BRAIN_TUMOR_PREDICTION

1. Importing System Libraries

```
import os

import itertools
```

- os: This module allows interaction with the operating system, such as file handling, directory navigation, and environment variables.
- itertools: This module provides functions for creating iterators for efficient looping, such as permutations, combinations, and infinite iterators.

---

2. Importing Data Handling Tools

```
import numpy as np

import pandas as pd

import seaborn as sns
```

- numpy (np): A powerful numerical computing library used for handling arrays, performing mathematical operations, and handling large datasets efficiently.
- pandas (pd): A popular library for data manipulation and analysis, providing data structures like DataFrames and Series for organizing and analyzing structured data.
- seaborn (sns): A statistical data visualization library built on top of matplotlib, providing a high-level interface for drawing attractive graphs.

```
sns.set_style('darkgrid')
```

- This sets the style of seaborn plots to 'darkgrid', which improves visibility by adding a grid with a dark background.

---

3. Importing Data Visualization Tools

```
import matplotlib.pyplot as plt
```

- matplotlib.pyplot (plt): A plotting library used to create static, animated, and interactive visualizations.

---

4. Importing Machine Learning Tools

```
from sklearn.model_selection import train_test_split
```

from sklearn.metrics import confusion_matrix, classification_report

- train_test_split: Splits a dataset into training and testing sets for model evaluation.
- confusion_matrix: A table used to evaluate the performance of a classification model by comparing predicted vs. actual values.
- classification_report: A summary report of a classification model's precision, recall, f1-score, and accuracy.

---

5. Importing TensorFlow and Keras (Deep Learning Frameworks)

import tensorflow as tf

from tensorflow import keras

- tensorflow (tf): An open-source machine learning and deep learning library developed by Google for building neural networks.
- keras: A high-level API for building deep learning models in TensorFlow.

from tensorflow.keras.models import Sequential

- Sequential: A Keras model type where layers are added sequentially, meaning each layer has a single input and output.

from tensorflow.keras.optimizers import Adamax

- Adamax: A variation of the Adam optimizer that is stable for large datasets and works well with deep learning models.

from tensorflow.keras.preprocessing.image import ImageDataGenerator

- ImageDataGenerator: A Keras utility for augmenting image data by applying transformations like rotation, zooming, flipping, and shifting to improve model generalization.

---

6. Importing Layers for Building a Neural Network

from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense

- Conv2D: A convolutional layer that extracts features from an image by applying filters (kernels).
- MaxPooling2D: A pooling layer that reduces the spatial dimensions of an image while retaining important features.
- Flatten: Converts a multi-dimensional array (like images) into a one-dimensional vector to be fed into a fully connected layer.

- Dense: A fully connected layer where each neuron is connected to every neuron in the previous layer.

---

7. Suppressing Warnings

import warnings

warnings.filterwarnings('ignore')

- This prevents warning messages from appearing in the output, making the code cleaner to read.

8. Importing predict Function

from predict import predict

- This imports a function named predict from a module named predict. This function is likely used for making predictions with a trained model.

1. Define the Training Data Directory

train_data_dir = 'Training'

- This specifies that the training data is stored in a folder named "Training".

---

2. Initialize Lists for File Paths and Labels

filepaths = []

labels = []

- filepaths: A list that will store the full path of each image file in the training dataset.
- labels: A list that will store the class (category) for each image.

---

3. Get the List of Class Folders

folds = os.listdir(train_data_dir)

- os.listdir(train_data_dir): Lists all folders inside "Training".
- Each folder represents a different class in a classification problem.
  For example, if "Training" contains:

4. Loop Through Each Class Folder and Store File Paths & Labels

for fold in folds:

    foldpath = os.path.join(train_data_dir, fold)

- foldpath creates the full path to the class folder.

5. Convert Lists into a Pandas DataFrame

Fseries = pd.Series(filepaths, name='filepaths')

Lseries = pd.Series(labels, name='labels')

- Creates two Pandas **Series** objects:
    - Fseries: Stores file paths.
    - Lseries: Stores corresponding labels.

train_df = pd.concat([Fseries, Lseries], axis=1)

- Combines both series into a **DataFrame** where:
    - Column **filepaths** contains image paths.
    - Column **labels** contains the corresponding category.

train_df

OUTPUT:-

```
[31]: train_df
```

[31]:

	filepaths	labels
0	Training\glioma_tumor\gg (1).jpg	glioma_tumor
1	Training\glioma_tumor\gg (10).jpg	glioma_tumor
2	Training\glioma_tumor\gg (100).jpg	glioma_tumor
3	Training\glioma_tumor\gg (101).jpg	glioma_tumor
4	Training\glioma_tumor\gg (102).jpg	glioma_tumor
...	...	...
2865	Training\pituitary_tumor\p (95).jpg	pituitary_tumor
2866	Training\pituitary_tumor\p (96).jpg	pituitary_tumor
2867	Training\pituitary_tumor\p (97).jpg	pituitary_tumor
2868	Training\pituitary_tumor\p (98).jpg	pituitary_tumor
2869	Training\pituitary_tumor\p (99).jpg	pituitary_tumor

2870 rows × 2 columns

1.Define the Testing Data Directory

train_data_dir = 'Testing'

- Instead of 'Training', now the dataset is stored in a folder named 'Testing'.

2. Initialize Lists for File Paths and Labels

filepaths = []

labels = []

- filepaths: Stores the full paths of test images.
- labels: Stores the corresponding class/category for each image.

3. Get the List of Class Folders

folds = os.listdir(train_data_dir)

- Retrieves all folders inside the "Testing" directory.
  Example folder structure:

5. Convert Lists into a Pandas DataFrame

Fseries = pd.Series(filepaths, name='filepaths')

Lseries = pd.Series(labels, name='labels')

- Creates two Pandas **Series** objects:
  - Fseries: Stores file paths.
  - Lseries: Stores corresponding labels.

ts_df = pd.concat([Fseries, Lseries], axis=1)

- Combines both series into a **DataFrame** where:
  - Column **filepaths** contains test image paths.
  - Column **labels** contains the corresponding category.

ts_df

OUTPUT:-

```
ts_df
```

	filepaths	labels
0	Testing\glioma_tumor\image(1).jpg	glioma_tumor
1	Testing\glioma_tumor\image(10).jpg	glioma_tumor
2	Testing\glioma_tumor\image(100).jpg	glioma_tumor
3	Testing\glioma_tumor\image(11).jpg	glioma_tumor
4	Testing\glioma_tumor\image(12).jpg	glioma_tumor
...	...	...
389	Testing\pituitary_tumor\image(95).jpg	pituitary_tumor
390	Testing\pituitary_tumor\image(96).jpg	pituitary_tumor
391	Testing\pituitary_tumor\image(97).jpg	pituitary_tumor
392	Testing\pituitary_tumor\image(98).jpg	pituitary_tumor
393	Testing\pituitary_tumor\image.jpg	pituitary_tumor

394 rows × 2 columns

1. Calculate the Class Distribution

data_balance = train_df.labels.value_counts()

- This counts how many images belong to each class in train_df (training dataset).
- The result is a **Pandas Series**, where:
    - The **index** contains the class names.
    - The **values** contain the number of images in each class.

Example Output (data_balance)

Labels	**Count**
Cats	500
Dogs	450
Birds	300

---

2. Define a Custom Percentage Formatter for Pie Chart

```
def custom_autopct(pct):
    total = sum(data_balance)
    val = int(round(pct * total / 100.0))
    return "{:.1f}%\n({:d})".format(pct, val)
```

- pct: Represents the percentage of the total dataset that a class occupies.
- total = sum(data_balance): Computes the total number of training images.
- val = int(round(pct * total / 100.0)): Converts the percentage into the actual number of images.
- Returns a **formatted string** showing:
    - The percentage with **1 decimal place** (e.g., 35.4%).
    - The exact number of images (e.g., (500)).

Example Output from custom_autopct(35.4)

35.4%

(500)

3. Create a Pie Chart to Visualize Class Distribution

```
plt.pie(
    data_balance,
    labels=data_balance.index,
    autopct=custom_autopct,
    colors=["#2092E6","#6D8CE6","#20D0E6","#A579EB"]
)
```

- **data_balance**: Provides the counts for each class.
- **labels = data_balance.index**: Uses the class names as labels.
- **autopct = custom_autopct**: Displays percentage and image count.
- **colors = [...]**: Specifies a custom color palette.

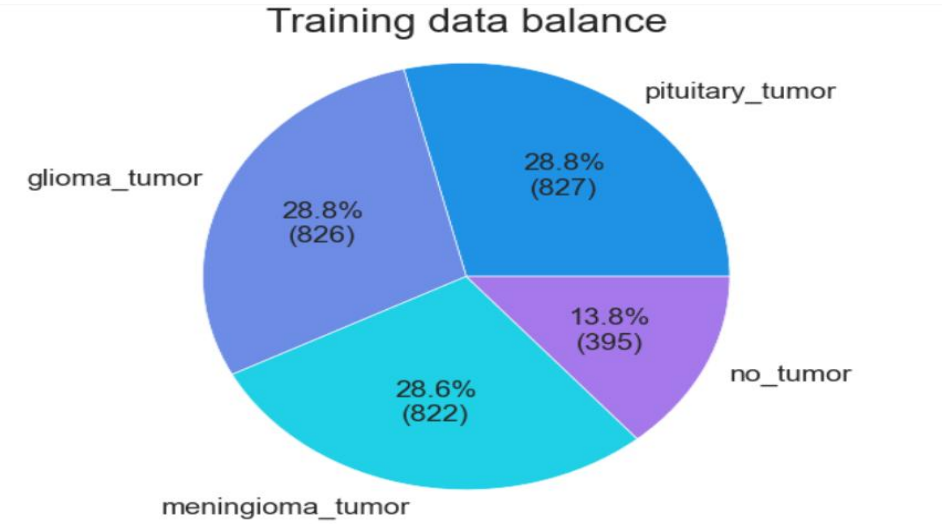
---

4. Add Title & Formatting

```
plt.title("Training data balance")
plt.axis("equal")  # Ensures the pie chart is a perfect circle.
plt.show()
```

- **plt.title("Training data balance")**: Adds a title.
- **plt.axis("equal")**: Ensures the pie chart remains circular.
- **plt.show()**: Displays the pie chart.

OUTPUT:-



data is balanced.

now we will split the testing data to test and validation data

Breaking Down the Code: Splitting Testing Data into Validation and Test Sets

valid_df, test_df = train_test_split(ts_df, train_size=0.5, shuffle=True, random_state=42)

This line **splits** the ts_df (testing dataset) into two new datasets:

1. **valid_df (Validation Set)** → 50% of ts_df
2. **test_df (Final Test Set)** → 50% of ts_df

Understanding Each Parameter

1. train_test_split(ts_df, train_size=0.5, shuffle=True, random_state=42)

- ts_df: The **original test dataset** containing images and labels.
- train_size=0.5:
    - Specifies that **50%** of the ts_df data will be assigned to valid_df (validation set).
    - The remaining **50%** will be assigned to test_df (final test set).
- shuffle=True:
    - **Randomly shuffles** the dataset before splitting.
    - Ensures data is **not split in order**, preventing bias.
- random_state=42:
    - **Fixes the random seed** to make sure the split is **reproducible** every time the code runs.
    - If you change this number, the data will be shuffled differently.

Why Split the Test Data?

1. **Validation Set (valid_df)**
    - Used **during training** to **tune hyperparameters** (e.g., learning rate, dropout rate).
    - Helps prevent **overfitting** by monitoring model performance on unseen data.
2. **Test Set (test_df)**
    - Used **only after training is complete**.
    - **Final evaluation** of model accuracy on completely new data.

Example

Before Splitting (ts_df contains 1000 images)

Filepaths	Labels
Testing/Cats/cat1.jpg	Cats
Testing/Dogs/dog2.jpg	Dogs

Total Images = 1000

After Splitting

- **Validation Set (valid_df)** → 500 images (50% of ts_df)
- **Test Set (test_df)** → 500 images (50% of ts_df)

Breaking Down the Code: Data Generators for Deep Learning Model Training

This code prepares image data for training, validation, and testing using **ImageDataGenerator** from TensorFlow's Keras library.

---

1. Define Image Parameters

batch_size = 16

img_size = (224, 224)

- batch_size = 16 → The model processes **16 images per batch** during training.
- img_size = (224, 224) → Each image is **resized** to **224x224 pixels** before being fed into the model.
    - This is a **common size** for pre-trained models like **ResNet, VGG16, MobileNet**, etc.

---

2. Initialize Data Generators

tr_gen = ImageDataGenerator()

ts_gen = ImageDataGenerator()

- **ImageDataGenerator()** is a **data augmentation tool** in TensorFlow.

- It **rescales and preprocesses** images before passing them to the model.
- Here, two generators are created:
    - **tr_gen** → For training data.
    - **ts_gen** → For validation & test data.

---

3. Create Data Generators for Training, Validation, and Testing

Each **flow_from_dataframe()** method loads images **from a Pandas DataFrame**.

---

a) Training Data Generator

```
train_gen = tr_gen.flow_from_dataframe(
    train_df,
    x_col='filepaths',
    y_col='labels',
    target_size=img_size,
    class_mode='categorical',
    color_mode='rgb',
    shuffle=True,
    batch_size=batch_size
)
```

- **train_df** → The training dataset DataFrame.
- **x_col='filepaths'** → Uses the **file path** column as input.
- **y_col='labels'** → Uses the **labels** column as output.
- **target_size=img_size** → Resizes images to **(224, 224)** pixels.
- **class_mode='categorical'** → Converts labels into **one-hot encoded** format for multi-class classification.
- **color_mode='rgb'** → Loads images in **RGB (3 channels)**.
- **shuffle=True** → Shuffles the data for better training.
- **batch_size=batch_size** → Loads **16 images per batch**.

💡 **This generator will feed batches of images & labels into the model during training.**

---

b) Validation Data Generator

```
valid_gen = ts_gen.flow_from_dataframe(
    valid_df,
    x_col='filepaths',
    y_col='labels',
    target_size=img_size,
    class_mode='categorical',
    color_mode='rgb',
    shuffle=True,
    batch_size=batch_size
)
```

- **Same settings as train_gen**, but uses valid_df (validation set).
- **Used during training to check model performance** on unseen data.

---

c) Testing Data Generator

```
test_gen = ts_gen.flow_from_dataframe(
    test_df,
    x_col='filepaths',
    y_col='labels',
    target_size=img_size,
    class_mode='categorical',
    color_mode='rgb',
    shuffle=False,
    batch_size=batch_size
)
```

- **Same settings as valid_gen**, but uses test_df (final test set).

- **shuffle=False** → Keeps data **in order**, so predictions align with original labels.

💡 **Used for final model evaluation** after training is complete.

Breaking Down the Code: Visualizing a Batch of Images

This code **retrieves and displays a batch of images** from the training dataset using train_gen (image generator).

---

1. Get Class Labels and Sample Batch

g_dict = train_gen.class_indices # Defines dictionary {'class_name': index}

classes = list(g_dict.keys()) # List of class names

images, labels = next(train_gen) # Get a batch of images & labels

What's Happening Here?

- train_gen.class_indices → Creates a **dictionary** mapping class names to numerical indices.

{'Cats': 0, 'Dogs': 1, 'Birds': 2}

- list(g_dict.keys()) → Extracts **only class names** into a list:

['Cats', 'Dogs', 'Birds']

- next(train_gen) → **Retrieves the next batch of images** (16 images since batch_size=16).
    - images → A **batch of image tensors** ((16, 224, 224, 3))
    - labels → Corresponding **one-hot encoded labels** ((16, num_classes)).

---

2. Difference Between next(iterator) and for loop

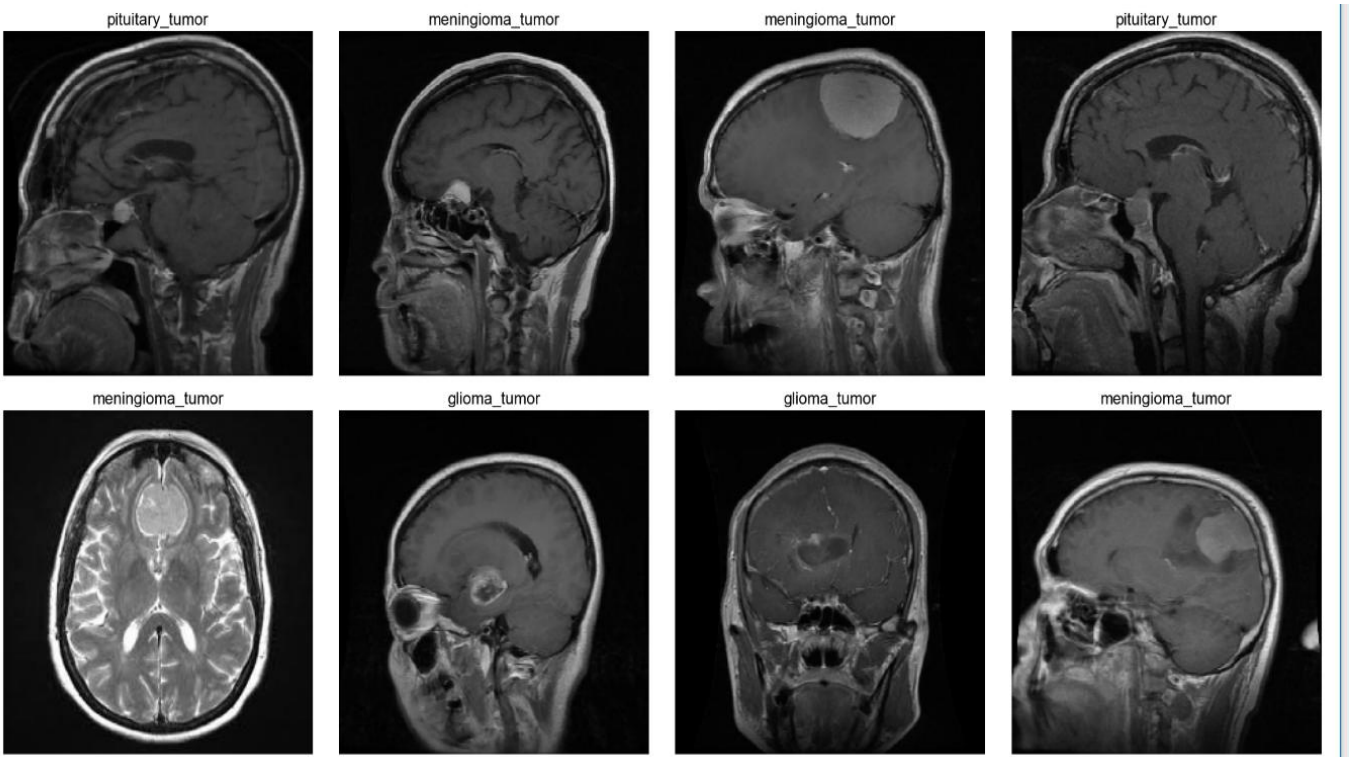
Feature	**next(iterator)**	**for loop**
How it Works	Retrieves the **next** batch	Iterates over the entire dataset
Usage	Fetches data in **small chunks** (useful for large datasets)	Useful when **processing all data**
Example Use Case	Loading a **mini-batch** for training	Looping through **all images**

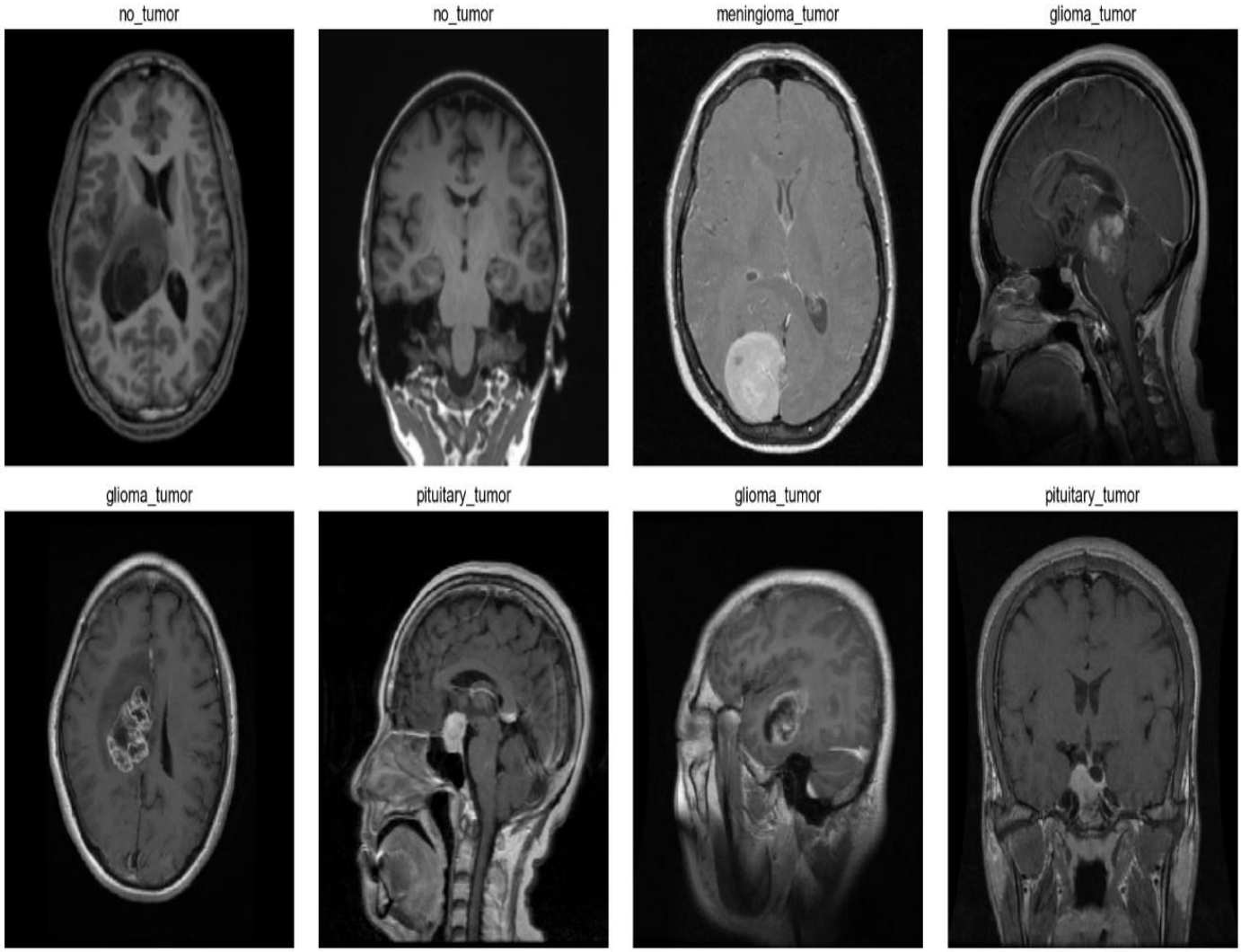
---

3. Display 16 Sample Images

```
plt.figure(figsize=(20, 20))  # Set figure size
for i in range(16):
    plt.subplot(4, 4, i + 1)  # Create 4x4 grid
    image = images[i] / 255   # Normalize pixel values (0-255 range)
    plt.imshow(image)  # Display image

    index = np.argmax(labels[i])  # Get the predicted class index
    class_name = classes[index]  # Convert index to class name

    plt.title(class_name, color='black', fontsize=16)  # Display class name
    plt.axis('off')  # Hide axes

plt.tight_layout()  # Adjust spacing
plt.show()  # Show the final grid of images
```

OUTPUT:-





4. Understanding the Key Operations

Line of Code	What It Does
plt.figure(figsize=(20, 20))	Creates a **large figure** for 16 images
plt.subplot(4, 4, i + 1)	Arranges images in a **4x4 grid**
image = images[i] / 255	**Normalizes** pixel values (0-1 range)
plt.imshow(image)	Displays the **image**
index = np.argmax(labels[i])	Converts **one-hot label** to class index
class_name = classes[index]	Gets **class name** from the index

Line of Code	What It Does
plt.title(class_name, color='black', fontsize=16)	Adds class name as **title**
plt.axis('off')	Hides **axes** for cleaner display

Breaking Down the Code: Building a Convolutional Neural Network (CNN)

This code defines a **CNN (Convolutional Neural Network)** using **Keras' Sequential API**. The model is designed for **multi-class image classification**.

1. Define Image Input Shape & Class Count

```
img_size = (224, 224)  # Image resolution (height, width)
channels = 3  # Number of color channels (RGB)
img_shape = (img_size[0], img_size[1], channels)  # Final input shape (224, 224, 3)
```

- Since images are **RGB**, they have **3 channels**.
- The input shape for the model is **(224, 224, 3)**, meaning each image has **224×224 pixels and 3 color channels**.

```
class_count = len(list(train_gen.class_indices.keys()))
```

- train_gen.class_indices.keys() → Extracts **all class names** from the training generator.
- len(...) → Counts the **number of unique classes** (for final classification layer).

---

2. Define the CNN Model

```
model = Sequential()
```

- Uses **Keras' Sequential API**, which **stacks layers one after another**.

---

3. Convolutional & Pooling Layers

First Convolutional Block

```
model.add(Conv2D(filters=32, kernel_size=(3,3), padding="same", activation="relu", input_shape=img_shape))
model.add(MaxPooling2D())
```

- **Conv2D(filters=32, kernel_size=(3,3), padding="same", activation="relu", input_shape=img_shape)**

- Adds **32 filters** (each filter is a **3×3 matrix**).
- **"same" padding** ensures the output size is the same as the input size.
- Uses **ReLU activation** to introduce non-linearity.
- **input_shape=img_shape** → First layer needs the **input shape**.

- **MaxPooling2D()**
  - Reduces **spatial dimensions** (height & width) by **taking the max value** in a **2×2 window**.
  - This helps **reduce computation** and extract the most important features.

Second Convolutional Block

```
model.add(Conv2D(filters=64, kernel_size=(3,3), padding="same", activation="relu"))
model.add(MaxPooling2D())
```

- Uses **64 filters** to detect more complex patterns.
- Again applies **max pooling** to reduce dimensions.

---

4. Flattening the Output

```
model.add(Flatten())
```

- **Converts 2D feature maps into a 1D vector** to pass into fully connected layers.

---

5. Fully Connected (Dense) Layers

```
model.add(Dense(64, activation="relu"))  # Fully connected layer with 64 neurons
model.add(Dense(32, activation="relu"))  # Fully connected layer with 32 neurons
```

- **Dense Layers** act as **final decision makers** in the classification task.
- **ReLU activation** ensures non-linearity and better learning.

---

6. Output Layer

```
model.add(Dense(class_count, activation="softmax"))
```

- **Final layer has class_count neurons**, one for each class.
- Uses **Softmax activation** to output **probabilities** for each class.

model.compile(Adamax(learning_rate= 0.001), loss= 'categorical_crossentropy', metrics= ['accuracy'])

model.summary()

OUTPUT:-

Model: "sequential_1"

Layer (type)	Output Shape	Param #
conv2d_2 (Conv2D)	(None, 224, 224, 32)	896
max_pooling2d_2 (MaxPooling2D)	(None, 112, 112, 32)	0
conv2d_3 (Conv2D)	(None, 112, 112, 64)	18,496
max_pooling2d_3 (MaxPooling2D)	(None, 56, 56, 64)	0
flatten_1 (Flatten)	(None, 200704)	0
dense_3 (Dense)	(None, 64)	12,845,120
dense_4 (Dense)	(None, 32)	2,080
dense_5 (Dense)	(None, 4)	132

Total params: 12,866,724 (49.08 MB)

Trainable params: 12,866,724 (49.08 MB)

Non-trainable params: 0 (0.00 B)

Breaking Down the Training Process

This code trains the CNN model using the training data generator (train_gen) and validates it using the validation generator (valid_gen).

---

1. Set the Number of Epochs

epochs = 20 # Total training cycles

- **Epoch** → One full pass of the training dataset through the model.
- The model will be trained for **20 epochs**.

---

2. Train the Model

history = model.fit(train_gen, epochs=epochs, verbose=1, validation_data=valid_gen, shuffle=False)

Understanding the Parameters:

Parameter	Explanation
train_gen	The training data generator that provides batches of images & labels.
epochs=epochs	Number of times the entire dataset is passed through the model.
verbose=1	Displays training progress (1 = detailed progress bar).
validation_data=valid_gen	Uses the validation dataset to check model performance after each epoch.
shuffle=False	**No shuffling** during training (data is already shuffled in generator).

---

3. What Happens During Training?

- **For each epoch**:
    1. The model **takes batches of images** from train_gen.
    2. It **passes them through the CNN layers** to extract features.
    3. The **final Dense layer predicts probabilities** for each class.
    4. The model **compares predictions with actual labels** using a **loss function**.
    5. The **optimizer adjusts weights** to minimize the loss.
    6. After each epoch, the model **evaluates performance on the validation set**.

OUTPUT:-

- Epoch 1/20
- **180/180** ━━━━━━━━━━━━━━━━━━━━ **43s** 236ms/step - accuracy: 0.5421 - loss: 6 1.1851 - val_accuracy: 0.5482 - val_loss: 2.0903
- Epoch 2/20
- **180/180** ━━━━━━━━━━━━━━━━━━━━ **43s** 238ms/step - accuracy: 0.8977 - loss: 0 .3121 - val_accuracy: 0.6853 - val_loss: 2.8812
- Epoch 3/20
- **180/180** ━━━━━━━━━━━━━━━━━━━━ **42s** 235ms/step - accuracy: 0.9719 - loss: 0 .1070 - val_accuracy: 0.7056 - val_loss: 3.2406
- Epoch 4/20
- **180/180** ━━━━━━━━━━━━━━━━━━━━ **43s** 237ms/step - accuracy: 0.9933 - loss: 0 .0337 - val_accuracy: 0.7157 - val_loss: 3.8589
- Epoch 5/20
- **180/180** ━━━━━━━━━━━━━━━━━━━━ **43s** 236ms/step - accuracy: 0.9977 - loss: 0 .0129 - val_accuracy: 0.7107 - val_loss: 3.7276
- Epoch 6/20
- **180/180** ━━━━━━━━━━━━━━━━━━━━ **42s** 236ms/step - accuracy: 0.9995 - loss: 0 .0107 - val_accuracy: 0.7462 - val_loss: 4.4989
- Epoch 7/20

- **180/180** ━━━━━━━━━━━━━━━━━━━━ **43s** 241ms/step - accuracy: 0.9985 - loss: 0.0084 - val_accuracy: 0.7107 - val_loss: 4.1770
- Epoch 8/20
- **180/180** ━━━━━━━━━━━━━━━━━━━━ **46s** 255ms/step - accuracy: 0.9989 - loss: 0.0079 - val_accuracy: 0.7411 - val_loss: 5.3027
- Epoch 9/20
- **180/180** ━━━━━━━━━━━━━━━━━━━━ **43s** 239ms/step - accuracy: 0.9990 - loss: 0.0031 - val_accuracy: 0.7411 - val_loss: 5.4885
- Epoch 10/20
- **180/180** ━━━━━━━━━━━━━━━━━━━━ **45s** 252ms/step - accuracy: 1.0000 - loss: 6.3974e-04 - val_accuracy: 0.7259 - val_loss: 5.7234
- Epoch 11/20
- **180/180** ━━━━━━━━━━━━━━━━━━━━ **45s** 248ms/step - accuracy: 1.0000 - loss: 3.4204e-04 - val_accuracy: 0.7360 - val_loss: 5.8420
- Epoch 12/20
- **180/180** ━━━━━━━━━━━━━━━━━━━━ **45s** 251ms/step - accuracy: 1.0000 - loss: 2.2506e-04 - val_accuracy: 0.7360 - val_loss: 6.0016
- Epoch 13/20
- **180/180** ━━━━━━━━━━━━━━━━━━━━ **46s** 255ms/step - accuracy: 1.0000 - loss: 1.6081e-04 - val_accuracy: 0.7513 - val_loss: 6.1716
- Epoch 14/20
- **180/180** ━━━━━━━━━━━━━━━━━━━━ **45s** 250ms/step - accuracy: 1.0000 - loss: 1.2712e-04 - val_accuracy: 0.7513 - val_loss: 6.2271
- Epoch 15/20
- **180/180** ━━━━━━━━━━━━━━━━━━━━ **47s** 259ms/step - accuracy: 1.0000 - loss: 9.0216e-05 - val_accuracy: 0.7513 - val_loss: 6.3479
- Epoch 16/20
- **180/180** ━━━━━━━━━━━━━━━━━━━━ **47s** 263ms/step - accuracy: 1.0000 - loss: 7.1983e-05 - val_accuracy: 0.7513 - val_loss: 6.4434
- Epoch 17/20
- **180/180** ━━━━━━━━━━━━━━━━━━━━ **68s** 379ms/step - accuracy: 1.0000 - loss: 5.7108e-05 - val_accuracy: 0.7513 - val_loss: 6.5044
- Epoch 18/20
- **180/180** ━━━━━━━━━━━━━━━━━━━━ **68s** 375ms/step - accuracy: 1.0000 - loss: 4.4564e-05 - val_accuracy: 0.7513 - val_loss: 6.6294
- Epoch 19/20
- **180/180** ━━━━━━━━━━━━━━━━━━━━ **68s** 380ms/step - accuracy: 1.0000 - loss: 3.4513e-05 - val_accuracy: 0.7513 - val_loss: 6.7346
- Epoch 20/20
- **180/180** ━━━━━━━━━━━━━━━━━━━━ **68s** 376ms/step - accuracy: 1.0000 - loss: 2.2896e-05 - val_accuracy: 0.7513 - val_loss: 6.8797

Breaking Down the Training History Visualization

This code **extracts training history** from the model and plots **loss & accuracy trends** over the epochs.

1. Extract Training and Validation Metrics

```
tr_acc = history.history['accuracy']     # Training accuracy per epoch
```

```
tr_loss = history.history['loss']        # Training loss per epoch
val_acc = history.history['val_accuracy'] # Validation accuracy per epoch
val_loss = history.history['val_loss']    # Validation loss per epoch
```

- history.history['accuracy'] → Stores training accuracy for each epoch.
- history.history['loss'] → Stores training loss.
- history.history['val_accuracy'] → Stores validation accuracy.
- history.history['val_loss'] → Stores validation loss.

---

2. Find Best Epochs

```
index_loss = np.argmin(val_loss)    # Finds epoch with lowest validation loss
val_lowest = val_loss[index_loss]   # Retrieves the lowest validation loss

index_acc = np.argmax(val_acc)      # Finds epoch with highest validation accuracy
acc_highest = val_acc[index_acc]    # Retrieves highest validation accuracy
```

- **np.argmin(val_loss)** → Finds the **epoch index** where validation loss was **lowest**.
- **np.argmax(val_acc)** → Finds the **epoch index** where validation accuracy was **highest**.

```
Epochs = [i+1 for i in range(len(tr_acc))]  # Creates a list of epoch numbers
loss_label = f'best epoch= {str(index_loss + 1)}'
acc_label = f'best epoch= {str(index_acc + 1)}'
```

- Epochs = [i+1 for i in range(len(tr_acc))] → Creates a **list of epoch numbers**.
- **index_loss + 1 & index_acc + 1** → Converts **zero-based indexing** to **one-based**.

---

3. Plot Training & Validation Loss

```
plt.figure(figsize=(20, 8))  # Set figure size
plt.style.use('fivethirtyeight')  # Set style for better visualization

plt.subplot(1, 2, 1)  # Create subplot (1 row, 2 columns, first plot)
plt.plot(Epochs, tr_loss, 'r', label='Training loss')  # Red line for training loss
```

```
plt.plot(Epochs, val_loss, 'g', label='Validation loss')  # Green line for validation loss
plt.scatter(index_loss + 1, val_lowest, s=150, c='blue', label=loss_label)  # Highlight best epoch
plt.title('Training and Validation Loss')  # Title
plt.xlabel('Epochs')  # X-axis label
plt.ylabel('Loss')  # Y-axis label
plt.legend()  # Show legend
```

- Plots **Training Loss (Red) & Validation Loss (Green)**.
- Highlights the **epoch with the lowest validation loss**.

---

4. Plot Training & Validation Accuracy

```
plt.subplot(1, 2, 2)  # Create second subplot
plt.plot(Epochs, tr_acc, 'r', label='Training Accuracy')  # Red line for training accuracy
plt.plot(Epochs, val_acc, 'g', label='Validation Accuracy')  # Green line for validation accuracy
plt.scatter(index_acc + 1, acc_highest, s=150, c='blue', label=acc_label)  # Highlight best epoch
plt.title('Training and Validation Accuracy')  # Title
plt.xlabel('Epochs')  # X-axis label
plt.ylabel('Accuracy')  # Y-axis label
plt.legend()  # Show legend
```

- Plots **Training Accuracy (Red) & Validation Accuracy (Green)**.
- Highlights the **epoch with the highest validation accuracy**.

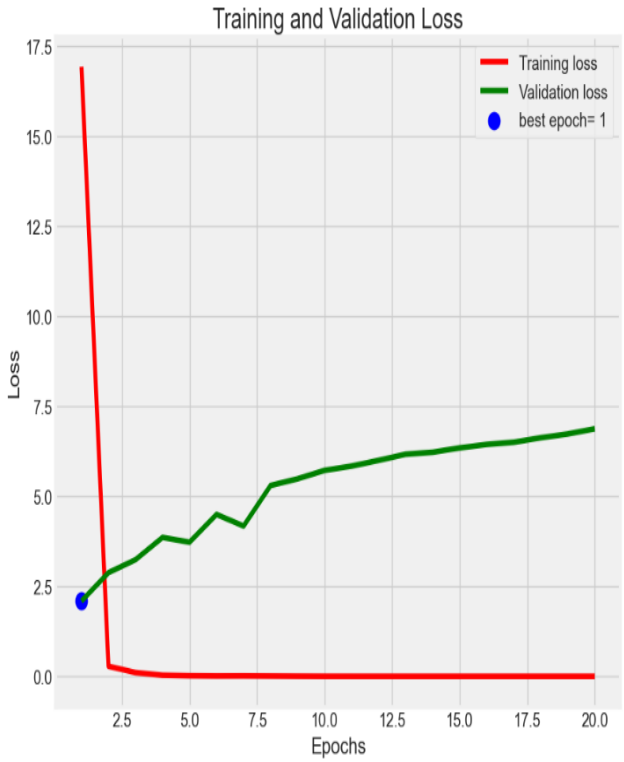
---

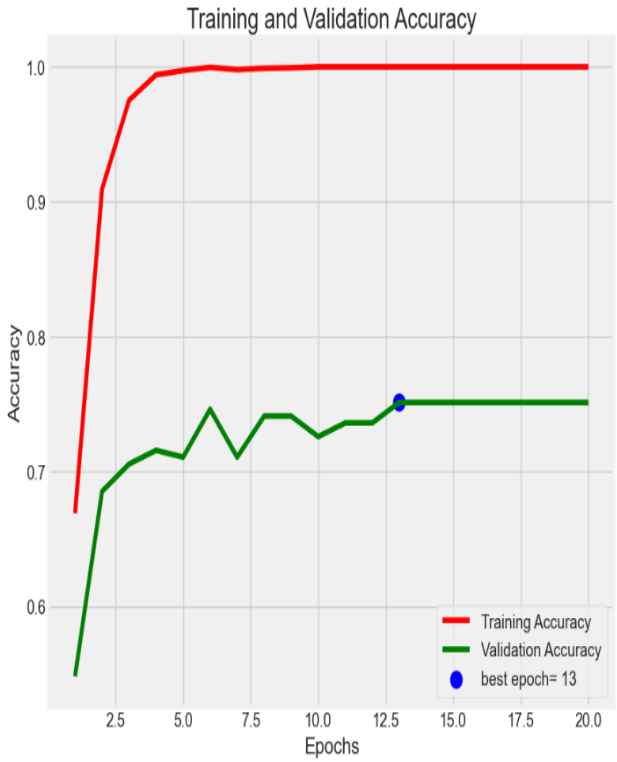
5. Display the Plots

```
plt.tight_layout()
plt.show()
```

- **plt.tight_layout()** ensures **proper spacing** between subplots.
- **plt.show()** displays the plots.

OUTPUT:-





Breaking Down Model Evaluation

This code evaluates the trained model's performance on the **training data**, **validation data**, and **test data**.

---

1. Evaluate the Model on Training Data

train_score = model.evaluate(train_gen, verbose=1)

- **model.evaluate(train_gen)** evaluates the model's performance on the **training dataset** provided by train_gen.
- The result is a **list of metrics**: the **first element** is the **loss** and the **second element** is the **accuracy** of the model on the training set.

2. Evaluate the Model on Validation Data

valid_score = model.evaluate(valid_gen, verbose=1)

- Similarly, **model.evaluate(valid_gen)** evaluates the model's performance on the **validation dataset** (valid_gen).
- This gives the **validation loss** and **validation accuracy**.

3. Evaluate the Model on Test Data

test_score = model.evaluate(test_gen, verbose=1)

- Finally, **model.evaluate(test_gen)** evaluates the model's performance on the **test dataset** (test_gen), which was not seen by the model during training.

---

4. Output the Scores

print("Train Loss: ", train_score[0])

print("Train Accuracy: ", train_score[1])

print('-' * 20)

print("Validation Loss: ", valid_score[0])

print("Validation Accuracy: ", valid_score[1])

print('-' * 20)

print("Test Loss: ", test_score[0])

print("Test Accuracy: ", test_score[1])

- **train_score[0] → Training loss**.
- **train_score[1] → Training accuracy**.
- **valid_score[0] → Validation loss**.
- **valid_score[1] → Validation accuracy**.
- **test_score[0] → Test loss**.
- **test_score[1] → Test accuracy**.

OUTPUT:-

180/180 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **27s** 150ms/step - accuracy: 1.0000 - loss: 1.9944e-05

13/13 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **2s** 132ms/step - accuracy: 0.7528 - loss: 6.7319

13/13 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **2s** 156ms/step - accuracy: 0.7033 - loss: 9.2100

Train Loss: 1.969781078514643e-05

Train Accuracy: 1.0

Validation Loss: 6.879691123962402

Validation Accuracy: 0.7512690424919128

Test Loss: 9.08182430267334

Test Accuracy: 0.7157360315322876

Breaking Down the Prediction Code

This code predicts the classes for the **test dataset** and converts the predicted probabilities into **class labels**.

---

1. Predict with the Model

preds = model.predict(test_gen)

- **model.predict(test_gen)**: This generates predictions for the **test dataset** using the model.
- test_gen provides batches of images that are passed to the model for prediction.
- **preds** will contain the **predicted probabilities** for each class, as the final layer of the model uses the **softmax activation**. Each prediction in preds will be a vector of probabilities, one for each class.

---

2. Convert Predicted Probabilities to Class Labels

python

CopyEdit

y_pred = np.argmax(preds, axis=1)

- **np.argmax(preds, axis=1)**: This function returns the **index of the maximum value** in each prediction vector (i.e., the class with the highest probability).
    - **axis=1** ensures that **maximum value** is selected along the **columns** (for each prediction vector).
- **y_pred** will now contain the **predicted class labels** corresponding to the **highest probability** for each test sample.

---

y_pred = [1, 0, 0]

- **Predicted Class** for the first sample is class **1** (since 0.7 is the highest probability).
- **Predicted Class** for the second sample is class **0** (since 0.4 is the highest probability, but it's a tie; it picks the first occurrence).
- **Predicted Class** for the third sample is class **0** (since 0.9 is the highest probability).

y_pred

OUTPUT:-

```
array([2, 2, 2, 0, 3, 1, 2, 2, 1, 1, 3, 1, 2, 1, 3, 2, 1, 2, 2, 3, 2, 3,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 3, 2, 1, 3, 2, 1, 0, 0, 2, 0, 1, 3, 1,
       1, 2, 2, 3, 1, 1, 2, 1, 2, 3, 3, 1, 1, 1, 1, 2, 2, 1, 3, 1, 3, 1,
       1, 1, 2, 1, 1, 3, 2, 1, 1, 2, 2, 3, 2, 3, 1, 1, 0, 3, 2, 3, 3, 2,
       2, 1, 2, 3, 2, 3, 2, 1, 2, 1, 2, 2, 0, 1, 2, 2, 2, 1, 3, 1, 1, 2,
       3, 2, 0, 2, 2, 1, 2, 1, 1, 2, 2, 2, 1, 1, 1, 2, 1, 2, 1, 2, 3, 1,
       1, 1, 2, 1, 1, 2, 1, 2, 1, 2, 1, 0, 1, 3, 1, 1, 2, 1, 2, 1, 1, 2,
       2, 2, 3, 1, 2, 3, 3, 2, 1, 3, 2, 0, 1, 2, 1, 2, 2, 2, 0, 1, 2, 1,
       1, 2, 3, 2, 3, 2, 3, 1, 1, 2, 1, 3, 1, 3, 1, 2, 2, 1, 1, 2, 2])
```

Confusion Matrics and Classification Report

Breaking Down the Confusion Matrix Visualization

The code below visualizes the performance of the model by plotting a **confusion matrix**. It helps you understand how well the model is predicting each class in comparison to the true labels.

---

1. Get Class Indices and Class Labels

```
g_dict = test_gen.class_indices  # Get class indices from the generator
classes = list(g_dict.keys())    # Get class names (keys from dictionary)
```

- **test_gen.class_indices** gives a dictionary that maps **class names** to **integer indices**.
- **classes** is a list of class names (e.g., [class_1, class_2, ..., class_n]).

---

2. Compute the Confusion Matrix

```
cm = confusion_matrix(test_gen.classes, y_pred)
```

- **test_gen.classes** contains the **true labels** for all images in the test set.
- **y_pred** contains the predicted labels (output of np.argmax(preds, axis=1)).
- **confusion_matrix(test_gen.classes, y_pred)** compares the true labels with the predicted labels, producing a **confusion matrix (cm)**.

Confusion Matrix:

The confusion matrix is a table that summarizes the performance of the model. It tells you:

- How many times each class was predicted correctly (diagonal elements).
- How many times a class was incorrectly predicted as another class (off-diagonal elements).

---

3. Plot the Confusion Matrix

```
plt.figure(figsize=(10, 10))  # Set figure size
plt.imshow(cm, interpolation='nearest', cmap=plt.cm.Blues)  # Plot the confusion matrix
plt.title('Confusion Matrix')  # Title of the plot
plt.colorbar()  # Add color bar to indicate intensity of values
```

- **plt.imshow(cm)** visualizes the matrix as an image.
- **cmap=plt.cm.Blues** uses a **blue color map** to visualize the matrix with intensity.

---

4. Add Class Labels and Formatting

```
tick_marks = np.arange(len(classes))  # Create ticks for each class
plt.xticks(tick_marks, classes, rotation=45)  # Set X-axis labels (class names)
plt.yticks(tick_marks, classes)  # Set Y-axis labels (class names)
```

- **np.arange(len(classes))** generates tick positions on the axes.
- **plt.xticks() and plt.yticks()** add class names to the axes of the confusion matrix.

---

5. Display Confusion Matrix Values

```
thresh = cm.max() / 2.  # Set threshold for text color (white or black)
for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
    plt.text(j, i, cm[i, j], horizontalalignment='center', color='white' if cm[i, j] > thresh else 'black')
```

- This loop iterates through each cell of the confusion matrix and places the **numerical value** of the confusion matrix in the corresponding position.
- The color of the text is chosen based on the background intensity, ensuring it is **readable**.

---

6. Final Layout and Labels

plt.tight_layout() # Adjust spacing to prevent overlap

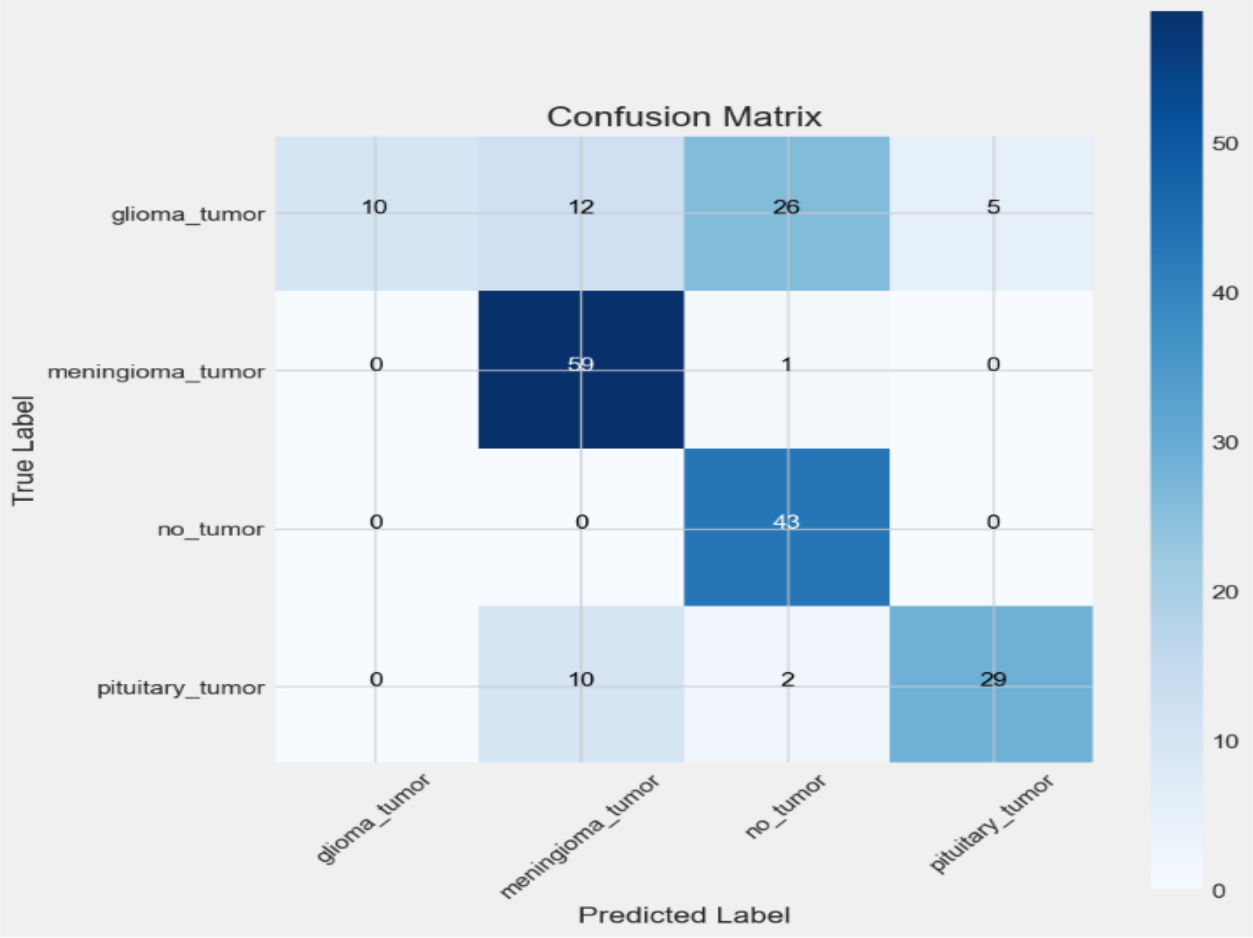
plt.ylabel('True Label') # Y-axis label

plt.xlabel('Predicted Label') # X-axis label

plt.show() # Display the plot

- **plt.tight_layout()** ensures that all elements are **properly spaced**.
- **plt.ylabel()** and **plt.xlabel()** add labels to the axes to indicate that the **Y-axis is the true label** and the **X-axis is the predicted label**.

OUTPUT:-



Classification report

print(classification_report(test_gen.classes, y_pred, target_names= classes))

OUTPUT:-

	precision	recall	f1-score	support
glioma_tumor	1.00	0.19	0.32	53
meningioma_tumor	0.73	0.98	0.84	60
no_tumor	0.60	1.00	0.75	43
pituitary_tumor	0.85	0.71	0.77	41
accuracy			0.72	197
macro avg	0.79	0.72	0.67	197
weighted avg	0.80	0.72	0.66	197

from predict import predict

is importing a function named **predict** from a Python module called **predict**. Here's a breakdown of how it works:

1. Importing from a Module:

- **from predict import predict**:
    - The Python interpreter is looking for a module named predict.py in the current directory or Python path.
    - The **predict** function is defined inside this predict.py file and is being imported into the current script.

2. Understanding predict Function:

- The **predict** function could be designed to handle specific tasks related to making predictions, like:
    - Loading the model.
    - Taking input data (e.g., images or numerical data).
    - Running the data through the model to make predictions.
    - Returning the predicted class, probability, or other relevant information.

Model Testing:

1. Loading the Model:

```
from tensorflow.keras.models import load_model

model = load_model('Brain Tumor.h5')
```

- **load_model('Brain Tumor.h5')** loads a pre-trained **Keras model** saved as 'Brain Tumor.h5'.
- This model is likely used for **classifying brain tumor images** based on the problem context.

---

2. Preprocessing the Image:

```
from PIL import Image

import numpy as np


def preprocess_image(image_path):
    image = Image.open(image_path).convert('RGB')
    image = image.resize((224, 224))
    image_array = np.array(image)
    image_array = image_array / 255.0
    image_array = np.expand_dims(image_array, axis=0)
    return image_array
```

- **Image.open(image_path).convert('RGB')**: This opens the image from the specified path and converts it to RGB color mode (in case it's grayscale or another format).
- **image.resize((224, 224))**: Resizes the image to the required size (224, 224) to match the model's input size.
- **np.array(image)**: Converts the image object into a numpy array, which is required for input into the model.
- **image_array / 255.0**: Normalizes the pixel values (scales them between 0 and 1).
- **np.expand_dims(image_array, axis=0)**: Adds an extra dimension to the array, creating a batch of size 1. This is required because models in Keras expect the input to have the shape (batch_size, height, width, channels).

---

3. Prediction Function:

```
def predic(image_path):
    processed_image = preprocess_image(image_path)
    prediction = model.predict(processed_image)
```

- **processed_image = preprocess_image(image_path)**: The function first preprocesses the input image.
- **prediction = model.predict(processed_image)**: This runs the **preprocessed image** through the model to get a prediction. The model returns the predicted probabilities for each class.

However, you should include a return statement to **return the prediction result** from the function. Without it, the predic() function does not return anything.

---

4. Fixing the Issue with predict() and Printing the Prediction:

```
prediction_result = predic(image_path)
print("Prediction:", prediction_result)
```

- **prediction_result = predic(image_path)** stores the result of the prediction in the prediction_result variable.
- **print("Prediction:", prediction_result)** tries to print the prediction, but since predic() doesn't return anything, this will print None.

You should modify the predic() function to return the prediction:

```
def predic(image_path):
    processed_image = preprocess_image(image_path)
    prediction = model.predict(processed_image)
    return prediction  # Return the prediction result
```

Now, the prediction_result will hold the output of the model, which is the predicted probabilities for each class.

5. Optionally Converting Prediction to Class Label

If the output of the prediction is probabilities (as is common for classification tasks), you likely want to convert these probabilities to a **predicted class label**.

You can use **np.argmax()** to get the index of the highest probability, which corresponds to the predicted class:

```
def predic(image_path):
```

```
    processed_image = preprocess_image(image_path)
    prediction = model.predict(processed_image)
    predicted_class = np.argmax(prediction, axis=1)  # Get the class with the highest probability
    return predicted_class
prediction_result = predic(image_path)
print("Predicted Class:", prediction_result)
```

OUTPUT:-

1/1 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **0s** 245ms/step

Prediction: pitutary

```
image_path = "Te-pi_0133.jpg"
image = Image.open(image_path)
image
```

does the following:

1. Loading the Image:

```
image = Image.open(image_path)
```

- **Image.open(image_path)**: This function opens the image located at image_path, which is "Te-pi_0133.jpg" in this case, using the PIL (Python Imaging Library) module.
- This creates an **Image object** that represents the loaded image.

2. Displaying the Image Object:

```
image
```

- When executed in a Jupyter notebook or an interactive Python environment (like IPython), this line will display a representation of the image.
- The **image** will be shown inline, allowing you to visually inspect it.

If you're working in a standard Python script, calling image by itself will not display it. To view the image, you need to use something like:

```
image.show()
```

This will open the image using the default image viewer on your system.

OUTPUT:-