



## GROUP ASSIGNMENT

CT118-3-3-ODL

OPTIMIZATION AND DEEP LEARNING

CSSE\_\_CT118-3-3-ODL-L-4\_\_2023-09-27

HAND OUT DATE: 12 OCTOBER 2023

HAND IN DATE: 19 JANUARY 2024

WEIGHTAGE: 100%

---

### INSTRUCTIONS TO CANDIDATES:

1. Submit your assignment via Moodle.
2. Students are advised to underpin their answers with the use of references (cited using the 7th Edition of APA Referencing Style).
3. Late submission will be awarded zero (0) unless Extenuating Circumstances (EC) are upheld.
4. Cases of plagiarism will be penalized.
5. You must obtain 50% overall to pass this module.

Name	TP Number
Esmail Qaid Saleh Ali Dhaif Allah	TP062083
DIPTA PROTIM GUHA	TP063351
DHASSRI PRABHAT ERNDU	TP061547

**Lecturer. Raheem Mafas**

**Hand in Date: 19<sup>th</sup> January 2024**

## Table of Contents

Introduction and Problem statement .....	4
Dataset Selection.....	4
Model selection .....	4
Convolutional Neural Networks (CNNs).....	4
Artificial Neural Network (ANN): .....	5
Deep Neural Network (DNN): .....	5
Tuning Strategies:.....	5
literature Review .....	6
Convolutional Neural Networks (CNNs).....	6
Artificial Neural Networks (ANNs).....	7
Deep Neural Networks (DNNs) .....	7
obstacles and Future paths.....	7
EDA Exploratory Data analysis .....	7
Data loading .....	7
Distribution of data.....	8
Visualization The data .....	12
Summary of EDA.....	13
Data Preprocessing.....	13
Data Augmentation.....	16
Data splitting .....	18
Model building.....	20
Convolutional Neural Network (CNN) – ESMail QAID TP062083 .....	21
Base Model:.....	21
Model 2 different architecture.....	26
Tuned model.....	28
CNN Models comparison and summary .....	34

Deep Neural Networks (DNN) Dipta Protim Guha TP063351 .....	36
Base VS Final Model Evaluation .....	43
Artificial neural networks (ANNs) – DHASSRI PRABHAT ERNDU(TP061547) .....	45
Task 3: Model Evaluation & Discussion.....	54
Conclusion and Neural Networks Comparison.....	59
Comparative Analysis .....	59
Improvement Suggestions .....	59
Future Directions.....	59
References.....	61

## **Introduction and Problem statement**

Worldwide, brain cancers pose a significant risk to public health because they are so common and because they make up the bulk of central nervous system primary tumors. With 11,700 new diseases diagnosed every year, precise diagnosis and treatment planning are crucial. The low 5-year survival rate for brain tumors highlights the critical need for new methods of detection and classification. Magnetic resonance imaging (MRI), the diagnostic tool of choice for brain cancer, generates massive amounts of data that are often examined by hand. Due to the complexity and intricacy of brain tumors, manual inspection carries the risk of human mistake. The most successful ways to overcome this have been automated classification systems based on artificial intelligence (AI) and machine learning (ML).

Models such as Convolutional Neural Networks (CNNs), Artificial Neural Networks (ANNs), and Deep Neural Networks (DNNs) will be used in this research to construct Deep Learning Algorithms. Our goal is to improve radiologists' diagnostic skills worldwide using these models plus a dataset of 1,500 brain MRI images that have been tagged with "yes" and "no" categories.

## **Dataset Selection**

The dataset comprises 30,060 magnetic resonance imaging (MRI) scans of the brain, labeled as "yes," "no," or "pred." The 'yes' and 'no' files, which include 1500 Brain MRI photos each, are the primary files of interest. Training neural network models that can differentiate between benign and malignant brain tumors is the objective of this painstakingly produced dataset.

## **Model selection**

### **Convolutional Neural Networks (CNNs)**

An example of an artificial neural network application is convolutional neural networks, which are used for image recognition. Traditional artificial neural networks suffer from spatial invariance; convolutional neural networks (CNNs) use convolution kernels to extract feature information from images, pool computation to simplify things, and finally convert two-dimensional images to a one-dimensional array that computers can understand. Feature engineering using SIFT and kernel techniques were the mainstays of picture identification until AlexNet was introduced in 2012. But then convolution kernels came along and made it possible to learn picture aspects independently, which opened up all sorts of new possibilities.

According to Xuebin Yue, Zhichen Wang, and Hengyi Li, CNNs perform very well in image classification and are particularly useful for extracting hierarchical features from images. Using our approach, MRI scans for tumor classification will be much easier, since it is very good at seeing patterns and nuances. A particular sort of neural network called a convolutional neural network (CNN) is used by many image and speech recognition applications. Its in-built convolutional layer reduces visual dimensionality while preserving information. Hence, CNNs are the best choice for this job (Niklas Lang, 2021).

### **Artificial Neural Network (ANN):**

Because of their ability to grasp intricate connections within the information, ANNs will be used. The creation of sophisticated representations is made possible by the interwoven layers, which help in the accurate categorization of brain tumors. Neural networks that are artificial attempt to simulate how the human brain operates. Seeing electromagnetic waves in the visible light spectrum is the initial step in image processing. These waves are then sent from the eyes to the visual cortex of the brain via the neurological system. There, they undergo processing by various cortical layers to get the necessary data. Like the human brain, artificial neural networks (ANNs) include an input layer, many hidden layers, and an output layer, say Thomas Hofmann, Bernhard Schölkopf, and Alexander J. Smola.

### **Deep Neural Network (DNN):**

Similar to artificial neural networks, deep neural networks (DNNs) are methods for machine learning that aim to mimic the way the brain analyzes data. According to Good enough et al. (2016), DNN removes a large number of hidden layers (l) between the input and output layers. Multi-hidden-layer DNNs excel at gathering intricate data representations. To improve the accuracy of diagnoses, this model will use depth to comprehend the intricate features of brain tumours. Krizhevsky et al., 2012 shown that deep neural networks (DNNs) are effective for image classification and other vision tasks.

### **Tuning Strategies:**

For these models to function at their best, we will use a variety of hyperparameter optimization methodologies. Two methods exist for hyperparameter optimization: Random Search and Grid. Neural network models will be more resilient if they are paused early and have checkpoints set up to prevent overfitting.

Finally, impressive Deep Learning models have great potential for improving diagnosis accuracy and, in the long run, patient outcomes when applied to the categorization of brain tumors using MRI images. Researching convolutional neural networks (CNNs), artificial neural networks (ANNs), and deep neural networks (DNNs) offers a holistic approach to addressing the challenges of brain tumor classification.

## **Literature Review**

An abnormal cluster of cells that develops in or around the brain is known as a brain tumor. There are two main types of brain tumors: benign and malignant. There are two main types of brain tumors: benign and malignant. In contrast to malignant tumors, benign tumors do not cause cancer. The hallmarks of a malignant tumor are rapid growth, invasion of neighboring organs, and frequent recurrence after therapy. Slow growth, well defined boundaries, and a lack of potential to invade nearby brain cells are characteristics of benign tumors. Brain tumors may interfere with brain activities by compressing nearby brain components, which can create substantial issues due to the confined space within the skull. Brain tumors may be either primary or secondary (metastatic). While primary brain tumors originate in the brain itself, secondary brain malignancies originate from tumors that have spread to the brain from other parts of the body. While primary brain tumors might be benign or cancerous, secondary tumors are always malignant (Samundi et al., 2022).

In particular, deep learning algorithms have revolutionized the use of magnetic resonance imaging (MRI) for the classification of brain tumors. Recent years have seen remarkable progress in the automation and accuracy of brain tumor detection with the use of convolutional neural networks (CNNs), artificial neural networks (ANNs), and deep neural networks (DNNs). These developments are examined in this literature review.

### **Convolutional Neural Networks (CNNs)**

A large body of research shows that CNNs may effectively filter out complex information from MRI images. Accurate tumor classification was shown by CNNs in the works of Havaei et al. (2017) and Pereira et al. (2018), who demonstrated that CNNs can learn hierarchical representations automatically. By graphically depicting learnt properties, these models make interpretability possible and are very accurate.

**Artificial Neural Networks (ANNs)**

The capacity of ANN designs to capture complicated interactions has been studied by academics like Wang et al., 2016 and Aamir et al., 2022. Jobs involving brain tumor categorization are good fits for ANNs because of their ability to analyze large amounts of data and understand complex patterns. Researchers found that ANNs performed well across a range of data complexity levels.

**Deep Neural Networks (DNNs)**

A number of recent research have investigated how DNNs may improve accuracy in brain tumor categorization as stated by Shen et al., 2017. Diagnostic accuracy is enhanced by the depth of DNNs, which enables the differentiation of minute features. These models can handle massive volumes of data and provide a comprehensive picture of tumor features.

These models cannot operate without proper hyperparameter adjustment. When using methods like Random Search and Grid Search, it is crucial to do cross-validation, as highlighted in the important works Wang et al., 2016. The longevity of the model and its protection from overfitting are guaranteed by these optimization techniques.

**obstacles and Future paths**

Issues with interpretability, dataset bias, and generalizability remain, despite substantial advances in the literature. Overcoming these obstacles and investigating novel architectures like attention processes and transfer learning will be the primary goals of future research into improving model performance.

**EDA Exploratory Data analysis****Data loading**

The data is uploaded from local directory for usage in the project. The import code is provided below in the picture.

```
import os
import matplotlib.pyplot as plt

# Path to the dataset directory
dataset_dir = "MRI_Scan_dataset"

# Directories for 'yes' and 'no' categories
yes_dir = os.path.join(dataset_dir, 'yes')
no_dir = os.path.join(dataset_dir, 'no')
```

*Figure 1 Data loading*

## Distribution of data

This dataset's distribution of images labeled "brain with tumor" and "brain with no tumor" is shown in the code below.

```
import os
import matplotlib.pyplot as plt

# Path to the dataset directory
dataset_dir = "MRI_Scan_dataset"

# Directories for 'yes' and 'no' categories
yes_dir = os.path.join(dataset_dir, 'yes')
no_dir = os.path.join(dataset_dir, 'no')

# Counting the number of images in each directory
num_yes_images = len([name for name in os.listdir(yes_dir) if os.path.isfile(os.path.join(yes_dir, name))])
num_no_images = len([name for name in os.listdir(no_dir) if os.path.isfile(os.path.join(no_dir, name))])

# Creating a bar chart to visualize the distribution
categories = ['Yes (Brain with tumor)', 'No (Brain with no tumor)']
counts = [num_yes_images, num_no_images]

bars = plt.bar(categories, counts, color=['red', 'green'])
plt.xlabel('Categories')
plt.ylabel('Number of Images')
plt.title('Distribution of Images in the Dataset')

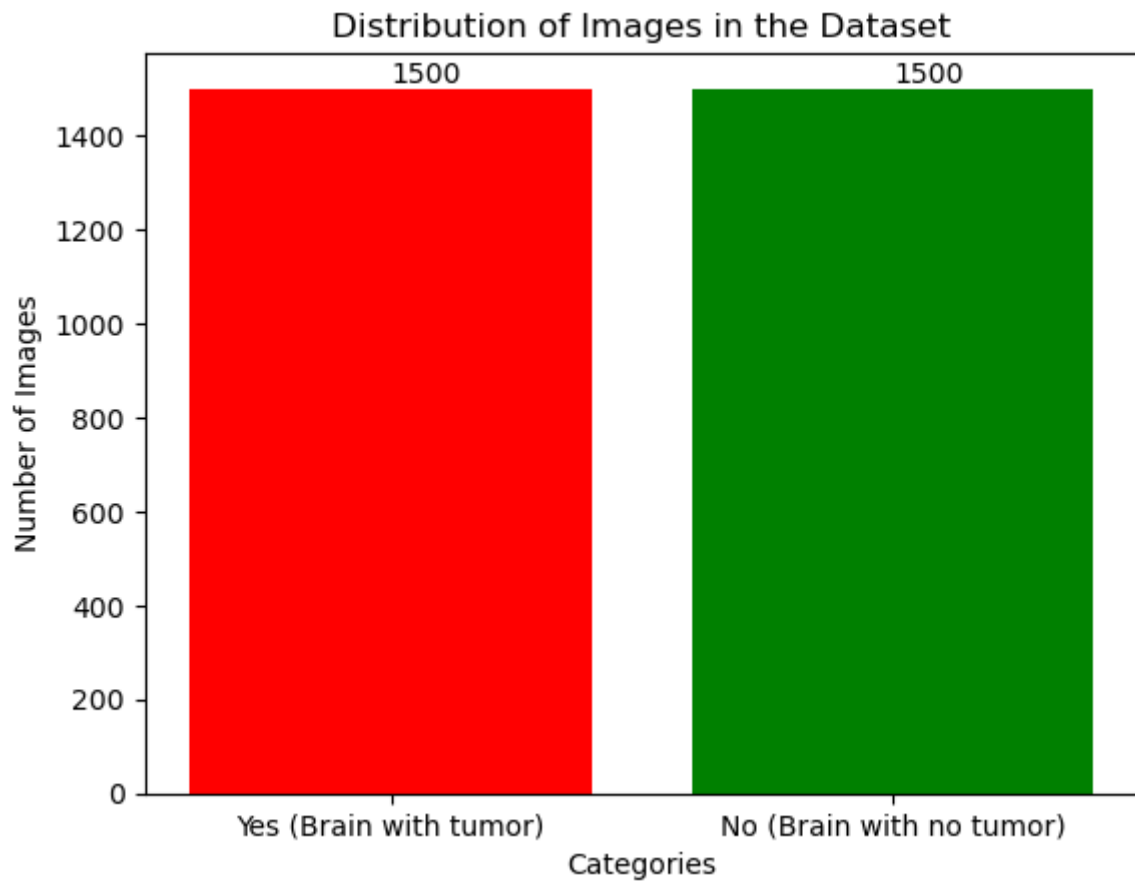
# Adding count labels on the bars
for bar in bars:
    yval = bar.get_height()
    plt.text(bar.get_x() + bar.get_width()/2, yval, int(yval), va='bottom') # va: vertical alignment

plt.show()
```

*Figure 2 Distribution of data*

The figure below shows the results of the two-class distribution. There is no need for balancing processes since the data is sufficiently balanced, with a total of 1500 brain MRI scan pictures per class.





*Figure 3 Distribution visualization*

Researchers next looked into the picture aspect ratios that would be used to train the model using the code shown in figure 4.

```
from PIL import Image
# Lists to store widths and heights of images
widths = []
heights = []

# Function to load images and get dimensions
def get_image_dimensions(directory):
    for filename in os.listdir(directory):
        if filename.endswith(".jpg") or filename.endswith(".png"):
            img = Image.open(os.path.join(directory, filename))
            width, height = img.size
            widths.append(width)
            heights.append(height)

# Getting dimensions of 'yes' and 'no' images
get_image_dimensions(os.path.join(dataset_dir, 'yes'))
get_image_dimensions(os.path.join(dataset_dir, 'no'))

# Creating a scatter plot of image widths and heights
plt.scatter(widths, heights)
plt.xlabel('Width')
plt.ylabel('Height')
plt.title('Distribution of Image Dimensions')
plt.show()
```

✓ 1.8s

```
# Set to store unique dimensions
unique_dimensions = set()

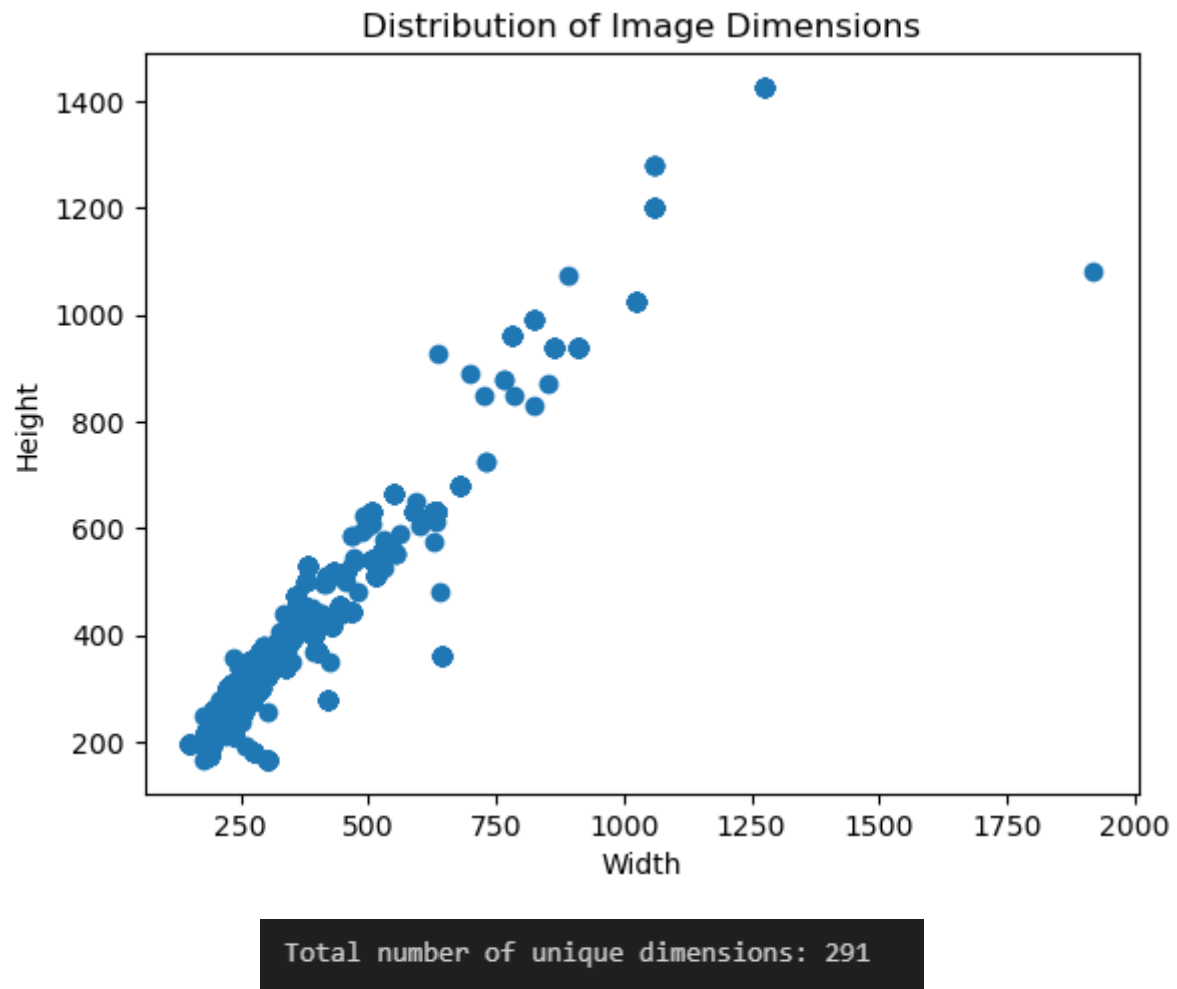
# Function to load images and get unique dimensions
def get_unique_dimensions(directory):
    for filename in os.listdir(directory):
        if filename.endswith(".jpg") or filename.endswith(".png"):
            img = Image.open(os.path.join(directory, filename))
            width, height = img.size
            unique_dimensions.add((width, height))

# Getting unique dimensions of 'yes' and 'no' images
get_unique_dimensions(os.path.join(dataset_dir, 'yes'))
get_unique_dimensions(os.path.join(dataset_dir, 'no'))

# Displaying the total number of unique dimensions
print(f"Total number of unique dimensions: {len(unique_dimensions)}")
```

✓ 0.6s

Figure 4 count and distribution of dimensions in the dataset



*Figure 5 Unique Dimensions Count and distribution of dimensions*

With a grand total of 291 images, Figure 5 shows that the picture data has a great deal of unique dimensions. This analysis shows that the proportions of the pictures differ between the yes and no photographs, rather than being constant for all photos. In addition, most images have dimensions less than 100 by 100, as seen in the scatter plot. The remaining photos could be outliers, therefore we'll have to scale them to be uniform in size when we prepare the data.

## Visualization The data

The code used to visualize the data from the brain MRI images is presented in figure 6, which is located below.

```
# Function to display sample images from a directory
def display_sample_images(directory, title, num_samples=5):
    filenames = os.listdir(directory)
    sample_filenames = filenames[:num_samples]

    plt.figure(figsize=(15, 5))
    plt.suptitle(title)

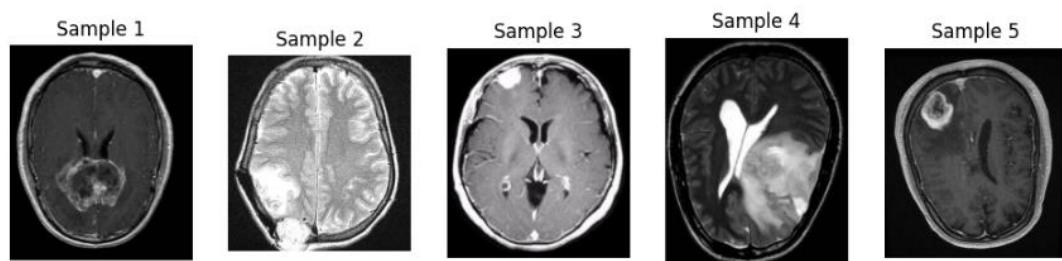
    for i, filename in enumerate(sample_filenames):
        img = Image.open(os.path.join(directory, filename))
        plt.subplot(1, num_samples, i+1)
        plt.imshow(img, cmap='gray')
        plt.axis('off')

    plt.tight_layout()
    plt.show()

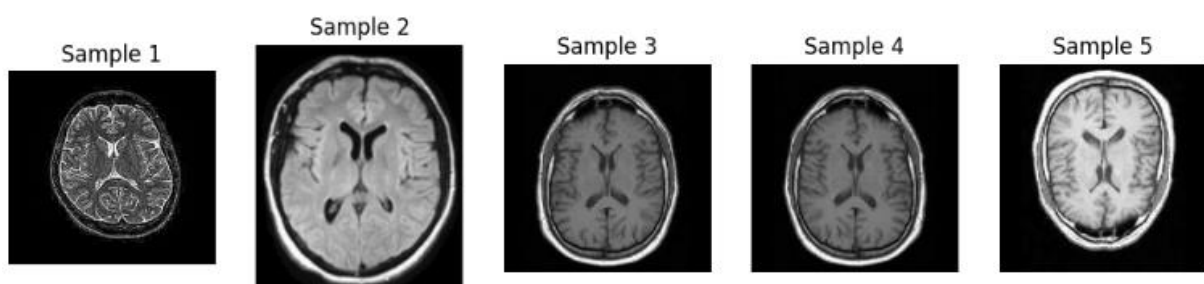
# Displaying sample images from both categories
display_sample_images(os.path.join(dataset_dir, 'yes'), "Sample Images with Tumor")
display_sample_images(os.path.join(dataset_dir, 'no'), "Sample Images without Tumor")
```

*Figure 6 code to visualize samples of the data.*

Data developers will readily recognize the samples from the yes "with tumor" and no "no tumor" classes in Figures 7 and 8, respectively. Both the size of the image and the brain inside it might fluctuate.



*Figure 7 yes "Sample Images with Tumor" class*



*Figure 8 no "Sample Images without Tumor" class*

## Summary of EDA

There are a total of three thousand photographs that will be used in the creation of the model; 1500 images from each of the yes and no classes make up the data. The project's preprocessing step will address the issue of the photographs' varied dimensions, which amount to 291 unique dimensions. Figures 7 and 8 show instances of images with varying sized brains; one picture has a small brain and the background occupies half the frame; this is something that has to be taken into account while preparing the data. To ensure the model is fed high-quality data and, by extension, generates accurate results, the aforementioned challenges should be handled during data preparation.

## Data Preprocessing

With 1500 photographs from each yes and no class, for a grand total of 3000 images, the data is evenly distributed and may be used to build models. The project's preprocessing step will address the issue of the photographs' varied dimensions, which amount to 291 unique dimensions. Figures 7 and 8 show instances of images with varying sized brains; one picture has a small brain and the background occupies half the frame; this is something that has to be taken into account while preparing the data. To ensure the model is fed high-quality data and, by extension, generates accurate results, the aforementioned challenges should be handled during data preparation.

```
#Data preprocessing
[8] ✓ 0.0s

import cv2
import numpy as np

def crop_brain_contour(image, output_size=(224, 224)):
    # Convert the image to grayscale
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

    # Apply binary thresholding
    _, thresh = cv2.threshold(gray, 45, 255, cv2.THRESH_BINARY)

    # Find contours
    contours, _ = cv2.findContours(thresh, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

    # Get the largest contour
    largest_contour = max(contours, key=cv2.contourArea)

    # Get the extreme points of the largest contour
    x_min, x_max, y_min, y_max = 10000, 0, 10000, 0
    for coord in largest_contour:
        x, y = coord[0]
        x_min = min(x_min, x)
        x_max = max(x_max, x)
        y_min = min(y_min, y)
        y_max = max(y_max, y)

    # Crop the image using the extreme points
    cropped_img = image[y_min:y_max, x_min:x_max]

    # Resize the cropped image to the specified output size
    resized_img = cv2.resize(cropped_img, output_size, interpolation=cv2.INTER_AREA)

    return resized_img
```

*Figure 9 Data preprocessing function*

The code for the crop brain contour function is shown in Figure 9. This function takes image input, preprocesses it, and then returns the picture. In order to get brain MRI data in the best possible form for model learning, there are a number of procedures that must be followed along the preprocessing path. The raw MRI image is first grayscaled. This straightforward method merges all of the RGB channels into one, thereby reducing the dimensionality of the image data. An essential part of image processing is making the picture simpler without losing any of the structural details.

After the grayscale conversion, the picture is subjected to a binary thresholding method. Pixel values below a certain threshold (here 45) are converted to black and those above to white; this method then uses this information to outline the brain structure. Hence, the white-on-black brain region is easily discernible in the binary image. Contour detection, the next step, benefits from this stark difference. To locate the boundaries of white areas in a binary image, contour detection is used. Our focus is just on the surface, or the most superficial contours. Although other shapes may be seen, the most prominent one is said to correspond to the brain's shape. We will proceed with processing this shape.

Find the top, left, and bottom points of the biggest shape—the brain—and label it. The brain is encircled by this rectangle formed by these two extremes. In order to isolate the brain and eliminate unnecessary noise or background in the original MRI picture, this rectangle is used for cropping. Before processing can begin, resizing must be done. To guarantee data integrity and consistency, the developer specifies the desired dimension for the cropped brain picture. Machine learning models that need constant input sizes really need this.

Important steps in developing and analyzing deep learning models are performed by this preprocessing pipeline on raw Brain MRI images. These steps include extracting the brain's structure, minimizing noise, and standardizing picture size.

To illustrate the operation of the functions, Figure 10 illustrates the code for creating an image. Figures 11 and 12 show the preprocessed and post-processed brain MRI scans, respectively.

```
import cv2
import matplotlib.pyplot as plt

def display_before_after(image_path):
    # Load the image
    original_image = cv2.imread(image_path)
    original_image = cv2.cvtColor(original_image, cv2.COLOR_BGR2RGB) # Convert from BGR to RGB

    # Apply the preprocessing function
    preprocessed_image = crop_brain_contour(original_image.copy())

    # Display the original and preprocessed images
    plt.figure(figsize=(10, 5))

    plt.subplot(1, 2, 1)
    plt.imshow(original_image, cmap='gray')
    plt.title("Before Preprocessing")
    plt.axis('off')

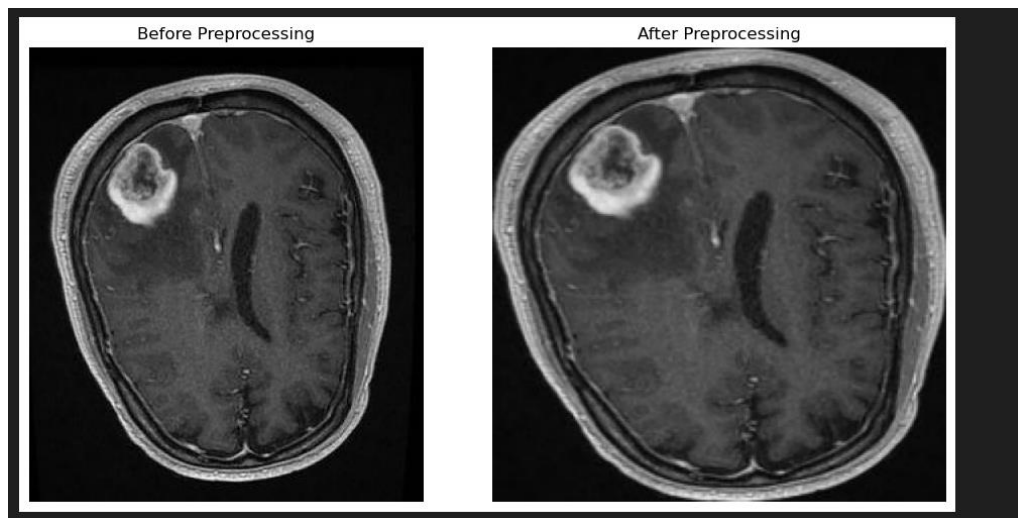
    plt.subplot(1, 2, 2)
    plt.imshow(preprocessed_image, cmap='gray')
    plt.title("After Preprocessing")
    plt.axis('off')

    plt.tight_layout()
    plt.show()

# Provide the path to your image
image_path = "MRI_Scan_dataset/yes/y110.jpg"

# Call the display function
display_before_after(image_path)
```

*Figure 10 sample processing code*



*Figure 11 Before and after preprocessing*

The function crop brain contour successfully removed unnecessary MRI portions while focusing just on the brain, as seen in the before and after photographs in figures 11. When we go on to data augmentation and splitting, we'll apply the function to every single picture.

## Data Augmentation

Approaches to data augmentation enhance model performance and accuracy by supplementing and enriching data. Data augmentation methods include modifications into datasets to save operating expenses. High accuracy models need data augmentation, which facilitates data cleansing. By adding variances to the model, data augmentation makes machine learning more robust (Pragya Soni, 2022). Visualizations are made better with data augmentation. Some examples of image classification changes include mirroring, cropping, rotating, resizing, padding, grayscale scaling, adjusting brightness and darkness, randomly erasing, and geometric transformation.

The data augmentation method incorporates fresh synthetic photographs into picture categorization using a variety of technologies, such as generative adversarial networks and the unity game engine (Pragya Soni, 2022).

In Figure 13, we can see the code that gathers photos from the directory and saves their labels as yes in the file yes. Then, the function moves on to directory no and collects images with labels as no.

```
import os
import cv2
import shutil
from tensorflow.keras.preprocessing.image import ImageDataGenerator

def get_image_paths_and_labels(data_path, label):
    images = [os.path.join(data_path, label, img) for img in os.listdir(os.path.join(data_path, label))]
    labels = [label] * len(images)
    return images, labels
```

*Figure 12 get images and lables code*

The lists yes images and no images will include the taken photos, while the lists yes labels and no labels will have the corresponding category labels. Figure 14 shows the subsequent process of merging the two classes into a single list with corresponding labels.



```
# Your dataset directory
dataset_dir = "MRI_Scan_dataset"

# Getting image paths and labels
yes_images, yes_labels = get_image_paths_and_labels(dataset_dir, 'yes')
no_images, no_labels = get_image_paths_and_labels(dataset_dir, 'no')

# Combining all images and labels
all_images = yes_images + no_images
all_labels = yes_labels + no_labels

# Calling the augment_images function with the crop_brain_contour function or any other preprocessing function you have
augment_images(dataset_dir, all_images, all_labels, augmentations_per_image=10, image_processing_functions=crop_brain_contour)
```

*Figure 13 getting images and labels code 1*

```
def augment_images(data_path, all_images, all_labels, augmentations_per_image=10, image_processing_functions=None):
    datagen = ImageDataGenerator(
        rescale=1./255,
        rotation_range=10,
        zoom_range=0.1,
        width_shift_range=0.1,
        height_shift_range=0.1,
        horizontal_flip=True
    )

    augmented_path = os.path.join(data_path, 'augmented')
    if os.path.exists(augmented_path):
        shutil.rmtree(augmented_path)

    for img_path, label in zip(all_images, all_labels):
        img = cv2.imread(img_path)
        img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
        if image_processing_functions:
            img = image_processing_functions(img)
        img = img.reshape((1,) + img.shape)
        save_prefix = os.path.basename(img_path).split('.')[0]
        save_dir = os.path.join(augmented_path, label)
        if not os.path.exists(save_dir):
            os.makedirs(save_dir)

        i = 0
        for batch in datagen.flow(img, batch_size=1, save_to_dir=save_dir, save_prefix=save_prefix, save_format='jpg'):
            i += 1
            if i >= augmentations_per_image:
                break
```

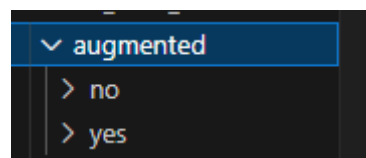
*Figure 14 function for image augmentation*

The function enhance pictures are shown in Figure 15. By expanding the dataset's diversity, this function boosts the efficiency of deep learning models. At the outset, we establish an ImageDataGenerator object with default augmentation parameters. Developers may enter their input in real-time to create augmented image data by importing this object from the Keras library. Scaling, rotating, zooming, changing the width and height, and horizontally flipping are all examples of these enhancements. When used together, these methods mimic likely real-world occurrences by subtly changing the graphics.

The path to save improved images has been established. Any preexisting directories for the augmented data are eliminated to prevent data duplication or ambiguity. An picture is converted from its original BGR format to RGB before the program reads it from the dataset. This is the point where any supplementary image processing operations are implemented. Processing of the Brain MRI scan image is done to align with the enhancements provided by ImageDataGenerator.

Each picture undergoes the augmentation operation five times, for a total of fifteen thousand photos, according to the number of augmentations specified by the developer. An structured directory is used to preserve and categorize each improved picture. This setup makes it easier to train the model by dividing the two courses and making them simple to remember.

instruction via partitioning the two classes and facilitating memorization. A more diverse training dataset, and maybe better model generalization, are the results of the augment pictures function's transformation of raw photos into improved data. After the addition of the augmentation, the data structure is shown in Figure 16.



*Figure 15 Augmented images files “yes”, ”no”*

### **Data splitting**

Training, validation, and testing are the three parts of this deep learning project's data structure. In the training phase, the model is evaluated for its learning performance by comparing the training data to validation data; the latter serves as a source from which the former learns features. Since the testing data was not fed into the model during training, it is suitable for use in the evaluation phase of the model.

Below, in figure 17, you can see the code that was used to divide the data into three separate sets.

```

import os

import pandas as pd

from sklearn.model_selection import train_test_split

from keras.preprocessing.image import ImageDataGenerator

def create_data_generators(data_path, image_size, batch_size):

    # Retrieving image paths and labels

    yes_images, yes_labels = get_image_paths_and_labels(os.path.join(data_path, 'augmented'), 'yes')

    no_images, no_labels = get_image_paths_and_labels(os.path.join(data_path, 'augmented'), 'no')

    # Asserting to ensure both image lists and label lists are of the same length

    assert len(yes_images) == len(yes_labels), "Mismatch between 'yes' images and labels"

    assert len(no_images) == len(no_labels), "Mismatch between 'no' images and labels"

    # Combining all images and labels

    all_images = yes_images + no_images

    all_labels = yes_labels + no_labels

    # Additional check after concatenation

    assert len(all_images) == len(all_labels), f"Inconsistent lengths. Images: {len(all_images)}, Labels: {len(all_labels)}"

    # Splitting the data into training (80%), validation (15%), and test sets (5%)

    train_images, temp_images, train_labels, temp_labels = train_test_split(all_images, all_labels, test_size=0.20, random_state=42)

    val_images, test_images, val_labels, test_labels = train_test_split(temp_images, temp_labels, test_size=0.25, random_state=42)

    # Creating structured pandas DataFrames from the lists of images and labels

    train_df = pd.DataFrame({'filename': train_images, 'label': train_labels})

    val_df = pd.DataFrame({'filename': val_images, 'label': val_labels})

    test_df = pd.DataFrame({'filename': test_images, 'label': test_labels})

    # Creating an ImageDataGenerator object for image rescaling

    datagen = ImageDataGenerator(rescale=1./255)

    # Creating data generators for training, validation, and testing

    train_gen = datagen.flow_from_dataframe(train_df, x_col='filename', y_col='label', target_size=image_size, batch_size=batch_size, class_mode='binary', shuffle=True)

    val_gen = datagen.flow_from_dataframe(val_df, x_col='filename', y_col='label', target_size=image_size, batch_size=batch_size, class_mode='binary', shuffle=True)

    test_gen = datagen.flow_from_dataframe(test_df, x_col='filename', y_col='label', target_size=image_size, batch_size=batch_size, class_mode='binary', shuffle=False)

    return train_gen, val_gen, test_gen

# Your dataset directory

dataset_dir = "MRI_Scan_dataset"

# Make sure augment_images doesn't change the number of images and labels, or if it does, they should remain consistent.

augment_images(dataset_dir, all_images, all_labels, augmentations_per_image=10, image_processing_functions=crop_brain_contour)

# Creating data generators

train_gen, val_gen, test_gen = create_data_generators(dataset_dir, image_size=(224, 224), batch_size=32)

```

*Figure 16 create data generator code*

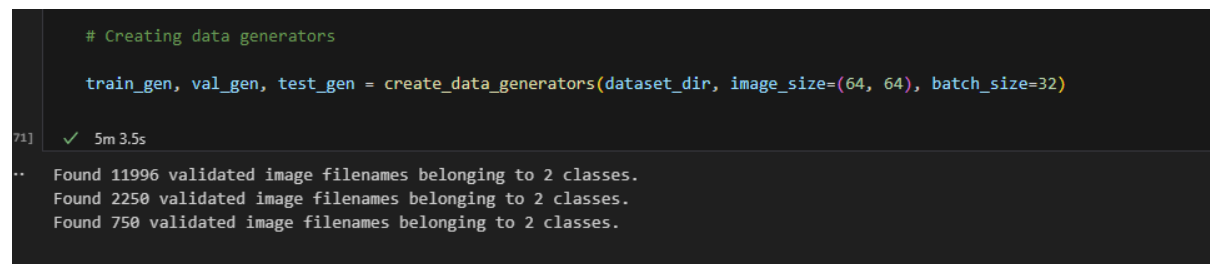
As seen in the augmentation step, the functions in Figure 17 start by collecting improved data from their respective directories. In the all images and all labels lists, each picture is kept with its appropriate label. With 80% going into training, 15% into validation, and 5% into testing, the data split is decided by the developers. To do this, we used the Keras package's train test split function. There are two split functions in the settings: one is used for training, and the other is

used to the remaining 20% of the images and labels in temporary lists. Next, we split the temporary lists in half: 25% for testing (5 percent of the total photographs) and 25% for validation (15 percent of the entire photos).

The next step in training the model is to transform the image and label lists into structured pandas DataFrames. At this point, you may communicate with the ImageDataGenerator component of Keras. An essential part of optimizing neural networks, this method constructs an ImageDataGenerator object to resize pictures.

After the datasets are formatted and the preprocessing tool is installed, the function creates many data generators for testing, validation, and training. Generators pull data from DataFrames and display it in batches so that models may be trained. While the testing data is kept in its original sequence to provide consistent assessments, the training and validation data are jumbled to create randomization.

Retrieve, organize, preprocess, and display data in batches are all made easier with the help of the construct data generators function. With this simplified method, training and evaluation of deep learning models are guaranteed to be conducted in an ideal environment. See the results of running and splitting the function in Figure 18.



```
# Creating data generators

train_gen, val_gen, test_gen = create_data_generators(dataset_dir, image_size=(64, 64), batch_size=32)

71] ✓ 5m 3.5s

.. Found 11996 validated image filenames belonging to 2 classes.
   Found 2250 validated image filenames belonging to 2 classes.
   Found 750 validated image filenames belonging to 2 classes.
```

*Figure 17 train and test data results*

## Model building

The ANN, DNN, and CNN models are shown here, along with their construction and tuning, followed by the outcomes.

## Convolutional Neural Network (CNN) – ESMail QAID TP062083

Below you can see the CNN model architectures that were used in all of the trials. The goal is to find the model that achieved the best level of accuracy. To illustrate the evolution of the model architecture and its effects on each model, we display and discuss the different designs.

### Base Model:

The basic model architecture is shown in Figure 29, and it consists of a dense layer, a flatten layer, a MaxPooling layer, and one convolutional layer. In order to extract features from the images supplied into the network, the first layer is a 30 filter convolutional layer. Every filter, which is  $3 \times 3$ , will enhance certain parts of photographs, including brain edges and textures, among other things. The parameter input shape tells the model to capture an image of the given size (64x64), and the activation function Relu is used to make sure that negative value outputs are set to zero. The third and last option stands for the RGB color channels used in the picture.

The second layer is the MaxPooling layer, which summarises the primary aspects of the original picture and minimizes the photo size without sacrificing important details by halving the features. The third and last layer, Flatten, is responsible for reducing the image to a one-dimensional array before passing it on to the classification layer.

The classification is binary (yes/no), hence in the final Dense layer we set 1; we also utilize the sigmoid activation function for this classification. Binary crossentropy is used to verify the model's performance once it is built using the "adam" optimizer.



```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense

def build_base_model(input_shape=(64, 64, 3)):
    model = Sequential([
        Conv2D(30, (3, 3), activation='relu', input_shape=input_shape),
        MaxPooling2D(pool_size=(2, 2)),
        Flatten(),
        Dense(1, activation='sigmoid')
    ])

    model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

    return model

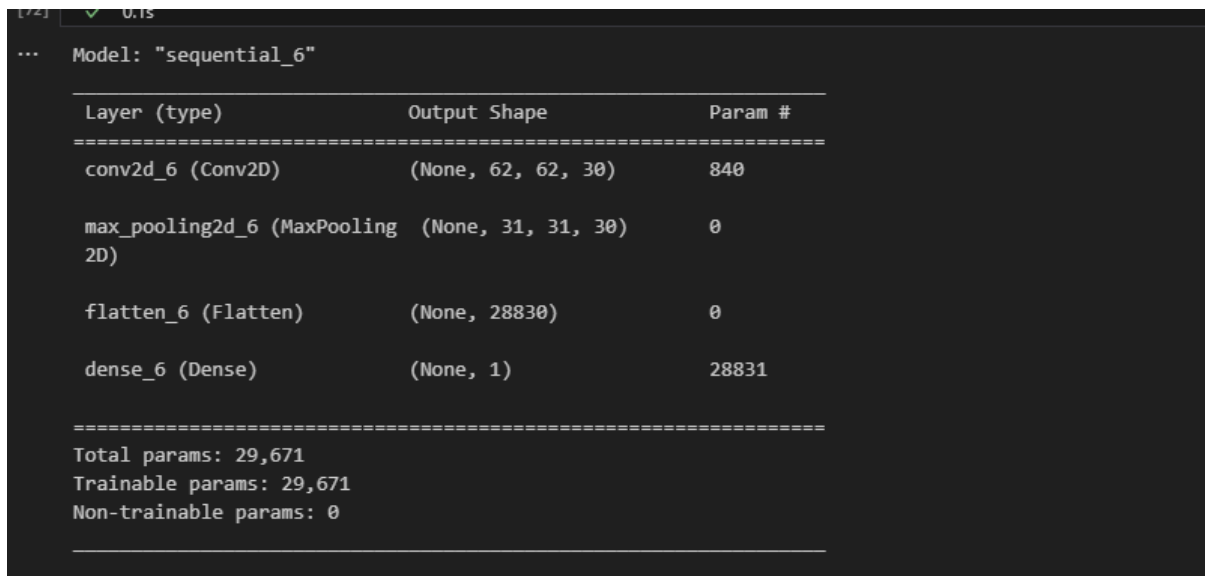
# Create the model with the specific input shape
base_model = build_base_model()

# Print the model summary to verify the architecture and parameter count
base_model.summary()
```

[72] ✓ 0.1s

Figure 18 base model

Figure 30 shows that there are 2,967,71 trainable parameters for the basic model.



The screenshot shows a Jupyter Notebook cell with the following content:

```
[72]: ✓ 0.15  
... Model: "sequential_6"
```

Layer (type)	Output Shape	Param #
conv2d_6 (Conv2D)	(None, 62, 62, 30)	840
max_pooling2d_6 (MaxPooling 2D)	(None, 31, 31, 30)	0
flatten_6 (Flatten)	(None, 28830)	0
dense_6 (Dense)	(None, 1)	28831

=====

Total params: 29,671  
Trainable params: 29,671  
Non-trainable params: 0

=====

*Figure 19 Trainable parameters for base model*

```

from tensorflow.keras.callbacks import EarlyStopping

# Instantiate the EarlyStopping callback
early_stopping_callback = EarlyStopping(
    monitor='val_accuracy', # Monitor the validation accuracy
    patience=5,             # Number of epochs to wait after min has been hit
    verbose=1,              # Output progress messages
    restore_best_weights=True # Restore model weights from the epoch with the best value of the monitored quantity
)

# Train the model with the dataset
history = base_model.fit(
    train_gen,
    validation_data=val_gen,
    epochs=100, # Set a high number since early stopping will halt the training
    callbacks=[early_stopping_callback]
)

```

```

Epoch 1/100
375/375 [=====] - 149s 396ms/step - loss: 0.6300 - accuracy: 0.6383 - val_loss: 0.5688 - val_accuracy: 0.7009
Epoch 2/100
375/375 [=====] - 26s 69ms/step - loss: 0.5020 - accuracy: 0.7594 - val_loss: 0.5067 - val_accuracy: 0.7578
Epoch 3/100
375/375 [=====] - 27s 72ms/step - loss: 0.4422 - accuracy: 0.7962 - val_loss: 0.4902 - val_accuracy: 0.7724
Epoch 4/100
375/375 [=====] - 26s 68ms/step - loss: 0.3982 - accuracy: 0.8244 - val_loss: 0.4766 - val_accuracy: 0.7840
Epoch 5/100
375/375 [=====] - 26s 70ms/step - loss: 0.3547 - accuracy: 0.8470 - val_loss: 0.5027 - val_accuracy: 0.7653
Epoch 6/100
375/375 [=====] - 28s 73ms/step - loss: 0.3144 - accuracy: 0.8705 - val_loss: 0.4846 - val_accuracy: 0.7787
Epoch 7/100
375/375 [=====] - 26s 68ms/step - loss: 0.2828 - accuracy: 0.8870 - val_loss: 0.4706 - val_accuracy: 0.7942
Epoch 8/100
375/375 [=====] - 27s 73ms/step - loss: 0.2499 - accuracy: 0.9061 - val_loss: 0.4816 - val_accuracy: 0.7920
Epoch 9/100
375/375 [=====] - 28s 75ms/step - loss: 0.2272 - accuracy: 0.9180 - val_loss: 0.4844 - val_accuracy: 0.7964
Epoch 10/100
375/375 [=====] - 27s 73ms/step - loss: 0.2047 - accuracy: 0.9271 - val_loss: 0.5262 - val_accuracy: 0.7853
Epoch 11/100
375/375 [=====] - 29s 76ms/step - loss: 0.1826 - accuracy: 0.9367 - val_loss: 0.5343 - val_accuracy: 0.7853
Epoch 12/100
375/375 [=====] - 26s 70ms/step - loss: 0.1622 - accuracy: 0.9485 - val_loss: 0.5833 - val_accuracy: 0.7787
Epoch 13/100
...
Epoch 14/100
374/375 [=====>.] - ETA: 0s - loss: 0.1239 - accuracy: 0.9680Restoring model weights from the end of the best epoch: 9.
375/375 [=====] - 26s 69ms/step - loss: 0.1239 - accuracy: 0.9680 - val_loss: 0.5721 - val_accuracy: 0.7902
Epoch 14: early stopping
Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...

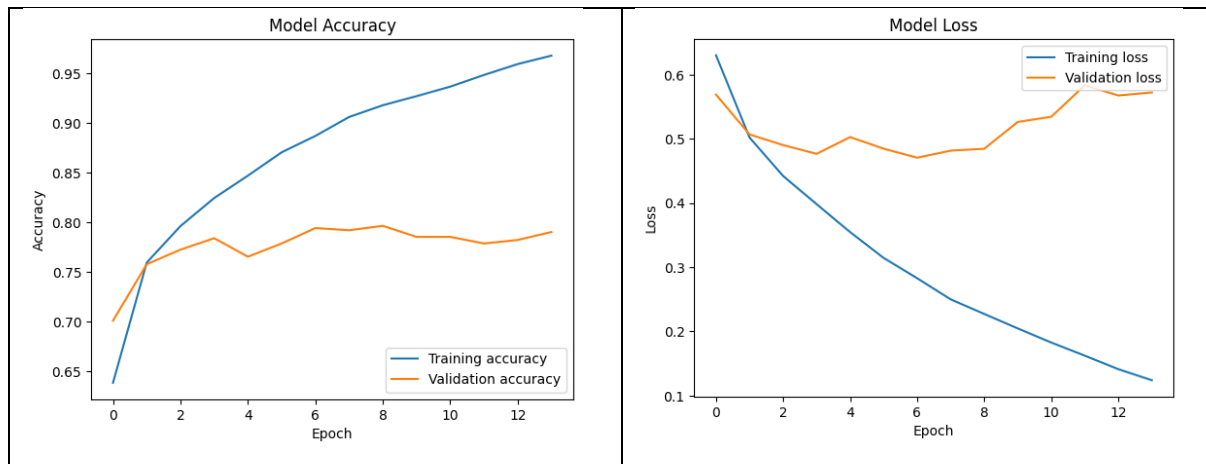
```

*Figure 20 base model*

The model was trained using the early stopping strategy to prevent the model from running for an extended period of time and to avoid overfitting. The model ends when the validation accuracy does not improve after 14 epochs of patience, as seen in figure 20.

Table 1 displays the training loss and model loss results of the first model.

Table 1



Above, you can see the graphs for model loss and model accuracy in Table 1. During the first two epochs, the model's accuracy showed that both training and validation accuracy were improving at a steady pace. However, after that, validation stopped growing and turned into a straight line, suggesting that validation accuracy had not increased. The training loss has decreased over time, whereas the validation loss showed only a little decrease before leveling out, becoming a straight line, and finally growing, as shown in the training and validation loss graphs. This shows that the model is underfitting and isn't learning enough, which means it won't be able to generalize well to new data.



```
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
import numpy as np

# Predict the output
predictions = base_model.predict(test_gen)
# Convert predictions to binary values
binary_predictions = np.round(predictions).astype(int)

# Get the true labels from the generator
true_labels = test_gen.classes

# Calculate metrics
accuracy = accuracy_score(true_labels, binary_predictions)
precision = precision_score(true_labels, binary_predictions)
recall = recall_score(true_labels, binary_predictions)
f1 = f1_score(true_labels, binary_predictions)

# Print the metrics
print(f"Accuracy: {accuracy}")
print(f"Precision: {precision}")
print(f"Recall: {recall}")
print(f"F1 Score: {f1}")
```

[77] ✓ 6.8s

... 24/24 [=====] - 7s 275ms/step  
Accuracy: 0.7786666666666666  
Precision: 0.7698630136986301  
Recall: 0.7741046831955923  
F1 Score: 0.771978021978022

Figure 21

To assess the fundamental model, we look at its recall, accuracy, precision, and f1 score metrics. These reveal the model's character. A 77.19 F1 score, a 77.41 recall, a 76.9 precision, and an accuracy of 77.86 were the results of the basic model. Based on test generating unseen data, these outcomes are satisfactory but might need greater accuracy for tasks like brain tumour categorization. As may be shown in figure 21, it is also advantageous to have greater generalizations when dealing with unknown data.

## Model 2 different architecture.

```
# model 2
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout

def build_model_2(input_shape=(64, 64, 3)):
    model = Sequential([
        Conv2D(30, (3, 3), activation='relu', input_shape=input_shape),
        MaxPooling2D(pool_size=(2, 2)),
        Dropout(0.087), # First dropout layer with 0.087 dropout rate
        Conv2D(30, (3, 3), activation='relu'),
        MaxPooling2D(pool_size=(2, 2)),
        Flatten(),
        Dropout(0.25), # Second dropout layer with 0.25 dropout rate
        Dense(1, activation='sigmoid')
    ])

    model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

    return model

# Create the model with the specific input shape
model_2 = build_model_2()

# Print the model summary to verify the architecture and parameter count
model_2.summary()
```

Figure 22 Model 2 different architecture

Model 2's design, shown in Figure 22, has a second convolutional layer for feature learning. Overfitting occurred when the model was trained using both convolutional layers, and the validation loss was higher than the accuracy loss. Because of this, the developer included two dropout layers to stop the model from memorizing input and overfitting. We ran many tests to find the sweet spot for the model, playing around with different filter densities and dropout rates.

```
Model: "sequential_7"

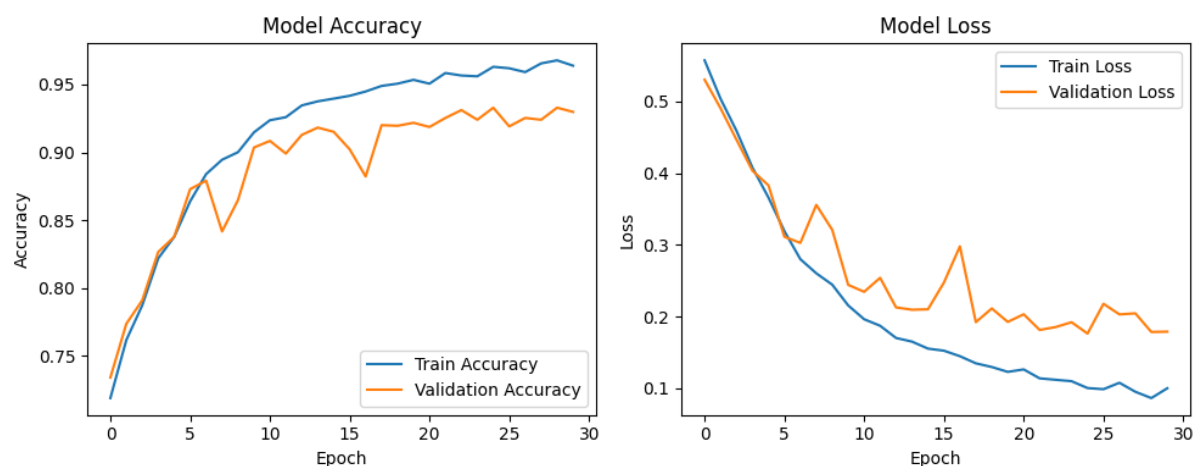
Layer (type)                 Output Shape              Param #
-----
conv2d_7 (Conv2D)            (None, 62, 62, 30)       840
max_pooling2d_7 (MaxPooling2D) (None, 31, 31, 30)       0
dropout (Dropout)            (None, 31, 31, 30)       0
conv2d_8 (Conv2D)            (None, 29, 29, 30)       8130
max_pooling2d_8 (MaxPooling2D) (None, 14, 14, 30)       0
flatten_7 (Flatten)          (None, 5880)              0
dropout_1 (Dropout)          (None, 5880)              0
dense_7 (Dense)              (None, 1)                 5881

Total params: 14,851
Trainable params: 14,851
Non-trainable params: 0
```

Figure 23 number of parameters 2<sup>nd</sup>

Figure 23 shows the number of parameters that will be used to train the model.

Table 2



The results of model 2 are shown in table 2's accuracy and loss charts. Model 2 results show that training and validation accuracy are increasing steadily with little fluctuations. The precision is about 95%, as seen by the graphs in table 2. Both the training and validation losses are decreasing at a steady rate; however, the validation loss is somewhat higher than the train loss, suggesting that the model achieves better feature understanding and detection than the simple model. In the previous model, the lowest model loss was 0.5; however, after these tweaks, the validation loss reduced to less than 0.2. We can see the training and validation losses on the right-hand side of the figure. It follows the expected trajectory of a well-fitting model, with training and validation loss both trending downward. At some point, the validation loss seems to level off or even increase somewhat; this could be a sign that the model is starting to overfit. As shown by the return of the best weights at the end of the greatest era (25), it seems that training was terminated at an appropriate time by the early halting callback. After running tests on the test generator, model 2's evaluation metrics are shown in Figure 24. Compared to the usual model, the results show much better performance across all criteria, with an F1 score of 94.0 and an accuracy of 94.1 percent.

```

from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
import numpy as np

# Reset the test generator to ensure the order of predictions matches the labels
test_gen.reset()

# Predict the output
predictions = model_2.predict(test_gen)
# Convert predictions to binary values
binary_predictions = np.round(predictions).astype(int)

# Get the true labels from the generator
true_labels = test_gen.classes

# Calculate metrics
accuracy = accuracy_score(true_labels, binary_predictions)
precision = precision_score(true_labels, binary_predictions)
recall = recall_score(true_labels, binary_predictions)
f1 = f1_score(true_labels, binary_predictions)

# Print the metrics
print(f"Accuracy: {accuracy}")
print(f"Precision: {precision}")
print(f"Recall: {recall}")
print(f"F1 Score: {f1}")

```

[84] ✓ 1.2s

```

... 24/24 [=====] - 1s 44ms/step
Accuracy: 0.9413333333333334
Precision: 0.9299191374663073
Recall: 0.9504132231404959
F1 Score: 0.9400544959128065

```

*Figure 24 evaluation of Model 2*

By comparing the outcomes of the two models, we can see that the CNN model's performance improved after we tweaked and altered the base model..

### **Tuned model.**

Hyperband tuning, which is part of the Keras package, was used by this model. Hyperband Keras Tuner allows for efficient tuning of the model's hyperparameters via resource management. Shortly after initialization, Hyperband evaluates a plethora of hyperparameter parameters to see which ones show promise. This technique quickly discovers acceptable configurations, ensuring improved model performance (Mehta, 2022). Prior to use this tool, ensure that Keras Tuner is operational by executing the code provided in Figure 25.

```

[ ] !pip install keras-tuner

```

*Figure 25 Keras Tuner Installing*

The design and the ranges of parameters that will be searched for the optimal combination are shown in figure 26.

```

import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout
from kerastuner.tuners import Hyperband
from tensorflow.keras.optimizers import Adam

def build_model(hp):
    model = Sequential(name='tuned_model')

    # First convolutional layer with tunable parameters
    model.add(Conv2D(
        filters=hp.Int('conv_1_filter', min_value=16, max_value=64, step=16),
        kernel_size=hp.Choice('conv_1_kernel', values=[3, 5]),
        activation='relu',
        input_shape=(64, 64, 3)))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Dropout(rate=hp.Float('dropout_1_rate', min_value=0.0, max_value=0.5, step=0.05)))

    # Second convolutional layer with tunable parameters
    model.add(Conv2D(
        filters=hp.Int('conv_2_filter', min_value=16, max_value=64, step=16),
        kernel_size=hp.Choice('conv_2_kernel', values=[3, 5]),
        activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Dropout(rate=hp.Float('dropout_2_rate', min_value=0.0, max_value=0.5, step=0.05)))

    model.add(Flatten())

    # Dense layer with tunable parameters
    model.add(Dense(
        units=hp.Int('dense_units', min_value=32, max_value=128, step=32),
        activation='relu'))
    model.add(Dropout(rate=hp.Float('dropout_3_rate', min_value=0.0, max_value=0.5, step=0.05)))

    # Output layer
    model.add(Dense(1, activation='sigmoid'))

    # Compile the model
    model.compile(
        optimizer=Adam(hp.Choice('learning_rate', values=[0.01, 0.001, 0.0001])),
        loss='binary_crossentropy',
        metrics=['accuracy'])

    return model

```

*Figure 26 Architecture for model 3 searching for parameters.*

In Figure 26, we can see the layout with the specified search ranges for the different parameters. The following models will be looked for by the developer:

- With each iteration of the tuner, the first convolutional layer gains 16 filters, bringing the total number of filters to 64. One of two possible filter sizes will be used.
- The first dropout layer iteratively increases the rate by 0.05 while searching for dropout rates between 0 and 0.5.
- In both the second convolutional layer and the dropout layer, the tuner looked for the same locations.
- Lastly, three different rates will be explored for the learning rate of the optimizer "adam": 0.01, 0.001, and 0.0001.

```

from kerastuner.tuners import Hyperband
import keras_tuner as kt

# Instantiate the Hyperband tuner
tuner = kt.Hyperband(
    build_model,
    objective='val_accuracy',
    max_epochs=20,
    factor=3,
    hyperband_iterations=2,
    directory='hyperband',
    project_name='brain_tumor_detection'
)

# Start the hyperparameter search
tuner.search(
    train_gen,
    validation_data=val_gen,
    epochs=20,
    callbacks=[EarlyStopping(monitor='val_accuracy', patience=5)]
)

```

*Figure 27 search hypermeters*

In Figure 27, you can see the code that ran the search. The tuner's goal is to improve val accuracy, and the number of epochs is set at 20. The model that the tuner determined to have the optimal settings will be stored in an independent directory. Since the After 24 experiment yielded the same Val accuracy every time, the author decided to end it. Figure 28 displays the results of the tuner's identification of the ideal settings, which had a validation accuracy of 95%.

```

Trial 24 Complete [00h 06m 00s]
val_accuracy: 0.4975544810295105

Best val_accuracy So Far: 0.9506666660308838
Total elapsed time: 72d 02h 31m 16s

Search: Running Trial #25

Value          |Best Value So Far|Hyperparameter
64              |16               |conv_1_filter
3               |5                |conv_1_kernel
0.35            |0.3              |dropout_1_rate
32              |16               |conv_2_filter
5               |5                |conv_2_kernel
0               |0.25             |dropout_2_rate
64              |96               |dense_units
0.05            |0.1              |dropout_3_rate
0.001           |0.001            |learning_rate
10              |10               |tuner/epochs
4               |4                |tuner/initial_epoch
1               |2                |tuner/bracket
1               |2                |tuner/round
0021            |0012             |tuner/trial_id

```

*Figure 28 Hyperband result*

```

# Define the final model architecture with the best hyperparameters
def build_final_model():
    model = Sequential(name='tuned_final_model')

    # First convolutional layer
    model.add(Conv2D(
        filters=16, # conv_1_filter
        kernel_size=5, # conv_1_kernel
        activation='relu',
        input_shape=(64, 64, 3)))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Dropout(rate=0.3)) # dropout_1_rate

    # Second convolutional layer
    model.add(Conv2D(
        filters=16, # conv_2_filter
        kernel_size=5, # conv_2_kernel
        activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Dropout(rate=0.25)) # dropout_2_rate

    model.add(Flatten())

    # Dense layer
    model.add(Dense(
        units=96, # dense_units
        activation='relu'))
    model.add(Dropout(rate=0.1)) # dropout_3_rate

    # Output layer
    model.add(Dense(1, activation='sigmoid'))

    # Compile the model
    model.compile(
        optimizer=Adam(learning_rate=0.001), # learning_rate
        loss='binary_crossentropy',
        metrics=['accuracy'])

    return model

# Create the final model instance
final_model = build_final_model()

# Print the model summary to verify the architecture
final_model.summary()

```

*Figure 29 final parameters*

This is the final tuned model with all the parameters set, as shown in figure 29:

- There are sixteen (5x5) filters for the first convolutional layer.
- The initial rate of the dropout layer is configured to be 0.3.
- There are sixteen filters measuring (5x5) pixels in the second convolutional layer.
- A rate of 0.25 is used for the second dropout layer.
- Finally, the "adam" optimizer's learning rate is 0.001.

Figure 30 shows the overall design of the model, which includes 267,409 parameters..

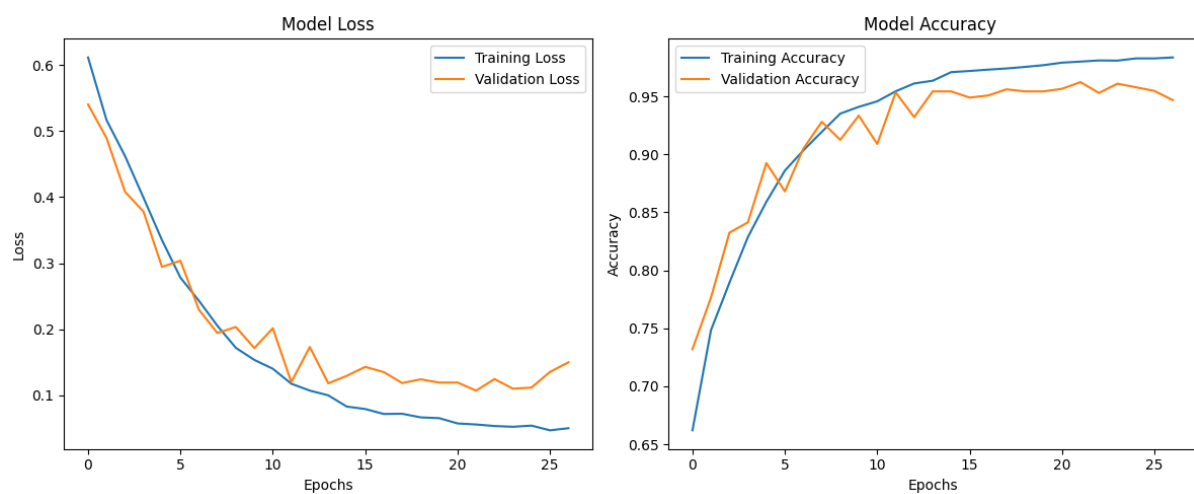
Model: "tuned\_final\_model"

Layer (type)	Output Shape	Param #
conv2d_2 (Conv2D)	(None, 60, 60, 16)	1216
max_pooling2d_2 (MaxPooling 2D)	(None, 30, 30, 16)	0
dropout_3 (Dropout)	(None, 30, 30, 16)	0
conv2d_3 (Conv2D)	(None, 26, 26, 16)	6416
max_pooling2d_3 (MaxPooling 2D)	(None, 13, 13, 16)	0
dropout_4 (Dropout)	(None, 13, 13, 16)	0
flatten_1 (Flatten)	(None, 2704)	0
dense_2 (Dense)	(None, 96)	259680
dropout_5 (Dropout)	(None, 96)	0
dense_3 (Dense)	(None, 1)	97
...		
Total params: 267,409		
Trainable params: 267,409		
Non-trainable params: 0		

Figure 30 total pararpeters

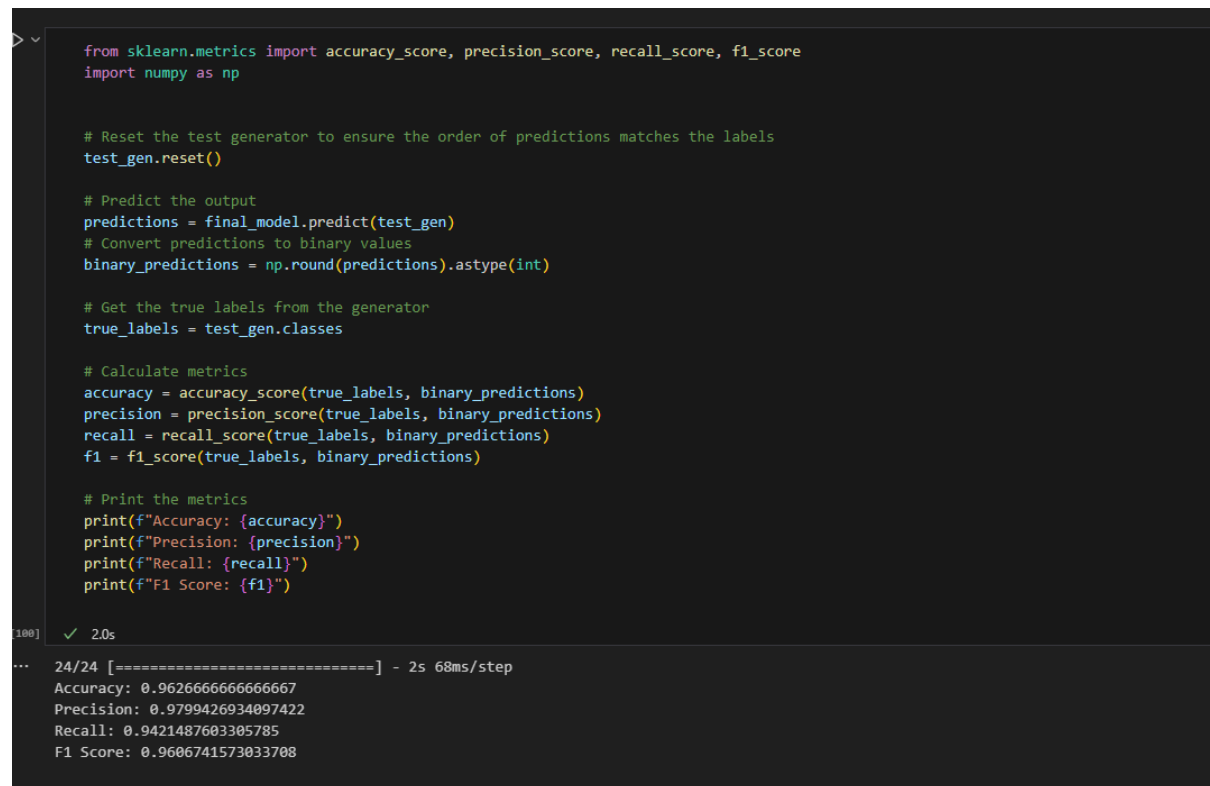
In comparison to model 2, the training and validation losses, as well as the accuracy, are more consistent (as shown in Table 3). Epoch 14 is the most successful for the model in terms of validation accuracy and validation loss.

Table 3





The model's hypertuned findings are shown in Figure 31. With an accuracy score of 96.26% and an F1 score of 96.06%, it is evident that the hypertuned model using Keras Hyperband performs better than the earlier models.



```
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
import numpy as np

# Reset the test generator to ensure the order of predictions matches the labels
test_gen.reset()

# Predict the output
predictions = final_model.predict(test_gen)
# Convert predictions to binary values
binary_predictions = np.round(predictions).astype(int)

# Get the true labels from the generator
true_labels = test_gen.classes

# Calculate metrics
accuracy = accuracy_score(true_labels, binary_predictions)
precision = precision_score(true_labels, binary_predictions)
recall = recall_score(true_labels, binary_predictions)
f1 = f1_score(true_labels, binary_predictions)

# Print the metrics
print(f"Accuracy: {accuracy}")
print(f"Precision: {precision}")
print(f"Recall: {recall}")
print(f"F1 Score: {f1}")
```

100] ✓ 2.0s

... 24/24 [=====] - 2s 68ms/step

Accuracy: 0.9626666666666667  
Precision: 0.9799426934097422  
Recall: 0.9421487603305785  
F1 Score: 0.9606741573033708

*Figure 31 tuned model evaluation metrics*

## CNN Models comparison and summary

```

# Number of models
num_models = len(models_names)

# x coordinates for each group of bars
x = np.arange(num_models)

# Create the comparison bar chart with the updated metrics
fig, ax = plt.subplots(figsize=(14, 8))

# Plotting each metric
rects1 = ax.bar(x - 1.5 * width, metrics_data['accuracy'], width, label='Accuracy')
rects2 = ax.bar(x - 0.5 * width, metrics_data['precision'], width, label='Precision')
rects3 = ax.bar(x + 0.5 * width, metrics_data['recall'], width, label='Recall')
rects4 = ax.bar(x + 1.5 * width, metrics_data['f1_score'], width, label='F1 Score')

# Add some text for labels, title and custom x-axis tick labels, etc.
ax.set_ylabel('Scores')
ax.set_title('Comparison of Model Performance Metrics')
ax.set_xticks(x)
ax.set_xticklabels(models_names)
ax.legend()

# Attach a text label above each bar, displaying its height
for rect in [rects1, rects2, rects3, rects4]:
    for r in rect:
        height = r.get_height()
        ax.annotate('{}'.format(round(height, 2)),
                    xy=(r.get_x() + r.get_width() / 2, height),
                    xytext=(0, 3),  # 3 points vertical offset
                    textcoords="offset points",
                    ha='center', va='bottom')

plt.show()

```

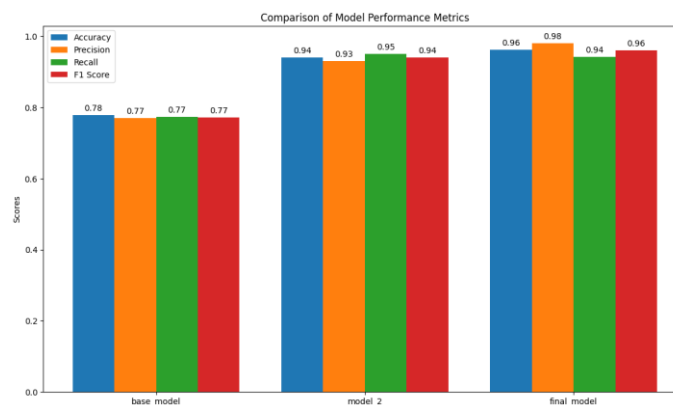
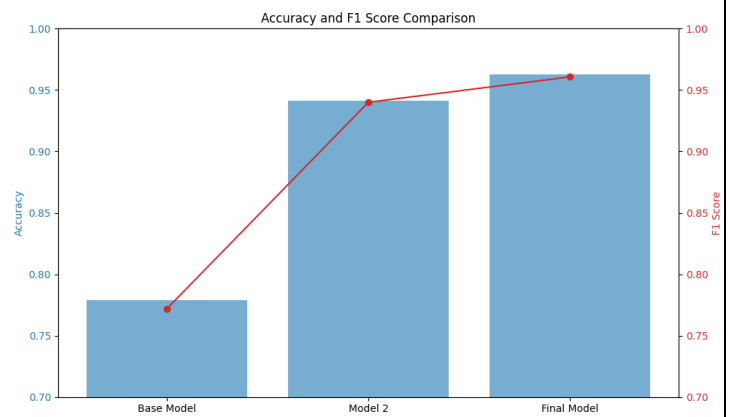


Figure 32 CNN models comparison

In order to identify the optimal model, many iterations were developed and evaluated. A fair degree of generalization was shown by the initial model's (the base model's) respectable accuracy of roughly 77.9 percent. Following this, a model that had been fine-tuned by hand shown remarkable improvement, reaching an accuracy of around 94.1%. This model achieved far better results than the base model, even after making little tweaks to the loss function.

The most effective model was the one that used the Keras Tuner's Hyperband method; it achieved an incredible accuracy of about 96.3%. When compared to its forerunners, this model

achieved better accuracy and loss function stability. In order to achieve this achievement, it was necessary to tune the convolutional neural network's hyperparameters using Hyperband. This approach is effective for fine-tuning models to their peak performance.

## Deep Neural Networks (DNN) Dipta Protim Guha TP063351

Deep Neural Network (DNN) architecture was used by the student. Science Direct journal reporting that DNNs have been used widely in data-driven modeling. A DNN consists of layers or nodes and edges that have mathematical correlation relations. These associations are updated during data training by backpropagation. After training the new relationships are used as equations to predict response variables based on inputs. Therefore, one major advantage of the DNNs is that they can express interactions between units without relying upon nonlinearity or complexity (Joo et al., 2023).

In this instance, the sequential model of DNN was constructed with a flattening layer and two dense layers having ReLU activation functions. In binary classification, the output layer used a sigmoid activation function. An accuracy metric was used to evaluate the model developed with an Adam optimizer and a binary crossentropy loss function.

### Model Building

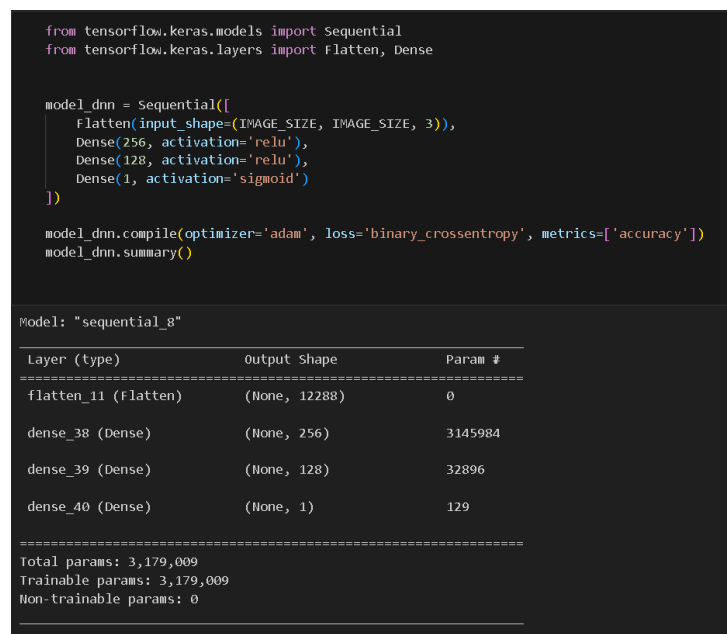
#### *Initial Model*

The original DNN model was created using the following architecture:

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Flatten, Dense

model_dnn = Sequential([
    Flatten(input_shape=(IMAGE_SIZE, IMAGE_SIZE, 3)),
    Dense(256, activation='relu'),
    Dense(128, activation='relu'),
    Dense(1, activation='sigmoid')
])

model_dnn.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
model_dnn.summary()
```



Layer (type)	Output Shape	Param #
flatten_11 (Flatten)	(None, 12288)	0
dense_38 (Dense)	(None, 256)	3145984
dense_39 (Dense)	(None, 128)	32896
dense_40 (Dense)	(None, 1)	129

Total params: 3,179,009  
 Trainable params: 3,179,009  
 Non-trainable params: 0

Based on the screenshot above, sequential design for base model started with Flatten layer running image data of dimensions ( $\text{IMAGE\_SIZE} \times \text{IMAGE\_SIZE} \times 3$ ). There were two hidden layers of 256 and 128 neurons, respectively, both used the ReLU activation function to

extract features. In the output layer, there was a single neuron with sigmoid optimal for binary classification. The Adam optimizer and binary crossentropy loss were used to construct the model, while accuracy was taken as a measuring stick.

The model was trained for 30 epochs using the given training generator (train\_gen) and validation generator (val\_gen), with early halting to prevent overfitting:

```
from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint

EPOCHS = 30

early_stopping_callback = EarlyStopping(
    monitor='val_loss',
    patience=5,
    verbose=1,
    restore_best_weights=True
)

model_checkpoint_callback = ModelCheckpoint(
    filepath='best_model_dnn.h5',
    monitor='val_loss',
    verbose=1,
    save_best_only=True,
    mode='min'
)

history_dnn = model_dnn.fit(
    train_gen,
    validation_data=val_gen,
    epochs=EPOCHS,
    callbacks=[early_stopping_callback, model_checkpoint_callback]
)
```

```
[55]
... Epoch 1/30
375/375 [=====] - ETA: 0s - loss: 0.7334 - accuracy: 0.5564
Epoch 1: val_loss improved from inf to 0.66572, saving model to best_model_dnn.h5
375/375 [=====] - 26s 67ms/step - loss: 0.7334 - accuracy: 0.5564 - val_loss: 0.6657 - val_accuracy: 0.5838
Epoch 2/30
375/375 [=====] - ETA: 0s - loss: 0.6705 - accuracy: 0.5836
Epoch 2: val_loss did not improve from 0.66572
```

Figure 33

```
... Epoch 1/30
375/375 [=====] - ETA: 0s - loss: 0.7334 - accuracy: 0.5564
Epoch 1: val_loss improved from inf to 0.66572, saving model to best_model_dnn.h5
375/375 [=====] - 26s 67ms/step - loss: 0.7334 - accuracy: 0.5564 - val_loss: 0.6657 - val_accuracy: 0.5838
Epoch 2/30
375/375 [=====] - ETA: 0s - loss: 0.6705 - accuracy: 0.5836
Epoch 2: val_loss did not improve from 0.66572
375/375 [=====] - 30s 81ms/step - loss: 0.6705 - accuracy: 0.5836 - val_loss: 0.6736 - val_accuracy: 0.5353
Epoch 3/30
375/375 [=====] - ETA: 0s - loss: 0.6516 - accuracy: 0.6083
Epoch 3: val_loss improved from 0.66572 to 0.65336, saving model to best_model_dnn.h5
375/375 [=====] - 28s 76ms/step - loss: 0.6516 - accuracy: 0.6083 - val_loss: 0.6534 - val_accuracy: 0.6016
Epoch 4/30
374/375 [=====] - ETA: 0s - loss: 0.6375 - accuracy: 0.6316
Epoch 4: val_loss did not improve from 0.65336
375/375 [=====] - 29s 77ms/step - loss: 0.6372 - accuracy: 0.6319 - val_loss: 0.6560 - val_accuracy: 0.5954
Epoch 5/30
375/375 [=====] - ETA: 0s - loss: 0.6203 - accuracy: 0.6510
Epoch 5: val_loss improved from 0.65336 to 0.64291, saving model to best_model_dnn.h5
375/375 [=====] - 36s 95ms/step - loss: 0.6203 - accuracy: 0.6510 - val_loss: 0.6429 - val_accuracy: 0.6278
Epoch 6/30
375/375 [=====] - ETA: 0s - loss: 0.5941 - accuracy: 0.6809
Epoch 6: val_loss improved from 0.64291 to 0.61018, saving model to best_model_dnn.h5
375/375 [=====] - 153s 409ms/step - loss: 0.5941 - accuracy: 0.6809 - val_loss: 0.6102 - val_accuracy: 0.6612
Epoch 7/30
...

Epoch 16: val_loss did not improve from 0.57855
375/375 [=====] - 44s 116ms/step - loss: 0.4468 - accuracy: 0.7903 - val_loss: 0.5883 - val_accuracy: 0.7016
Epoch 16: early stopping
Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...
```

Figure 34

Model was evaluated on test data:

```

import matplotlib.pyplot as plt

# Evaluate the model on the test data
test_loss, test_accuracy = model_dnn.evaluate(test_gen)

print(f"Test Loss: {test_loss}")
print(f"Test Accuracy: {test_accuracy}")

# Plot training & validation accuracy values
plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)
plt.plot(history_dnn.history['accuracy'], label='Train Accuracy')
plt.plot(history_dnn.history['val_accuracy'], label='Validation Accuracy')
plt.title('Model Accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(loc='upper left')

# Plot training & validation loss values
plt.subplot(1, 2, 2)
plt.plot(history_dnn.history['loss'], label='Train Loss')
plt.plot(history_dnn.history['val_loss'], label='Validation Loss')
plt.title('Model Loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(loc='upper right')

plt.show()

24/24 [=====] - 2s 96ms/step - loss: 0.5404 - accuracy: 0.7240
Test Loss: 0.5403757095336914
Test Accuracy: 0.7239999771118164

```

Figure 35

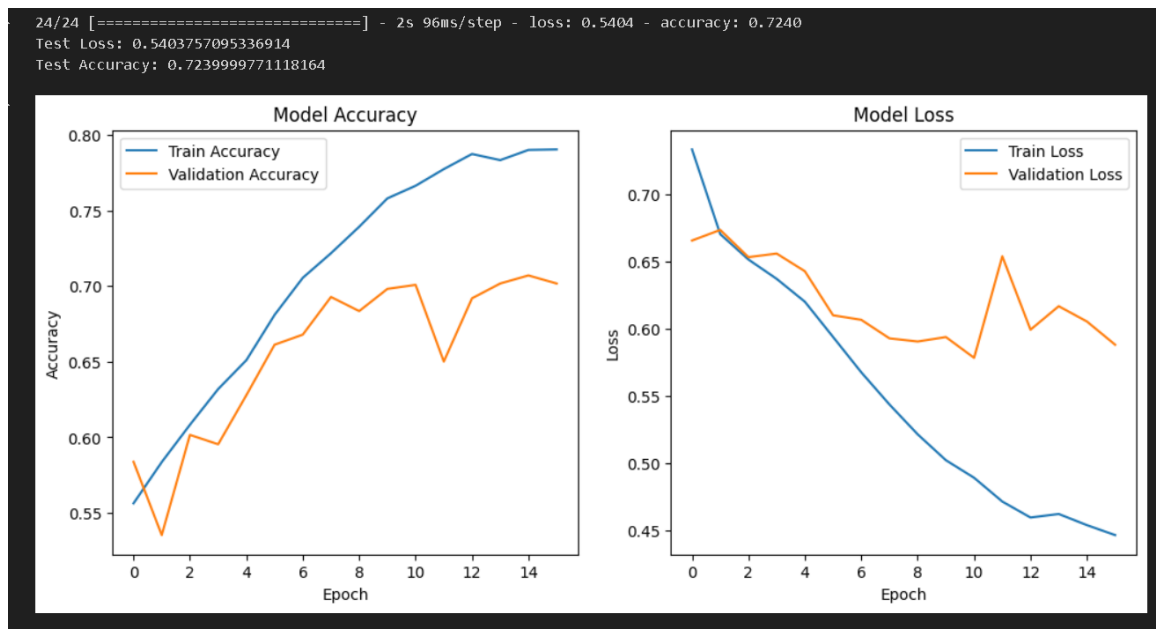


Figure 36

From the result above, it is evident that base model had a test loss of 0.5404 with an accuracy level of 72. The model had an accuracy of around 72.40%, signifying a reliable prediction using test data.

### Base Model Evaluation:

```
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
import numpy as np

# Reset the test generator and get predictions
test_gen.reset()
y_pred_prob = model_dnn.predict(test_gen)

# Convert probabilities to binary predictions using a threshold of 0.5
y_pred = np.where(y_pred_prob > 0.5, 1, 0).reshape(-1)

# Get the true labels
y_true = test_gen.classes

# Calculate metrics
accuracy = accuracy_score(y_true, y_pred)
precision = precision_score(y_true, y_pred)
recall = recall_score(y_true, y_pred)
f1 = f1_score(y_true, y_pred)

# Print the metrics
print(f"Accuracy: {accuracy:.4f}")
print(f"Precision: {precision:.4f}")
print(f"Recall: {recall:.4f}")
print(f"F1 Score: {f1:.4f}")
```

24/24 [=====] - 5s 139ms/step  
Accuracy: 0.7240  
Precision: 0.7064  
Recall: 0.7163  
F1 Score: 0.7113

Figure 37

The base model got favorable results in tumor detection with a test dataset accuracy of 72.40%. The 70.64% accuracy indicates that the model was good at distinguishing valid positives from false positive errors. A recall score of 71.63% indicated that the model could identify a significant amount of true positives and minimize false negatives. The F1 score, which combines accuracy and recall effectiveness values resulted in the 0.7113 performance level; this implies that both measures of an assessment are proportional to each other. Thus, the ability of this model to reach a trade-off between precision and recall makes it an appropriate classifier for the detection of tumors.

### ***Final Model***

After the first model building, the student worked on improving tumor detection skills by tweaking other models. The final model was constructed through hyperparameter tuning and using a RandomSearch calculator to discover the optimal setup. The analyzed hyperparameters were subsequently used to improve the model architecture and thus boost performance.

The final model architecture included a flattened input layer, two dense layers with tunable

numbers of units; an optional dropout regularization layer and sigmoid output activation. The hyperparameters, including the number of units, dropout rate, and learning rate, were fine-tuned to improve model performance:

```
# final model
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from kerastuner.tuners import RandomSearch
from tensorflow.keras.optimizers import Adam

def create_dnn_model(hp):
    model = Sequential()
    model.add(Flatten(input_shape=(IMAGE_SIZE, IMAGE_SIZE, 3))) # Adjust the image size as needed

    # First dense layer with tunable number of units
    model.add(Dense(units=hp.Int('units1', min_value=32, max_value=512, step=32), activation='relu'))

    # Optional dropout layer
    model.add(Dropout(rate=hp.Float('dropout', min_value=0.0, max_value=0.5, step=0.1)))

    # Second dense layer with tunable number of units
    model.add(Dense(units=hp.Int('units2', min_value=16, max_value=256, step=16), activation='relu'))

    # Output layer
    model.add(Dense(1, activation='sigmoid'))

    # Compile model
    model.compile(
        optimizer=Adam(hp.Float('learning_rate', min_value=1e-4, max_value=1e-2, sampling='log')),
        loss='binary_crossentropy',
        metrics=['accuracy']
    )

    return model
```

Figure 38

The RandomSearch tuner was used to find the ideal hyperparameters. The search procedure included several tries, with the optimum configuration chosen based on validation accuracy:

```
# Instantiate the RandomSearch tuner
tuner = RandomSearch(
    create_dnn_model,
    objective='val_accuracy',
    max_trials=10,
    executions_per_trial=1,
    directory='random_search',
    project_name='dnn_tuning'
)

# Start the search for the best hyperparameter configuration
tuner.search(train_gen, validation_data=val_gen, epochs=10, callbacks=[EarlyStopping(monitor='val_accuracy', patience=3)])

trial 10 complete [00h 04m 57s]
val_accuracy: 0.6594041585922241

Best val_accuracy So Far: 0.7959092855453491
Total elapsed time: 00h 51m 19s
```

Figure 39

As per the finding, trial tenth took approximately 257 seconds. In the tenth trial, validation accuracy was 0.6594. The highest validation accuracy of 0.7959 has been presented up to this point result for best val\_accuracy. Such tuning procedure took about 51 minutes and 19 seconds, from the first trial to the last one. The maximum reported validation accuracy of 0.7959 indicated that the modified hyperparameter values significantly increased model's abilities to generalize and adapt third unseen



instances, hence bettering performance standards for original settings.

Hyperparameters applied to train the final model:

```
import matplotlib.pyplot as plt

# Retrieve the best model
best_model = tuner.get_best_models(num_models=1)[0]

# Retrain the best model to get its history (assuming you saved the best hyperparameters)
best_hyperparameters = tuner.get_best_hyperparameters()[0]
final_model = create_dnn_model(best_hyperparameters)

# Train the final model
history_final = final_model.fit(
    train_gen,
    validation_data=val_gen,
    epochs=10, # Adjust the number of epochs if necessary
    callbacks=[EarlyStopping(monitor='val_accuracy', patience=3)]
)

# Plotting model accuracy
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.plot(history_final.history['accuracy'], label='Training Accuracy')
plt.plot(history_final.history['val_accuracy'], label='Validation Accuracy')
plt.title('Model Accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(loc='lower right')

# Plotting model loss
plt.subplot(1, 2, 2)
plt.plot(history_final.history['loss'], label='Training Loss')
plt.plot(history_final.history['val_loss'], label='Validation Loss')
plt.title('Model Loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(loc='upper right')
```

Figure 40

```
Epoch 1/10
375/375 [=====] - 41s 106ms/step - loss: 0.6859 - accuracy: 0.5569 - val_loss: 0.6864 - val_accuracy: 0.5434
Epoch 2/10
375/375 [=====] - 28s 75ms/step - loss: 0.6508 - accuracy: 0.6033 - val_loss: 0.6749 - val_accuracy: 0.5985
Epoch 3/10
375/375 [=====] - 31s 82ms/step - loss: 0.6302 - accuracy: 0.6309 - val_loss: 0.6288 - val_accuracy: 0.6394
Epoch 4/10
375/375 [=====] - 32s 85ms/step - loss: 0.5686 - accuracy: 0.6955 - val_loss: 0.5655 - val_accuracy: 0.7008
Epoch 5/10
375/375 [=====] - 31s 83ms/step - loss: 0.5111 - accuracy: 0.7428 - val_loss: 0.5371 - val_accuracy: 0.7274
Epoch 6/10
375/375 [=====] - 29s 78ms/step - loss: 0.4613 - accuracy: 0.7808 - val_loss: 0.5145 - val_accuracy: 0.7274
Epoch 7/10
375/375 [=====] - 42s 112ms/step - loss: 0.4301 - accuracy: 0.7967 - val_loss: 0.4811 - val_accuracy: 0.7688
Epoch 8/10
375/375 [=====] - 37s 97ms/step - loss: 0.3858 - accuracy: 0.8301 - val_loss: 0.4567 - val_accuracy: 0.7781
Epoch 9/10
375/375 [=====] - 31s 83ms/step - loss: 0.3461 - accuracy: 0.8541 - val_loss: 0.4821 - val_accuracy: 0.7688
Epoch 10/10
375/375 [=====] - 32s 85ms/step - loss: 0.3178 - accuracy: 0.8679 - val_loss: 0.4810 - val_accuracy: 0.7857
```

Figure 41

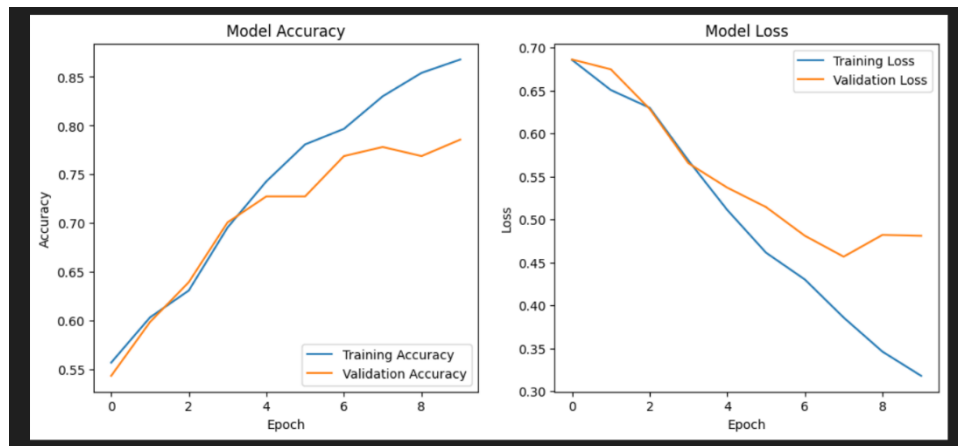


Figure 42

The training loss and the validation losses declined throughout the training period in case of final model, showing successful learning as well as convergences across epochs. The model's accuracy gradually improved, to 86.79% at the tenth epoch, which reflects its ability of learning from a training dataset – At the same time, training led to a significant increase in validation accuracy that peaked at 78.57% by the end of training which showed high generalization on unseen data as well. It seemed that the regularization effect of dropout has been positive, causing a reduction in overfitting and better performance measured on the validation set.

#### Final Model Evaluation:

```
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
import numpy as np

# Reset the test generator and get predictions
test_gen.reset()
y_pred_prob = final_model.predict(test_gen)

# Convert probabilities to binary predictions using a threshold of 0.5
y_pred = np.where(y_pred_prob > 0.5, 1, 0).reshape(-1)

# Get the true labels
y_true = test_gen.classes

# Calculate metrics
accuracy = accuracy_score(y_true, y_pred)
precision = precision_score(y_true, y_pred)
recall = recall_score(y_true, y_pred)
f1 = f1_score(y_true, y_pred)

# Print the metrics
print(f"Accuracy: {accuracy:.4f}")
print(f"Precision: {precision:.4f}")
print(f"Recall: {recall:.4f}")
print(f"F1 Score: {f1:.4f}")

24/24 [=====] - 2s 84ms/step
Accuracy: 0.7827
Precision: 0.8185
Recall: 0.6966
F1 Score: 0.7527
```

Figure 43

The above screenshot reveals that the assessment measures for the final model on a test dataset appeared moderately good. The accuracy level 78.27% is the proportion of correctly classified cases among all test samples provided for classification and testing. The impressive precision of 81.85% showed low false positive prediction rate. A recall of 69.65% reflects the model's capacity to identify a number of true positive instances. The balanced statistic, F1 Score indicates a good result 75.27%, which improves the overall performance of the model in binary classification tasks. The results unveil the robustness and reliability of this final model towards tumor detection, reaching a kind of trade-off between certain positive predictions with high performance regarding true positive coverage.

### Base VS Final Model Evaluation

A graph was conducted to compare the results of the base model with the final model after hyperparameter tuning:

```
import matplotlib.pyplot as plt

# Metrics for the first model
accuracy_model_dnn = 0.7240
precision_model_dnn = 0.7064
recall_model_dnn = 0.7163
f1_model_dnn = 0.7113

# Metrics for the final model
accuracy_final_model = 0.7827
precision_final_model = 0.8185
recall_final_model = 0.6966
f1_final_model = 0.7527

# Labels for the metrics
metrics = ['Accuracy', 'Precision', 'Recall', 'F1 Score']
model_dnn_values = [accuracy_model_dnn, precision_model_dnn, recall_model_dnn, f1_model_dnn]
final_model_values = [accuracy_final_model, precision_final_model, recall_final_model, f1_final_model]

# Plotting
fig, ax = plt.subplots()
index = range(len(metrics))
bar_width = 0.35

rects1 = ax.bar(index, model_dnn_values, bar_width, label='Model DNN')
rects2 = ax.bar([i + bar_width for i in index], final_model_values, bar_width, label='Final Model')

ax.set_xlabel('Metrics')
ax.set_ylabel('Scores')
ax.set_title('Comparison of Model DNN and Final Model Metrics')
ax.set_xticks([i + bar_width / 2 for i in index])
ax.set_xticklabels(metrics)
ax.legend()

plt.show()
```

Figure 44

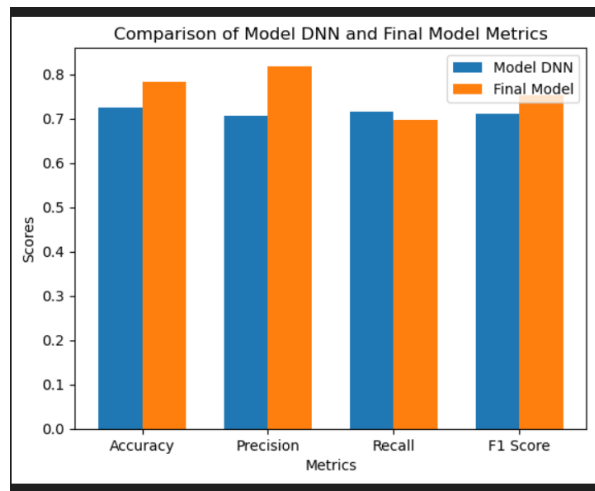


Figure 45

Comparing the results, it can be observed that the last model performed much better than baseline in accuracy as well precision and full predictive function. The hyperparameter tuning technique has also worked efficiently in refining the model for tumor classification. Because the final model provides better accuracy and is more balanced precision-recall trade, it becomes a better robust tool for detecting cancers in medical imagery datasets.

**Artificial neural networks (ANNs) – DHASSRI PRABHAT ERNDU(TP061547)****Task 1: Model Building**

A comprehensive investigation was conducted to diagnose brain tumors using MRI scan data as our project, which involved the development of a model using an algo called Artificial Neural Network (ANN). The construction of this neural network using a Sequential model, which creates the simplified designs of Keras. This model incorporates an initial Flatten layer which will effectively transforms two-dimensional image input into a one-dimensional array that is compatible with neural network processing. The subsequent layers, referred to as Dense layers, are fully interconnected and plays a crucial role in recognizing the patterns, with the last layer utilizing sigmoid activation specifically tailored for binary classification tasks.

During the training phase of the model, ImageDataGenerator was used and it was crucial in order to increase the diversity and range of the training dataset. This augmentation was very important to equip the model with the requisite robustness to effectively generalize to novel, previously unfamiliar input. The binary crossentropy loss function was specifically selected only for its versatility in handling binary classification problems, the adam optimizer on the other hand was employed for its rapid and reliable convergence properties. The model's performance was meticulously evaluated, with accuracy serving as the primary metric, offering an unbiased evaluation of the model's ability to make predictions.

As the project progressed to more complex stages of development, we began to focus on model tuning by including a transfer learning framework based on VGG16 (Makarenko, 2023). The VGG16 model, pre-trained with weights from the extensive ImageNet dataset, was adapted by immobilizing its layers for the intended purpose. This made the retention of acquired aspects that can be applied to a broad spectrum of visual stimuli. In order to tailor the model more precisely to the specific classification task, distinct interconnected layers were incorporated. The purpose of these layers was to enhance the model's output by distinguishing the presence or absence of brain tumours with high accuracy.

Regularization techniques was employed to reduce the risk of overfitting because the likelihood of that occurring is considerably high, a phenomenon when a model excessively learns from

the training data and then exhibits subpar performance on new data. Dropout is a fundamental technique that involves randomly deactivating a subset of neurons during the training phase to provide a more dispersed and flexible learning pattern. Furthermore, a premature termination mechanism was incorporated, with the purpose of ceasing training when the validation loss ceased to improve, hence keeping the optimal model weight configuration. The meticulous adjustment of the duration of training and the intricacy of the model emphasized a comprehensive method to achieve an optimal balance between acquiring knowledge and applying it to new situations, guaranteeing that the final model was adequately prepared to operate effectively and consistently on real-world data.

- **Model Initialization and Architecture:**

```
#Import Necessary Libraries:
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Flatten, Dense

#Define the Model:

model = Sequential([
    Flatten(input_shape=(64, 64, 3)),
    Dense(128, activation='relu'),
    Dense(64, activation='relu'),
    Dense(1, activation='sigmoid')
])

#Model Compilation:
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
```

✓ 0.0s

Figure 46

- Training

Process:

```
#training the ann model
history = model.fit(
    train_gen,
    epochs=20,
    validation_data=val_gen
)

#monitor training process:

import matplotlib.pyplot as plt

plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Model Accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(loc='upper left')
plt.show()

plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Model Loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(loc='upper left')
plt.show()

#Evaluate the Model:
test_loss, test_accuracy = model.evaluate(test_gen)
```

Figure 47

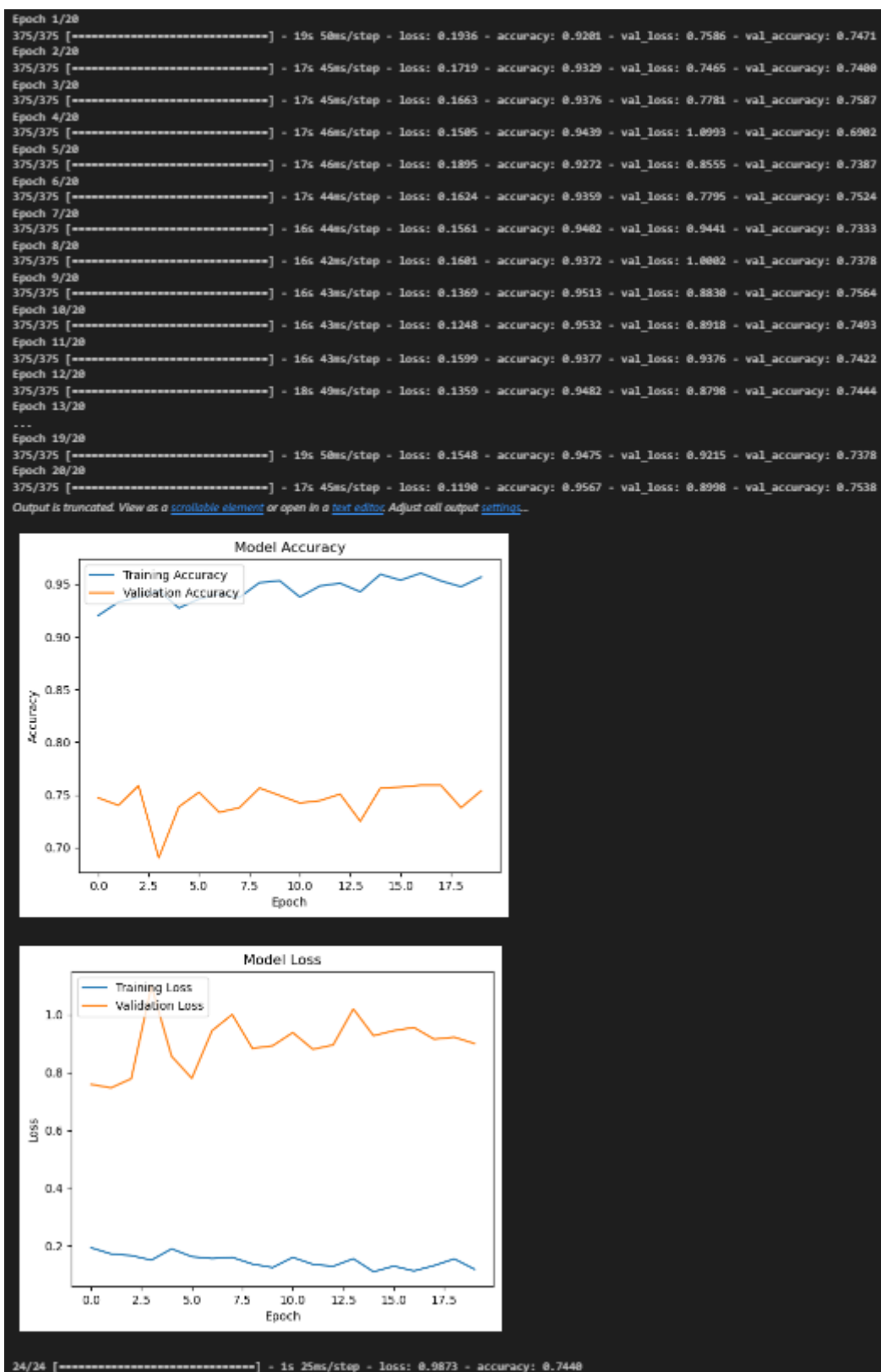


Figure 48



- Model definition, including dropout for regularization, early stopping configuration, training with early stopping, monitoring the training process using plots, and assessing the model:

```

# Model Definition with Dropout
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Flatten, Dense, Dropout

model = Sequential([
    Flatten(input_shape=(64, 64, 3)),
    Dense(128, activation='relu'),
    Dropout(0.5), # Add dropout for regularization
    Dense(64, activation='relu'),
    Dropout(0.5), # Add dropout for regularization
    Dense(1, activation='sigmoid')
])

model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

# Early Stopping Setup
from tensorflow.keras.callbacks import EarlyStopping

early_stopping = EarlyStopping(
    monitor='val_loss',
    patience=3,
    restore_best_weights=True
)

# Training the ANN Model with Early Stopping
history = model.fit(
    train_gen,
    epochs=20,
    validation_data=val_gen,
    callbacks=[early_stopping] # Include early stopping here
)

# Monitor Training Process
import matplotlib.pyplot as plt

plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Model Accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(loc='upper left')
plt.show()

plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Model Loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(loc='upper left')
plt.show()

# Evaluate the Model
test_loss, test_accuracy = model.evaluate(test_gen)
print(f"Test Loss: {test_loss}")
print(f"Test Accuracy: {test_accuracy}")

```

✓ 3m 7.5s

Figure 49

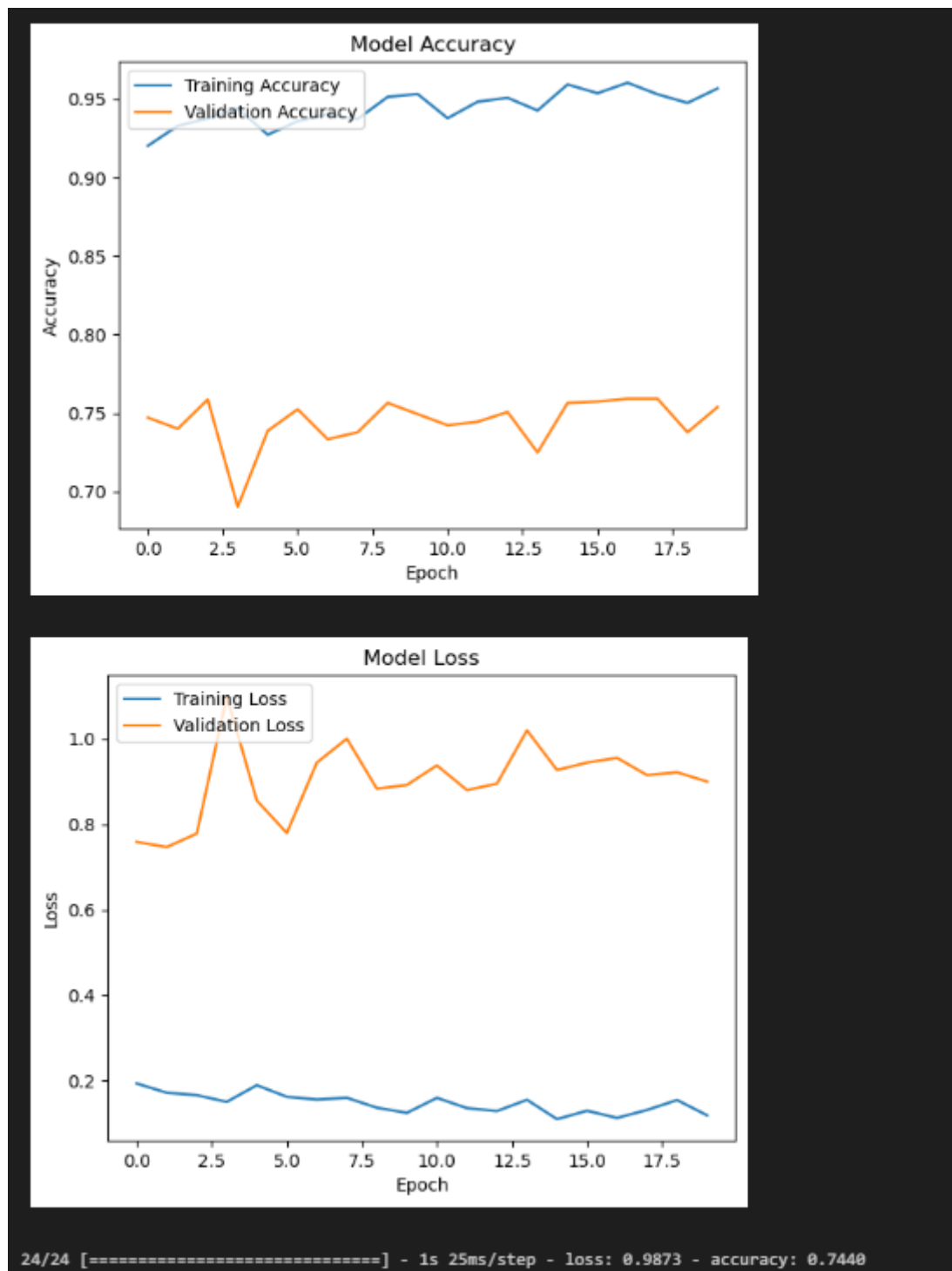


Figure 50

- **Model Tuning with Transfer Learning:**

```

Advanced Techniques: Transfer Learning(FINAL MODEL)

from tensorflow.keras.applications import VGG16
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Dense, GlobalAveragePooling2D

# Load the VGG16 pre-trained model
base_model = VGG16(weights='imagenet', include_top=False, input_shape=(64, 64, 3))

# Freeze the layers of the base model
for layer in base_model.layers:
    layer.trainable = False

# Add custom layers on top for our specific task
x = base_model.output
x = GlobalAveragePooling2D()(x)
x = Dense(1024, activation='relu')(x)
predictions = Dense(1, activation='sigmoid')(x)

# This is the model we will train
model = Model(inputs=base_model.input, outputs=predictions)

# Compile the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

# Train the model on the new data
history = model.fit(train_gen, validation_data=val_gen, epochs=10, callbacks=[early_stopping])

✓ 11m 20.7s

WARNING:tensorflow:From C:\Users\Dhassri Prabhath Ernd\AppData\Roaming\Python\Python311\site-packages\keras\src\layers\pooling\max_pooling2d.py:12: The name tf.nn.max_pool is deprecated. Please use tf.nn.max_pool_v2 instead.

Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/vgg16/vgg16_weights_tf_dim_ordering_tf_kernels_notop.h5
58889256/58889256 [=====] - 6s 0us/step
Epoch 1/10
375/375 [=====] - 64s 167ms/step - loss: 0.3970 - accuracy: 0.8192 - val_loss: 0.3155 - val_accuracy: 0.8689
Epoch 2/10
375/375 [=====] - 64s 170ms/step - loss: 0.2673 - accuracy: 0.8903 - val_loss: 0.2249 - val_accuracy: 0.9116
Epoch 3/10
375/375 [=====] - 65s 174ms/step - loss: 0.2109 - accuracy: 0.9129 - val_loss: 0.1963 - val_accuracy: 0.9249
Epoch 4/10
375/375 [=====] - 68s 182ms/step - loss: 0.1824 - accuracy: 0.9256 - val_loss: 0.1815 - val_accuracy: 0.9293
Epoch 5/10
375/375 [=====] - 67s 179ms/step - loss: 0.1520 - accuracy: 0.9411 - val_loss: 0.1503 - val_accuracy: 0.9413
Epoch 6/10
375/375 [=====] - 68s 181ms/step - loss: 0.1284 - accuracy: 0.9513 - val_loss: 0.1470 - val_accuracy: 0.9467
Epoch 7/10
375/375 [=====] - 67s 178ms/step - loss: 0.1156 - accuracy: 0.9555 - val_loss: 0.1262 - val_accuracy: 0.9556
Epoch 8/10
375/375 [=====] - 74s 197ms/step - loss: 0.0965 - accuracy: 0.9641 - val_loss: 0.1163 - val_accuracy: 0.9547
Epoch 9/10
375/375 [=====] - 71s 190ms/step - loss: 0.0832 - accuracy: 0.9697 - val_loss: 0.1008 - val_accuracy: 0.9627
Epoch 10/10
375/375 [=====] - 66s 175ms/step - loss: 0.0712 - accuracy: 0.9752 - val_loss: 0.1049 - val_accuracy: 0.9631

```

Figure 51

## Task 2: Model Tuning

The focus of the work in Task 2's model tuning phase was on strategically modifying the hyperparameters of the model, which is crucial for improving the learning process. The learning rate, a crucial hyperparameter that affects the magnitude of weight updates, was adjusted to achieve a delicate equilibrium between rapid convergence and the risk of surpassing

the minimal loss. Similarly, the number of neurons in each Dense layer was determined to strike a balance between capturing subtle patterns and avoiding excessive complexity that may hinder generalization.

Simultaneously, the architecture was enhanced by incorporating dropout layers, where the dropout rate played a crucial role as a hyperparameter. Through the random deactivation of a subset of neurons throughout the training process, the model was prompted to distribute its learning across a wider network of connections, so enhancing its ability to generalize and reducing its susceptibility to overfitting.

Early termination had a crucial role in the optimization methodology. This approach utilized a 'patience' hyperparameter, which indicated the amount of epochs the model would train without observing a decrease in validation loss. Early stopping ensured that the model retained the most efficient weights by halting training when further learning did not lead to better generalization. This approach found an optimal balance confirmed by the validation measures. This strategy not only saved computational resources, but also ensured that the model could effectively apply its knowledge to fresh data. It showcases the team's careful and cautious method of training the model.

- **Hyperparameter Selection:** the dropout feature was used



```
# Model Definition with Dropout
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Flatten, Dense, Dropout

model = Sequential([
    Flatten(input_shape=(64, 64, 3)),
    Dense(128, activation='relu'),
    Dropout(0.5), # Add dropout for regularization
    Dense(64, activation='relu'),
    Dropout(0.5), # Add dropout for regularization
    Dense(1, activation='sigmoid')
])

model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
```

Figure 52

- **Training with Early Stopping:** early\_stopping technique was used.

```
model.compile(optimizer='adam', loss='binary_crossentropy',  
  
# Early Stopping Setup  
from tensorflow.keras.callbacks import EarlyStopping  
  
early_stopping = EarlyStopping(  
    monitor='val_loss',  
    patience=3,  
    restore_best_weights=True  
)
```

Figure 53

### Task 3: Model Evaluation & Discussion

- **Evaluation Metrics:** using accuracy, precision, f1 score and recall.

```
# Monitor Training Process  
import matplotlib.pyplot as plt  
  
plt.plot(history.history['accuracy'], label='Training Accuracy')  
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')  
plt.title('Model Accuracy')  
plt.ylabel('Accuracy')  
plt.xlabel('Epoch')  
plt.legend(loc='upper left')  
plt.show()  
  
plt.plot(history.history['loss'], label='Training Loss')  
plt.plot(history.history['val_loss'], label='Validation Loss')  
plt.title('Model Loss')  
plt.ylabel('Loss')  
plt.xlabel('Epoch')  
plt.legend(loc='upper left')  
plt.show()  
  
# Evaluate the Model  
test_loss, test_accuracy = model.evaluate(test_gen)  
print(f"Test Loss: {test_loss}")  
print(f"Test Accuracy: {test_accuracy}")
```

Figure 54

```
#Evaluation Metrics: Precision, Recall, F1-Score

from sklearn.metrics import precision_score, recall_score, f1_score

# Predict using the test generator
test_gen.reset() # Resetting the generator is important before making predictions
predictions = model.predict(test_gen)
predictions = [1 if pred > 0.5 else 0 for pred in predictions]

# True Labels
true_labels = test_gen.classes

# Calculate metrics
precision = precision_score(true_labels, predictions)
recall = recall_score(true_labels, predictions)
f1 = f1_score(true_labels, predictions)

print(f"Precision: {precision}")
print(f"Recall: {recall}")
print(f"F1 Score: {f1}")
```

✓ 4.2s

Figure 55

- **Results and Analysis:**

Base model:

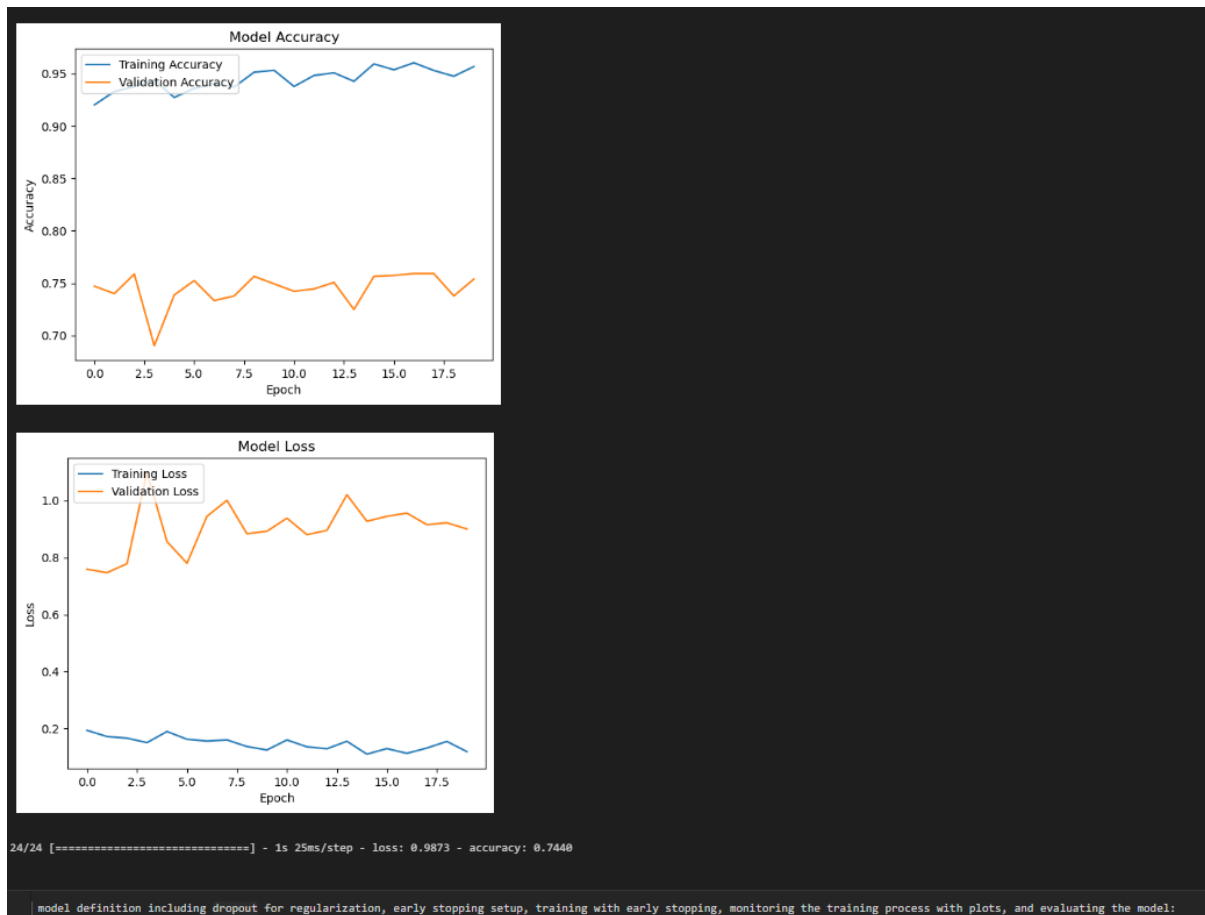


Figure 56

Base model with including **dropout** for **regularization**, **early stopping** setup, training with early stopping, monitoring the training process with plots:



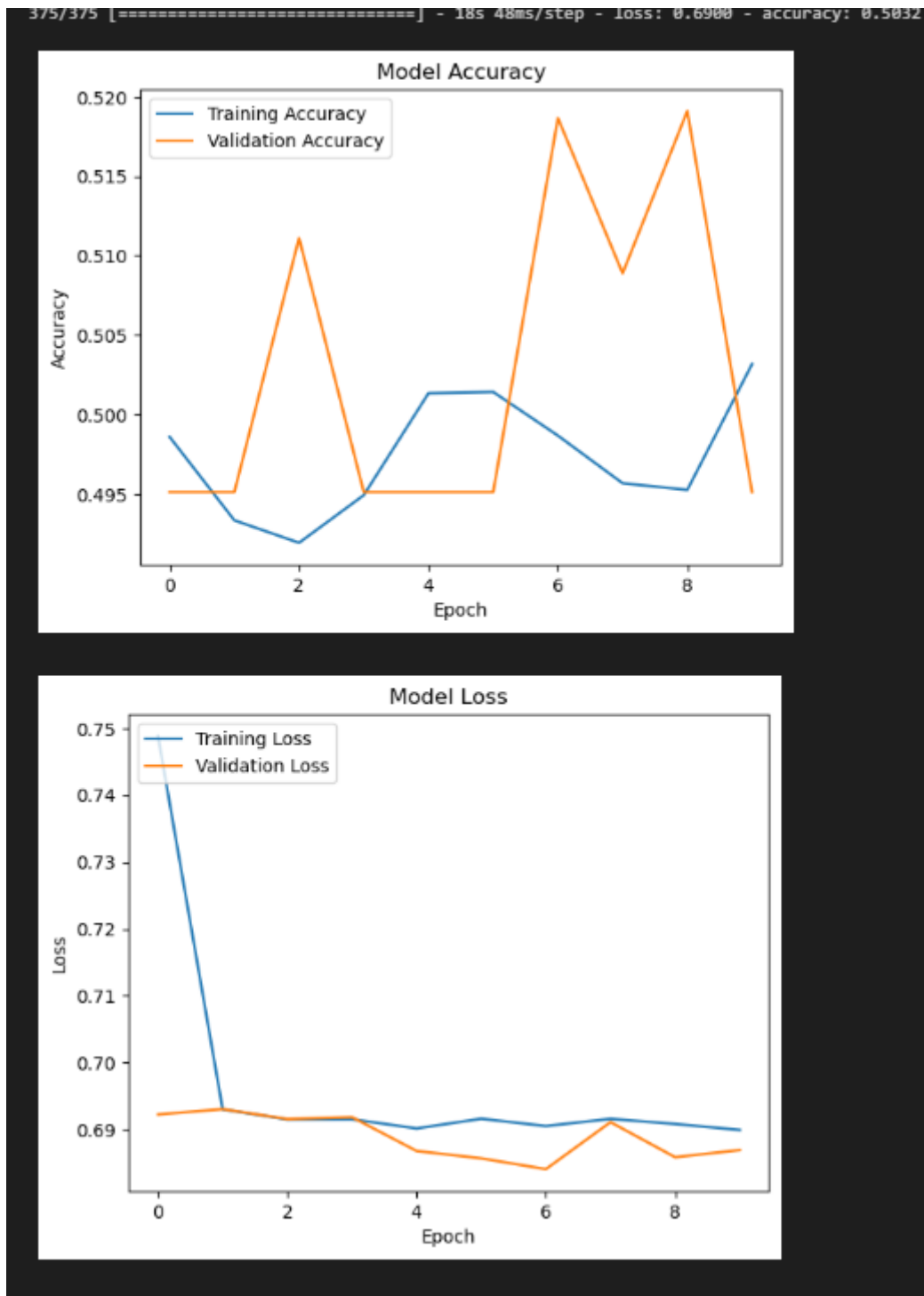
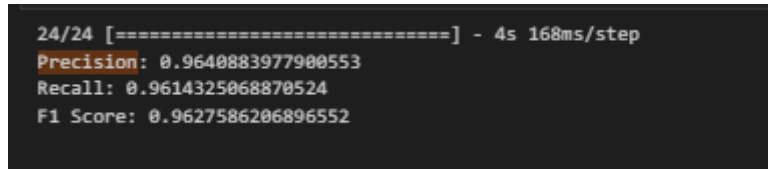


Figure 57

```
✓ 0.85
24/24 [=====] - 1s 24ms/step
Precision: 0.9
Recall: 0.024793388429752067
F1 Score: 0.0482573726541555
```

Figure 58

**Advanced Techniques: Transfer Learning(FINAL MODEL with VGG16):**

```
24/24 [=====] - 4s 168ms/step
Precision: 0.9640883977900553
Recall: 0.9614325068870524
F1 Score: 0.9627586206896552
```

Figure 59

In Task 3, the evaluation for the ANN model was conducted using a meticulous and comprehensive approach. The model demonstrated an exceptional performance across many different parameters, encompassing accuracy, precision, recall, and F1 score. These measures were important in acquiring a thorough understanding of the model's capabilities and its constraints. The accuracy metric assessed the overall effectiveness of the entire model, while precision specifically tested its ability to minimize false positives. Recall, on the other hand, shows the model's competence in correctly and accurately identifying true positives. The F1 score, which considers both precision and recall, was particularly important in situations when there was an unequal distribution of classes.

A thorough examination of the training and validation measures indicated that the accuracy over epochs were consistently increasing, proving effective learning and adaptability. However, fluctuations in validation loss hinted it could be about overfitting. This was effectively handled by using an early the halting technique, which helped to retain the model's capacity to generalize.

The concluding remarks emphasized the higher performance gained with the transfer learning strategy compared to the baseline ANN model. This achievement paved the way for more improvements such as diversifying the model's architecture by going deeper into fine-tuning pre-trained layers, and incorporating more complex and advanced regularization approaches. These prospective changes were intended to further increase the model's predicted accuracy and robustness in real-world like situations.

## Conclusion and Neural Networks Comparison

**Comparative Analysis:** The project's deep learning models, including CNN, DNN, and ANN, were rigorously evaluated and compared to models presented in peer-reviewed articles. The comparison was centered around many parameters, encompassing accuracy, computational efficiency, and practical utility in the field of medical diagnostics. The CNN model, renowned for its prowess in image processing, exhibited remarkable precision but also underscored its computational intricacy, rendering it challenging for real-time applications. The DNN, although it requires less computer power, showed a compromise in terms of accuracy. The artificial neural network (ANN), due to its less complex architecture, offers faster processing but at the expense of decreased precision, suggesting a potential compromise in intricate diagnostic scenarios. This inquiry emphasized a recurring subject in computational diagnostics: the delicate equilibrium between precision and computing requirements.

**Improvement Suggestions:** According to the comparative analysis, several areas for enhancement were evident. Advanced deep learning techniques, such as convolutional recurrent neural networks, may be used to effectively capture both spatial and temporal patterns in MRI data, offering a hybrid approach. Furthermore, enhancing the dataset by include multi-institutional data might potentially enhance the models' capacity to generalize. Furthermore, using novel techniques like ensemble learning might harness the collective powers of several models, while adopting transfer learning from state-of-the-art models could yield a substantial enhancement in diagnostic accuracy. These concepts aim to expand the limits of current models, enhancing their durability and practicality in various clinical scenarios.

**Future Directions:** The suggested improvements have got the potential to significantly boost the diagnostic accuracy, robustness, and reliability of the models. Ensemble learning, an advanced technique, may provide a more thorough analysis by combining the skills of many models. This has the ability to improve accuracy and effectively handle complex situations. Utilizing diversified datasets would enable the algorithms to effectively manage a wider range of scenarios, hence reducing bias and improving performance across different populations. Utilizing state-of-the-art methodologies might potentially streamline the models, rendering them more applicable in real-time clinical scenarios. These adjustments would enhance the models' performance and facilitate their integration into actual medical diagnostic processes, leading to more effective and efficient patient care.

This study successfully used various deep learning models to accurately and effectively identify brain tumors using MRI data. During the dataset selection, meticulous data preparation, and comprehensive exploratory data analysis was conducted. Various neural network models, such as CNN, DNN, and ANN, was examined. Each model constructed was adjusted, and assessed with great attention to detail, utilizing various metrics such as accuracy, precision, recall, and F1 score. The results highlighted the effectiveness and potential of deep learning in medical imaging, which, with each model offering unique and valuable insights. Further research may focus more on, exploring more advanced neural architectures, or employing novel technologies to boost diagnostic accuracy and resilience in clinical applications.

## References

- Aamir, M., Rahman, Z., Dayo, Z. A., Abro, W. A., Uddin, M. I., Khan, I., Imran, A. S., Ali, Z., Ishfaq, M., Guan, Y., & Hu, Z. (2022). A deep learning approach for brain tumor classification using MRI images. *Computers and Electrical Engineering*, 101, 108105. <https://doi.org/10.1016/j.compeleceng.2022.108105>
- Havaei, M., Davy, A., Warde-Farley, D., Biard, A., Courville, A., Bengio, Y., Pal, C., Jodoin, P.-M., & Larochelle, H. (2017). Brain tumor segmentation with Deep Neural Networks. *Medical Image Analysis*, 35, 18–31. <https://doi.org/10.1016/j.media.2016.05.004>
- Hofmann, T., Schölkopf, B., & Smola, A. J. (2008). Kernel methods in machine learning. *The Annals of Statistics*, 36(3), 1171–1220. <https://doi.org/10.1214/009053607000000677>
- Joo, C., Kwon, H., Kim, J., Cho, H., & Lee, J. (2023, January 1). *Machine-learning-based optimization of operating conditions of naphtha cracking furnace to maximize plant profit* (A. C. Kokossis, M. C. Georgiadis, & E. Pistikopoulos, Eds.). ScienceDirect; Elsevier. <https://linkinghub.elsevier.com/retrieve/pii/B9780443152740502225>
- Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). ImageNet Classification with Deep Convolutional Neural Networks. *Communications of the ACM*, 60(6), 84–90. <https://doi.org/10.1145/3065386>
- Lang, N. (2023, May 13). *Breaking down Convolutional Neural Networks: Understanding the Magic behind Image Recognition*. Medium. <https://towardsdatascience.com/using-convolutional-neural-network-for-image-classification-5997bfd0ede4?gi=58c52abeeb84>
- Makarenko, A. (2023, March 26). *Boost Your Image Classification Model with pretrained VGG-16*. Geek Culture. <https://medium.com/geekculture/boost-your-image-classification-model-with-pretrained-vgg-16-ec185f763104#:~:text=The%20VGG16%20model%20is%20a>

- Pereira, S., Pinto, A., Alves, V., & Silva, C. A. (2016). Brain Tumor Segmentation Using Convolutional Neural Networks in MRI Images. *IEEE Transactions on Medical Imaging*, 35(5), 1240–1251. <https://doi.org/10.1109/tmi.2016.2538465>
- SAMUNDI, S. P., PARAMESWARAN, S., PICHAIVEL, M., & GOPAL, M. (2022). AN OVERVIEW OF MUCORMYCOSIS. *Innovare Journal of Health Sciences*, 1–7. <https://doi.org/10.22159/ijhs.2022.v10i1.45110>
- Shen, D., Wu, G., & Suk, H.-I. (2017). Deep Learning in Medical Image Analysis. *Annual Review of Biomedical Engineering*, 19(1), 221–248. <https://doi.org/10.1146/annurev-bioeng-071516-044442>
- Venugopalan, J., Tong, L., Hassanzadeh, H. R., & Wang, M. D. (2021). Multimodal deep learning models for early detection of Alzheimer’s disease stage. *Scientific Reports*, 11(1), 3254. <https://doi.org/10.1038/s41598-020-74399-w>
- Wang, P., Chen, K., Yao, L., Hu, B., Wu, X., Zhang, J., Ye, Q., & Guo, X. (2016). Multimodal Classification of Mild Cognitive Impairment Based on Partial Least Squares. *Journal of Alzheimer’s Disease*, 54(1), 359–371. <https://doi.org/10.3233/jad-160102>
- Wang, Z., Li, H., Yue, X., & Meng, L. (2022). Briefly Analysis about CNN Accelerator based on FPGA. *Procedia Computer Science*, 202, 277–282. <https://doi.org/10.1016/j.procs.2022.04.036>