RAFIK HARIRI UNIVERSITY

BRAIN TUMOR SEGMENTATION AND CLASSIFICATION

Done by

ABDALLAH KAADAN

AYA BAASEERI

AYA El SAOUDI

ISSA EL MOUSLEH

MOHAMAD AL JANNOUN

Submitted to

DR. ROAA SOLOH

This senior project is submitted in partial fulfillment of the requirements of BS degree to

The Computer Science Major of the College of Art and Sciences at the Rafik Hariri

University

MECHREF - LEBANON

April 2023

Abdullah Kaadan (2020-0556)

Aya Baaseeri (2020-0604)

Aya El Saoudi (2020-0131)

Issa El Mousleh (2020-0415)

Mohamad Al Jannoun (2020-0439)

ACKNOWLEDGMENTS

We would like to express our deep gratitude and appreciation to Dr. Roaa Soloh for her invaluable support and unwavering guidance throughout our project. Her dedication, unwavering support, and motivation have been instrumental in helping us steer and complete this project. Her expert advice, constant follow-up, and willingness to help have been an immense source of encouragement andassistance to us.

Additionally, we extend our thanks to Dr. Mohamad Al Abed for his consistent backing, trust in us, and endurance throughout our project. His faith in our capabilities and continuous motivation have been extremely valuable in assisting us in overcoming challenges and reaching our objectives. We especially appreciate his patience and comprehension during the moments when things didn't unfold asanticipated. Their combined contributions have made a significant impact on our project's success, and we are deeply grateful for their unwavering support.

ABSTRACT

In this project, our main focus has been on addressing the critical issue of brain tumor identification and classification using the power of advanced computer technologies and medical imaging techniques. By employing CNN and U-Net models on Kaggle datasets, one can achieve strong results in brain tumor segmentation and classification. Preprocess MRI scans, train models with appropriate metrics, and fine-tune hyperparameters. Deploy successful models for clinical automation.

We aimed to develop a comprehensive solution that combines Convolutional Neural Networks (CNNs) and U-Net to achieve accurate tumor classification and precise segmentation in MRIscans. To ensure the effectiveness of our models, we diligently preprocessed the MRI images by applying various adjustments, such as normalizing pixel values and resizing images to a common resolution. Additionally, we utilized data augmentation techniques to generate a more diverse and comprehensive training dataset, enhancing the model's ability to generalize and perform well on unseen data. Moreover, we took a step further by creating a user-friendly graphical interface powered by thePyQt5 library. This interface enables medical professionals to interact seamlessly with our technology, making the process of brain tumor analysis more intuitive and accessible.

TABLE OF CONTENTS

List of Figures

CHAPTER 1

INTRODUCTION

1.1. Problem Definition

Brain tumor segmentation and classification pose important issues when performing medical imaging because they require recognizing and characterizing malignancies within the brain. Brain tumor segmentation and categorization provide important aid to medical practitioners in properly diagnosing patients, developing appropriate treatment regimens, and closely monitoring brain tumor growth.

According to ASCO (2023), after measuring relative survival rate statistics for brain tumors and the damage caused by brain tumors over a 5-year period, they concluded the following survival rates: For people younger than age 15, the survival rate is approximately 75%. For individuals aged 15 to 39, the rate nears 72%. However, for people aged 40 and older, the survival rate drops to 21%. Experts measure relative survival rate statistics for brain tumors every 5 years. ASCO also mentioned that Brain and other nervous system cancer is the 10th leading cause of death for men and women in the world.

Challenges in brain tumor segmentation and classification come from the tumor's variety in shape, size, location, and the existence of overlapping features. Tumors can have uneven borders, infiltrate into healthy tissue, and have a variety of appearances, making correct segmentation and classification difficult.

Advances in medical imaging technology such as magnetic resonance imaging (MRI) and computed tomography (CT), as well as the increased availability of big datasets, have made deep learning-based techniques for brain tumor segmentation and classification increasingly popular. These advanced technologies identify, segment, and categorize brain tumors using computer tools, image processing, pattern recognition, and artificial intelligence approaches. Deep learning models, namely convolutional neural networks (CNNs), have demonstrated promising results by learning from large amounts of annotated medical pictures, allowing for precise and efficient tumor segmentation andclassification methods.

Bhattacharjee et al. (2022) discussed that magnetic resonance imaging (MRI) is used to diagnose brain tumors. X-rays, computed tomography (CT) scans, and positron emission tomography (PET) scans are also used to diagnose brain tumors. MRI, on the other hand, is more informative than other imaging modalities because it offers extensive information on the tumor's nature, size and shape, position, architecture, and vascular supply. As a result, MRI is the best method for studying brain tumors (Healthcare Informatics Research).

Accurately identifying and categorizing brain tumors is crucial for better patient care. It helps doctors choose the right treatments and monitor how the tumor is responding. By spotting tumors early, pinpointing their location, and understanding their features, improved imaging and analysis techniques lead to better patient results, cost savings in healthcare, and higher-quality treatment in neuro-oncology.

Manual segmentation is a common method for segmenting and classifying brain tumors, typically performed by expert radiologists or specialized physicians. This approach involves delineating tumor areas slice-by-slice or in 3D, utilizing expert knowledge for accurate identification. However, manual segmentation has drawbacks, such as being time-consuming and limiting scalability in large-scale investigations or clinical settings. Consequently, researchers are exploring automated segmentation methods, including machine learning and deep learning techniques, to improve reproducibility and efficiency in brain tumor analysis and overcome the inherent unpredictability of manual segmentation (ScienceDirect, 2022).

As a solution, we created a desktop program that employs two machine learning models: a CNN for classification and a U-Net for segmentation. These models were carefully chosen after significant study and testing to verify their appropriateness for the specific purpose of brain tumor investigation.

We employed the U-Net architecture for brain tumor segmentation and a convolutional neural network (CNN) for tumor classification in our study. The U-Net is a well-known deep learning network that excels in picture segmentation tasks, notably in the medical area. Its unique design, which comprises of an encoder and a decoder, enables exact tumor boundary delineation.

We used CNN, a deep learning model commonly used for image classification, to classify tumors. Because of the CNN's capacity to acquire hierarchical information from input pictures, it was well-suited for properly diagnosing brain tumors based on their kinds or characteristics.

Throughout the report, we will provide a detailed description of the U-Net architecture, its implementation for brain tumor segmentation, and the training process. Similarly, we will discuss the design and training of the CNN for tumor classification, highlighting the features or attributes considered for classification purposes.

1.2. Project Aim

The application provides a convenient platform for users to upload brain MRI scans, where the tumors are automatically classified into appropriate groups using a CNN and tumor regions are accurately delineated through the application of U-Net. This combined methodology revolutionizes diagnosis, reducing manual efforts, and expediting analysis, resulting in improved patient care and progress in brain tumor research. Additionally, the application's user-friendly interface and efficient computation widen its accessibility to a diverse range of users, fostering the adoption of cutting-edge machine-learning methods in the medical domain.

The overall layout of the application is shown in Figure 1's Data Flow Diagram (DFD), where the system's input comprises of brain MRI images and the main output is the outcome of segmenting and classifying brain tumors.

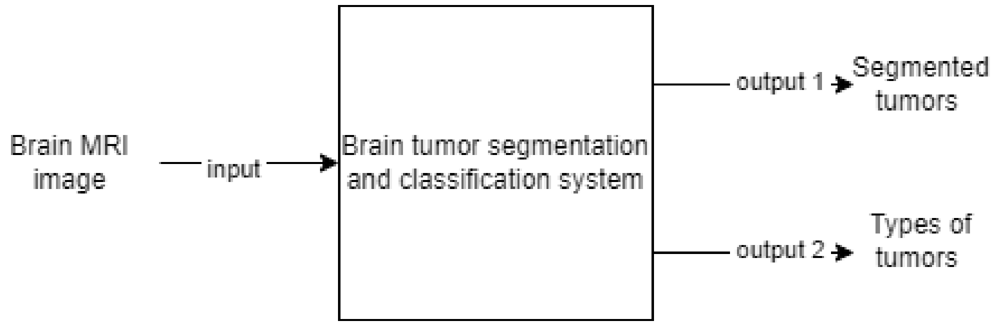


Figure 1. DFD level 0 of the application

We've developed an application that simplifies brain tumor segmentation and classification. Users input an MRI image, and our system utilizes U-Net for segmentation and CNN for classification, providing the results promptly.

1.3. Report Structure

In this report, we follow a structured approach, beginning with the Literature Review in chapter 2, where we investigate the use of U-Net for segmentation and CNN for classification in brain tumor analysis. We explore U-Net's fundamentals, advantages, and drawbacks, along with dataset considerations and model comparisons. In Chapter 3, we delve into our methodology and implementation, detailing the steps for training and testing both CNN and U-Net models. We outline preprocessing, model architecture, training, and testing processes. Conclusion and perspectives are presented in chapter 4.

CHAPTER 2

LITERATURE REVIEW

2.1. Brain tumor types

Let's start by introducing the brain tumor types. Brain tumors are collections of cells in the brain that can be benign, pre-cancerous, or malignant. Tumors, whether malignant or non-cancerous, can produce increased pressure inside the skull (Bhattacharjee et al, 2022). As a result, these tumors can be fatal and cause brain damage. Brain tumors are classified as either primary (originating in the brain) or secondary (occurring in the brain as a result of cancer cells spreading from other organs such as the lungs, kidney, or breast).

Some of the common tumors that grow gradually are:

a) **Meningioma tumor:** occurs in the meninges (membranes that enclose the brain and spinal cord), and is more common in women than men. Figure 2 provides a visual representation of meningioma tumors in various planes of the brain, offering valuable insights into their location and distribution within the cranial cavity.

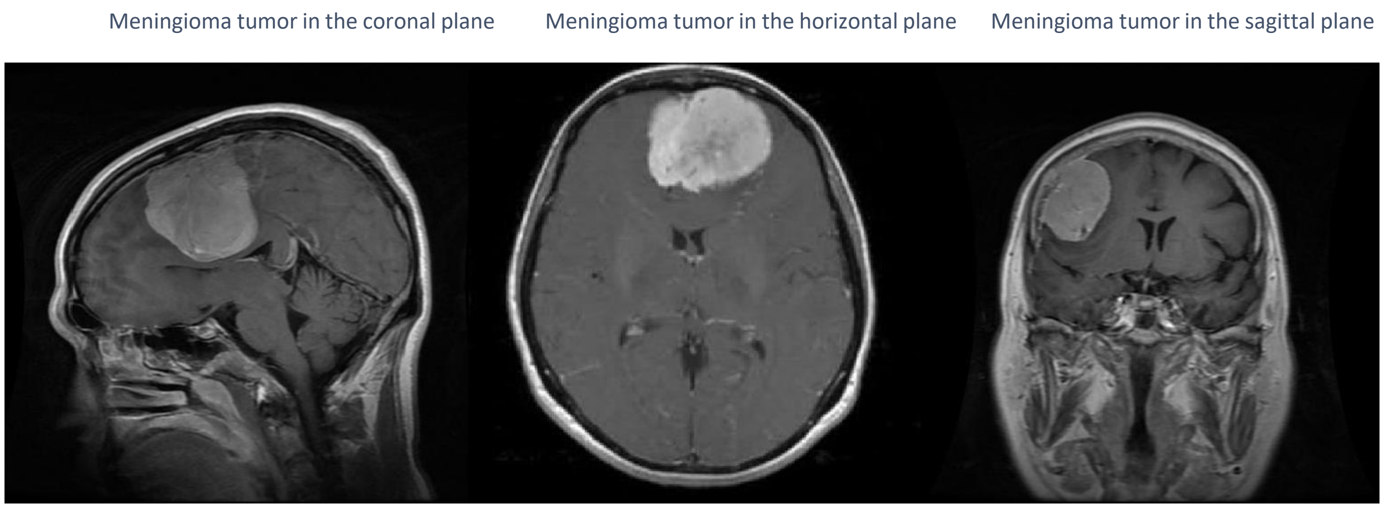

Meningioma tumor in the coronal plane
Meningioma tumor in the horizontal plane
Meningioma tumor in the sagittal plane

Figure 2. Meningioma tumors in different planes of the brain

b) **Glioma tumor:** developed from glial cells. Figure 3 visually depicts glioma tumors observed from different perspectives within the brain, offering a comprehensive view of their spatial distribution and localization within the cranial cavity. This visual representation aids in understanding the characteristics and presentation of glioma tumors.

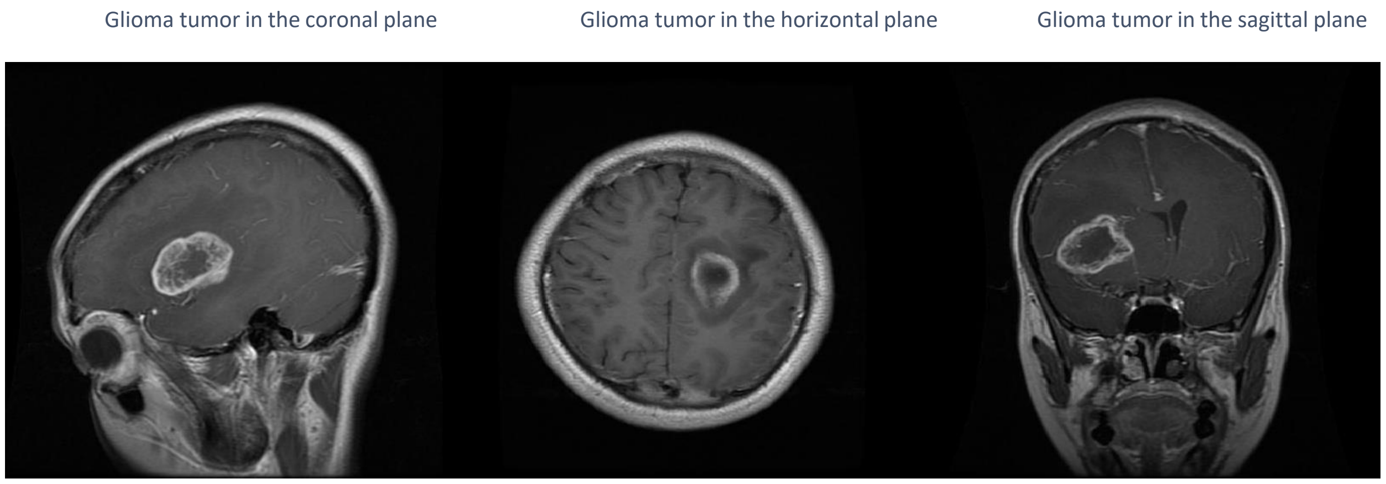

Glioma tumor in the coronal plane
Glioma tumor in the horizontal plane
Glioma tumor in the sagittal plane

Figure 3. Glioma tumors in different planes of the brain

c) **Pituitary tumor:** grows on the area of the pituitary gland. Figure 4 provides a visual representation of pituitary tumors as seen from different perspectives within the brain. This illustration helps elucidate the location and variations in the presentation of pituitary tumors, enhancing our understanding of this specific type of brain tumor.

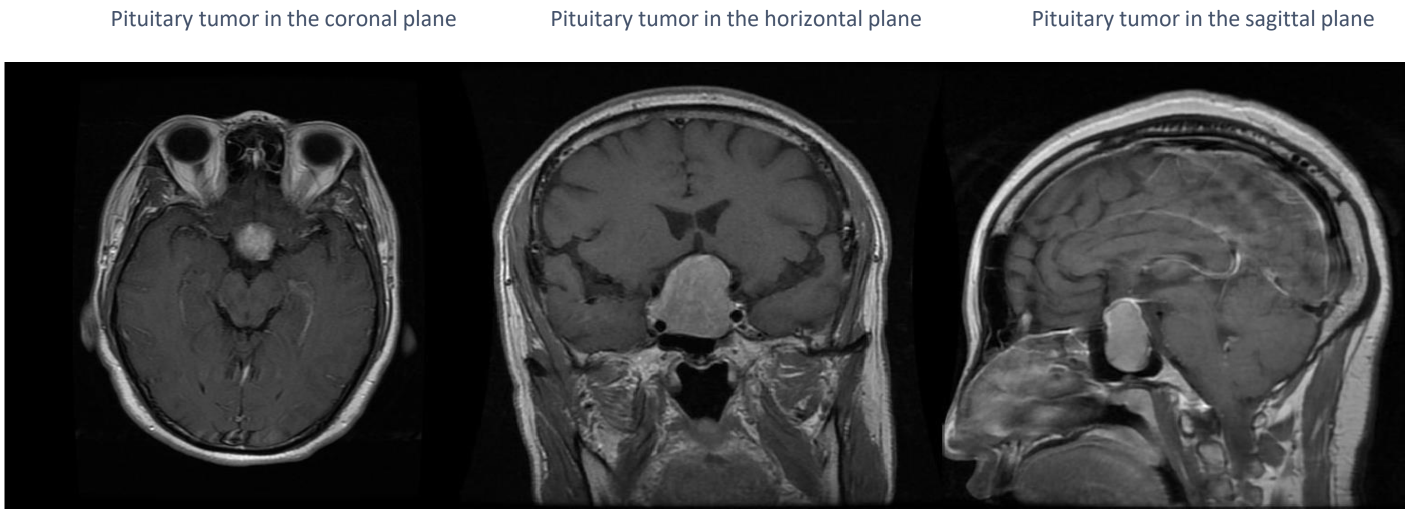

Pituitary tumor in the coronal plane
Pituitary tumor in the horizontal plane
Pituitary tumor in the sagittal plane

Figure 4. Pituitary tumors in different planes of the brain

2.2 U-Net for segmentation and CNN for classification

The deep learning models U-Net and CNN have undergone extensive research and demonstrated strong performance in their respective tasks. These models hold the potential to enhance patient outcomes and treatment approaches in neuro-oncology when combined. Further exploration will shed light on specific insights and findings from various research, highlighting the critical role of U-Net andCNN in advancing brain tumor analysis in medical imaging.

2.3 Segmentation of brain tumor using U-Net

2.3.1 What is U-Net?

U-Net is a deep learning architecture specifically crafted for image segmentation tasks, including brain tumor segmentation. Created by Olaf Ronnenberg et al. in 2015, U-Net has gained prominence asan efficient model for addressing the challenge of limited training data in medical imaging (Yousef et al., 2023). This convolutional neural network (CNN) features an encoder (contracting path) and a decoder (expanding path). The encoder employs convolutional and pooling layers to down sample the input image, while the decoder utilizes convolutional and up sampling layers to restore feature maps to the original size. The encoder and decoder are connected via skip connections that concatenate corresponding feature maps, enabling the decoder to enhance segmentation accuracy by leveraging high-level features from the encoder.

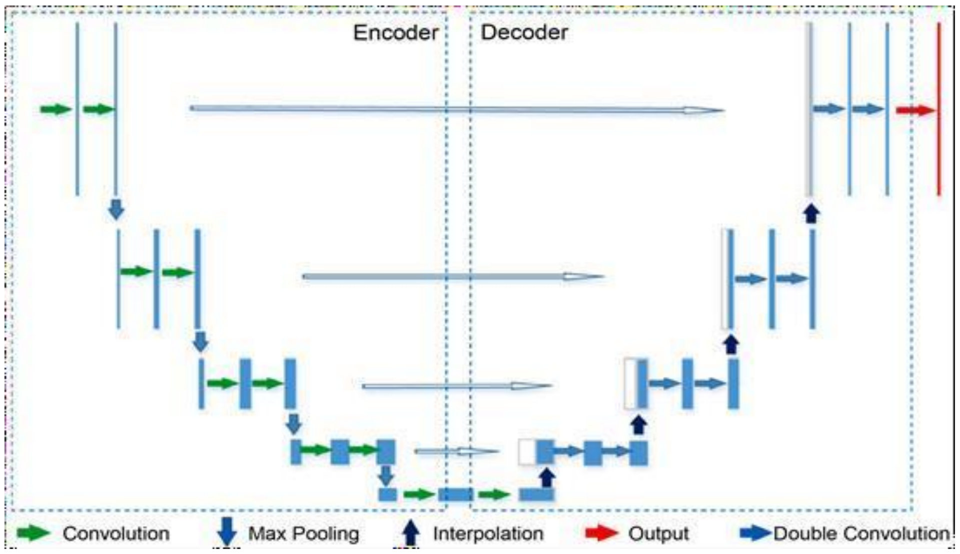


Figure 5. General architecture of U-Net

2.3.2 Advantages and disadvantages of U-Net:

Prior to selecting a specific model for a particular task or problem, it is vital to thoroughly evaluate its merits and limitations. Each machine learning or deep learning model possesses unique strengths and weaknesses that significantly influence its performance and suitability for various applications. Understanding the benefits of a model allows us to leverage its strengths and capitalize on its distinctive characteristics, potentially leading to enhanced accuracy and efficiency in solving the problem at hand. Conversely, being aware of a model's limitations enables us to anticipate potential challenges and drawbacks during its implementation or deployment. This rigorous assessment ensures that the chosen model aligns precisely with the specific requirements of the task, maximizes its advantages, and minimizes the risk of underperformance or unexpected outcomes. Ultimately, making an informed decision regarding the model to be employed can profoundly impact the overall performance and efficacy of the machine learning or deep learning solution.

- **Advantages of U-Net for Brain Tumor Segmentation:**

Excellent Segmentation Accuracy: The U-Net model has demonstrated outstanding segmentation accuracy in various studies, making it a popular choice for brain tumor segmentation tasks. For instance, Walsh et al. (2022) conducted experiments comparing different architectures, including an improved version of U-Net called Linknet. Despite the attempt to enhance the U-Net architecture, the results consistently showed that U-Net outperformed other models in terms of accuracy. This robust performance can be attributed to U-Net's inherent ability to effectively capture relevant features and details, leading to precise tumor segmentation.

Efficient Use of Data: U-Net has proven to be highly efficient in utilizing limited training data, making it suitable for medical imaging tasks where acquiring extensive labeled data can be challenging. According to Xu et al. (2020), among popular convolutional neural networks (CNNs), U-

Net achieves state-of-the-art performance even with a limited amount of training data. This capabilityis particularly advantageous in medical applications, where obtaining a large annotated dataset may be expensive or time-consuming. The U-Net's ability to learn meaningful representations from small datasets enables efficient model training and encourages broader applications in medical image segmentation.

Captures Contextual Information: One of the essential advantages of the U-Net architecture is its ability to capture contextual information, which is critical for accurate segmentation. Du et al. (2020) highlighted this aspect, emphasizing the large number of feature channels in the up-sampling part of U-Net. This allows the network to propagate contextual information to higher resolution layers during the decoding process. As a result, the U-Net model can effectively integrate local details with global context, enhancing its ability to precisely delineate complex tumor structures and boundaries in brain MRI scans.

- **Disadvantages of U-Net for Brain Tumor Segmentation:**

Sensitive to Image Quality: U-Net's performance can be affected by the quality of input images, particularly in the border regions. As noted by Ronneberger et al. (2015), when predicting pixels in the border region of an image, the missing context is extrapolated by mirroring the input image. While this tiling strategy is essential to apply the network to large images without limiting resolution by GPU memory, it may lead to artifacts and inaccuracies at the borders. In cases where image quality is compromised due to noise, artifacts, or variations in imaging techniques, U-Net's sensitivity can resultin less accurate segmentation around the image edges.

High Computational Cost (Long Training Time): U-Net's architecture, especially with the inclusion of skip connections, can contribute to high computational costs during training. As mentioned in a study by S (2023), the additional computations required by skip connections can make U-Net more computationally expensive compared to other architectures. The extensive use of skip connections helpsin capturing multi-scale features but can lead to longer training times, especially when dealing with large datasets. This high computational demand can be a limiting factor for resource-constrainedenvironments and may require access to powerful computing hardware for efficient training.

Limited Interpretability (Black Box): U-Net is considered a black box model, lacking transparency and interpretability in its predictions. This drawback was highlighted by Yin et al. (2022), emphasizing the importance of interpretable deep learning models in medical image analysis. In clinical applications, where segmentation results could impact patient care and diagnosis, the lack of confidence and interpretability in U-Net's predictions can be a concern for medical professionals. Understanding the rationale behind a model's decisions and having trust in its accuracy are essential for clinical acceptance. As U-Net lacks interpretability, its predictions may not be readily trusted and recognized by medical practitioners for critical clinical applications.

2.3.3 Comparing U-Net with other models

- **U-Net vs Atlas-based segmentation**

U-Net is a powerful deep-learning network that has demonstrated tremendous potential in medical picture segmentation applications. A detailed study of U-Net's features is required to successfully utilize its strengths and overcome its weaknesses. To begin, the size and variety of the training datasethave a significant impact on U-Net performance, as proven by works such as Yousef et al. (2023) andYogananda et al. (2020). With access to huge and diverse datasets, the model may learn robust featuresand improve segmentation accuracy. This condition, however, makes acquiring labeled data with reliable ground truth segmentations difficult. Furthermore, as several publications have shown, the architectural complexity of U-Net requires significant memory and processing capacity during training and inference, making it resource-intensive. Regardless of these computing needs, U-Net is successful at producing accurate segmentations, particularly for complex medical images, because to its ability to capture contextual information and U-shaped design.

Atlas-based segmentation, on the other hand, provides an alternate strategy with significant advantages. Atlas-based segmentation reduces the need for significant training by relying on pre- defined atlases or templates, as proven by Bakas et al. (2017), and can be computationally quicker. The use of image registration techniques to match the atlas picture with the new image enables for anatomical knowledge to be transferred. However, the capacity of atlas-based segmentation to accommodate differences in patient anatomy and picture variability may limit its usefulness. When theanatomical discrepancies between atlases and the new picture are significant, this technique may need manual tuning and modifications and struggles to get ideal results. As a result, atlas-based segmentation may not provide consistently accurate segmentations in all circumstances, posing difficulties when dealing with various datasets.

- **U-Net vs K-Means segmentation**

U-Net and K-Means present distinct requirements for their training process in brain tumorsegmentation, each with its advantages and limitations.

According to Yousef et al. (2023) and Yogananda et al. (2020), U-Net is a commonly used architecture for brain tumor segmentation, noted for its capacity to provide accurate segmentations, particularly for complicated pictures. However, realizing U-Net's full potential demands a large amount of labelled training data. The model's effectiveness is strongly reliant on having access to a broad dataset with precise ground truth segmentations, making data annotation a critical and time-consuming step. Furthermore, the architectural complexity of U-Net necessitates substantial quantities of memory and computing power during both training and inference, which might be a bottleneck in resource- constrained applications. During the training phase, the network is optimized using a loss function that quantifies the difference between anticipated and actual segmentations, allowing U-Net to learn the detailed mapping between input pictures and their associated segmentations.

K-Means, on the other hand, is not often used for brain tumor segmentation because to its difficulties in dealing with complicated picture structures, as stated in Thakur (2022) and Khan et al. (2021). When confronted with detailed picture characteristics, the algorithm may suffer with over- segmentation or under-segmentation, making it less suitable for precise tumor delineation. K-Means, on the other hand, has the distinct benefit of not requiring labelled data for training. Instead, it is a clustering algorithm that groups pixels based on their intensity value similarity. The clusters formed are then used to segment the picture, eliminating the requirement for ground truth annotations. While this eliminates the need for data labelling, K-Means' shortcomings in dealing with complicated picture structures restrict its application in brain tumor segmentation settings.

2.4 Classification of brain tumor using CNN:

After employing the U-Net for brain tumor segmentation, let's shift our focus to CNN, the model used for tumor classification

2.4.1 What is CNN (Convolutional Neural Networks)?

According to Dickson (2020), Convolutional neural networks, often referred to as ConvNets, made their initial appearance in the 1980s, pioneered by Yann LeCun, a postdoctoral researcher in computer science. LeCun's contributions built upon the earlier work of Kunihiko Fukushima, a Japanese scientist who had developed a rudimentary image recognition neural network known as recognition a few years prior.

Convolutional neural networks (CNNs) are made up of artificial neurons arranged in layers, inspired by the functioning of biological neurons. These artificial neurons compute the weighted sum of inputs and yield an activation value. When an image is fed into a CNN, each layer produces activation maps, which pinpoint significant visual features. Within the CNN, individual neurons receive input as pixel patches, apply their assigned weights to the pixel values, aggregate the results, and pass them through an activation function.

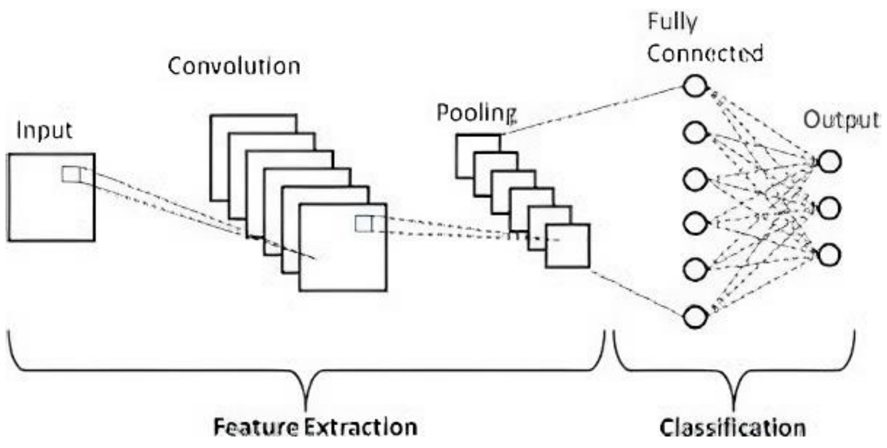


Figure 6. Brief representation of CNN architecture

2.4.2 Advantages and disadvantages of CNN

Before choosing a model for a certain activity or problem, it is critical to properly assess its benefits and drawbacks. Each machine learning or deep learning model has strengths and limitations that can have a substantial influence on its performance and applicability. Understanding a model's benefits enables us to capitalize on its strengths and capitalize on its distinctive features, perhaps leading to better accuracy and efficiency. On the other side, being aware of the model's shortcomings allows us to predict probable issues and drawbacks during implementation or deployment. This rigorous review ensures that the chosen model is aligned with the precise needs of the problem at hand, that its advantages are maximized, and that the danger of poor performance or unexpected effects is minimized. Finally, making an informed decision on the model to use may have a significant impact on the overall performance and efficacy of the machine learning or deep learning solution.

- **Advantages of CNN**

High accuracy: Krizhevsky et al. (2012) explained that due to a number of characteristics, CNNs are very accurate in computer vision applications. First, during training, they may automatically learn pertinent characteristics from the input, collecting complex patterns and representations, which improves their performance. Second, to gradually distinguish higher-level characteristics from lower-level ones and learn complicated spatial hierarchies of features, CNNs employ hierarchical structures with numerous layers, which increases their accuracy. Additionally, weight sharing and convolutional filters help CNNs use less memory and have fewer parameters, which improves generalization with less training data. Furthermore, they are able to generalize to previously undiscovered data thanks to their translation-invariant nature, which makes them resilient to slight translations or input shifts and enables them to recognize patterns in many spatial locations.

Robust to variation and noise: According to Krizhevsky et al. (2012), through a number of processes, Convolutional Neural Networks (CNNs) exhibit resilience to changes and noise. In order to focus on local patterns and features and become less susceptible to global changes and noise, they first apply local feature learning utilizing modest filters. Second, CNNs use pooling layers to down-sample feature maps, keeping just the most important features and rejecting extraneous information and noise to increase their resilience. Additionally, they are immune to small changes or translations in the input because to their intrinsic translation invariance, which enables them to recognize patterns regardless of their precise position. Additionally, data augmentation approaches help CNNs build more robust and generalized representations by adding fake noise and fluctuations to the training dataset. Finally, to avoid overfitting and enhance the network's capacity to generalize to noisy or unknown input, regularization techniques like dropout, batch normalization, or weight decay are used.

Automatic feature extraction: According to LeCun et al. (2015), CNNs learn specific features in layers, starting with low-level features like edges and textures in the early layers and progressing to higher-level features like shapes and objects in deeper layers.

This hierarchical architecture allows CNNs to automatically extract features. CNNs use convolutional filters to scan input data for local patterns and important characteristics. These relevant features are then learned via convolution operations with weights that are shared across various regions. This makes it possible to identify comparable patterns, which helps with automated feature extraction. CNNs can automatically capture and highlight significant characteristics that are crucial for the job at hand during training by applying weight learning using back propagation and optimization strategies. CNNs may also learn feature representations directly from raw input data without the need for handmade feature engineering since they are trained in an end-to-end way, concurrently optimizing the whole network.

Efficient use of memory: According to Han et al. (2015), through a variety of methods, Convolutional Neural Networks (CNNs) efficiently use memory. In order to minimize the number of parameters and increase memory efficiency, they first make use of parameter sharing and convolutional filters, especially for huge pictures. Second, down sampling feature maps using pooling layers helps optimize memory use by keeping important details while eliminating less important ones. To further improve memory effectiveness, CNNs also show sparsity in their activation maps, firing just the appropriate neurons for each input data point. Reduced precision is made possible via weight quantization, which greatly reduces memory needs without sacrificing accuracy. After training, fewer important connections or neurons are removed from the model to reduce its memory footprint while maintaining performance. Last but not least, deep compression uses methods like weight sharing, quantization, and Huffman coding to significantly reduce the size of CNN models and improve memory utilization.

- **Disadvantages of CNN**

Require large amounts of data: According to Krizhevsky et al. (2012), for effective training, Convolutional Neural Networks (CNNs) need a lot of data for a variety of reasons. First off, because CNNs train by directly learning features from data, a large and representative dataset is required to capture complex patterns and variations. The model can generalize successfully to novel and untested cases thanks to enough data. Second, during training, millions of parameters in CNNs must be adjusted. To correctly optimize these parameters and prevent overfitting, when the model memorizes training samples rather than learning generalizable characteristics, sufficient data is required. Large datasets offer a wider variety of samples, which helps prevent overfitting. Finally, for CNNs to learn meaningful abstractions, a variety of instances at various levels are needed to learn hierarchical representations of features. In conclusion, big datasets are essential for CNNs to function accurately and robustly because they make it easier to learn features effectively, adjust parameters, prevent overfitting, and capture hierarchical patterns.

Computationally expensive: Due to a number of causes, Convolutional Neural Networks (CNNs) may be computationally costly. First, the computational effort is increased by the introduction of convolutional layers with numerous filters to scan input data and extract features as well as pooling layers for down sampling. Secondly, state-of-the-art models, especially those with deep CNN architectures with many layers, need a large number of operations during training and inference. Additionally, since extensive data processing is required for efficient learning, training CNNs with big datasets increases the computing cost. Additionally, processing and analyzing high-resolution pictures, which are typical in computer vision jobs, necessitates additional calculations. All in all, these elements add to CNNs' computational cost (Szegedy et al, 2015).

Susceptible to overfitting: According to Krizhevsky et al. (2012), for a variety of reasons, Convolutional Neural Networks (CNNs) are prone to overfitting. First off, deep architecture CNNs sometimes include a huge number of learnable parameters, which might cause users to memorize training data rather than learn generalizable features. Intricate patterns in the training data can be captured by complicated models with numerous layers, but this complexity also raises the possibility of overfitting, particularly when the dataset is limited or lacking in variety. Additionally, having insufficient training data can lead to overfitting since CNNs may pick up on noise or certain traits from the small dataset rather than actual patterns. Overfitting can also occur as a result of inadequate or incorrect regularisation procedures, such as dropout, batch normalisation, or weight decay. Data augmentation should also be used cautiously to ensure that the enhanced data stays reflective of the real-world distribution, even while it can assist prevent overfitting. For CNNs to be more generalizable and to reduce overfitting, these elements must be managed properly.

Limited interpretability (black box): Due to a variety of factors, Convolutional Neural Networks (CNNs) display low interpretability (black box). First of all, their intricate structures, which include several layers, a large number of neurons, and filters, make it difficult to comprehend the inner workings and justification of specific judgments, especially as the model gets more complicated. Second, non-linear activation functions used by CNNs to alter input data make it difficult to directly evaluate how input attributes relate to output predictions. Thirdly, it is challenging to comprehend the precise characteristics or patterns that contribute to the final judgment since CNNs provide high-dimensional feature representations at various levels. Additionally, during training, CNNs automatically pick up characteristics from the data that might not correspond to ideas that people can understand or be directly related to certain input features. Last but not least, the dispersed structure of deep networks makes it difficult to comprehend how certain parameters affect model predictions, which adds to the black box aspect of CNNs.

2.4.3 Comparing CNN with other models

It is essential to thoroughly compare all potential models before selecting one for a particular machine learning assignment in order to get to the optimal conclusion. Machine learning includes a wide range of methods, each with advantages and disadvantages, and selecting the best model can have a big influence on the results. We may evaluate the performance of several models across a range of measures, including accuracy and computing efficiency. To fully comprehend the trade-offs between accuracy and explain ability, it is also crucial to take the interpretability and complexity of the models into account. By taking into account variables like the size of the dataset, the kind of features, and the required level of interpretability, the comparison process aids in choosing the model that best fits the needs of the work at hand. Model comparison increases the likelihood of precise forecasts or insightful analysis and assures that the chosen model is well-suited to handle the given situation.

CNN vs KNN (K-Nearest Neighbors):

According to Havaei (2017), Convolutional Neural Networks (CNNs) have outperformed K- Nearest Neighbors (KNN) in the classification of brain tumors. Processing medical pictures, such as MRI scans, is a common step in brain tumor classification assignments. CNNs have succeeded in this area because they can learn hierarchical features directly from the raw image data.

The ability of CNNs to extract complex patterns and representations from medical pictures is crucial for differentiating between various forms of brain tumors. They perform incredibly well in image-based classification tasks thanks to their deep design, which enables them to automatically learn pertinent characteristics.

KNN, on the other hand, is a straightforward algorithm that cannot learn features from data. It uses metrics that are dependent on distance to categorize new data points, which might not be appropriate for complicated medical picture data with many dimensions.

CNN vs Random forests:

In comparison to Random Forests, Convolutional Neural Networks (CNNs) have demonstrated better performance in the classification of brain tumors, according to Korfiatis (2016). Because they can automatically identify pertinent features from unprocessed picture data and recognize complicated patterns, CNNs are especially well-suited for image-based classification tasks. Due to the hierarchical representations that CNNs develop, they are quite good at identifying different forms of brain tumors in medical imaging.

Contrarily, the ensemble learning technique known as Random Forests can handle both categorical and continuous data. To make the final categorization, they build many decision trees and integrate their predictions. While durable and capable of performing well in some classification tasks, Random Forests might not be as efficient as CNNs when handling high-dimensional picture data with intricate patterns.

CHAPTER 3

METHODOLOGY AND IMPLEMENTATION

In this chapter, we adhere to a structured approach outlined in the project's protocol (Section 3.1) for training and evaluating two distinct neural network models. Section 3.2 focuses on the Convolutional Neural Network (CNN), encompassing data preprocessing, the creation of CNN model layers, training, model saving, and performance evaluation during testing. Shifting to the U-Net model in Section 3.3, we delve into its training and testing steps, including an explanation of U-Net layers, data preprocessing, defining the model's layers, training, saving the U-Net model, and assessing its performance through testing. This chapter provides a comprehensive guide to the key phases involved in training and evaluating both CNN and U-Net models, maintaining a structured approach as per the specified protocol.

The methodology employed consists of two main steps for both the CNN and U-Net models. In the CNN approach, data is initially preprocessed, then split into training and testing sets. The training data undergoes 30 epochs of training, and the resulting model is saved in an HDF5 file. Similarly, in the U-Net approach, data is preprocessed and trained for 250 epochs, with the model also saved in an HDF5 file. Subsequently, the system utilizes these saved models for predictions, as illustrated in Figure 7.

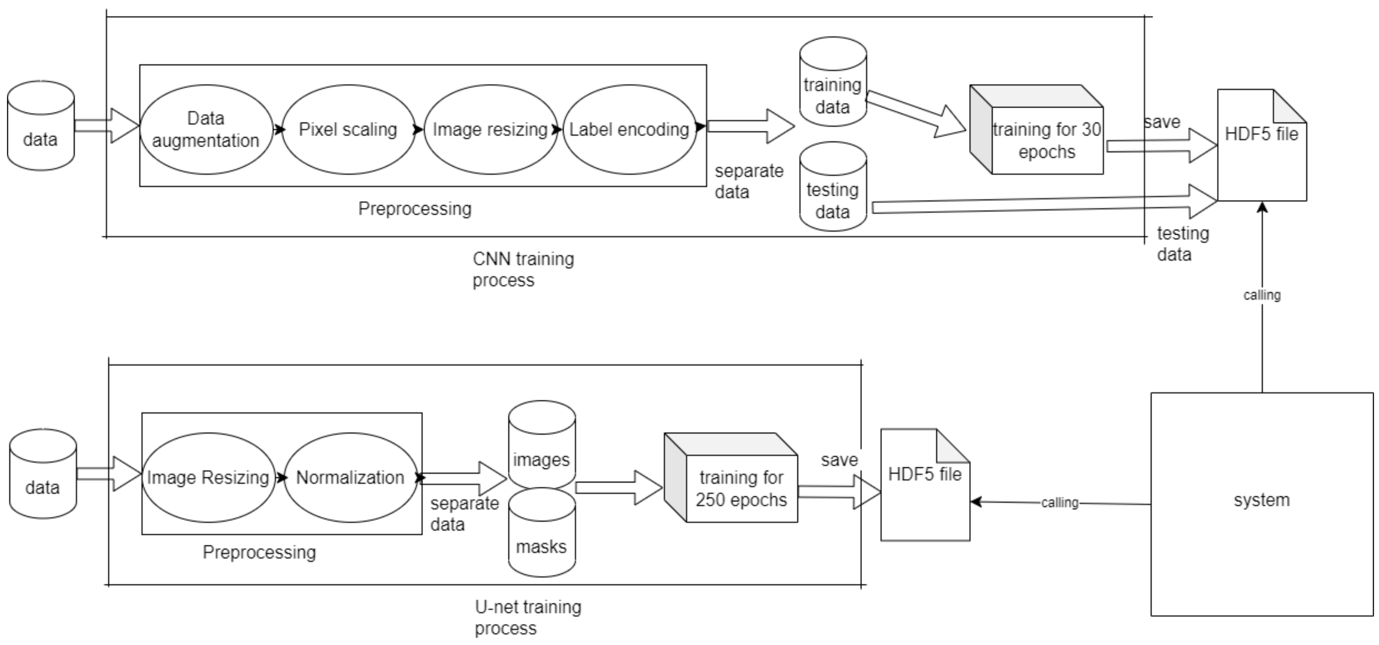


Figure 7. Brief representation the project's methodology

3.1 Protocol

a) Tools and Environment

The brain tumor segmentation and classification project utilized various tools and environments:

- Visual Studio Code: A versatile source code editor used for developing and editing the Python codebase of the desktop application.
- Jupyter Notebook: An interactive computing environment employed for exploratory data analysis, prototyping, and testing of machine learning algorithms and models.
- Oracle VM Virtualbox: A powerful virtualization tool used for creating and managing virtual machines to test the application on various operating systems and configurations.

b) Language and Libraries

- The project was developed using the following programming language and libraries:
- Python: A widely-used, high-level programming language known for its simplicity and readability, making it well-suited for various applications, including machine learning and image processing.
- TensorFlow: An open-source machine learning framework developed by Google that provides a flexible and efficient ecosystem for building and deploying machine learning models.
- OpenCV: A powerful library of computer vision algorithms, enabling tasks such as image processing, feature extraction, and object detection.
- PyQt5: A Python binding for the Qt application framework, allowing developers to create cross-platform graphical user interfaces (GUIs) for desktop applications.

c) Dataset and Code Repositories

The project utilized the following resources for data and code:

- Kaggle: A renowned online platform that offers a wide array of datasets and hosts data science competitions. Kaggle served as a valuable resource for accessing curated datasets relevant to the project, facilitating data-driven analysis and model training.
- GitHub: A widely-used web-based hosting service that provided an essential platform for version control and collaborative development. GitHub was leveraged to discover and access various code repositories, including open-source projects and libraries. These repositories provided valuable insights and code

snippets, aiding in the implementation and fine-tuning of the machine learning models utilized in the project.

d) Dataset Used for Training and Testing

1) Dataset used for CNN

The dataset employed in this study was sourced from Kaggle, a reputable platform for machine learning resources. It was meticulously organized into two distinct folders, each serving a crucial role inour research endeavors.

The first folder contained Brain MRI scans, specifically focusing on cases with tumors. These high-resolution MRI scans were stored in the Portable Network Graphics (PNG) format, ensuring optimal image quality and fidelity. The presence of tumors within the brain structures was accurately captured in these scans.

In the second folder, we found the masked tumors of the Brain MRI scans. The purpose of this folder was to provide labeled data, indicating the specific regions within the brain where tumors were present. The masks were carefully created to represent tumor regions in white, contrasting against the black background of the rest of the image. This binary representation facilitated precise localization of tumors, aiding our model in identifying and analyzing their spatial distribution.

A critical aspect of this dataset was its uniformity in size. Both the Brain MRI scans folder and the masked tumors folder contained an identical number of images, resulting in a total of 3064 pairs. This balance ensured fairness during the training and evaluation of our model.

By leveraging this comprehensive dataset, our model achieved remarkable performance in tumor detection and analysis. The use of high-quality Brain MRI scans and their corresponding masked tumor images provided a robust foundation for accurate outcomes, offering the potential to advance medical diagnostics and patient care significantly.

The provided bar graph presents the quantity of data employed for training the U-Net model, alongside the initially intended dataset size. This dataset encompasses MRI scans and annotated tumors.

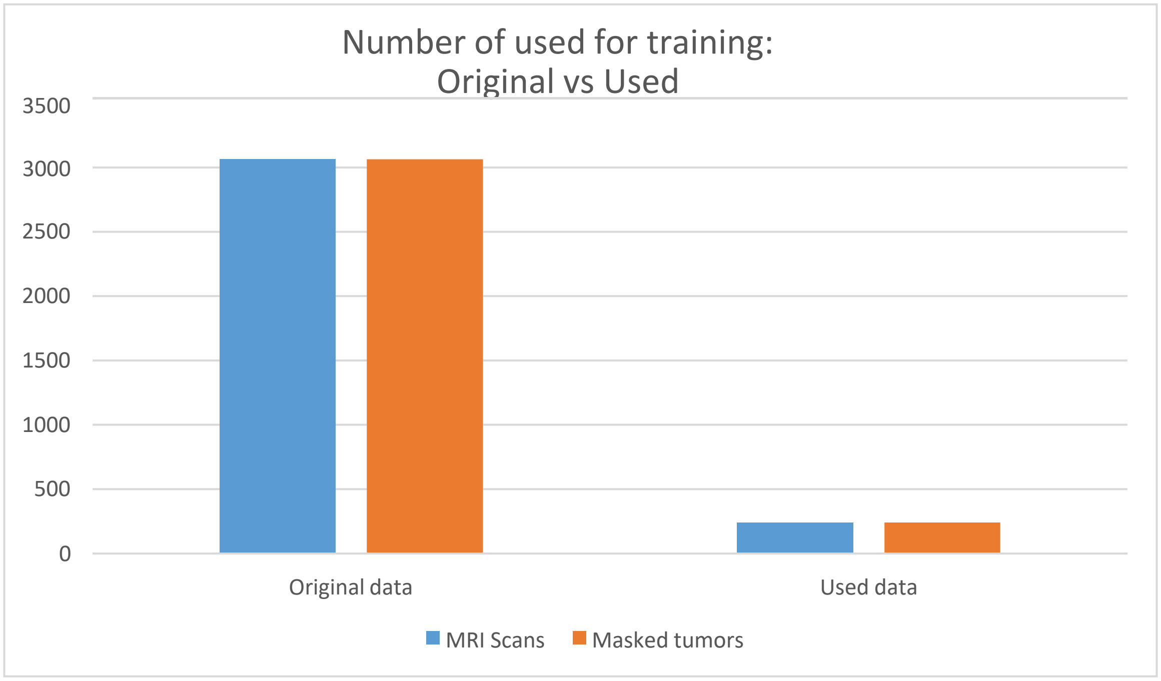


Figure 8. Bar-graph shows original vs used data

Here is the URL to the dataset we utilized for training of U-Net model: https://www.kaggle.com/datasets/nikhilroxtomar/brain-tumor segmentation

2) Data used to train and test the model

The following bar graph shows the different sets of data we used to train and test the CNN model. Each bar stands for a dataset and tells us how much data was used for training and testing. The graph helps us compare the sizes of the datasets and how they were split between teaching the program and checking how well it learned.

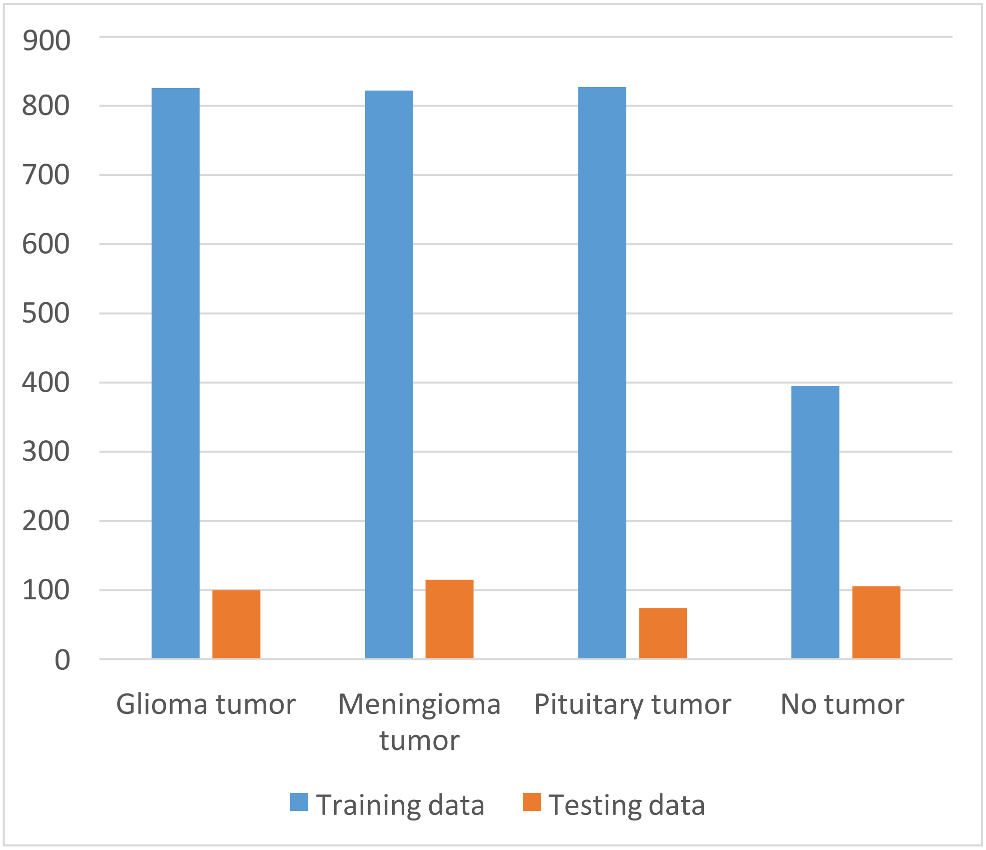


Figure 9. Bar-graph shows the data used for CNN model

Here is the URL to the dataset we utilized for training of CNN model:

https://www.kaggle.com/sartajbhuvaji/brain-tumor-classification-mri

3.2 CNN training and testing steps

3.2.1. Preprocessing

i. Data augmentation:

Convolutional neural networks (CNNs) and data augmentation are frequently used to improve the performance and generalizability of the model. It entails using several transformations and alterations to the original dataset to create fresh training data while preserving the label and meaning of the data. The diverse training set provided by the enhanced data enables CNN to extract more robust and consistent characteristics from the pictures (Perez el al, 2017). We employed Random Rotations, which introduce rotational invariance by rotating pictures at random angles. After augmentation, the testing data increased to 2,000 and the training data to 12,000 each.

ii. Pixel scaling:

The act of rescaling the pixel values of photographs to a typical range is known as "pixel scaling," sometimes known as "normalization" or "standardization." The goal of pixel scaling is to increase the convergence of the training process and to make the input data consistent. It entails converting the pixel values so that the data either falls inside a predetermined range [0, 1] or has a zero mean and unit variance.

Large pixel value ranges that could lead to numerical instability during training can be avoided with the use of pixel scaling. The scaling of the pixel values to a common range improves the stability and effectiveness of the optimization process. Additionally, it may be necessary to modify the learning rate utilized during training when the input data has a large range of pixel values. Pixel scaling makes it possible to regulate the learning rate to an acceptable level, resulting in quicker convergence and more dependable training.

iii. Image resizing:

Every picture in the dataset must have a defined input size for CNNs. Resizing ensures that all of thepictures are the same size, which makes it possible to handle them quickly during training and inference.

iv. Label encoding:

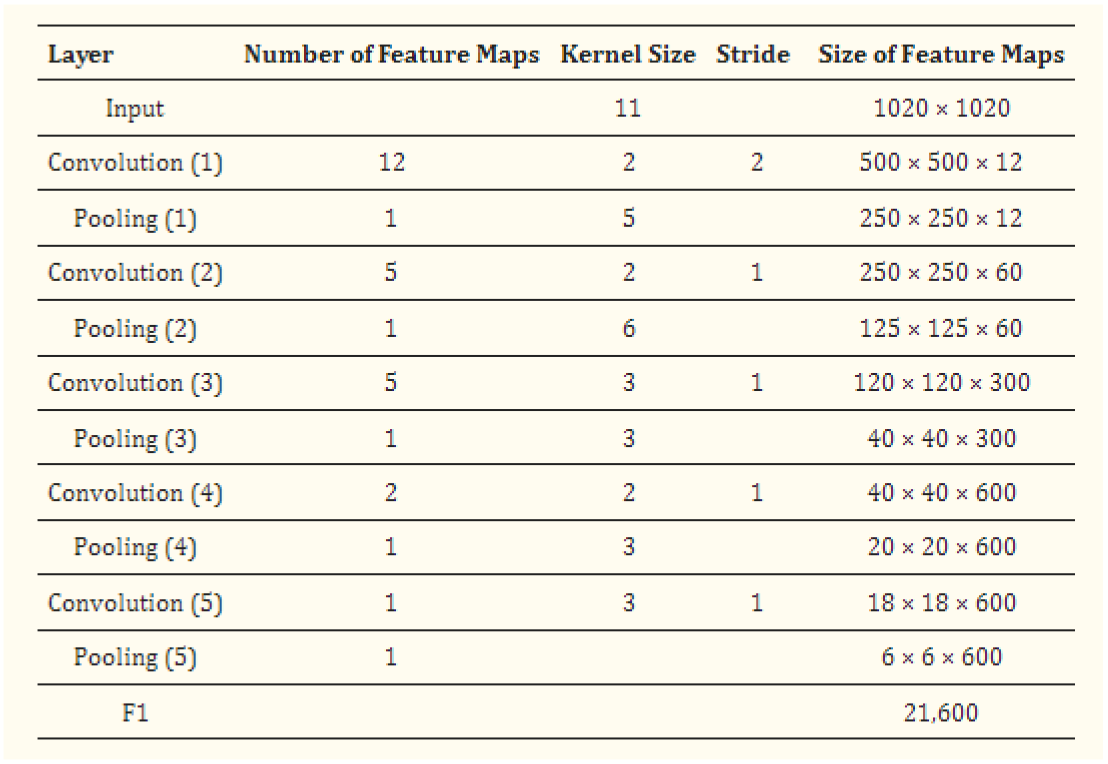
Convolutional Neural Networks (CNN) classification tasks frequently employ the preprocessing method known as label encoding to transform category labels into numerical notation. The target labels are frequently expressed in CNN applications as class names or categories, which are generally strings of text. However, for training and prediction, CNNs need numerical inputs.

Label encoding efficiently transforms categorical labels into numerical representations by assigninga distinct integer value to each class or category. For instance, label encoding may map the three classes A, B, and C to 0, 1, and 2, respectively. In our case, we set the glioma tumor as 0, the meningioma tumor as 1, the no tumor as 2, and the pituitary tumor as 3.

3.2.2. Creating the model's layers

To find the most effective model architecture, we tested three distinct ones with various amounts of layers in each. Finding the ideal balance between model complexity and predictive power was the aim.

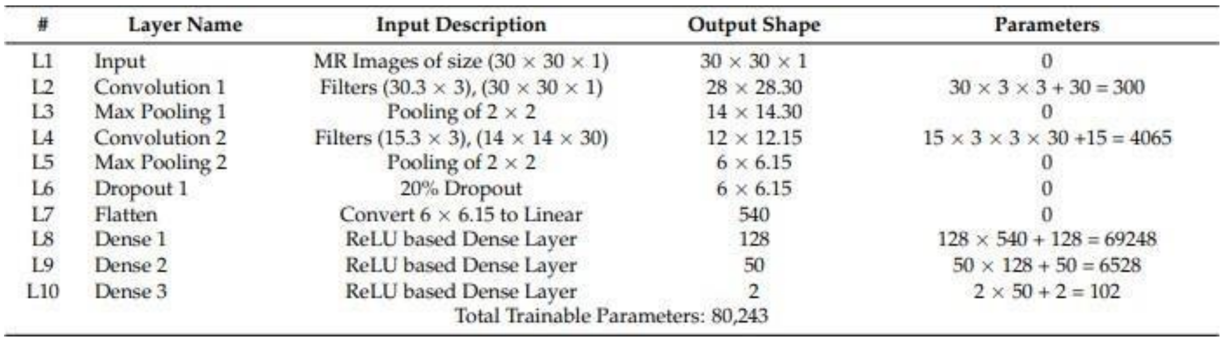
The initial model, taken from the "Differential Deep Convolutional Neural Network Model for Brain Tumor Classification" article (2021), was computationally effective because of its shallow designand limited number of layers, but it might not have been able to fully capture complex patterns in the data.



Layer	Number of Feature Maps	Kernel Size	Stride	Size of Feature Maps
Input		11		1020 × 1020
Convolution (1)	12	2	2	500 × 500 × 12
Pooling (1)	1	5		250 × 250 × 12
Convolution (2)	5	2	1	250 × 250 × 60
Pooling (2)	1	6		125 × 125 × 60
Convolution (3)	5	3	1	120 × 120 × 300
Pooling (3)	1	3		40 × 40 × 300
Convolution (4)	2	2	1	40 × 40 × 600
Pooling (4)	1	3		20 × 20 × 600
Convolution (5)	1	3	1	18 × 18 × 600
Pooling (5)	1			6 × 6 × 600
F1				21,600

Figure 10. First model later created

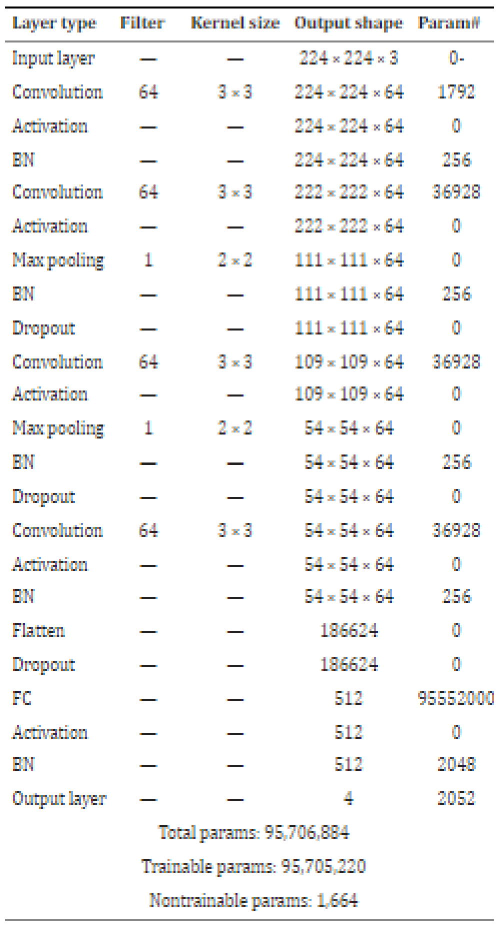
The second model, taken from the "DeepTumor: Framework for Brain MR Image Classification, Segmentation, and Tumor Detection" article, which offered a balance between complexity and processing capacity, had a modest number of layers.



#	Layer Name	Input Description	Output Shape	Parameters
L1	Input	MR Images of size (30 × 30 × 1)	30 × 30 × 1	0
L2	Convolution 1	Filters (30.3 × 3), (30 × 30 × 1)	28 × 28.30	30 × 3 × 3 + 30 = 300
L3	Max Pooling 1	Pooling of 2 × 2	14 × 14.30	0
L4	Convolution 2	Filters (15.3 × 3), (14 × 14 × 30)	12 × 12.15	15 × 3 × 3 × 30 +15 = 4065
L5	Max Pooling 2	Pooling of 2 × 2	6 × 6.15	0
L6	Dropout 1	20% Dropout	6 × 6.15	0
L7	Flatten	Convert 6 × 6.15 to Linear	540	0
L8	Dense 1	ReLU based Dense Layer	128	128 × 540 + 128 = 69248
L9	Dense 2	ReLU based Dense Layer	50	50 × 128 + 50 = 6528
L10	Dense 3	ReLU based Dense Layer	2	2 × 50 + 2 = 102
		Total Trainable Parameters: 80,243		

Figure 11. Second model layer created

The third model, from the article "CNN Based Multiclass Brain Tumor Detection Using Medical Imaging" (2022), which had numerous layers and had the ability to learn more complicated representations from the data, also required more processing power.



Layer type	Filter	Kernel size	Output shape	Param#
Input layer	—	—	224 × 224 × 3	0-
Convolution	64	3 × 3	224 × 224 × 64	1792
Activation	—	—	224 × 224 × 64	0
BN	—	—	224 × 224 × 64	256
Convolution	64	3 × 3	222 × 222 × 64	36928
Activation	—	—	222 × 222 × 64	0
Max pooling	1	2 × 2	111 × 111 × 64	0
BN	—	—	111 × 111 × 64	256
Dropout	—	—	111 × 111 × 64	0
Convolution	64	3 × 3	109 × 109 × 64	36928
Activation	—	—	109 × 109 × 64	0
Max pooling	1	2 × 2	54 × 54 × 64	0
BN	—	—	54 × 54 × 64	256
Dropout	—	—	54 × 54 × 64	0
Convolution	64	3 × 3	54 × 54 × 64	36928
Activation	—	—	54 × 54 × 64	0
BN	—	—	54 × 54 × 64	256
Flatten	—	—	186624	0
Dropout	—	—	186624	0
FC	—	—	512	95552000
Activation	—	—	512	0
BN	—	—	512	2048
Output layer	—	—	4	2052
Total params: 95,706,884				
Trainable params: 95,705,220				
Nontrainable params: 1,664				

Figure 12. Third model layers created and applied

We evaluated the models' generalizability and accuracy by thorough examination and comparison using acceptable performance criteria on a validation dataset. Based on the findings, we chose the model that performed the best, striking the ideal balance between depth and performance and providing a workable solution for the particular problem at hand. The third model was ultimately picked because it displays the ability to make precise forecasts while maintaining computational effectiveness.

All layers are created using Keras library. The Keras library is a high-level neural network API written in Python that provides an easy-to-use interface for building and training deep learning models,including CNNs (Chollet, F, 2015).

3.2.3. Training and saving the model

The model effectively learnt from the training data after 30 training iterations, adjusting its parameters to maximize performance. Based on the suggestions from the paper we cited while creating the layers for the model, the number of training epochs was chosen. The trained model was saved to a file using the Hierarchical Data Format version 5 (HDF5) file format to keep it for future usage and analysis. HDF5 is the perfect format for storing neural network models together with their corresponding weights and architecture since it offers a versatile and effective solution to store huge and complicated datasets. The trained model is now saved as an HDF5 file, which makes it simple to retrieve and use for a variety of purposes, such as generating predictions on fresh data or continuing training in later sessions, guaranteeing the model's insightful conclusions are simply accessible and repeatable. Figure 13 presents the result of the training.

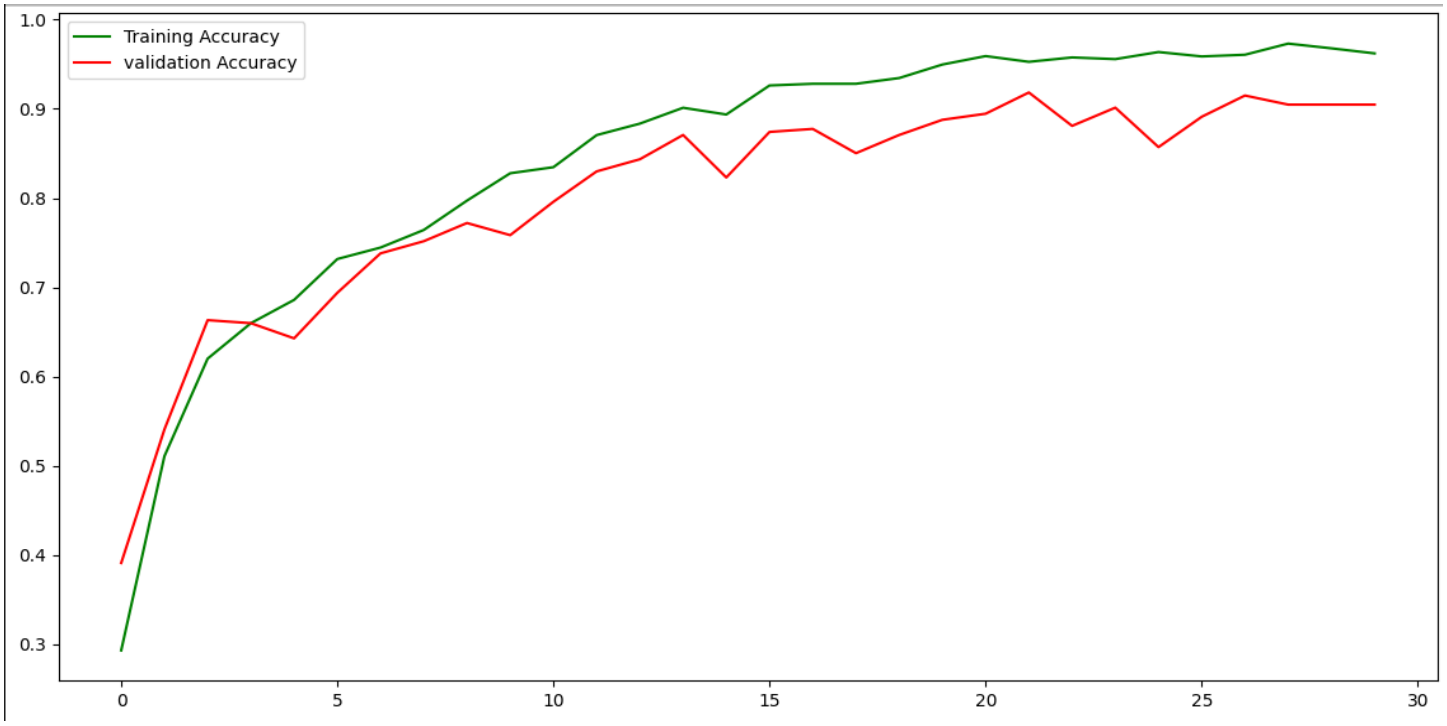
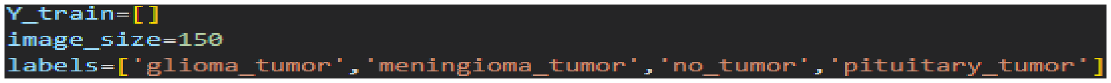


Figure 13. Graph showing the result of the training with respect to the epochs

Note that a small difference between the training accuracy and validation accuracy (less than 5-10%) is a good indication that the model is not overfitting and is generalizing well to new data.

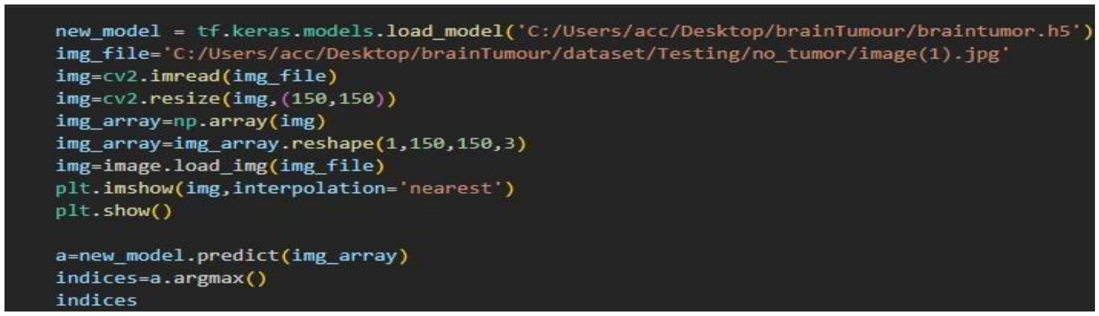
3.2.4. Testing the model

Figures 13 - 16 present the testing steps and the result:



```
Y_train=[]
image_size=150
labels=['glioma_tumor','meningioma_tumor','no_tumor','pituitary_tumor']
```

Figure 14. The labels generated in the preprocessing step



```
    new_model = tf.keras.models.load_model('C:/Users/acc/Desktop/brainTumour/braintumor.h5')
    img_file='C:/Users/acc/Desktop/brainTumour/dataset/Testing/no_tumor/image(1).jpg'
    img=cv2.imread(img_file)
    img=cv2.resize(img,(150,150))
    img_array=np.array(img)
    img_array=img_array.reshape(1,150,150,3)
    img=image.load_img(img_file)
    plt.imshow(img,interpolation='nearest')
    plt.show()

    a=new_model.predict(img_array)
    indices=a.argmax()
    indices
```

Figure 15. Testing a random image on the model created

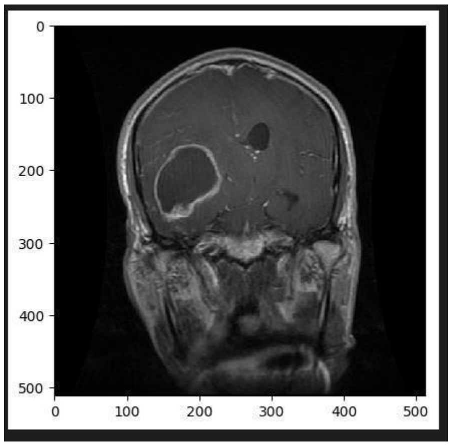
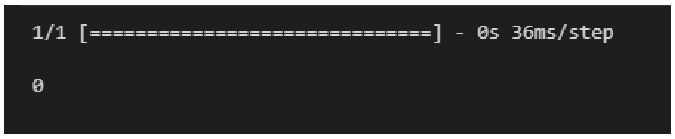


Figure 16. The random image that got selected to test



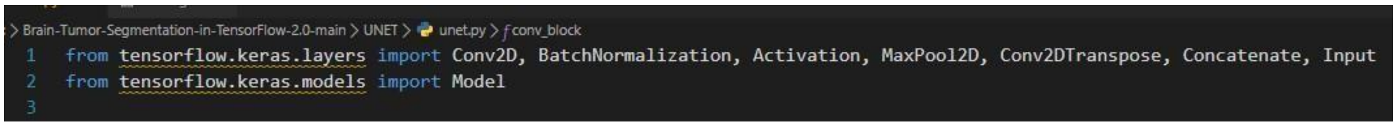
```
1/1 [==============================] - 0s 36ms/step

0
```

Figure 17. The result “0” indicates that there is a Glioma Tumor

3.3 U-Net training and testing steps

3.3.1. U-Net layers used

In this part, we'll go into the TensorFlow Keras code for a U-Net model for semantic segmentation. Due to its capability to precisely define objects inside pictures, U-Net is a strong and well-liked model that is frequently employed for image segmentation tasks.


> Brain-Tumor-Segmentation-in-TensorFlow-2.0-main > UNET > unet.py > conv_block

```python
from tensorflow.keras.layers import Conv2D, BatchNormalization, Activation, MaxPool2D, Conv2DTranspose, Concatenate, Input
from tensorflow.keras.models import Model

```

Figure 18. Importing needed libraries

We start this part by importing the necessary TensorFlow Keras modules and layers. These libraries are essential for establishing the U-Net model for semantic segmentation and serve as building blocksfor neural network designs.

Conv2D: This layer performs 2D convolutions on the input data. It is used to extract spatial features from the input images, learning patterns and representations that are important for the segmentation task.

BatchNormalization: This layer normalizes the outputs of the previous layer. It helps stabilize and speed up training by ensuring that the inputs to subsequent layers are in a consistent range.

Activation: This layer applies activation functions to introduce non-linearity into the model. Here, the "relu" (Rectified Linear Unit) activation function is used, which is commonly employed in neural networks to enable them to learn complex patterns.

MaxPool2D: This layer performs max-pooling operations on the input data. It reduces the spatial dimensions of the feature maps, preserving the most important information while discarding less relevant details. Max-pooling helps down sample the feature maps, making the network more computationally efficient.

Conv2DTranspose: Also known as "deconvolution" or "up-sampling" layers, this type of layer performs 2D transposed convolutions. It is used for up sampling the feature maps, recovering spatial information lost during the down sampling process. It helps the decoder part of the U-Net to increase the resolution of the feature maps.

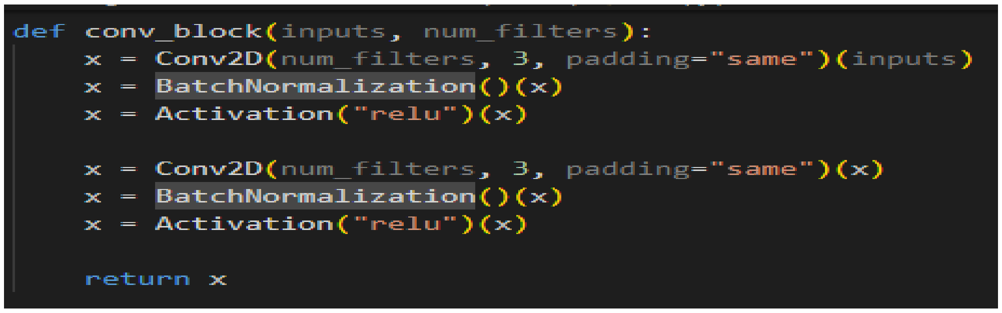
Concatenate: This layer concatenates the feature maps from different parts of the network. In the U- Net, skip connections are used to connect the encoder features with the corresponding decoder features. The concatenate layer is used to combine these features, preserving low-level spatial details while providing high-level context.

Input: This module is used to define the input layer of the neural network. It specifies the shape andtype of the input data that the network will process.

Model: This module is used to define the overall model architecture. It takes the input and output layers as arguments and constructs the neural network model. In this case, we'll use it to create the U-Net model by specifying the input and output layers later in the code. By integrating these libraries, we set the groundwork for the U-Net architecture. The next functions and code snippets will employ these layers to build the encoder, decoder, and overall U-Net model for semantic segmentation.

Following the import of the essential libraries, the code then defines numerous functions that play critical roles in the construction of the U-Net architecture. These functions are in charge of constructing the convolutional blocks, encoder blocks, decoder blocks, and the overall U-Net model.

First function is “conv_block”:



```
def conv_block(inputs, num_filters):
    x = Conv2D(num_filters, 3, padding="same")(inputs)
    x = BatchNormalization()(x)
    x = Activation("relu")(x)

    x = Conv2D(num_filters, 3, padding="same")(x)
    x = BatchNormalization()(x)
    x = Activation("relu")(x)

    return x
```

Figure 19. conv_block function

This function defines an essential building part of the U-Net architecture. It is made up of two successive 2D convolutional layers that have batch normalization and ReLU activation. These layers collaborate to discover essential aspects of the input data. The num_filters parameter determines the complexity of the learned features by controlling the number of filters (output channels) for each convolutional layer.

Second function is “encoder_block”:

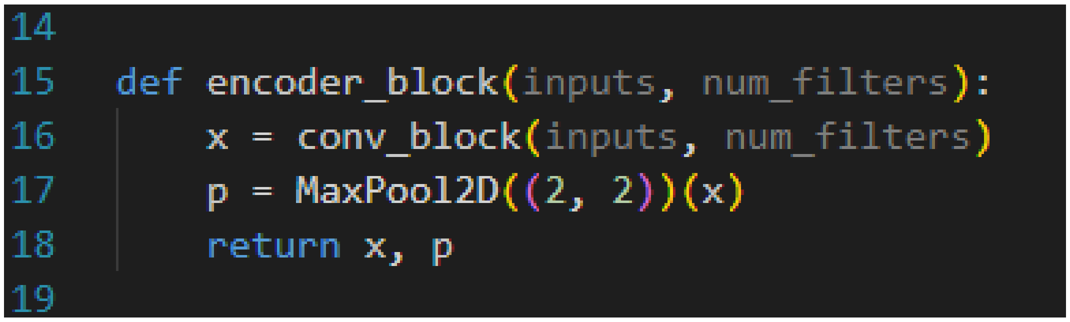
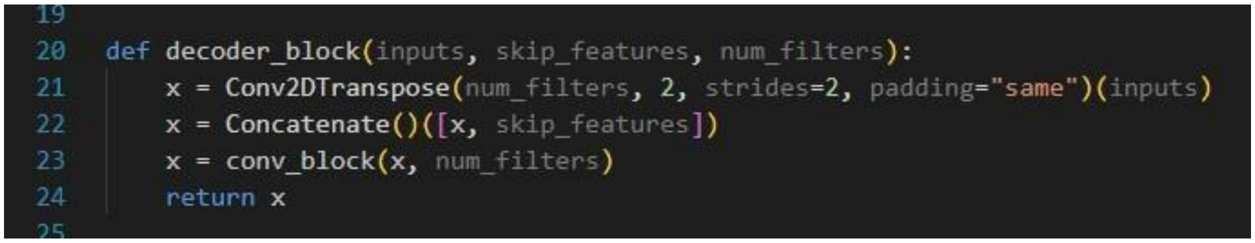

```python
def encoder_block(inputs, num_filters):
    x = conv_block(inputs, num_filters)
    p = MaxPool2D((2, 2))(x)
    return x, p
```

Figure 20. encoder_block function

The encoder_block function defines the encoder part of the U-Net. It uses the conv_block function to process the input inputs and extract meaningful features. The num_filters parameter controls the number of filters for the convolutional block, determining the complexity of learned features.

After applying the conv_block, the encoder block utilizes a max-pooling layer with a pool size of (2, 2). Max-pooling reduces the spatial resolution of the feature map while retaining the most salient features, allowing the model to focus on higher-level patterns in the data. The function returns two outputs: the processed feature map x, which is used in skip connections, and the down sampled feature map p, which is passed to the next encoder block for further processing.

After that the third function which is “decoder_block”:

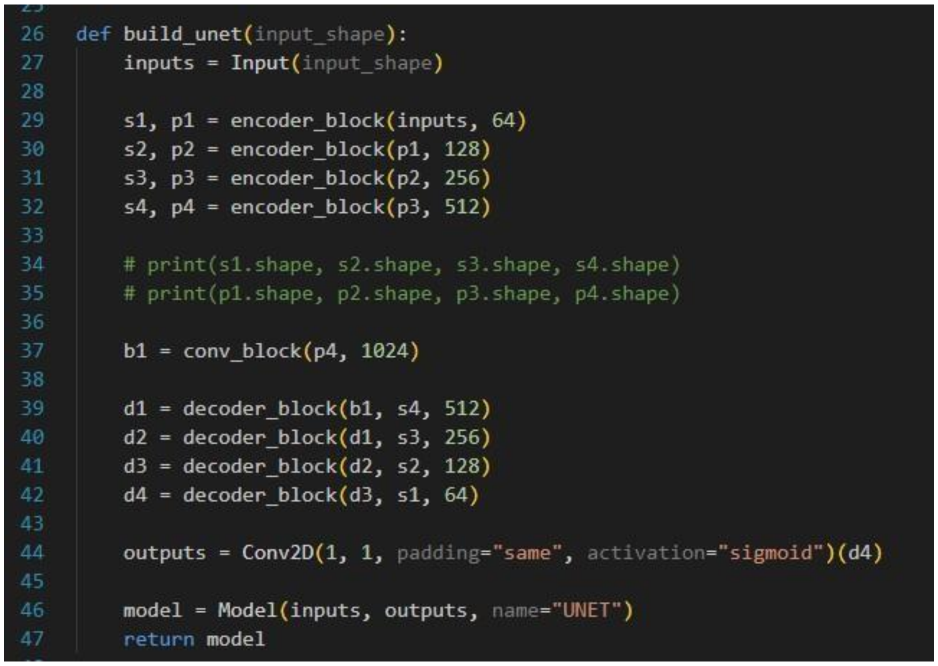


```python
def decoder_block(inputs, skip_features, num_filters):
    x = Conv2DTranspose(num_filters, 2, strides=2, padding="same")(inputs)
    x = Concatenate()([x, skip_features])
    x = conv_block(x, num_filters)
    return x
```

Figure 21. decoder_block function

The decoder_block function represents the U-Net's decoder. It up samples the input using 2D transposed convolutions (deconvolutions) and mixes it with the skip_features from the appropriate encoder block. The decoder's purpose is to retrieve the spatial information that were lost during the down sampling process. The skip connections are critical in merging low-level spatial data with high-level context, which results in correct segmentation.

After we have prepared all the required functions, we can now move on to building the complete U-Net architecture by defining the build_unet function:

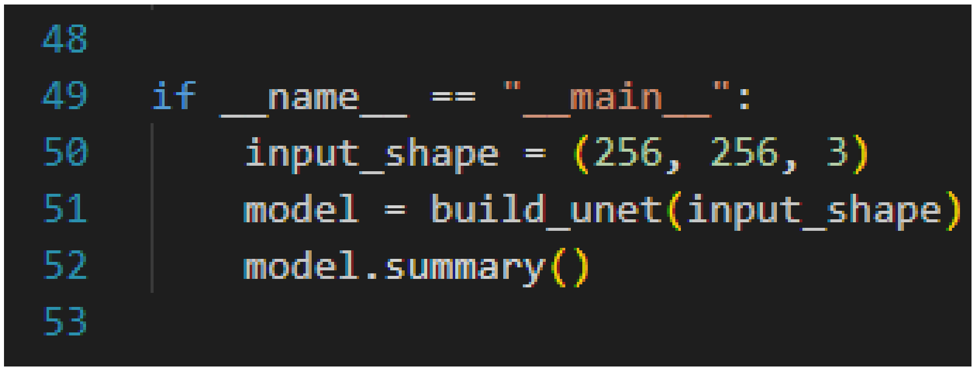


```python
def build_unet(input_shape):
    inputs = Input(input_shape)

    s1, p1 = encoder_block(inputs, 64)
    s2, p2 = encoder_block(p1, 128)
    s3, p3 = encoder_block(p2, 256)
    s4, p4 = encoder_block(p3, 512)

    # print(s1.shape, s2.shape, s3.shape, s4.shape)
    # print(p1.shape, p2.shape, p3.shape, p4.shape)

    b1 = conv_block(p4, 1024)

    d1 = decoder_block(b1, s4, 512)
    d2 = decoder_block(d1, s3, 256)
    d3 = decoder_block(d2, s2, 128)
    d4 = decoder_block(d3, s1, 64)

    outputs = Conv2D(1, 1, padding="same", activation="sigmoid")(d4)

    model = Model(inputs, outputs, name="UNET")
    return model
```

Figure 22. build_unet function

The build_unet function constructs the complete U-Net architecture by assembling the encoder, bottleneck, and decoder parts. It takes input_shape as an argument, representing the shape of the input images (height, width, channels). First, it defines the input layer using the Input module with the given input_shape. Next, it calls the encoder_block function multiple times to create a series of encoder blocks that progressively down sample the spatial dimensions and capture high-level features.The output feature maps from each encoder block are stored as s1, s2, s3, and s4, while the down sampled feature maps are stored as p1, p2, p3, and p4.

A bottleneck block (b1) is then created using the conv_block function. This block serves as the central representation of the U-Net and contains rich feature information. The decoder part of the U-Net is constructed using the decoder_block function. It performs up sampling with transposed convolutions and combines the up sampled features with the corresponding encoder features (s1, s2, s3, and s4) using skip connections. This helps in recovering the spatial detailsand combining the high-level context with low-level spatial information. Finally, a 1x1 convolutional layer with a sigmoid activation function is applied to generate the segmentation mask (outputs) for the input image.



```python

if __name__ == "__main__":
    input_shape = (256, 256, 3)
    model = build_unet(input_shape)
    model.summary()

```

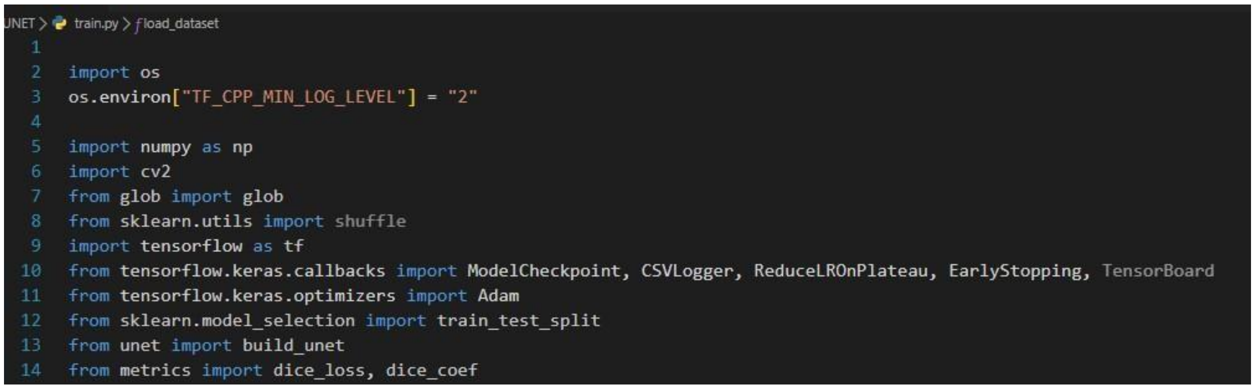
Figure 23. Main block function

After we build all the functions, including the build_unet function, we proceed to construct the main block of the code. In this section, we call the build_unet function to create the U-Net model, which is a key step in the process. U-Net is a sophisticated and widely used model specifically designed for semantic segmentation tasks, enabling it to accurately delineate objects within photos.

The build_unet function plays a central role in assembling the entire architecture. It seamlessly combines the encoder, bottleneck, and decoder components to create a comprehensive U-Net model. The encoder extracts meaningful features from the input images, the bottleneck captures high-dimensional representations, and the decoder efficiently recovers spatial details. The result is an end-to-end network with powerful segmentation capabilities.

After the successful creation of the U-Net model, we take the next step and print a summary of the architecture. This summary offers valuable insights into the organization and complexity of the model. Specifically, it reveals the number of parameters in each layer, which is essential in understanding the memory requirements and computational load of the network. With this succinct depiction of the U- Net model, we gain a clearer understanding of its structure, facilitating verification of its validity and uncovering potential optimization possibilities.

Then, we started with the training part coding starting first by importing the needed libraries



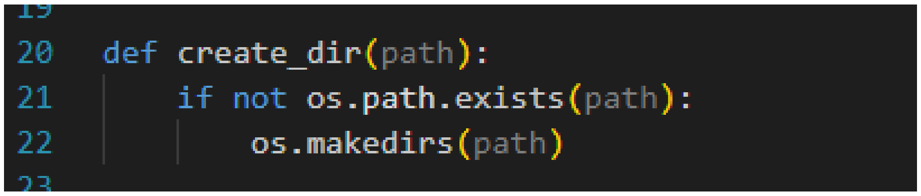
```

import os
os.environ["TF_CPP_MIN_LOG_LEVEL"] = "2"

import numpy as np
import cv2
from glob import glob
from sklearn.utils import shuffle
import tensorflow as tf
from tensorflow.keras.callbacks import ModelCheckpoint, CSVLogger, ReduceLROnPlateau, EarlyStopping, TensorBoard
from tensorflow.keras.optimizers import Adam
from sklearn.model_selection import train_test_split
from unet import build_unet
from metrics import dice_loss, dice_coef
```

Figure 24. Importing needed libraries for the training code part

The necessary libraries are imported in this block, and the TensorFlow logging level is set to suppress unnecessary information. NumPy, OpenCV (cv2), glob for file path handling, TensorFlow, and various TensorFlow Keras modules used for callbacks, optimizers, and metric functions are all important libraries. Furthermore, functions and classes from custom modules unet and metrics are imported. The U-Net architecture is implemented in the custom unet module, while metrics provides custom loss and evaluation functions.

After we imported the needed libraries, we stated the height and width needed for the picture in the training. Then we start writing the functions we need in the training part:

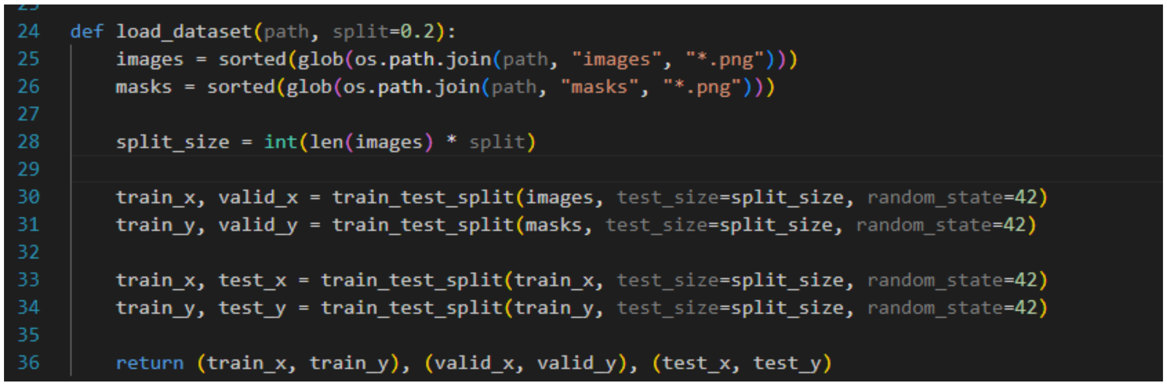
First function is “create_dir”:



```python
def create_dir(path):
    if not os.path.exists(path):
        os.makedirs(path)
```

Figure 25. create_dir function

Second function is “load_dataset”:

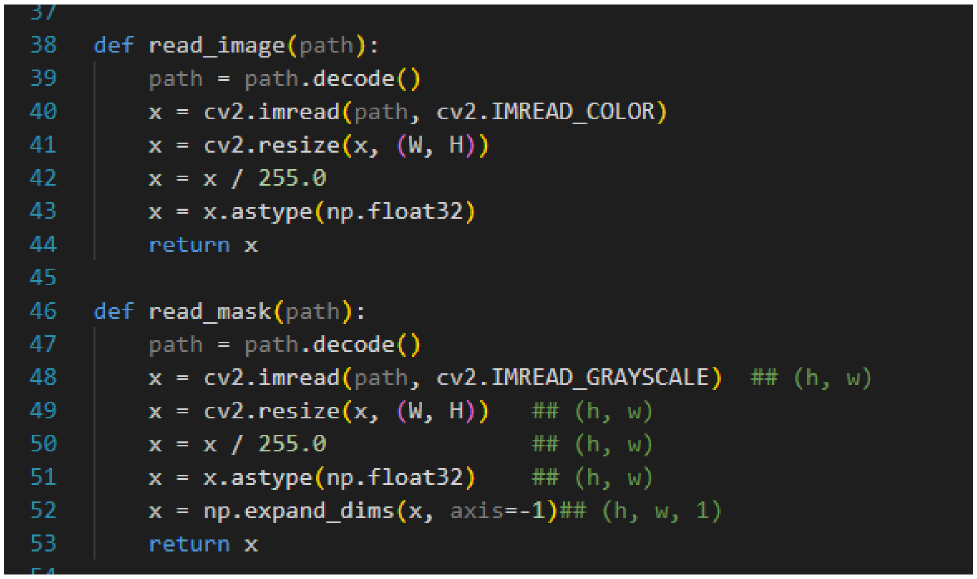


```python
def load_dataset(path, split=0.2):
    images = sorted(glob(os.path.join(path, "images", "*.png")))
    masks = sorted(glob(os.path.join(path, "masks", "*.png")))

    split_size = int(len(images) * split)

    train_x, valid_x = train_test_split(images, test_size=split_size, random_state=42)
    train_y, valid_y = train_test_split(masks, test_size=split_size, random_state=42)

    train_x, test_x = train_test_split(train_x, test_size=split_size, random_state=42)
    train_y, test_y = train_test_split(train_y, test_size=split_size, random_state=42)

    return (train_x, train_y), (valid_x, valid_y), (test_x, test_y)
```

Figure 26. load_dataset function

The load_dataset function loads the specified dataset images and masks. It anticipates that the dataset will contain separate directories named "images" and "masks" containing the corresponding PNG images and masks. The glob function is used to find and sort the file paths for images and masks. The dataset is then divided into training, validation, and testing sets using the sklearn train_test_split function.

Third and fourth function are “read_image” and “read_mask”:



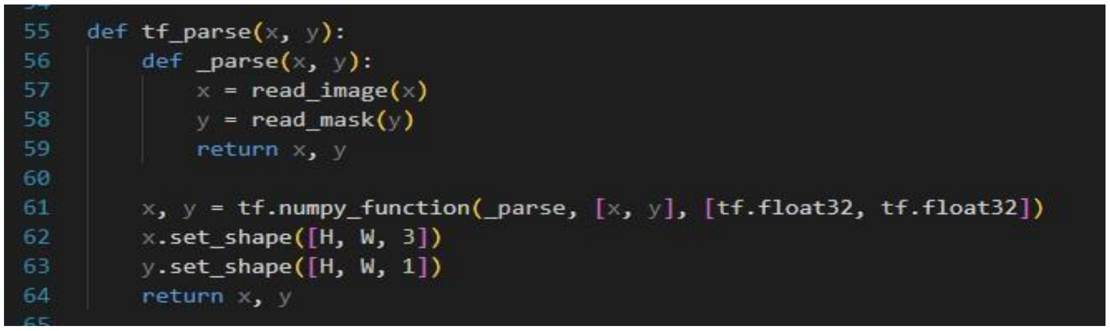
```python
def read_image(path):
    path = path.decode()
    x = cv2.imread(path, cv2.IMREAD_COLOR)
    x = cv2.resize(x, (W, H))
    x = x / 255.0
    x = x.astype(np.float32)
    return x

def read_mask(path):
    path = path.decode()
    x = cv2.imread(path, cv2.IMREAD_GRAYSCALE)  ## (h, w)
    x = cv2.resize(x, (W, H))   ## (h, w)
    x = x / 255.0               ## (h, w)
    x = x.astype(np.float32)    ## (h, w)
    x = np.expand_dims(x, axis=-1)## (h, w, 1)
    return x
```

Figure 27. read_image and read_mask functions

These functions, read_image and read_mask, are used to read and preprocess the images and masks from their file paths. The read_image function reads an image, resizes it to the specified height and width (H and W), normalizes the pixel values to the range [0, 1], and converts it to the data type np.float32.

The read_mask function reads a mask image (in grayscale), applies the same preprocessing steps as read_image, and additionally expands the mask's dimensions by adding a new axis at the end to convert it to a shape of (H, W, 1).

Fifth function is “tf_parse”:

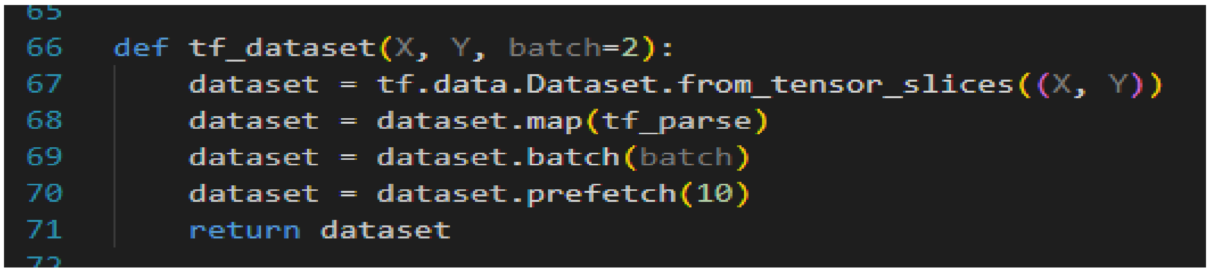


```python
def tf_parse(x, y):
    def _parse(x, y):
        x = read_image(x)
        y = read_mask(y)
        return x, y

    x, y = tf.numpy_function(_parse, [x, y], [tf.float32, tf.float32])
    x.set_shape([H, W, 3])
    y.set_shape([H, W, 1])
    return x, y
```

Figure 28. tf_parse function

The tf_parse function is implemented to process the dataset using TensorFlow operations. It accepts the file paths x and y as inputs. The read_image and read_mask functions are invoked within this function to preprocess the images and masks. TensorFlow's tf.numpy_function is used to convert Python functions to TensorFlow operations. The resulting x and y have the shapes [H, W, 3] and [H, W, 1], respectively.

Sixth function “tf_dataset”:



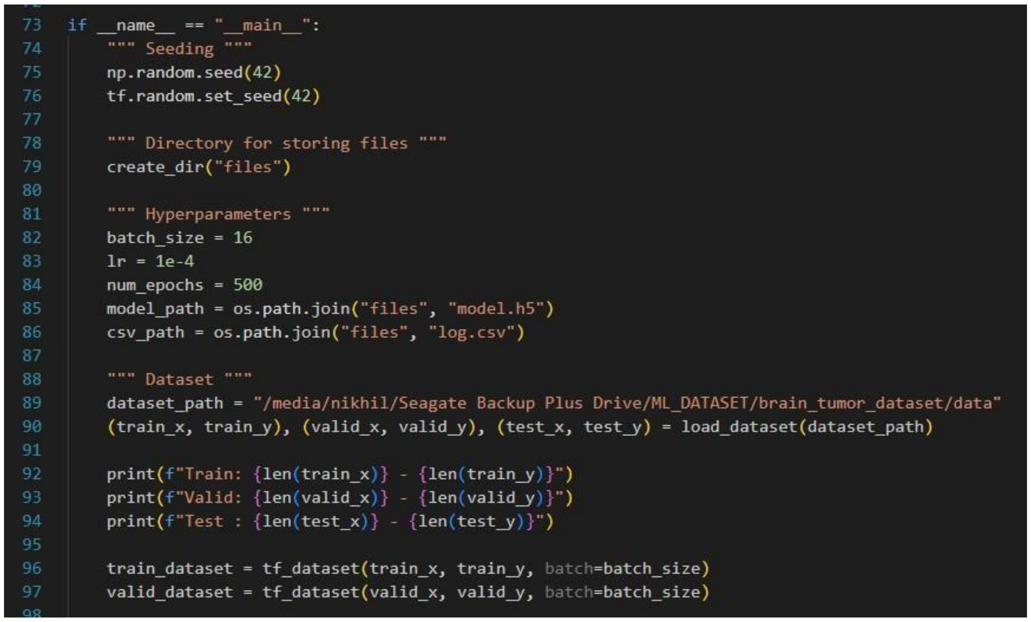
```python
def tf_dataset(X, Y, batch=2):
    dataset = tf.data.Dataset.from_tensor_slices((X, Y))
    dataset = dataset.map(tf_parse)
    dataset = dataset.batch(batch)
    dataset = dataset.prefetch(10)
    return dataset
```

Figure 29. tf_dataset function

TensorFlow datasets are created using the tf_dataset function from the image and mask file paths. It first uses tf.data to convert the file paths into a TensorFlow dataset.Dataset.from_tensor_slices. The tf_parse function is then applied to each element of the dataset using map, preprocessing the images and masks.

Following preprocessing, the dataset is batched to the specified batch size and pre-fetched to improve training performance. The function returns the TensorFlow dataset that was created.

Now the main block:



```python
if __name__ == "__main__":
    """ Seeding """
    np.random.seed(42)
    tf.random.set_seed(42)

    """ Directory for storing files """
    create_dir("files")

    """ Hyperparameters """
    batch_size = 16
    lr = 1e-4
    num_epochs = 500
    model_path = os.path.join("files", "model.h5")
    csv_path = os.path.join("files", "log.csv")

    """ Dataset """
    dataset_path = "/media/nikhil/Seagate Backup Plus Drive/ML_DATASET/brain_tumor_dataset/data"
    (train_x, train_y), (valid_x, valid_y), (test_x, test_y) = load_dataset(dataset_path)

    print(f"Train: {len(train_x)} - {len(train_y)}")
    print(f"Valid: {len(valid_x)} - {len(valid_y)}")
    print(f"Test : {len(test_x)} - {len(test_y)}")

    train_dataset = tf_dataset(train_x, train_y, batch=batch_size)
    valid_dataset = tf_dataset(valid_x, valid_y, batch=batch_size)
```

Figure 30. Main block

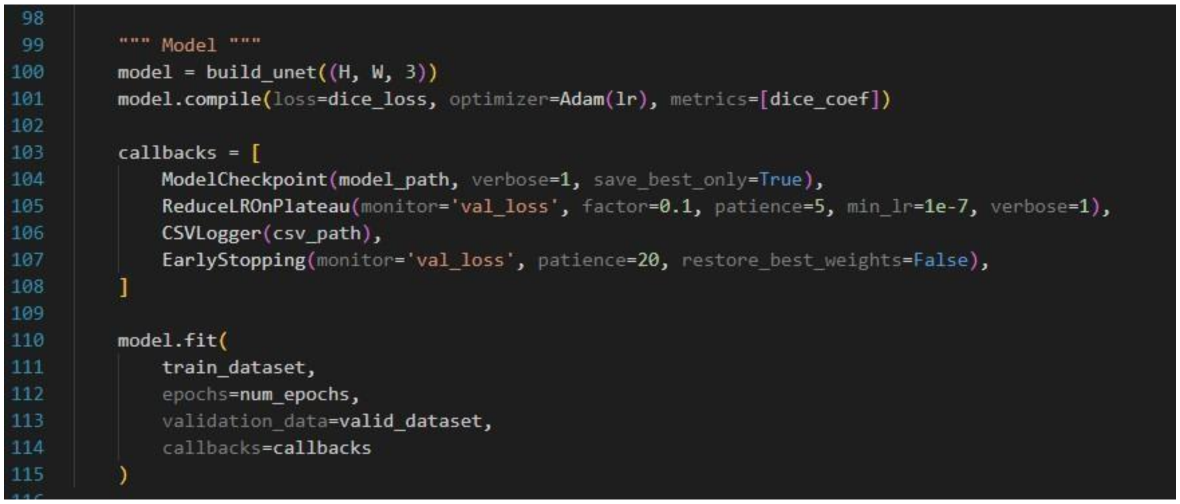
To ensure reproducibility in the experiments, the code in the main block begins by setting random seeds with np.random.seed(42) and tf.random.set_seed(42).

Then, using the create_dir function, it creates a directory named "files" to store various files generated during the training process, such as model weights and training logs.

Batch_size, lr (learning rate), and num_epochs (number of training epochs) are defined as hyperparameters. Model_path and csv_path are also used to specify file paths for saving the trained model and training logs.

The dataset path (dataset_path) is set to the directory that contains the image and mask data. To load and split the dataset into training, validation, and testing sets, the load_dataset function is used. The sizes of the datasets are printed to the console.

Then we added the last part of the main block which is the model compilation and training part:



```
    """ Model """
    model = build_unet((H, W, 3))
    model.compile(loss=dice_loss, optimizer=Adam(lr), metrics=[dice_coef])

    callbacks = [
        ModelCheckpoint(model_path, verbose=1, save_best_only=True),
        ReduceLROnPlateau(monitor='val_loss', factor=0.1, patience=5, min_lr=1e-7, verbose=1),
        CSVLogger(csv_path),
        EarlyStopping(monitor='val_loss', patience=20, restore_best_weights=False),
    ]

    model.fit(
        train_dataset,
        epochs=num_epochs,
        validation_data=valid_dataset,
        callbacks=callbacks
    )
```

Figure 31. Model compilation and training part

In this part of the code, we handle model compilation and training. Firstly, the U-Net model is created using the build_unet function, specifying the input shape (H, W, 3) for the height, width, and color channels of the input images.

Next, the model is compiled using the custom dice_loss as the loss function and the Adam optimizer with the specified learning rate (lr). Additionally, we include dice_coef as a metric to monitor during training.

A list of callbacks is defined to customize the training process. These callbacks serve different purposes:

ModelCheckpoint: It saves the model's weights to model_path when an improvement in validation loss is observed (verbose=1).

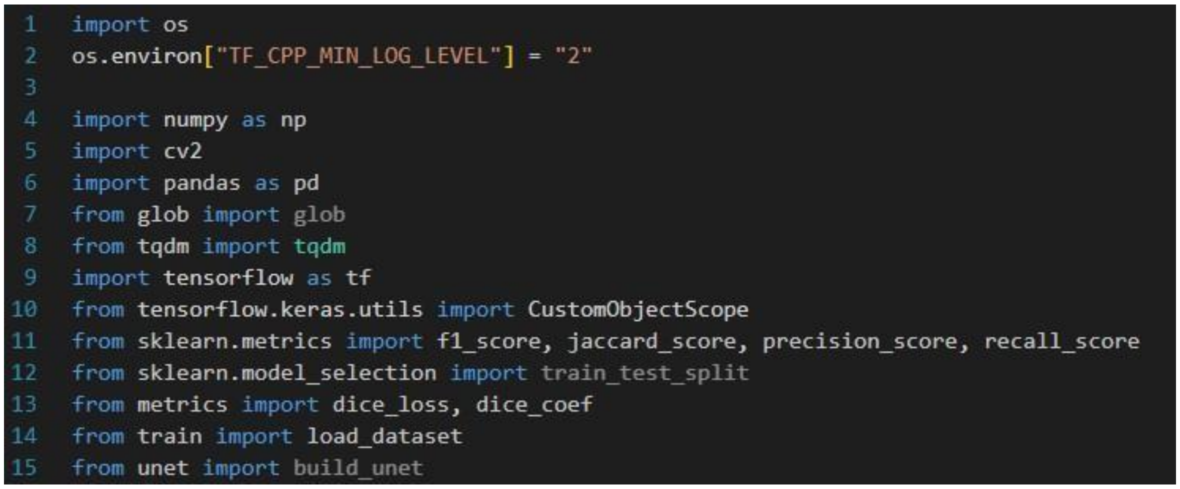
ReduceLROnPlateau: This callback reduces the learning rate (lr) by a factor of 0.1 when the validation loss plateaus for patience number of epochs. It helps fine-tune the training process.

CSVLogger: This callback logs the training progress to a CSV file specified by csv_path.

EarlyStopping: This callback stops the training if no improvement in validation loss is observed for patience number of epochs.

Finally, the model.fit function is called to train the U-Net model. It uses the train_dataset for training data and valid_dataset for validation data. The model is trained for num_epochs iterations, and the defined callbacks are used to monitor and control the training process. The training progress, including loss and metrics, is displayed in the console.

Now we will show the employment of the testing code starting by importing the necessities libraries,



```python
import os
os.environ["TF_CPP_MIN_LOG_LEVEL"] = "2"

import numpy as np
import cv2
import pandas as pd
from glob import glob
from tqdm import tqdm
import tensorflow as tf
from tensorflow.keras.utils import CustomObjectScope
from sklearn.metrics import f1_score, jaccard_score, precision_score, recall_score
from sklearn.model_selection import train_test_split
from metrics import dice_loss, dice_coef
from train import load_dataset
from unet import build_unet
```

Figure 32. Importing needed libraries for test code

This block imports required libraries, defines global parameters, and imports specific functions and classes from custom modules. The following are the main points:

The script modifies the TensorFlow logging level to suppress extraneous data (os.environ["TF_CPP_MIN_LOG_LEVEL"] = "2").

For various functionalities, libraries such as NumPy, OpenCV (cv2), Pandas (pd), and TensorFlow(tf) are imported.

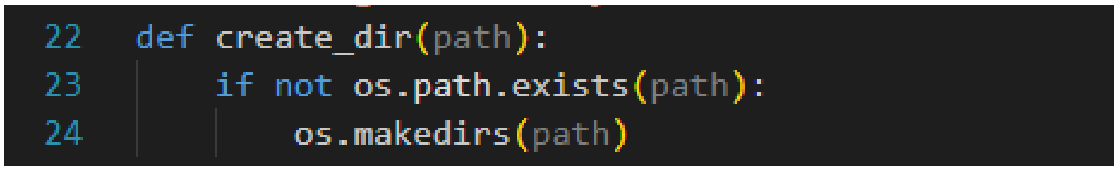
1. The H and W global parameters represent the desired image height and width for testing.
2. The script imports custom functions dice_loss and dice_coef from the metrics.py file, which wereused for loss calculation and evaluation during model training.
3. The load_dataset function from the training file (train.py) is also imported because it is

used to loadthe test dataset.

4. Lastly, the build_unet function is imported from the unet.py file, which is used to create the U- Netmodel.

Then we selected the sizes of height and weight, after that we started writing the functions of the testing.

Starting by the first function which is “create_dir”:

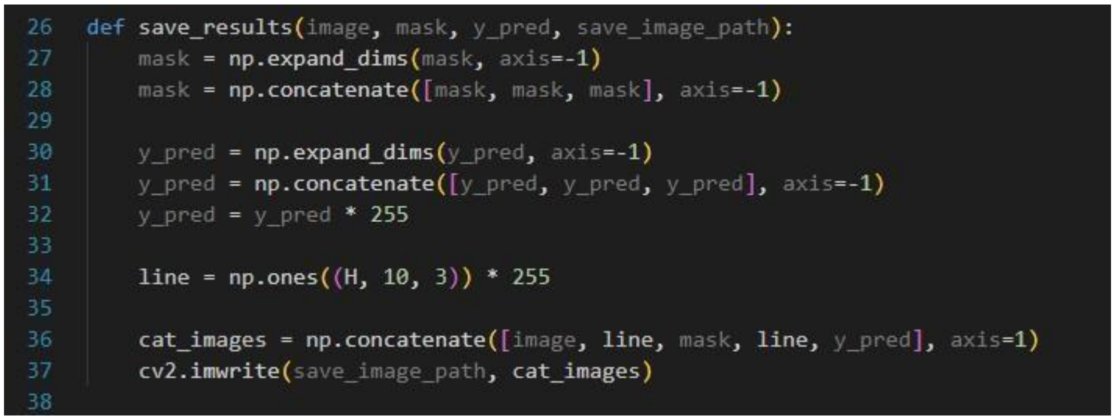


```python
def create_dir(path):
    if not os.path.exists(path):
        os.makedirs(path)
```

Figure 33. create_dir function

If a directory does not already exist, "create_dir" is defined to create it. This function is used to create a directory called "results" to keep the test results.

Then, the second function which is “save_results”:



```python
def save_results(image, mask, y_pred, save_image_path):
    mask = np.expand_dims(mask, axis=-1)
    mask = np.concatenate([mask, mask, mask], axis=-1)

    y_pred = np.expand_dims(y_pred, axis=-1)
    y_pred = np.concatenate([y_pred, y_pred, y_pred], axis=-1)
    y_pred = y_pred * 255

    line = np.ones((H, 10, 3)) * 255

    cat_images = np.concatenate([image, line, mask, line, y_pred], axis=1)
    cv2.imwrite(save_image_path, cat_images)

```

Figure 34. save_results function

The test results are visualized and saved using the save_results method. It requires four arguments: image: This is the original image.

mask: The ground truth mask.

y_pred: The anticipated mask.

save_image_path: The path to the file where the generated visualization will be saved.

The function is responsible for the following tasks:

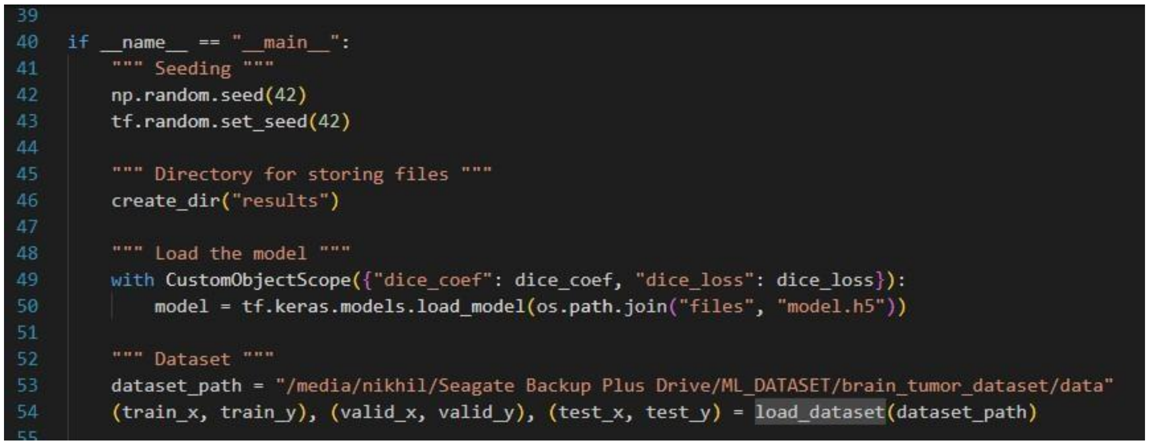
To concatenate the mask and y_pred, an extra channel (axis=-1) is added to the mask and y_pred.

Because it is a binary mask, the y_pred is multiplied by 255 to convert it to pixel values in the range [0, 255].

For visual separation, a white line is applied between the original images, ground truth mask, and forecasted mask.

Using OpenCV's cv2.imwrite function, the concatenated images are stored in the supplied file directory.

Now for the main block in the testing code,



```
if __name__ == "__main__":
    """ Seeding """
    np.random.seed(42)
    tf.random.set_seed(42)

    """ Directory for storing files """
    create_dir("results")

    """ Load the model """
    with CustomObjectScope({"dice_coef": dice_coef, "dice_loss": dice_loss}):
        model = tf.keras.models.load_model(os.path.join("files", "model.h5"))

    """ Dataset """
    dataset_path = "/media/nikhil/Seagate Backup Plus Drive/ML_DATASET/brain_tumor_dataset/data"
    (train_x, train_y), (valid_x, valid_y), (test_x, test_y) = load_dataset(dataset_path)
```

Figure 35. Main block

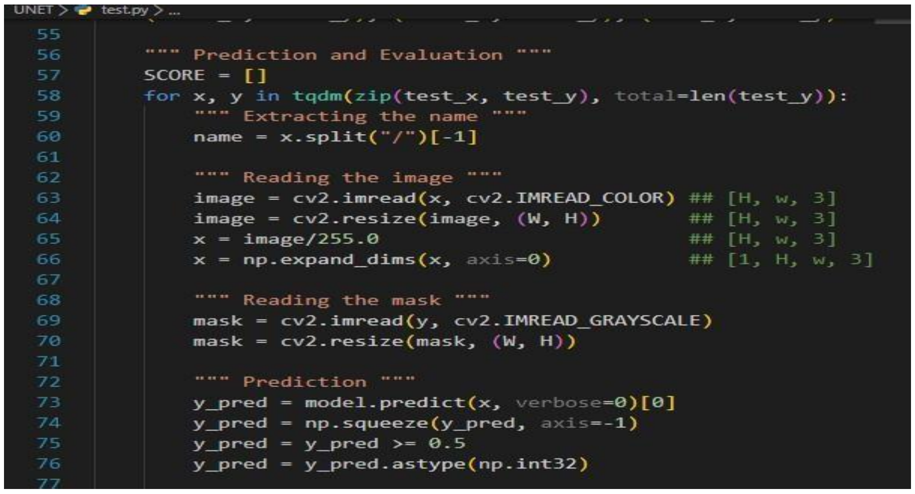
The main block of code in this section begins with the seeding of random number generators to ensure reproducibility.

The create_dir function is then invoked to create a directory named "results" to hold the test results.

TensorFlow Keras' load_model method is used to load the trained U-Net model. When loading the model, a special object scope is utilized to ensure that the custom metrics (dice_coef and dice_loss) are recognized. This ensures that the model is built using the same metrics that were used during training.

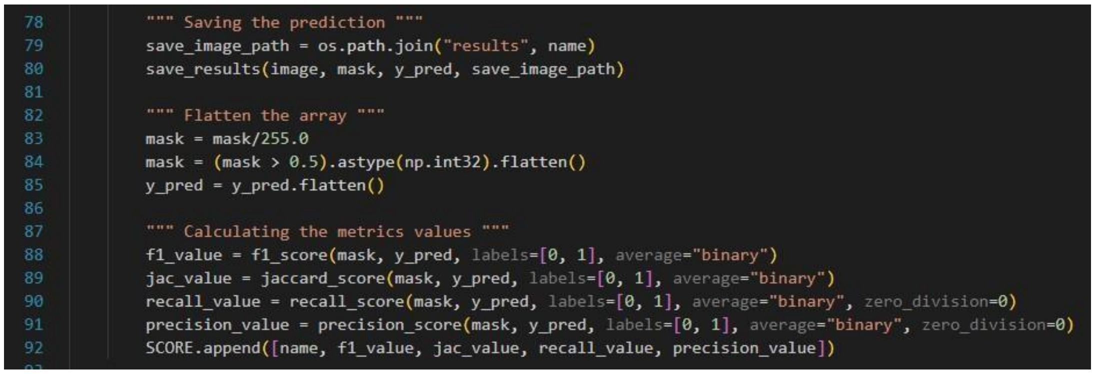
The load_dataset function, which was imported from the training file, is used to load the test dataset.

The function returns the test image and mask file directories when the dataset path is supplied.



```
UNET > test.py > ...

    """ Prediction and Evaluation """
    SCORE = []
    for x, y in tqdm(zip(test_x, test_y), total=len(test_y)):
        """ Extracting the name """
        name = x.split("/")[-1]

        """ Reading the image """
        image = cv2.imread(x, cv2.IMREAD_COLOR) ## [H, w, 3]
        image = cv2.resize(image, (W, H))       ## [H, w, 3]
        x = image/255.0                         ## [H, w, 3]
        x = np.expand_dims(x, axis=0)           ## [1, H, w, 3]

        """ Reading the mask """
        mask = cv2.imread(y, cv2.IMREAD_GRAYSCALE)
        mask = cv2.resize(mask, (W, H))

        """ Prediction """
        y_pred = model.predict(x, verbose=0)[0]
        y_pred = np.squeeze(y_pred, axis=-1)
        y_pred = y_pred >= 0.5
        y_pred = y_pred.astype(np.int32)
```

Figure 36. Predicting and evaluating code



```
        """ Saving the prediction """
        save_image_path = os.path.join("results", name)
        save_results(image, mask, y_pred, save_image_path)

        """ Flatten the array """
        mask = mask/255.0
        mask = (mask > 0.5).astype(np.int32).flatten()
        y_pred = y_pred.flatten()

        """ Calculating the metrics values """
        f1_value = f1_score(mask, y_pred, labels=[0, 1], average="binary")
        jac_value = jaccard_score(mask, y_pred, labels=[0, 1], average="binary")
        recall_value = recall_score(mask, y_pred, labels=[0, 1], average="binary", zero_division=0)
        precision_value = precision_score(mask, y_pred, labels=[0, 1], average="binary", zero_division=0)
        SCORE.append([name, f1_value, jac_value, recall_value, precision_value])
```

Figure 37. Continue of the predicting code

This section of code is at the heart of the testing procedure. It uses a loop to go over the test picture and mask file paths (for x, y in tqdm(zip(test_x, test_y), total=len(test_y)):). The following processes are carried out for each test image:

By splitting the file path and taking the last component, the name of the test image is extracted.

OpenCV (cv2.imread) is used to read the image and mask and resize it to the appropriate dimensions (H, W).

Preprocessing the image involves dividing pixel values by 255 to bring them into the range [0, 1].

Then, to produce a batch of size one, a new dimension is added (np.expand_dims(x, axis=0)).

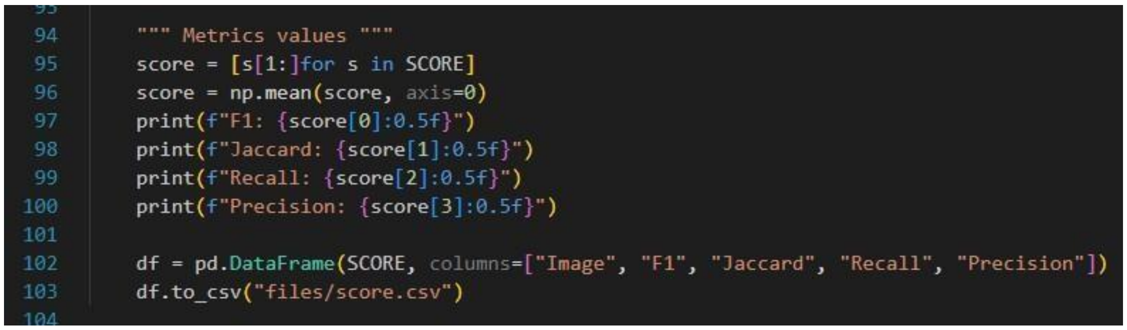
On the preprocessed test image, the U-Net model is utilised to produce a prediction. The output of the model is used to anticipate the mask.

The anticipated mask's values are threshold by comparing them to 0.5 and converting them to binary values (0 or 1). It's also flattened to look like the ground truth mask.

The save_results method is used to save the original image, ground truth mask, and forecasted mask as a visualization side by side.

To make the ground truth mask equivalent to the anticipated mask, it is flattened and threshold. The metrics (f1_score, jaccard_score, recall_score, precision_score) between the ground truth mask and predicted mask are computed, and the results are appended to the SCORE list.

After all that, the metrics values are computed based on the accumulated results from the test set. The following code block calculates the mean of various evaluation metrics to provide an overall assessment of the U-Net model's performance on the test data.



```python
    """ Metrics values """
    score = [s[1:]for s in SCORE]
    score = np.mean(score, axis=0)
    print(f"F1: {score[0]:0.5f}")
    print(f"Jaccard: {score[1]:0.5f}")
    print(f"Recall: {score[2]:0.5f}")
    print(f"Precision: {score[3]:0.5f}")

    df = pd.DataFrame(SCORE, columns=["Image", "F1", "Jaccard", "Recall", "Precision"])
    df.to_csv("files/score.csv")
```

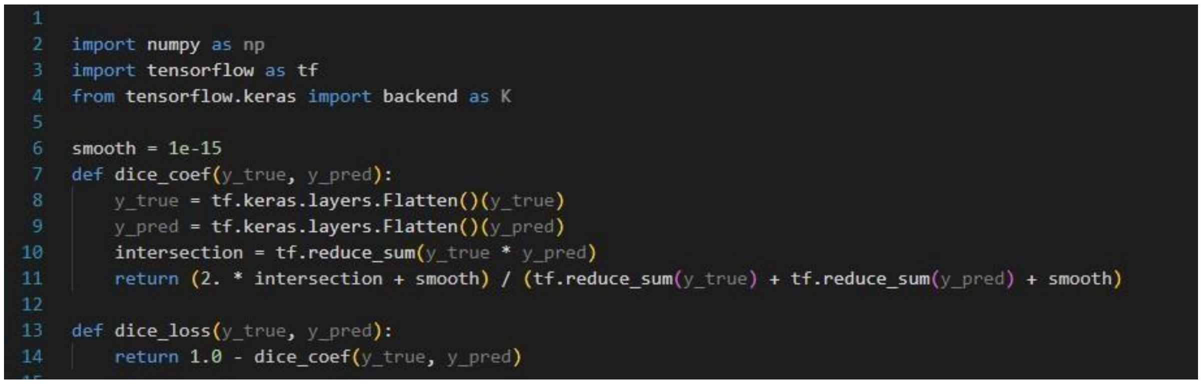
Figure 38. Metric values

In this block, the `score` list is processed to extract the evaluation metrics values (F1, Jaccard, Recall, Precision) from each test image. The `np.mean` function is then used to compute the mean of these metrics across all test images. The mean values represent an aggregated performance measure for the entire test dataset.

Finally, the average F1, Jaccard, Recall, and Precision scores are printed to the console for quick reference. Additionally, the individual metric values for each test image are stored in a Pandas DataFrame and saved to a CSV file named "score.csv" for further analysis or comparison.

Finally, with this, we completed the testing code for the U-Net model. The code loads the trained model, predicts the masks on the test dataset, evaluates the model's performance using different metrics, and saves the results and metrics for further analysis. The test script allows us to assess how well the model generalizes to unseen data and how effectively it can segment brain tumor regions from MRI scans.

In the training and testing code, the custom metrics functions are imported from the metrics.py file, which contains the following functions:



```

import numpy as np
import tensorflow as tf
from tensorflow.keras import backend as K

smooth = 1e-15
def dice_coef(y_true, y_pred):
    y_true = tf.keras.layers.Flatten()(y_true)
    y_pred = tf.keras.layers.Flatten()(y_pred)
    intersection = tf.reduce_sum(y_true * y_pred)
    return (2. * intersection + smooth) / (tf.reduce_sum(y_true) + tf.reduce_sum(y_pred) + smooth)

def dice_loss(y_true, y_pred):
    return 1.0 - dice_coef(y_true, y_pred)
```

Figure 39. Metrices functions

dice_coef(y_true, y_pred): This function calculates the Dice coefficient, which is a common metric used to evaluate the performance of image segmentation models. The Dice coefficient measures the similarity between the predicted mask (y_pred) and the ground truth mask (y_true). It is defined as the ratio of twice the intersection of y_true and y_pred to the sum of the total pixels in y_true and y_pred.

The smooth term is added to avoid division by zero in cases where the denominator is zero. dice_loss(y_true, y_pred): This function defines the Dice loss, which is the complement of the Dice coefficient. The Dice loss is used as a loss function during model training. It calculates the difference between 1 and the Dice coefficient to minimize the dissimilarity between the predicted and ground truth masks.

The U-Net's coding is complete; now we will execute the code and train a model. Using the dataset we generated, the U-Net architecture will learn to segregate brain tumor locations from MRI scans throughout the training phase. Custom loss functions will be used to optimize the model, and custom metrics will be used to assess it.

Run the code to see how the model performs on the training and validation datasets. After training, we will apply the trained model to the test dataset and assess its segmentation accuracy using several metrics. This will provide us with important information on how effectively the model generalizes to new data and its capacity to precisely identify brain tumor areas.

We will have a fully trained U-Net model capable of accurate brain tumor segmentation at the conclusion of the training and testing procedure.

3.3.2. Training steps

For the training steps we will start talking about the preprocessing methods we utilized in U-net which are only image-resizing and normalization.

a) Preprocessing:

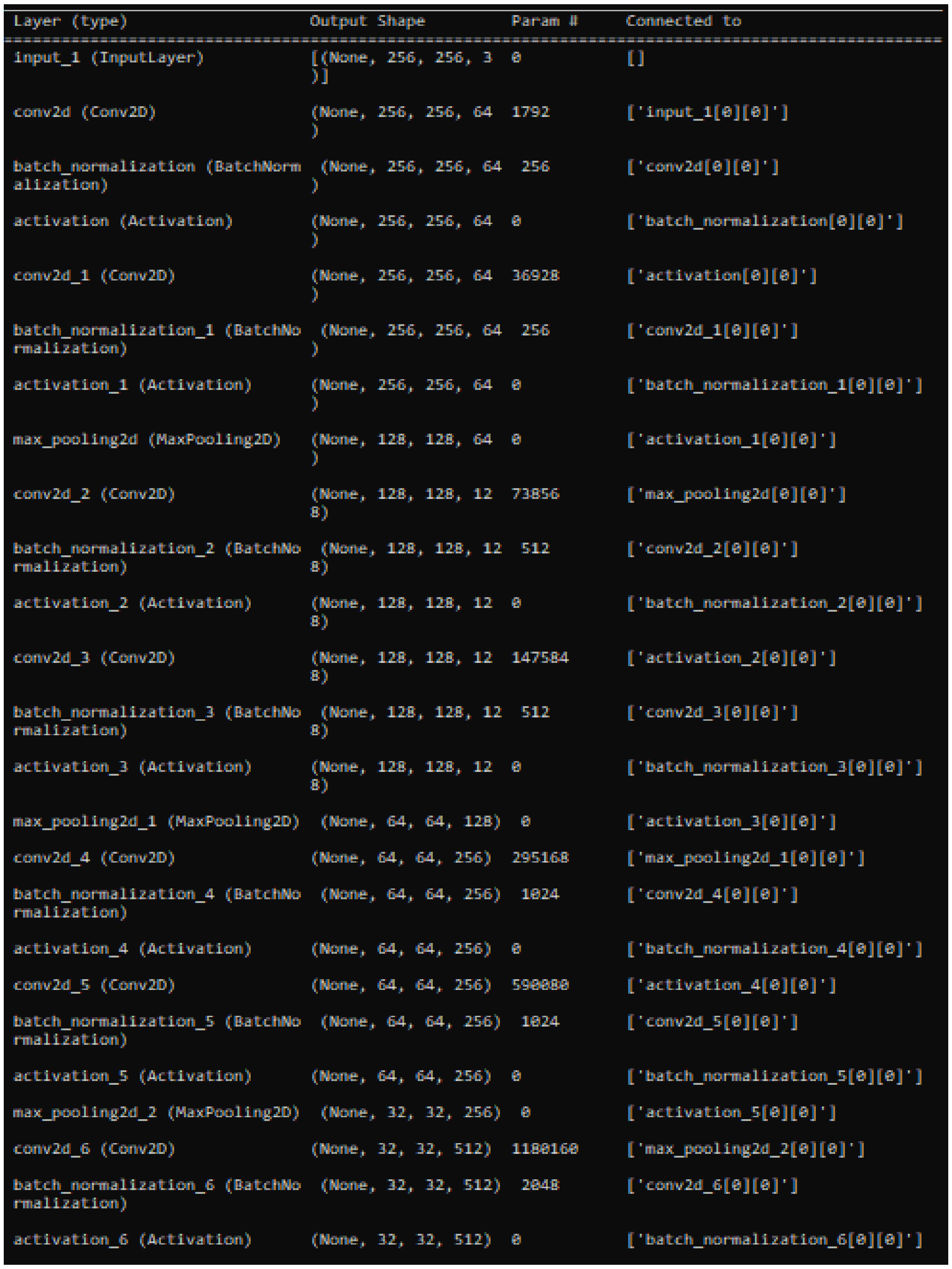
i. Image Resizing:

Using OpenCV's cv2.resize function, the original brain MRI images were downsized to a set sizethat we selected. Resizing photos to a constant size is a standard preprocessing procedure that ensuresall images have the same dimensions, which is necessary for effective batch processing during training and testing.

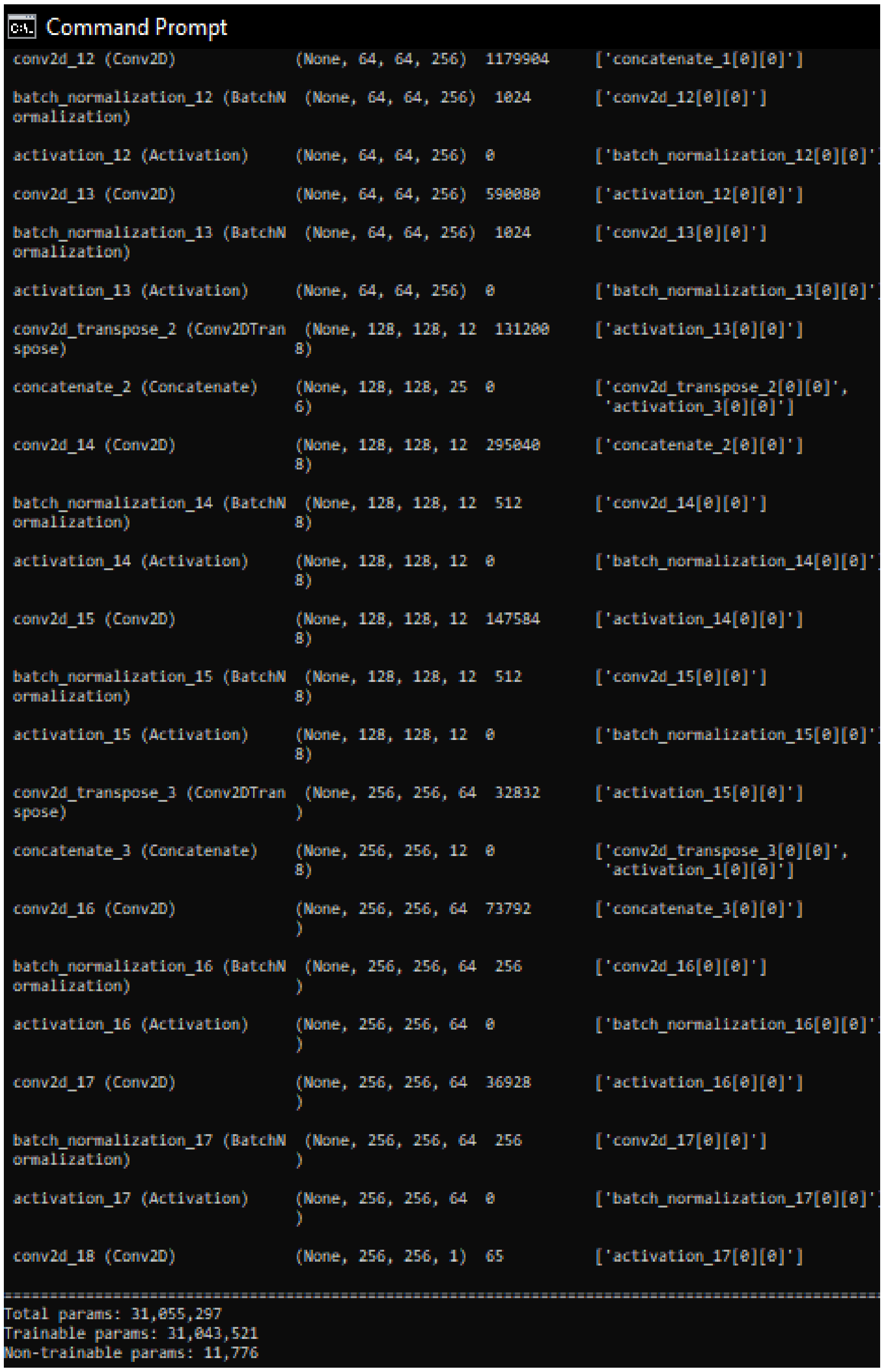
ii. Normalization:

The resized pictures' pixel values were normalized to fall within the range [0, 1]. This was accomplished by multiplying the pixel values by 255.0. Normalization helps to scale pixel intensitiesand prepare the data for neural network training.

b) Model's layers:



```
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================================
input_1 (InputLayer)            [(None, 256, 256, 3  0           []
                                )]

conv2d (Conv2D)                 (None, 256, 256, 64  1792        ['input_1[0][0]']
                                )

batch_normalization (BatchNorm  (None, 256, 256, 64  256         ['conv2d[0][0]']
alization)                      )

activation (Activation)         (None, 256, 256, 64  0           ['batch_normalization[0][0]']
                                )

conv2d_1 (Conv2D)               (None, 256, 256, 64  36928       ['activation[0][0]']
                                )

batch_normalization_1 (BatchNo  (None, 256, 256, 64  256         ['conv2d_1[0][0]']
rmalization)                    )

activation_1 (Activation)       (None, 256, 256, 64  0           ['batch_normalization_1[0][0]']
                                )

max_pooling2d (MaxPooling2D)    (None, 128, 128, 64  0           ['activation_1[0][0]']
                                )

conv2d_2 (Conv2D)               (None, 128, 128, 12  73856       ['max_pooling2d[0][0]']
                                8)

batch_normalization_2 (BatchNo  (None, 128, 128, 12  512         ['conv2d_2[0][0]']
rmalization)                    8)

activation_2 (Activation)       (None, 128, 128, 12  0           ['batch_normalization_2[0][0]']
                                8)

conv2d_3 (Conv2D)               (None, 128, 128, 12  147584      ['activation_2[0][0]']
                                8)

batch_normalization_3 (BatchNo  (None, 128, 128, 12  512         ['conv2d_3[0][0]']
rmalization)                    8)

activation_3 (Activation)       (None, 128, 128, 12  0           ['batch_normalization_3[0][0]']
                                8)

max_pooling2d_1 (MaxPooling2D)  (None, 64, 64, 128)  0           ['activation_3[0][0]']

conv2d_4 (Conv2D)               (None, 64, 64, 256)  295168      ['max_pooling2d_1[0][0]']

batch_normalization_4 (BatchNo  (None, 64, 64, 256)  1024        ['conv2d_4[0][0]']
rmalization)

activation_4 (Activation)       (None, 64, 64, 256)  0           ['batch_normalization_4[0][0]']

conv2d_5 (Conv2D)               (None, 64, 64, 256)  590080      ['activation_4[0][0]']

batch_normalization_5 (BatchNo  (None, 64, 64, 256)  1024        ['conv2d_5[0][0]']
rmalization)

activation_5 (Activation)       (None, 64, 64, 256)  0           ['batch_normalization_5[0][0]']

max_pooling2d_2 (MaxPooling2D)  (None, 32, 32, 256)  0           ['activation_5[0][0]']

conv2d_6 (Conv2D)               (None, 32, 32, 512)  1180160     ['max_pooling2d_2[0][0]']

batch_normalization_6 (BatchNo  (None, 32, 32, 512)  2048        ['conv2d_6[0][0]']
rmalization)

activation_6 (Activation)       (None, 32, 32, 512)  0           ['batch_normalization_6[0][0]']
```



```
Command Prompt
conv2d_12 (Conv2D)             (None, 64, 64, 256)  1179904     ['concatenate_1[0][0]']

batch_normalization_12 (BatchN  (None, 64, 64, 256)  1024       ['conv2d_12[0][0]']
ormalization)

activation_12 (Activation)     (None, 64, 64, 256)  0          ['batch_normalization_12[0][0]'

conv2d_13 (Conv2D)             (None, 64, 64, 256)  590080     ['activation_12[0][0]']

batch_normalization_13 (BatchN  (None, 64, 64, 256)  1024       ['conv2d_13[0][0]']
ormalization)

activation_13 (Activation)     (None, 64, 64, 256)  0          ['batch_normalization_13[0][0]'

conv2d_transpose_2 (Conv2DTran  (None, 128, 128, 12  131200     ['activation_13[0][0]']
spose)                         8)

concatenate_2 (Concatenate)    (None, 128, 128, 25  0          ['conv2d_transpose_2[0][0]',
                               6)                                'activation_3[0][0]']

conv2d_14 (Conv2D)             (None, 128, 128, 12  295040     ['concatenate_2[0][0]']
                               8)

batch_normalization_14 (BatchN  (None, 128, 128, 12  512        ['conv2d_14[0][0]']
ormalization)                  8)

activation_14 (Activation)     (None, 128, 128, 12  0          ['batch_normalization_14[0][0]'
                               8)

conv2d_15 (Conv2D)             (None, 128, 128, 12  147584     ['activation_14[0][0]']
                               8)

batch_normalization_15 (BatchN  (None, 128, 128, 12  512        ['conv2d_15[0][0]']
ormalization)                  8)

activation_15 (Activation)     (None, 128, 128, 12  0          ['batch_normalization_15[0][0]'
                               8)

conv2d_transpose_3 (Conv2DTran  (None, 256, 256, 64  32832      ['activation_15[0][0]']
spose)                         )

concatenate_3 (Concatenate)    (None, 256, 256, 12  0          ['conv2d_transpose_3[0][0]',
                               8)                                'activation_1[0][0]']

conv2d_16 (Conv2D)             (None, 256, 256, 64  73792      ['concatenate_3[0][0]']
                               )

batch_normalization_16 (BatchN  (None, 256, 256, 64  256        ['conv2d_16[0][0]']
ormalization)                  )

activation_16 (Activation)     (None, 256, 256, 64  0          ['batch_normalization_16[0][0]'
                               )

conv2d_17 (Conv2D)             (None, 256, 256, 64  36928      ['activation_16[0][0]']
                               )

batch_normalization_17 (BatchN  (None, 256, 256, 64  256        ['conv2d_17[0][0]']
ormalization)                  )

activation_17 (Activation)     (None, 256, 256, 64  0          ['batch_normalization_17[0][0]'
                               )

conv2d_18 (Conv2D)             (None, 256, 256, 1)  65         ['activation_17[0][0]']

==================================================================================================
Total params: 31,055,297
Trainable params: 31,043,521
Non-trainable params: 11,776
```

Figure 40. Continue of the summary of model's layers

We acquired the model summary after executing the U-Net code, which gives useful insights into the architecture's complexity and the amount of trainable parameters. The U-Net model is made up of several layers, including convolutional layers, pooling layers, and transpose convolutional layers, which serve as the encoder, bottleneck, and decoder components, respectively. Each layer helps with feature extraction and spatial transformations, allowing the model to learn hierarchical representations for accurate semantic segmentation.

The model summary shows how many layers are employed in the U-Net architecture, as well as the input and output shapes at each layer. It also displays the number of trainable parameters, which represents the model's ability to learn from the provided dataset. In our example, the model has a whopping 31,043,521 trainable parameters, indicating a large network capable of capturing subtle patterns and nuances in brain MRI scans. The huge number of parameters demonstrates the U-Net model's expressive capability, allowing it to modify and specialize its features to handle the difficult challenge of brain tumor segmentation.

Understanding the model summary assists in confirming the validity of the architecture, fine- tuning hyper parameters if necessary, and getting insights into the model's memory needs. It demonstrates the U-Net's intricacy, which, together with a huge number of trainable parameters, implies its potential for high accuracy in segmenting brain tumor areas. However, it also emphasizes the need of having sufficient computational resources for efficient training and inference. The model summary is an important diagnostic tool for the U-Net architecture, leading us through the process of constructing and fine-tuning the model to reach the best outcomes possible.

c) Training and saving the model:

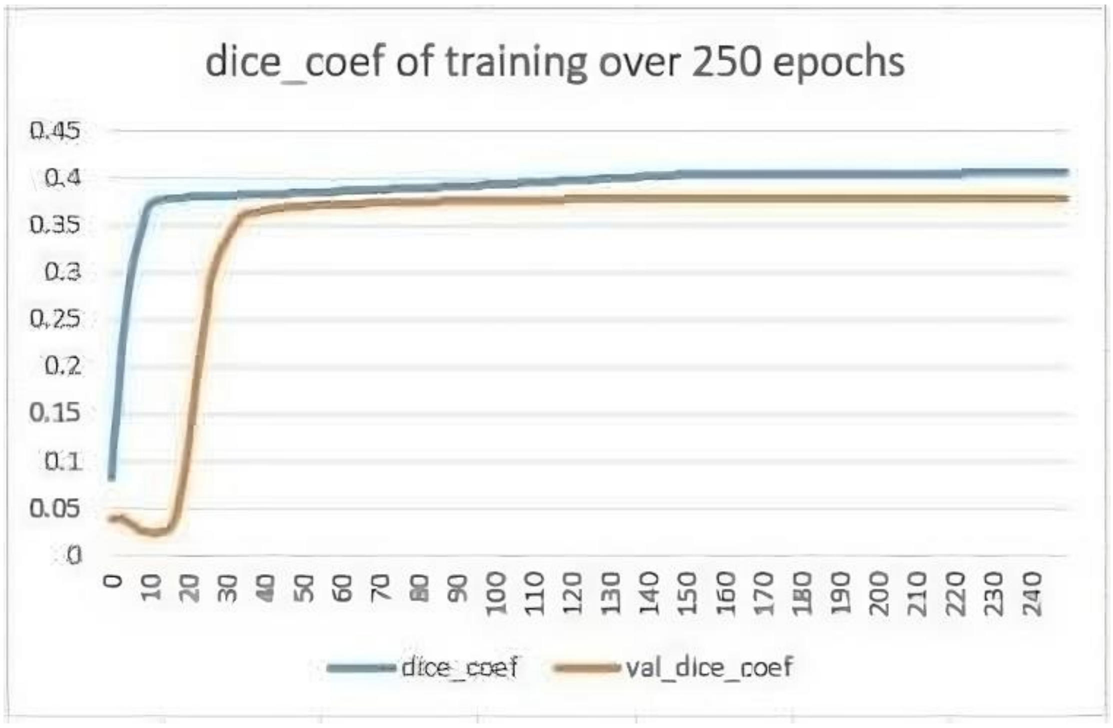


Figure 41. dice coef of training over 250 epochs

The U-Net model was trained on a subset of the dataset, which included 240 brain MRI scans and their matching masked tumor pictures, accounting for around 8% of the total available data (3064 images). Due to time restrictions, this sub-setting option was taken to allow for a more manageable training procedure while still giving a representative percentage of the data. Despite the smaller dataset size, the model demonstrated outstanding learning skills and efficiently fine-tuned its parameters during 250 training rounds (rather than the initially scheduled 500 iterations).Despite training on a limited dataset, the U-Net model demonstrated promising results. The model achieved a Dice coefficient of approximately 40% on the training data, indicating a reasonable level of agreement between the predicted and ground truth masks. Furthermore, during validation, the Dice coefficient reached around 38%, suggesting that the model's performance generalized well to unseen data.

After the training procedure was completed, the trained U-Net model was stored in the HDF5 file format, assuring the preservation of its architecture and learning weights for future use and analysis. Because of its versatility and effectiveness in managing big and complicated datasets, such as those found in deep learning applications, HDF5 is an excellent choice for storing neural network models. This stored model may be easily retrieved and used for a variety of activities, such as predicting new data or continuing training in later sessions, ensuring the model's essential insights are easily accessible and replicable.

The trained U-Net model performed well, with a Dice coefficient of 40% on the training data and 38% on the validation data. The high Dice coefficient suggests that the model was successful in segmenting brain tumor locations from MRI images. This advantageous feature demonstrates the model's capacity to collect key patterns and characteristics for successful segmentation.

However, it is important to recognize the constraints given by the smaller dataset size. With only 240 photos utilized for training and validation, the model's ability to generalize to new data may suffer. A larger dataset might give more diverse instances, thereby improving the model's resilience. Furthermore, the reduced training time (250 iterations instead of 500) may restrict the model's capacity to completely converge, thus resulting in somewhat worse validation results. As a result, future rounds might comprise training on a larger dataset and increasing the training duration to further enhance the model.

The U-Net model demonstrated good segmentation skills, obtaining significant Dice coefficients on training and validation data. Despite the limited dataset and training time, the model's ability to accurately segment brain tumors shows its importance for medical image processing. However, more optimization and training on a bigger dataset could be investigated in order to fully realize it’s potential.

In addition to the mentioned aspects, it's important to highlight that the U-Net model performed exceptionally well on a substantial number of images, showcasing its ability to accurately delineate brain tumor regions in various MRI scans. The model's remarkable performance on a significant portion of the test dataset signifies its capability to generalize effectively to diverse cases, demonstrating its potential value in real-world medical image segmentation tasks.

As a part of future work and to achieve the best possible accuracy percentage, the model will undergo further training and optimization. One key area of improvement would be training the model on a larger dataset, encompassing a more diverse range of brain MRI scans and tumor variations. This expanded dataset would enable the model to learn from a wider array of scenarios and generalize better to unseen examples, potentially leading to improved segmentation results.

Moreover, the training process will be extended to more iterations to allow the model to converge further and refine its learned representations. Fine-tuning hyper parameters, such as learning rate, batch size, and augmentation techniques, can also be explored to optimize the model's performance.

Additionally, the model architecture itself could be enhanced by incorporating advanced techniques like skip connections, residual blocks, or attention mechanisms to further boost the model's segmentation capabilities. Such architectural modifications can provide the model with additional context and help it focus on important regions during segmentation.

By iteratively refining the model and incorporating these enhancements, the aim is to achieve the highest accuracy percentage possible, making the U-Net model an indispensable tool for accurate and reliable brain tumor segmentation in medical imaging applications.

d) Testing the model

In the testing phase of the model, we seamlessly integrated the trained U-Net model into a user- friendly graphical interface. This interface allowed us to import new MRI scan images and observe the model's segmentation results interactively. After importing an image and running it through the model, we carefully examined the produced segmentations to ensure their accuracy and quality. The graphical interface provided an intuitive and visual representation of the model's performance, enabling us to verify that the predicted segmentations aligned with the actual tumor regions.

Upon verifying the satisfactory results, we further enhanced the graphical interface by adding a post-processing function. This function aimed to improve the visual appeal and comprehensibility of the segmentation results. Specifically, we implemented a contouring procedure to outline the masked tumorregion from the original MRI scan. By overlaying the contoured tumor region on top of the original scan,we created an informative visualization that highlighted the precise tumor localization. This added visual feature allowed us to better understand the model's segmentation performance and made it easier to communicate the results to medical professionals.

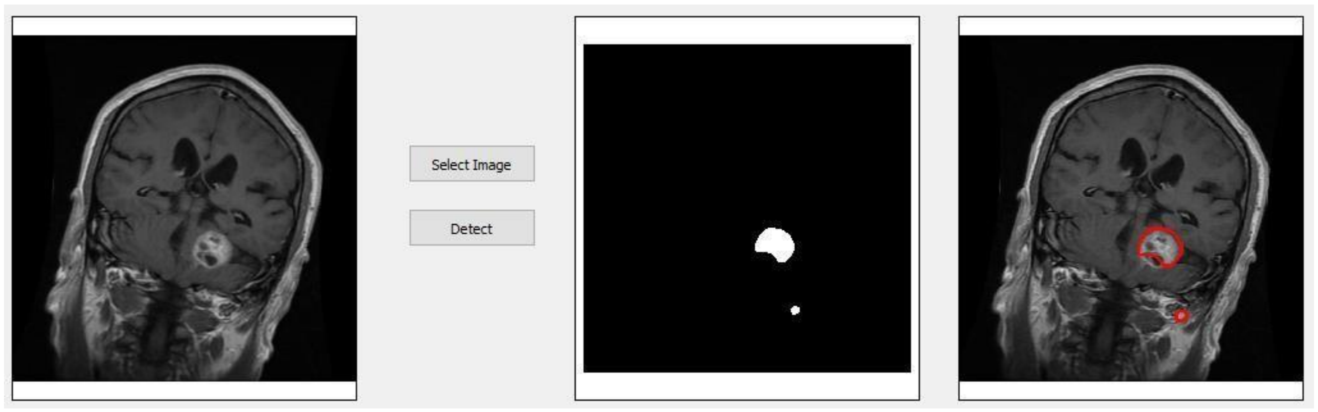


Figure 42. One of the testing tries

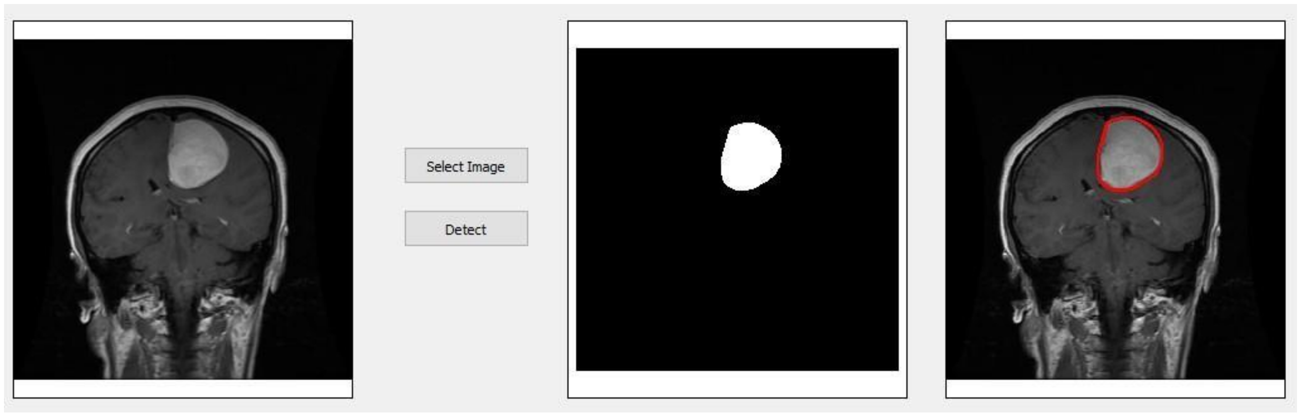


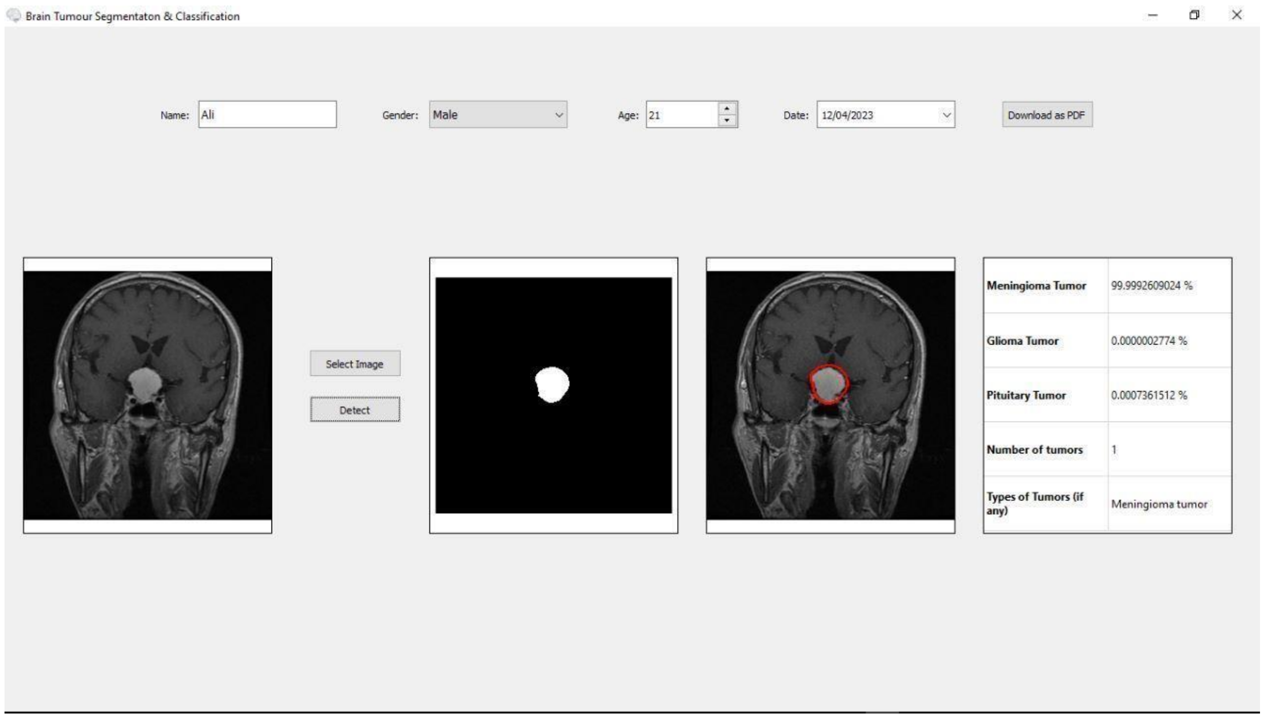
Figure 43. Another one of the testing tries

To summarize, the testing process entailed incorporating the learned U-Net model into a user- friendly graphical interface. The interface aided in the import of new MRI scan pictures, allowing for interactive evaluation of the model's segmentation findings. We created a contouring function to visually emphasize the masked tumor location on the original MRI scan after confirming the correctness of the segmentations. This new visualization improved the model's segmentation outputs' interpretability and utility, making it a powerful tool for medical picture analysis and brain tumor localization.

3.4 Graphical user interface

In our project, we made use of the PyQt5 library's strength and adaptability to create a graphical user interface (GUI) that was simple to understand and enjoyable to use. With the help of PyQt5, a Python binding for the Qt framework, we can easily build feature-rich desktop apps. We were able to create an interactive interface using PyQt5 that worked in unison with our technology for identifying and categorizing brain tumors. We were able to customize the GUI by using the library's large selection of widgets, layouts, and tools, giving users a productive and pleasurable experience while engaging with the system. Our project benefited further from PyQt5's powerful features and cross-platform interoperability, which made our GUI usable by a larger audience on a variety of operating systems. Overall, PyQt5 integration was a key factor in improving our system's usability and aesthetics, contributing much to the success of our project.

In Figure 43, we present the graphical user interface (GUI) of our application, showcasing a user-friendly platform where users can effortlessly upload an MRI image. Upon selecting the 'Detect' option, the application seamlessly processes the image and provides both a segmented representation of the tumor and a classification of its type. This intuitive and interactive interface enhances accessibility and usability for clinicians and researchers alike.



Name: Ali

Gender: Male

Age: 21

Date: 12/04/2023

Meningioma Tumor	99.9992609024 %
Glioma Tumor	0.0000002774 %
Pituitary Tumor	0.0007361512 %
Number of tumors	1
Types of Tumors (if any)	Meningioma tumor

Figure 44. The application designed by PyQt5 library

CHAPTER 4

CONCLUSION AND PERSPECTIVES

In conclusion, our application represents a significant leap forward in the field of medical imaging, showcasing impressive capabilities in accurately identifying and categorizing brain tumors. While there is room for enhancement, the dynamic nature of technology ensures ongoing progress. Future research endeavors should focus on refining the system's precision and efficacy to broaden its diagnostic potential. By incorporating additional classifications for various types of brain tumors, we can further extend its utility and impact in the realm of medical imaging.

This project has afforded us invaluable insights into the realms of machine learning and image processing, setting the stage for future advancements in this critical domain. Committed to pushing the boundaries of knowledge and technology, we remain poised to explore even more accurate and reliable medical solutions, with a positive impact on patient care and outcomes.

Looking ahead, the potential for our technology is boundless. As machine learning, data analytics, and medical imaging continue to advance, our system stands to benefit immensely from these developments. Integration of cutting-edge algorithms, access to larger and more diverse datasets, and real-time processing capabilities could usher in a new era of even more precise and efficient brain tumor diagnostics.

Collaborations with medical experts and practitioners will be instrumental in tailoring our technology to meet the evolving demands of patient care. Through the ongoing refinement and expansion of our system's capabilities, we envision a future where it becomes an indispensable tool for healthcare professionals, driving improved treatment outcomes and contributing to the overall advancement of medical science. Our dedication to innovation and the pursuit of excellence ensures that our journey towards better healthcare solutions will continue to be at the forefront of medical technology advancements

REFERENCES

Wadhwa, A., Bhardwaj, A., & Singh Verma, V. (2019, September). A review on brain tumor segmentation of MRI images. *Magnetic Resonance Imaging, 61*, 247–259. https://doi.org/10.1016/j.mri.2019.05.043

Gordillo, N., Montseny, E., & Sobrevilla, P. (2013, October). State of the art survey on MRI brain tumor segmentation. *Magnetic Resonance Imaging, 31*(8), 1426–1438. https://doi.org/10.1016/j.mri.2013.05.002

Yousef, R., Khan, S., Gupta, G., Siddiqui, T., Albahlal, B. M., Alajlan, S. A., & Haq, M. A. (2023, May 4). U-Net-Based Models towards Optimal MR Brain Image Segmentation. *Diagnostics, 13*(9), 1624. https://doi.org/10.3390/diagnostics13091624

Walsh, J., Othmani, A., Jain, M., & Dev, S. (2022, November). Using U-Net network for efficient brain tumor segmentation in MRI images. *Healthcare Analytics, 2*, 100098. https://doi.org/10.1016/j.health.2022.100098

Xu, W., Deng, X., Guo, S., Chen, J., Sun, L., Zheng, X., Xiong, Y., Shen, Y., & Wang, X. (2020, July 22). High-Resolution U-Net: Preserving Image Details for Cultivated Land Extraction. *Sensors, 20*(15), 4064. https://doi.org/10.3390/s20154064

Du, G., Cao, X., Liang, J., Chen, X., & Zhan, Y. (2020, March 1). Medical Image Segmentation based on U-Net: A Review. *Journal of Imaging Science and Technology, 64*(2), 20508–1. https://doi.org/10.2352/j.imagingsci.technol.2020.64.2.020508

Ronneberger, O., Fischer, P., & Brox, T. (2015). U-Net: Convolutional Networks for Biomedical Image Segmentation. *Lecture Notes in Computer Science*, 234–241. https://doi.org/10.1007/978-3-319-24574-4_28

S, P. (2023, January 6). *Top 8 Interview Questions on UNet Architecture*. Analytics

Vidhya.https://www.analyticsvidhya.com/blog/2023/01/top-8-interview-questions-on-unet-architecture/

Yin, X. X., Sun, L., Fu, Y., Lu, R., & Zhang, Y. (2022, April 15). U-Net-Based Medical Image Segmentation. *Journal of Healthcare Engineering*, *2022*, 1–16.https://doi.org/10.1155/2022/4189781

Yogananda, C. G. B., Shah, B. R., Vejdani-Jahromi, M., Nalawade, S. S., Murugesan, G. K., Yu, F. F., Pinho, M. C., Wagner, B. C., Emblem, K. E., Bjørnerud, A., Fei, B., Madhuranthakam, A. J., & Maldjian, J. A. (2020, June 1). A Fully Automated Deep Learning Network for Brain Tumor Segmentation. *Tomography*, *6*(2), 186–193. https://doi.org/10.18383/j.tom.2019.00026

Bakas, S., Akbari, H., Sotiras, A., Bilello, M., Rozycki, M., Kirby, J. S., Freymann, J. B., Farahani, K., & Davatzikos, C. (2017, September 5). Advancing The Cancer Genome Atlas glioma MRI collections with expert segmentation labels and radiomic features. *Scientific Data*, *4*(1).https://doi.org/10.1038/sdata.2017.117

Thakur, P. A. (2022, February 28). Brain Tumor Segmentation Using K-means Clustering Algorithm. *International Journal for Research in Applied Science and Engineering Technology*, *10*(2), 1–8.https://doi.org/10.22214/ijraset.2022.40112

Khan, A. R., Khan, S., Harouni, M., Abbasi, R., Iqbal, S., & Mehmood, Z. (2021, February). Brain tumor segmentation using K-means clustering and deep learning with synthetic data augmentation for classification. *Microscopy Research and Technique*, *84*(7), 1389–1399.https://doi.org/10.1002/jemt.23694

Brain Tumor - statistics. Cancer.Net. (2023, May 31). https://www.cancer.net/cancer-types/brain-tumor/statistics#:~:text=The%205%2Dyear%20relative%20survival%20rate%20for%20people%20younger%20than,brain%20tumor%20every%205%20years.

Bhattacharjee, S., Prakash, D., Kim, C.-H., Kim, H.-C., & Choi, H.-K. (2022, January). Texture, morphology, and statistical analysis to differentiate primary brain tumors on two-dimensional magnetic resonance imaging scans using artificial intelligence techniques. Healthcare informatics research.https://www.ncbi.nlm.nih.gov/pmc/articles/PMC8850171/#:~:text=Meningiomas%20occur%20in%20the%20meninges,be%20cancerous%20or%20non-cancerous.

Manual segmentation. Manual Segmentation - an overview | ScienceDirect Topics. (n.d.).

https://www.sciencedirect.com/topics/computer-science/manual-segmentation

Ben Dickson, By, Dickson, B., Ben Dickson. (2020, January 6). What are Convolutional Neural Networks (CNN)?. TechTalks. https://bdtechtalks.com/2020/01/06/convolutional-neural-networks-cnnconvnets/#:~:text=Convolutional%20neural%20networks%2C%20also%20called,a%20postdoctoral%20computer%20science%20researcher.

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). ImageNet Classification with Deep Convolutional Neural Networks. In Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1 (NIPS'12). Curran Associates Inc. https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf

LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. Nature, 521(7553), 436-444. doi: 10.1038/nature14539

Han, S., Mao, H., & Dally, W. J. (2015). Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. arXiv preprint arXiv:1510.00149.

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., & Rabinovich, A. (2015). Going deeper with convolutions. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (pp. 1-9).https://openaccess.thecvf.com/content_cvpr_2015/html/Szegedy_Going_Deeper_With_2015_CVPR_paper.html

Zhou, B., Khosla, A., Lapedriza, A., Oliva, A., & Torralba, A. (2016). Learning Deep Features for Discriminative Localization. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (pp. 2921-

2929).https://openaccess.thecvf.com/content_cvpr_2016/html/Zhou_Learning_Deep_Features_CVPR_2016_paper.html

Havaei, M., Davy, A., Warde-Farley, D., Biard, A., Courville, A., Bengio, Y., ... & Pal, C. (2017). Brain tumor segmentation with deep neural networks. Medical image analysis, 35, 18-31.https://doi.org/10.1016/j.media.2016.05.004

Korfiatis, P., Kline, T. L., Erickson, B. J. (2016). Automated segmentation of hyperintense regions in FLAIR MRI using deep learning. In Proceedings of SPIE Medical Imaging (pp. 97841P-97841P).https://doi.org/10.1117/12.2216271

Perez, L., & Wang, J. (2017). The Effectiveness of Data Augmentation in Image Classification using Deep Learning. arXiv preprint arXiv:1712.04621. https://arxiv.org/abs/1712.04621

Abd El Kader, I., Xu, G., Shuai, Z., Saminu, S., Javaid, I., &amp; Salim Ahmad, I. (2021, March 10). A differential deep convolutional neural network model for Brain Tumor Classification. Brain sciences. https://www.ncbi.nlm.nih.gov/pmc/articles/PMC8001442/

G; L. (n.d.). Deeptumor: Framework for brain mr image classification, segmentation, and tumor detection. Diagnostics (Basel, Switzerland). https://pubmed.ncbi.nlm.nih.gov/36428948/

Tiwari, P., Pant, B., Elarabawy, M. M., Abd-Elnaby, M., Mohd, N., Dhiman, G., &amp; Sharma, S. (2022, June 21). CNN-based multiclass brain tumor detection using medical imaging. Computational intelligence and neuroscience.https://www.ncbi.nlm.nih.gov/pmc/articles/PMC9239800/

Chollet, F. (2015). Keras. https://keras.io/

APPENDIX

The ACM/IEEE-CS Software Engineering Code of Ethics and Professional Practice has been kept in mind during the development of this project. The following is the shortened version of the code of ethics and professional practice taken from:

ACM Ethics, The Software Engineering Code of Ethics and Professional Practice, URL: https://ethics.acm.org/code-of-ethics/software-engineering-code/,

Software engineers shall commit themselves to making the analysis, specification, design, development, testing and maintenance of software a beneficial and respected profession. In accordance with their commitment to the health, safety and welfare of the public, software engineers shall adhere to the following Eight Principles:

1. *PUBLIC – Software engineers shall act consistently with the public interest.*
2. *CLIENT AND EMPLOYER – Software engineers shall act in a manner that is in the best interests of their client and employer consistent with the public interest.*
3. *PRODUCT – Software engineers shall ensure that their products and related modifications meet the highest professional standards possible.*
4. *JUDGMENT – Software engineers shall maintain integrity and independence in their professional judgment.*
5. *MANAGEMENT – Software engineering managers and leaders shall subscribe to and promote an ethical approach to the management of software development and maintenance.*
6. *PROFESSION – Software engineers shall advance the integrity and reputation of the profession consistent with the public interest.*
7. *COLLEAGUES – Software engineers shall be fair to and supportive of their colleagues.*
8. *SELF – Software engineers shall participate in lifelong learning regarding the practice of their profession and shall promote an ethical approach to the practice of the profession.*