   * plt.grid(True): This adds a grid to the plot.
   * mplcyberpunk.make\_lines\_glow(): This adds a cyberpunk-style glow effect to the lines in the plot.

* **plt.tight\_layout()**: This adjusts the spacing between subplots to prevent overlap.
* **plt.show()**: This displays the entire figure with both subplots.

**CONFUSION MATRIX**:

* **y\_true\_test = np.argmax(y\_test, axis=1)**: This line converts the one-hot encoded labels (y\_test) back into their original numerical labels by selecting the index of the highest value in each row.
* **y\_pred\_test = np.argmax(model.predict(x\_test), axis=1)**: This line makes predictions on the testing data (x\_test) using the trained model (model.predict(x\_test)) and then converts the predicted probabilities into numerical labels by selecting the index of the highest value in each row.
* **conf = confusion\_matrix(y\_true\_test, y\_pred\_test)**: This line computes the confusion matrix using the true labels (y\_true\_test) and the predicted labels (y\_pred\_test).
* **print(f'Confusion matrix:\n{conf}')**: This line prints the confusion matrix, which is a table that summarizes the performance of a classification model. It shows the number of true positives, true negatives, false positives, and false negatives for each class.

CLASSIFICATION REPORT:

* **print(classification\_report(y\_true\_test, y\_pred\_test))**: This line prints a classification report that summarizes the performance of a classification model.

The classification report includes:

* **Precision**: The proportion of true positive predictions among all positive predictions (higher is better).
* **Recall**: The proportion of true positive predictions among all actual positives (higher is better).
* **F1-score**: The harmonic mean of precision and recall, providing a single metric to evaluate the model's performance (higher is better).
* **Support**: The number of occurrences of each class in the true labels (y\_true\_test).
* **Accuracy**: The proportion of correct predictions out of all predictions made.

 **x = len(x\_test)**: This line calculates the number of samples in the x\_test array (testing data) and stores it in the variable x.

 **print(x)**: This line prints the value stored in x, which represents the total number of samples in your testing dataset.

CALCULATING CONFIDENCE VALUE FOR RANDOM DATASET:

 **Generating a Random Image and Making Predictions**:

* **random\_index = np.random.randint(0, x)**: This line generates a random index within the range of the number of samples in the testing dataset (x\_test).
* **random\_img = x\_test[random\_index]**: This selects a random image (random\_img) from the testing dataset using the randomly generated index.
* **predictions = model.predict(random\_img.reshape(1, 150, 150, 3))**: This line makes predictions on the random image by reshaping it to the expected input shape (1, 150, 150, 3) and using the trained model (model.predict).

 **Interpreting the Model's Predictions**:

* **predicted\_class = np.argmax(predictions)**: This gets the index of the class with the highest predicted probability (predictions).
* **predicted\_label = labels[predicted\_class]**: This converts the predicted class index into a human-readable label using the labels list.
* **confidence = predictions[0][predicted\_class]**: This calculates the confidence or probability score of the predicted class.

 **Getting Actual Label and Displaying Information**:

* **actual\_index = y\_test[random\_index]**: This gets the one-hot encoded actual class from y\_test corresponding to the random image.
* **actual\_class = np.argmax(actual\_index)**: This gets the index of the actual class from the one-hot encoded vector.
* **actual\_label = labels[actual\_class]**: This converts the actual class index into a human-readable label using the labels list.

 **Displaying the Image and Prediction Information**:

* **print(f"\033[94mPredicted label: {predicted\_label}\033[0m \n\033[92mActual label: {actual\_label}\033[0m \n\033[93mConfidence: {confidence\*100:.2f}%\033[0m\n")**: This prints the predicted label, actual label, and confidence level in a formatted manner with color-coded text.
* **plt.figure(figsize=(3, 3))**: This sets up a figure for plotting with a size of 3 inches by 3 inches.
* **plt.imshow(random\_img)**: This displays the random image (random\_img).
* **plt.axis('off')**: This removes axis labels from the plot for better visualization.

 **plt.show()**: This displays the plot with the random image and prediction information.

TESTING THE MODEL:

 **test\_loss, test\_accuracy = model.evaluate(x\_test, y\_test, verbose=1)**: This line evaluates the trained model (model) on the testing dataset (x\_test, y\_test).

 **verbose=1**: This parameter determines whether to display progress messages during evaluation.

 **test\_accuracy\*100**: This calculates the accuracy of the model on the testing dataset and converts it into a percentage.

 **print(f"Test Accuracy: {test\_accuracy\*100:.2f}%")**: This line prints the test accuracy of the model, formatted to two decimal places for clarity.