**CHAPTER 1**

**INTRODUCTION**

**Artificial Intelligence** is a domain of computer science that aims to build intelligent machines , capable of performing tasks typically requiring human cognition, such as decision-making, problem-solving, language understanding, and visual perception. AI systems simulate human intelligence by processing large amounts of data and identifying patterns or relationships that help them make informed decisions. In the medical field, AI helps streamline diagnostics, enhance decision support, and improve treatment planning.

four stages of AI development are commonly recognized.

1. **Reactive machines:** Limited AI that only reacts to different kinds of stimuli based on preprogrammed rules. Does not use memory and thus cannot learn with new data. **IBM’s Deep Blue that beat chess champion Garry Kasparov in 1997** was an example of a reactive machine.
2. **Limited memory:** Most modern AI is considered to be limited memory. It can use memory to improve over time by being trained with new data, typically through an artificial neural network or other training model. Deep learning, a subset of machine learning, is considered limited memory artificial intelligence.
3. **Theory of mind:** Theory of mind AI does not currently exist, but research is ongoing into its possibilities. It describes AI that can emulate the human mind and has decision-making capabilities equal to that of a human, including recognizing and remembering emotions and reacting in social situations as a human would.
4. **Self - aware:**A step above theory of mind AI, self-aware AI describes a mythical machine that is aware of its own existence and has the intellectual and emotional capabilities of a human. Like theory of mind AI, self-aware AI does not currently exist.

**Machine Learning** is a subset of AI that focuses on building algorithms that learn from data and improve over time without being explicitly programmed. Instead of using a fixed set of instructions, ML models find patterns and make predictions or classifications based on past data. For instance, by feeding a model thousands of MRI brain scans labeled as ‘tumor’ or ‘no tumor,’ the system can learn to classify new, unseen scans with increasing accuracy.

**Deep Learning** is a specialized area within machine learning that uses **artificial neural networks** with many layers—hence the term "deep"—to process data. These networks mimic how the human brain works, enabling the model to recognize intricate patterns. One key architecture in DL used for image analysis is the **Convolutional Neural Network (CNN)**, which automatically detects features such as edges, shapes, and textures, making it especially effective for medical image classification tasks like brain tumor detection.

1. **Feedforward Neural Networks (FF)** are one of the oldest types of neural networks, where data flows through layers of neurons to produce an output. Modern versions are often "deep" with multiple hidden layers and use backpropagation to correct errors and improve accuracy.
2. **Recurrent Neural Networks (RNN)** differ by using time-series or sequence data. They "remember" previous outputs, which helps in tasks like language processing, speech recognition, and image captioning.
3. **Long Short-Term Memory (LSTM)** is an advanced RNN that remembers information over longer periods using memory cells, making it ideal for tasks like speech recognition and predictions.
4. **Convolutional Neural Networks (CNN)** are commonly used for image recognition. They use multiple layers to detect features in images, starting from simple ones like edges to more complex patterns in later layers.
5. **Generative Adversarial Networks (GAN)** involve two networks competing: a generator creates examples, and a discriminator evaluates them. This competition improves the output, making GANs useful for generating realistic images and art.

**Role of AI in MRI-based Brain Tumor Detection**

With the help of AI—particularly CNN-based deep learning models—the analysis of MRI scans can be automated. These models are trained on large datasets of labeled brain images, allowing them to learn the features that distinguish tumor types. Once trained, they can:

* Detect tumors in new scans quickly and accurately
* Classify tumor types (e.g., glioma, meningioma, pituitary)
* Reduce the workload on radiologists
* Minimize human errors  
  This technological advancement supports more consistent, faster, and scalable diagnostic workflows, ultimately leading to better patient care and clinical outcomes.

**CHAPTER 2**

**EXECUTIVE SUMMARY**

**Deep learning** is applied to detect brain tumors from **MRI images** using the **Brain Tumor Classification (MRI) dataset** from Kaggle. The pipeline involves data preprocessing, CNN model building, training, and evaluation. Several tools and libraries like **TensorFlow**, **Keras**, and **scikit-learn** are used to support this process.

**Kaggle Dataset:** Brain Tumor MRI Detection

Kaggle is an online platform for data science and machine learning competitions, collaboration, and learning. It hosts a variety of datasets and challenges, where users can participate in competitions to solve real-world problems using data analysis and machine learning techniques. It is widely used by both beginners and experienced data scientists, researchers, and developers

* The dataset used is the **Brain Tumor Classification (MRI)** dataset from **Kaggle**.
* It contains **labeled MRI images** divided into different classes: **glioma, meningioma, pituitary tumors, and no tumor**.
* These labels help train a model to distinguish between tumorous and non-tumorous brain scans effectively.

**Changing Runtime Hardware to GPU**

* **GPU (Graphics Processing Unit)** is selected in the runtime environment to **accelerate model training**.
* Deep learning tasks, especially CNN-based image classification, involve heavy matrix operations that GPUs can handle faster than CPUs.
* This step significantly **reduces training time** and improves performance.

**Installing and Importing Dependencies**

Before model training begins, essential libraries are installed and imported.

**Key Deep Learning Libraries:**

* **TensorFlow**: A popular open-source framework for machine learning and deep learning.
* **Keras**: A high-level API built on top of TensorFlow; used to quickly build and train neural networks.
  + We import:
    - Sequential – a linear stack of layers, commonly used for **ANN, CNN, and RNN**.
    - layers – for defining neural network components (e.g., Conv2D, MaxPooling2D, Dense).

**Supporting Libraries:**

* **scikit-learn (sklearn)**: Provides tools like train\_test\_split for data division and performance evaluation metrics.
* **OpenCV (cv2)**: Used for image processing operations such as resizing, converting to grayscale, etc.
* **PIL (Python Imaging Library)**: For opening and manipulating images.
* **ipywidgets & IO**: Helpful for interactive notebook elements and input/output operations.
* **tqdm**: Used to display progress bars during data loading or training loops.

**CHAPTER 3**

**BRAIN TUMOR DETECTION**

Brain tumor detection is crucial as early diagnosis significantly improves treatment outcomes and survival rates. Manual detection through MRI scans is time-consuming and prone to human error, whereas **AI-powered models** can analyze medical images with high precision, aiding radiologists in faster and more accurate diagnosis. Automated detection helps in reducing misdiagnosis, improving patient care, and supporting timely medical intervention.

**Step 1:** **Dataset Usage**

The Brain Tumor Classification (MRI) dataset contains labeled MRI images categorized into tumor types and non-tumor cases. It helps in training machine learning models to identify and classify brain tumors with higher accuracy, aiding in medical diagnosis.

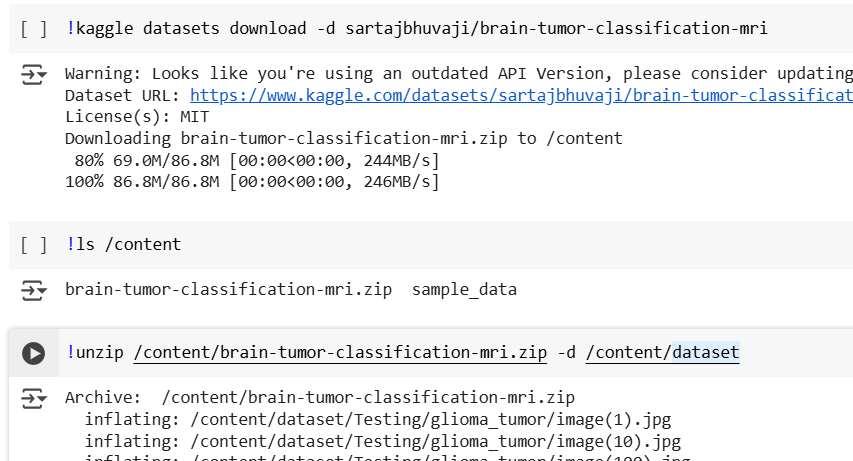
link: [https://www.kaggle.com/datasets/sartajbhuvaji/brain-tumor-classification-mri](https://www.google.com/url?q=https%3A%2F%2Fwww.kaggle.com%2Fdatasets%2Fsartajbhuvaji%2Fbrain-tumor-classification-mri)

**code:**

!kaggle datasets download -d sartajbhuvaji/brain-tumor-classification-mri

!ls /content

!unzip /content/brain-tumor-classification-mri.zip -d /content/dataset



**Step 2: Installing Dependencies**

Libraries like TensorFlow, Keras and scikit-learn provide essential tools for deep learning, image processing, and model evaluation. They help in building, training, and optimizing neural networks for tumor detection.

**Code:**

!pip install keras

import keras

from keras.models import Sequential

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

from sklearn.metrics import accuracy\_score

import ipywidgets as widgets

import io

from PIL import Image

import tqdm

from sklearn.model\_selection import train\_test\_split

import cv2

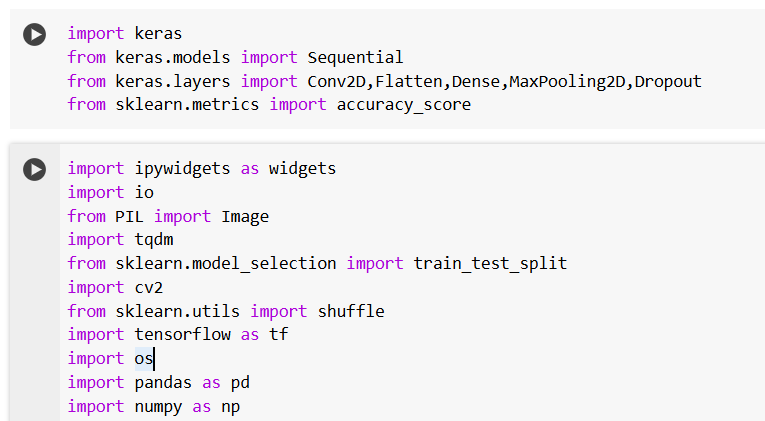
from sklearn.utils import shuffle

import tensorflow as tf

import os

import pandas as pd

import numpy as np



**Step 3: Divide the dataset**

The folder paths for the training and testing datasets are:

**Training Dataset:** [/content/train](https://colab.research.google.com/drive/1b4vwRoaIRdZazFp_IHfD1Yvcc3a2pB30) (Contains 80% of the images for model training)

**Testing Dataset:**[/content/test](https://colab.research.google.com/drive/1b4vwRoaIRdZazFp_IHfD1Yvcc3a2pB30) (Contains 20% of the images for model evaluation) Each of these folders will have subdirectories for different tumor categories, maintaining class labels.

**Code:**

#We are taking both training and testing varaibles(dataset) to the training variables in local system.

#The above is done because the image provided can be baised towards training or testing.

#So we combine the total and split according to the requirement.

X\_train = []

Y\_train = []

image\_size = 150

labels = ['glioma\_tumor','meningioma\_tumor','no\_tumor','pituitary\_tumor']

#Training folder path.

for i in labels:

    folderPath = os.path.join('/content/dataset/Training',i)

    for j in os.listdir(folderPath):

        img = cv2.imread(os.path.join(folderPath,j))

        img = cv2.resize(img,(image\_size,image\_size))

        X\_train.append(img)

        Y\_train.append(i)

#Testing Folder path

for i in labels:

    folderPath = os.path.join('/content/dataset/Testing',i)

    for j in os.listdir(folderPath):

        img = cv2.imread(os.path.join(folderPath,j))

        img = cv2.resize(img,(image\_size,image\_size))

        X\_train.append(img)

        Y\_train.append(i)

#The above training variables are list but we use array in DL.

X\_train = np.array(X\_train)

Y\_train = np.array(Y\_train)

shuffle the images in the training variables

X\_train,Y\_train = shuffle(X\_train,Y\_train,random\_state=101)

X\_train.shape

#The ouput is [no. of images, image size, image size, no. of channels(rgb)]

**Split the Dataset**

The dataset is divided into training (80%) and testing (20%) sets to train the model effectively and evaluate its performance. The split ensures that the model learns patterns from training data while being tested on unseen images. This helps in preventing overfitting and improves the accuracy of brain tumor classification.

**Code:**

X\_train,X\_test,y\_train,y\_test = train\_test\_split(X\_train,Y\_train,test\_size=0.1,random\_state=101)

#X\_train contains images converted to array.

#Y\_train contains the labels.

y\_train\_new = []

for i in y\_train:

    y\_train\_new = [np.argmax(i) for i in y\_train]

y\_train=np.array(y\_train\_new)

#label indexes are converted to categorical variables, so model doesn't give any error.

y\_train = tf.keras.utils.to\_categorical(y\_train)

y\_test\_new = []

for i in y\_test:

    y\_test\_new = [np.argmax(i) for i in y\_test]

y\_test=np.array(y\_test\_new)

y\_test = tf.keras.utils.to\_categorical(y\_test)

**Step 4: Convolutional Neural Network**

This is a **Convolutional Neural Network (CNN)** built using the Keras API in TensorFlow. It's designed for **image classification**, where the model processes input images and assigns them to one of four possible categories (based on the Dense(4, activation='softmax') output layer).

**1. Convolutional Layers (Conv2D)**

* The model starts with multiple **Conv2D layers** that apply convolution operations on the input image to extract features such as edges, textures, and patterns.
* The first convolutional layer has **32 filters** of size (3, 3), followed by additional layers with **64** and **128 filters**.
* These layers use **ReLU activation** ('relu'), which helps introduce non-linearity, enabling the model to learn complex patterns.

**2. Pooling Layers (MaxPooling2D)**

* After each group of convolutional layers, **MaxPooling2D layers** are used to reduce the spatial dimensions of the feature maps (i.e., downsampling). This helps reduce computation and captures the most important features.
* The pooling window size is (2, 2), meaning it reduces the width and height of the image by half.

**3. Dropout Layers**

* **Dropout layers** are added after convolution and dense layers (with a rate of 0.3), which randomly "drop" a fraction of neurons during training. This helps prevent **overfitting** by encouraging the model to generalize better.

**4. Flatten Layer**

* The **Flatten** layer converts the 2D matrix from the last convolutional or pooling layer into a 1D vector, which can be passed into the fully connected layers.

**5. Fully Connected Layers (Dense)**

* The model includes two fully connected (dense) layers with **512 units** each and **ReLU activation**. These layers learn higher-level combinations of the features extracted by the convolutional layers.
* The final **Dense(4)** layer uses **softmax activation**, which outputs a probability distribution over 4 classes. This is typical for multi-class classification problems.

**Code:**

model = Sequential()

#Adding the layer to the model.

#add(<model\_name>(no. of parameters,(matrix trianing on), <activation>, input\_shape))

model.add(Conv2D(32,(3,3),activation = 'relu',input\_shape=(150,150,3)))

#The 64 indicates how many output units layer will have.

model.add(Conv2D(64,(3,3),activation='relu'))

#Now we deal with pooling so we add MaxPooling2D layer, since images ar 2D.

model.add(MaxPooling2D(2,2))

#Some images should be dropped out, since not necessary.

model.add(Dropout(0.3))

#Add Multiple layers to the model.

model.add(Conv2D(64,(3,3),activation='relu'))

model.add(Conv2D(64,(3,3),activation='relu'))

model.add(Dropout(0.3))

model.add(MaxPooling2D(2,2))

model.add(Dropout(0.3))

#Expand the network since there are many pooling images.

model.add(Conv2D(128,(3,3),activation='relu'))

model.add(Conv2D(128,(3,3),activation='relu'))

model.add(Conv2D(128,(3,3),activation='relu'))

model.add(MaxPooling2D(2,2))

model.add(Dropout(0.3))

# Add Convolution layers because we have multiple layers.

model.add(Conv2D(128,(3,3),activation='relu'))

model.add(Conv2D(256,(3,3),activation='relu'))

model.add(MaxPooling2D(2,2))

model.add(Dropout(0.3))

#Neural network is created. Flatten the ouput layer.

model.add(Flatten())

model.add(Dense(512,activation = 'relu'))

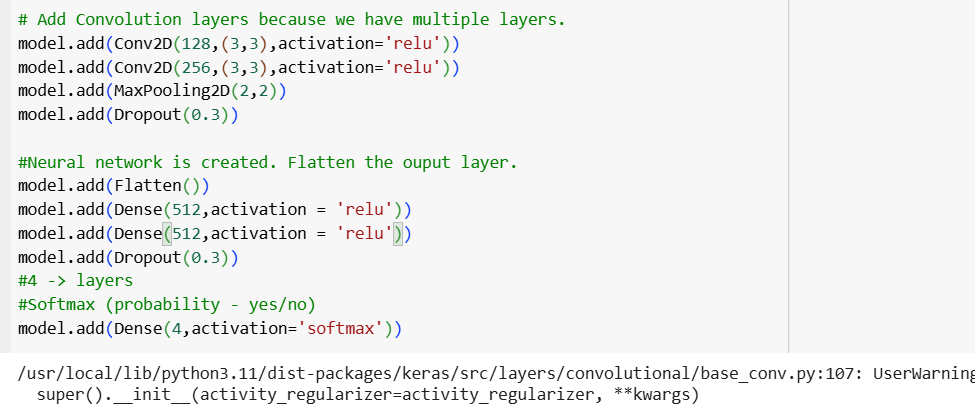
model.add(Dense(512,activation = 'relu'))

model.add(Dropout(0.3))

#4 -> layers

#Softmax (probability - yes/no)

model.add(Dense(4,activation='softmax'))

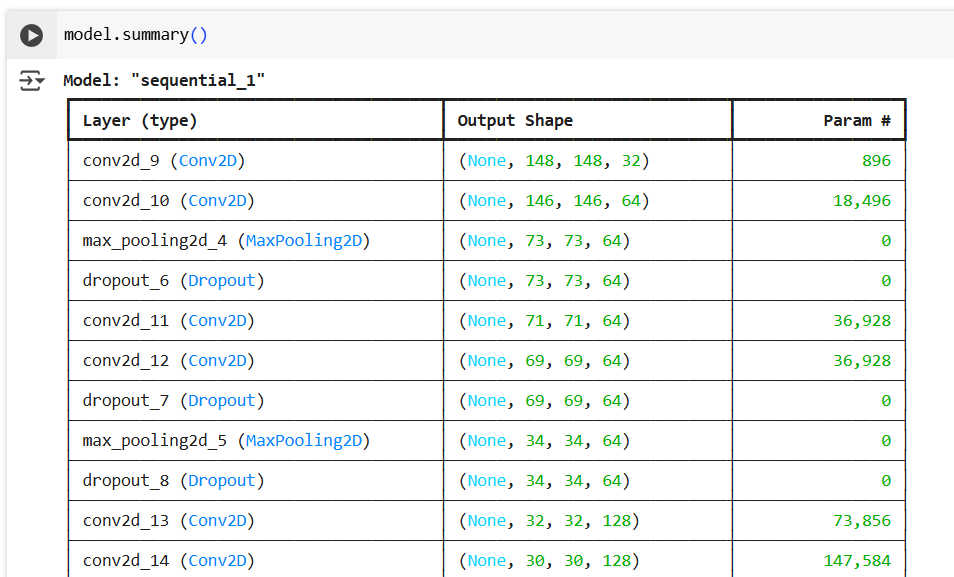


The **model.summary()** output provides a detailed breakdown of your Convolutional Neural Network (CNN) architecture.

* Layer Type: Each row represents a layer in the model (e.g., Conv2D, MaxPooling2D, Dense).
* Output Shape: The shape of the data after passing through each layer. Example: (None, 148, 148, 32) means a batch of images (None represents batch size) with dimensions 148x148 pixels and 32 filters.
* Param # (Parameters): The number of trainable parameters in each layer

**Code:**

model.summary()



**Step 5: Compile and Train the model**

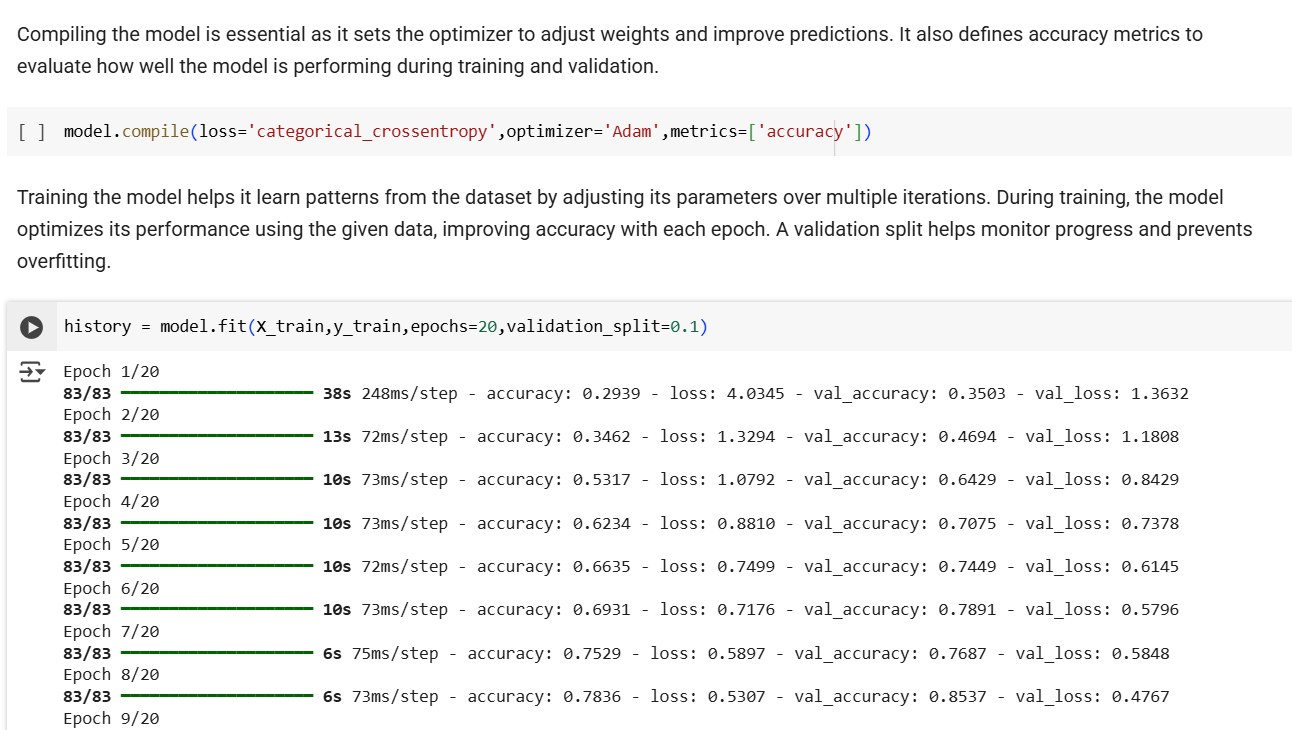
When **compiling** the model, we specify the **optimizer** (e.g., Adam), the **loss function** (e.g., categorical crossentropy for multi-class classification), and **metrics** (e.g., accuracy) to evaluate model performance. This setup is essential for guiding the training process and adjusting the model's weights to minimize the loss.

**Training** the model involves feeding it labeled data, allowing it to learn patterns and improve over time. By specifying the number of **epochs** (iterations) and **batch size**, we control how long and how often the model learns from the data. Training helps the model generalize to new, unseen examples.

**Code:**

model.compile(loss='categorical\_crossentropy',optimizer='Adam',metrics=['accuracy'])

history = model.fit(X\_train,y\_train,epochs=20,validation\_split=0.1)



**Step 6: Visualizing the model**

**Matplotlib** and **Seaborn** are both powerful Python visualization libraries that complement each other. **Matplotlib** offers a high degree of flexibility and customization, allowing users to create a wide range of static, animated, and interactive plots. It's great for fine-tuning visualizations and creating detailed, publication-ready charts. On the other hand, **Seaborn** builds on Matplotlib by providing a more aesthetically pleasing interface and simpler syntax for creating complex statistical plots. It also integrates well with Pandas data structures, making it easier to visualize data directly from DataFrames. Together, they enable users to create insightful and visually appealing data visualizations with minimal effort.

**Code:**

import matplotlib.pyplot as plt

import seaborn as sns

acc = history.history['accuracy']

val\_acc = history.history['val\_accuracy']

epochs = range(len(acc))

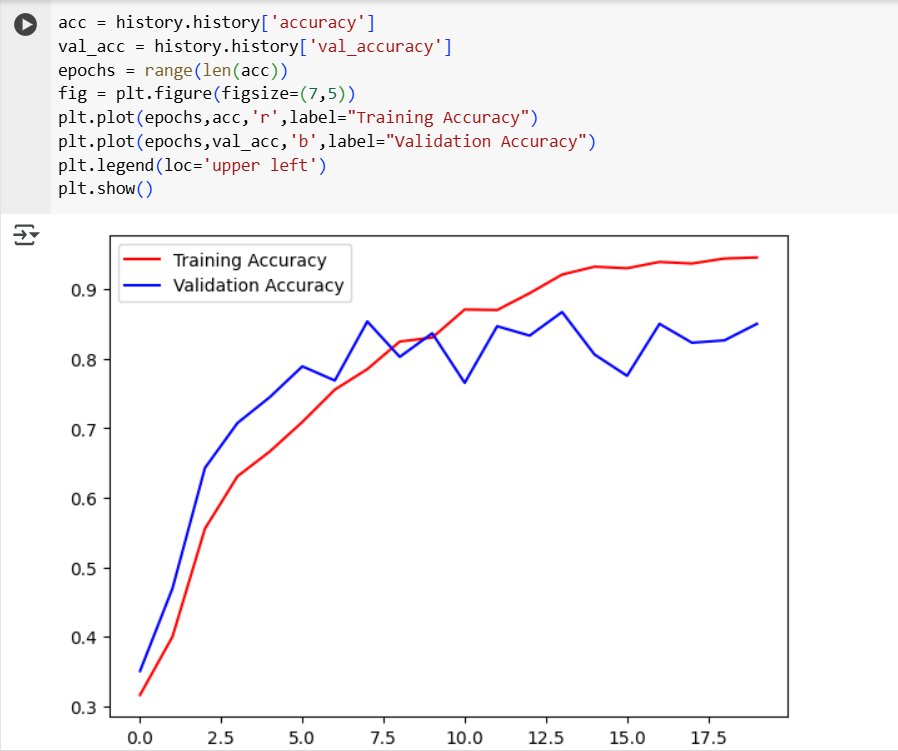
fig = plt.figure(figsize=(7,5))

plt.plot(epochs,acc,'r',label="Training Accuracy")

plt.plot(epochs,val\_acc,'b',label="Validation Accuracy")

plt.legend(loc='upper left')

plt.show()



loss = history.history['loss']

val\_loss = history.history['val\_loss']

epochs = range(len(loss))

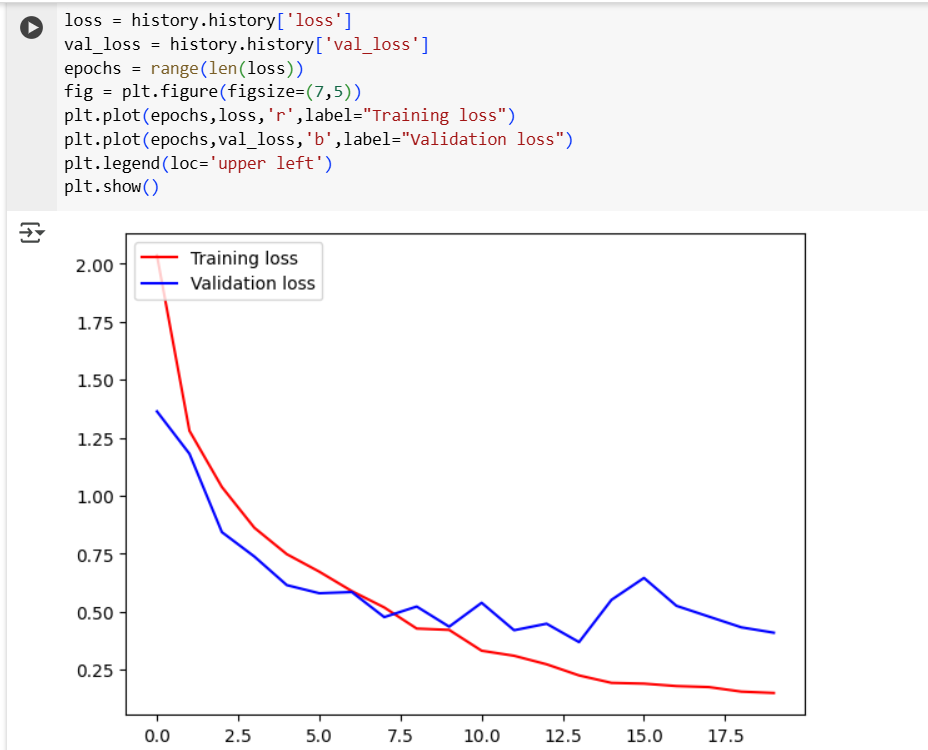
fig = plt.figure(figsize=(7,5))

plt.plot(epochs,loss,'r',label="Training loss")

plt.plot(epochs,val\_loss,'b',label="Validation loss")

plt.legend(loc='upper left')

plt.show()



**Step 7: Prediction**

**Prediction** in machine learning refers to the process of using a trained model to make inferences or forecasts about unseen data. After a model is trained on a dataset, it can predict outcomes for new, unseen inputs by applying the patterns it has learned. For instance, in classification tasks, a model might predict which class an image belongs to, while in regression tasks, it might forecast a continuous value, such as house prices.

Once the model is compiled and trained, predictions are typically made using the predict() method. The model applies the learned parameters to the new data and outputs the predicted values. In classification, the model often provides probabilities for each class, from which the final predicted class is selected (usually the one with the highest probability).

**Code:**

import cv2

import numpy as np

import requests

from io import BytesIO

from PIL import Image

# Load image from URL

url = 'https://media.springernature.com/full/springer-static/image/art%3A10.1038%2Fs41598-023-41576-6/MediaObjects/41598\_2023\_41576\_Fig1\_HTML.jpg'

response = requests.get(url)

img = Image.open(BytesIO(response.content))

# Convert to OpenCV format

img = np.array(img)

img = cv2.cvtColor(img, cv2.COLOR\_RGB2BGR)

# Resize

img = cv2.resize(img, (150, 150))

# Convert to numpy array

img\_array = np.array(img)

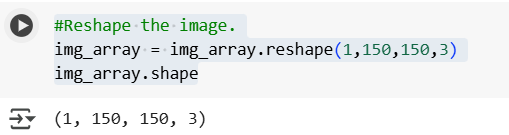
print(img\_array.shape)



#Reshape the image.

img\_array = img\_array.reshape(1,150,150,3)

img\_array.shape



import matplotlib.pyplot as plt

import requests

from PIL import Image

from io import BytesIO

# URL of the image

url = "https://media.springernature.com/full/springer-static/image/art%3A10.1038%2Fs41598-023-41576-6/MediaObjects/41598\_2023\_41576\_Fig1\_HTML.jpg"

# Load the image from the URL

response = requests.get(url)

img = Image.open(BytesIO(response.content))

# Display the image

plt.imshow(img)

plt.axis("off")  # Hide axes

plt.show()



#predict is a probability array of 4 values(labels).

a=model.predict(img\_array)

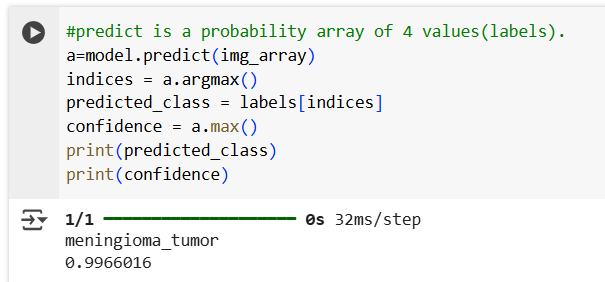
indices = a.argmax()

predicted\_class = labels[indices]

confidence = a.max()

print(predicted\_class)

print(confidence)



**CHAPTER 4**

**CONCLUSION**

The brain tumor detection model developed using Convolutional Neural Networks (CNN) provides an effective solution for automating the diagnosis of brain tumors from MRI images. By leveraging deep learning techniques, the model significantly reduces the time and potential errors associated with manual detection, **aiding radiologists in making faster** and more **accurate diagnoses**. The dataset was properly divided into training and testing subsets, and the CNN architecture, consisting of multiple convolutional, pooling, and dropout layers, allows the model to effectively extract important features from the images.

The use of libraries like TensorFlow, Keras, and scikit-learn made model building, training, and evaluation easier, while Matplotlib and Seaborn enabled the visualization of model performance metrics, such as accuracy and loss over epochs. The trained model can predict brain tumor types or identify non-tumor cases with high confidence, making it a valuable tool in clinical settings.

**Technologies Used:**

1. **TensorFlow/Keras**:
   * **Usage**: TensorFlow is an open-source machine learning framework, and Keras is its high-level API for building neural networks. In this project, TensorFlow and Keras were used to design, compile, and train the Convolutional Neural Network (CNN) model. The primary role of TensorFlow is to perform the heavy computational tasks required for deep learning, while Keras simplifies the process of building complex neural network architectures.
   * **Key Functions**:
     + **Model Building**: Layers of the CNN (e.g., Conv2D, MaxPooling2D, Dense) were added using the Keras Sequential model, providing a clear, step-by-step architecture of the neural network.
     + **Model Compilation**: Keras was used to compile the model by specifying the loss function (categorical cross-entropy), optimizer (Adam), and evaluation metrics (accuracy).
     + **Model Training**: The model was trained using the fit() function, which takes in the training data, adjusts weights using backpropagation, and evaluates the performance on the validation dataset during each epoch.
2. **OpenCV**:
   * **Usage**: OpenCV (Open Source Computer Vision Library) is a popular library for computer vision tasks. It was used here to handle image preprocessing, such as reading, resizing, and converting images to the correct format before feeding them into the neural network.
   * **Key Functions**:
     + **Image Loading**: OpenCV was used to load images from file paths and URLs.
     + **Image Resizing**: Before passing images to the CNN model, OpenCV resized them to a uniform size (150x150 pixels), which is necessary for consistency and compatibility with the network's input layer.
     + **Image Conversion**: OpenCV helped convert images from color formats (RGB to BGR) and normalized the pixel values to ensure proper model input.
3. **Matplotlib/Seaborn**:
   * **Usage**: Both Matplotlib and Seaborn are powerful Python libraries for creating data visualizations. In this project, they were used to visualize the model's training progress, specifically the accuracy and loss curves over multiple epochs.
   * **Key Functions**:
     + **Matplotlib**: Used to plot custom, high-quality visualizations of training and validation metrics, like accuracy and loss, over time. The plt.plot() function visualizes how the model's performance improves (or not) over the epochs.
     + **Seaborn**: Built on top of Matplotlib, Seaborn provides an easier syntax and aesthetic plots. In this project, while Seaborn wasn't directly used for model training, it can be leveraged to create visually appealing and statistically insightful visualizations in other aspects of data analysis, like confusion matrices or heatmaps.
     + **Example Visualizations**: Accuracy curves, loss curves, and performance comparisons between training and validation data. This helps in understanding whether the model is overfitting or underfitting.
4. **Scikit-Learn**:
   * **Usage:** Scikit-learn is a machine learning library in Python that provides tools for data analysis, including model evaluation and preprocessing. In this project, it was used for the following purposes:
     + **Train-Test Split:** The train\_test\_split() function was used to split the dataset into training and testing sets, ensuring the model could be trained on one portion of the data and evaluated on another.
     + **Evaluation Metrics:** Scikit-learn provides functions like accuracy\_score to compute and evaluate the model's performance after predictions.
     + **Data Preprocessing:** Functions like shuffle() were used to randomize the dataset, improving the model's ability to generalize.

**Further Improvements:**

1. **Transfer Learning**: Incorporating pre-trained models like VGG16 or ResNet could improve performance, especially when data is limited.
2. **Data Augmentation**: Techniques such as image rotation, flipping, or zooming can help increase the diversity of the training dataset and improve generalization.
3. **Real-time Prediction**: The model could be integrated into a real-time diagnostic tool for use in hospitals, making the detection process even more efficient.