Apr 28, 2025 Version 3

Lightweight Brain Tumor Segmentation on Low-Resource Systems: A Step-by-Step Guide with 3D U-Net V.3

DOI
https://dx.doi.org/10.17504/protocols.io.dm6gpdwmdgzp/v3

Ayomide Oladele[1], Raymond Confidence[1,2,3], Dong Zhang[1,4], Charity Umoren[1], Aondana M Iorumbur[5], Anu Gbadamosi[1], Farouk Dako[1,6], Maruf Adewole[1,6], Udunna Anazodo[1,2,3]

[1]Medical Artificial Intelligence Laboratory, Lagos, Nigeria;
[2]Montreal Neurological Institute, McGill University, Montreal, Canada;
[3]Department of Biomedical Engineering, McGill University, Montreal, Canada;
[4]Department of Electrical and Computer Engineering, University of British Columbia, Vancouver, Canada;
[5]Department of Physics, Federal University of Technology, Minna;
[6]Department of Radiology, University of Pennsylvania, USA

MAI Lab
Medical Aritifical Intelligence Laboratory, Lagos Nigeria

Create & collaborate more with a free account

Edit and publish protocols, collaborate in communities, share insights through comments, and track progress with run records.

Create free account

OPEN ACCESS

DOI: https://dx.doi.org/10.17504/protocols.io.dm6gpdwmdgzp/v3

Protocol Citation: Ayomide Oladele, Raymond Confidence, Dong Zhang, Charity Umoren, Aondana M Iorumbur, Anu Gbadamosi, Farouk Dako, Maruf Adewole, Udunna Anazodo 2025. Lightweight Brain Tumor Segmentation on Low-Resource Systems: A Step-by-Step Guide with 3D U-Net. **protocols.io** https://dx.doi.org/10.17504/protocols.io.dm6gpdwmdgzp/v3Version created by **MAI Lab**

License: This is an open access protocol distributed under the terms of the **Creative Commons Attribution License**, which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited

Protocol status: In development
We are still developing and optimizing this protocol

Created: April 28, 2025

Last Modified: April 28, 2025

Protocol Integer ID: 187693

Keywords: Deep learning, MRI Scans, GPUs, CPUs, Brain Tumor Segmentation, Resource-Constrained Settings, Segmentation, Validation, 3D U-Net Architecture, Deployment, lightweight brain tumor segmentation model, lightweight brain tumor segmentation, lightweight brain tumor segmentation on low, automated brain tumor segmentation, brain tumor segmentation, deep learning methods development, conventional deep learning model, deep learning method, skills training in deep learning method, implementation of conventional deep learning model, precise identification of tumor region, application of deep learning method, computational efficiency with segmentation accuracy, high computing resource, need for high computing resource, tumor region, dataset, segmentation accuracy, high computational resource, computing resource, expensive computational resource, segmentation output, mri

Disclaimer

About this Tutorial
This tutorial was collaboratively developed by the Research Staff at the Medical Artificial Intelligence (MAI) Lab, Lagos, Nigeria, and participants of the Sprint AI Training for African Medical Imaging Knowledge Translation (SPARK). It provides a hands-on, end-to-end guide for building and training a lightweight deep learning model for automated brain tumor segmentation using the BraTS-Africa 2024 dataset.

Designed with accessibility in mind, this tutorial is optimized for devices with limited computing power and shows how to obtain solid results using just a CPU—making deep learning more approachable for a broader community of learners and practitioners.

DISCLAIMER – FOR INFORMATIONAL PURPOSES ONLY; USE AT YOUR OWN RISK
The content provided in this tutorial is for informational purposes only and does not constitute legal, clinical, medical, or safety advice. It has not undergone peer review or formal approval processes. Always exercise your own professional judgment before acting on any information presented. Neither the authors nor any affiliated organization can be held liable for the use or misuse of this material.

Abstract

The application of deep learning methods in the healthcare has gained significant popularity and relevance among researchers and consumers in clinical, academic, and industry settings, leading to impactful discoveries that are improving human health. One such application is in automated brain tumor segmentation, which aids in the precise identification of tumor regions on magnetic resonance imaging (MRI) scans for accurate diagnosis, treatment planning, and prognosis.

However, the implementation of conventional deep learning models for this task often requires high computational resources, limiting their use by researchers in resource-constrained settings. This computational burden also limits skills training in deep learning methods in resource-constrained settings.

This tutorial presents an approach to address the need for high computing resources in deep learning methods development. It provides a step-by-step guide to developing a **lightweight** brain tumor segmentation model using a 3D U-Net architecture, optimized for low-resource systems. Using the Brain Tumor Segmentation (BraTS) in Sub-Sharan African Population (BraTS-Africa) 2024 dataset, the architecture was trained efficiently and evaluated on standard CPUs, without relying on GPUs. The approach taken in this tutorial seeks to balance computational efficiency with segmentation accuracy. The lightweight model achieved a Dice score of 0.67% on the validation data, and the segmentation output was visually compared with the ground truth. Despite being trained on low computing resources, the model showed promising results.

The main objective of this tutorial is to empower researchers in resource-constrained settings to learn how to develop, validate and deploy deep learning methods using existing frameworks and without reliance on expensive computational resources such as GPUs. More importantly, the tutorial will enable a wider audience to gain practical AI skills, facilitating development of local relevant tools for early detection, ultimately improving patient outcomes.

INTRODUCTION

1 *This comprehensive tutorial guides you through the end-to-end process of developing and training a lightweight deep learning model for brain tumor segmentation using the BraTS-Africa 2024 dataset. Specifically designed for computers with limited computing resources, this tutorial demonstrates how to achieve promising results on a CPU, making deep learning accessible to a wider range of learners and users.*

Brain tumor segmentation is a critical task in medical imaging, where accurate identification and delineation of tumor regions in brain magnetic resonance imaging (MRI) scans are crucial for diagnosis, treatment planning, and monitoring disease progression. However, this process is challenging due to the complex variability of tumor shapes, sizes, and locations, as well as the intricate structure of the brain. In resource-constrained settings, where access to skilled radiologists and medical facilities is limited, delayed diagnosis can lead to poor outcomes including mortalities.

To address this issue, AI-powered tools can facilitate early detection and initial assessment. Nevertheless, most operational AI models require high-performance computing resources (e.g., GPUs or TPUs), which can hinder their deployment on low-resource systems.

To overcome this limitation, lightweight models that balance accuracy and efficiency are essential. These models must be capable of operating effectively on standard CPUs, without relying on high-end GPUs or extensive memory. By developing and deploying such models, we can expand access to timely and accurate diagnosis, ultimately improving health outcomes in resource-constrained communities.

The tutorial is divided into four phases:

1: Data Collection, Preparation and Preprocessing Phase
2: Data Loading and Model Building Phase
3: Model Training and Evaluation Phase
4: Model Deployment and Practical Stages.

(Learn visually: Watch the tutorial video for this section.)

To learn more about automated brain tumor segmentation with deep learning, review these videos:

*1. **Overview of U-Net***
*2. **Semantic Segmentation of BraTS 2020 with 3D Unet***

3. Deep learning approaches for MRI research

MATERIALS SECTION

2 **(Learn visually: Watch the tutorial video for this section - Mac PC | Windows PC.)**

This section will help you prepare your computer with the right materials needed to complete this tutorial successfully.

Table 2.1: Computing Hardware Requirement

	Specification	Minimum System Requirements	System Specification Used for the tutorial
	Processor	Core i5 and more	Core i5
	Python Version	3.12+	3.12
	RAM	4GB and more	8GB
	Storage	5GB of free disk space (can either be internal or external hard drive)	12GB of free disk space

The Lightweight model is designed to run on a CPU system.

2.1 **Computing Software Requirements**

These two software packages are required to be installed on your computer to complete this tutorial:

(note: Recommended browser is Google Chrome)

a. **IBM-Aspera-Connect** *(further instructions on this are under Phase 1 below)*

b. **Visual Studio Code (VS code)**

The Integrated Development Environment (IDE)used for this tutorial is Visual Studio Code (VS Code)).

An IDE is a software application that helps programmers develop software codes, like a container that allows programmers to type in their python codes. Google colab, Kaggle notebooks, VS Code are some of the popular python IDEs.

Setting up a Visual Studio Code Application

1. To install your Visual Studio Code App, click on the VS code link above.

2. By clicking on the link, it will redirect you to a web page where you can download the VS code.

Setting Up Your Project in Visual Studio Code

To get started, follow the steps below to set up your project in Visual Studio Code:

Step 1: Create a New Project Folder

1. On your local computer, create a new folder named "BT Segmentation".
2. This folder will serve as the main directory for your project.

Step 2: Open the Project Folder in Visual Studio Code

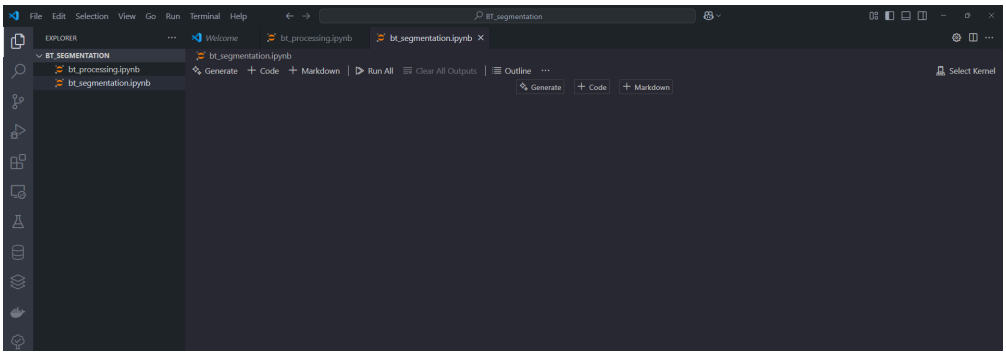
1. Open "**Visual Studio Code"** on your computer.
2. Click on "**File"** in the top menu bar and select **"Open Folder..."**.
3. Navigate to the "**BT Segmentation"** folder and select it.

Step 3: Create a New Python Notebook

1. With the **"BT Segmentation**" folder open in Visual Studio Code, create a new file by clicking on the **"New File..."** button in the Explorer panel or by using the keyboard shortcut **Ctrl + N**(Windows/Linux) or **Cmd + N** (Mac).
2. Save the new file with a **.ipynb** extension (**"bt_processing.ipynb"**) inside the **"BT Segmentation"** folder.
3. Repeat *Step 1 and 2 and create a new .ipynb file* ("**bt_segmentation.ipynb**")
4. These notebooks are where you'll input the code provided in this tutorial.

Your final folder structure should look like this:

BT Segmentation/
└── bt_processing.ipynb
|___bt_segmentation.ipynb



*Then you can click on the **"+ Code"** inside the **.ipynb** file to create your first code block to input your code.*

2.2 Creating a Virtual Environment and Installing Dependencies in VS Code

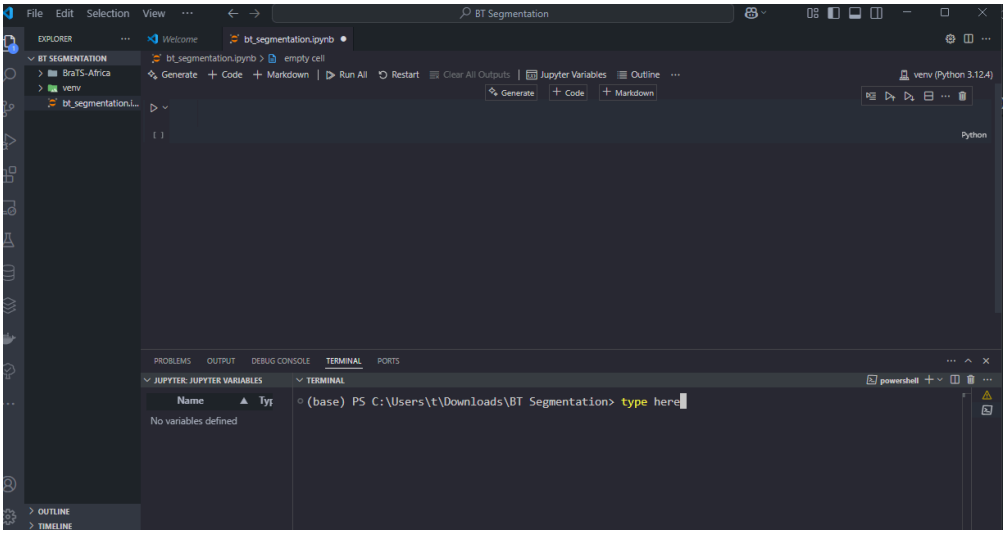
Follow the steps below to create a virtual environment and install the required dependencies using the following steps in **Visual Studio Code (VS Code)**:

Step 1: Open the Terminal in VS Code

1. Click on **"Terminal"** in the left top menu and select **"New Terminal"**
2. Or use the keyboard shortcut:

`Ctrl + `` (backtick) on Windows/Linux
`Cmd + `` on macOS

A terminal will open at the bottom part of VS code as seen in the picture below:



Step 2: Create and activate Virtual Environment

Run the following command in the terminal to create a virtual environment in conda named

Step 2a: Check if Conda is installed on your system with this code:

```
conda --version
```

if you see an output like this "**Conda 24.5.0**", then it means Conda is installed. Proceed to **Step 2c.**

If You See:

'conda' is not recognized as an internal or external command (Windows)

or

command not found: conda (MacOS/Linux)

Then Conda is **not installed** — proceed to **Step 2b**.

Step 2b: Install Conda (Anaconda or Miniconda)

Go to this **website** to install Miniconda on your computer.

Step 2c: Create and Activate the Conda Environment

After you have installed Conda from **Step 2b,** Close your VS code and open it again, then run the following commands in your VS code terminal.

```
conda create -n bt_venv python=3.12
```

Now you have created the environment, then we activate the environment.
(For Windows Users only):

```
conda init powershell
```

Then close and reopen your vscode.

```
conda activate bt_venv
```

This will create a new folder called **bt_venv** in your project directory that contains the isolated environment.

Step 3: Install Required Dependencies

Now that your virtual environment is activated, install your project's dependencies with the following codes in your terminal:

```
conda install -y -c conda-forge ipykernel pandas scikit-learn
glob2 nibabel matplotlib
```

then,

```
pip install numpy tensorflow keras segmentation_models_3D split-
folders
```

Step 4: Add the Virtual Environment as a Kernel in Jupyter

To use your virtual environment as a kernel in Jupyter notebooks inside VS Code:

1. Register your environment as a Jupyter kernel:

```
python -m ipykernel install --user --name=bt_venv --display-name
"Python (bt_venv)"
```

2. Click on your Jupyter notebook ("bt_segmentation.ipynb" and "bt_processing.ipynb") in VS Code.
In the top right corner, click the **kernel selector** , Select **Python Environments** and choose **"Python (bt_venv) (Python 3.12) "** from the list.

Your Jupyter notebook is now running with the environment you just created!

Now you are ready to begin the tutorial**. All is set !!!**

PHASE 1: DATA COLLECTION, PREPARATION AND PREPROCESSING

3 **Objective:** To collect, prepare and preprocess your tutorial dataset.

The BraTS-Africa dataset used in this tutorial was collected from ***The Cancer Imaging Archive (TCIA)***, a publicly available repository of medical images (Adewole, *et al., 2024*). The BraTS-Africa dataset has been fully described elsewhere (Adewole, *et al., 2025*).

(Learn visually: Watch the tutorial video for this section.)

3.1 **A. Dataset Overview**

The BraTS-Africa dataset contains 146 MR Images of 95 glioma and 51 tumor cases in a NIFTI format (`.nii.gz`), which is a common format for storing 3D medical imaging data. Each case has 4 MRI scans and the corresponding segmentation masks of 3 tumor sub-regions generated by expert readers, as reference standard:

a) T1n → (T1-weighted, native) – *t1n.nii.gz*
b) T1c →(T1-weighted, contrast-enhanced - *t1c.nii.gz*
c) T2f → (T2-weighted, fluid sensitive) – *t2f.nii.gz*
d) T2w → (T2 weighted) – *t2w.nii.gz*
e) Segmentation mask - The *seg.nii.gz*

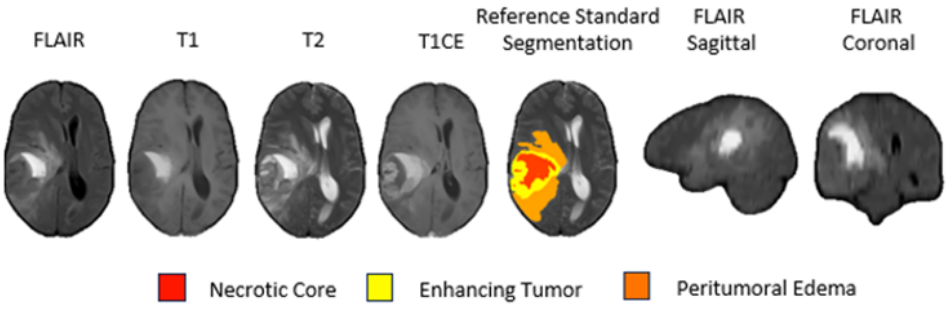


Illustration of the BraTS-Africa Dataset showing brain slices of the four MRI image contrasts and Segmentation masks of the three tumor sub-regions in one representative patient *(Adewole, et al., 2025)*.

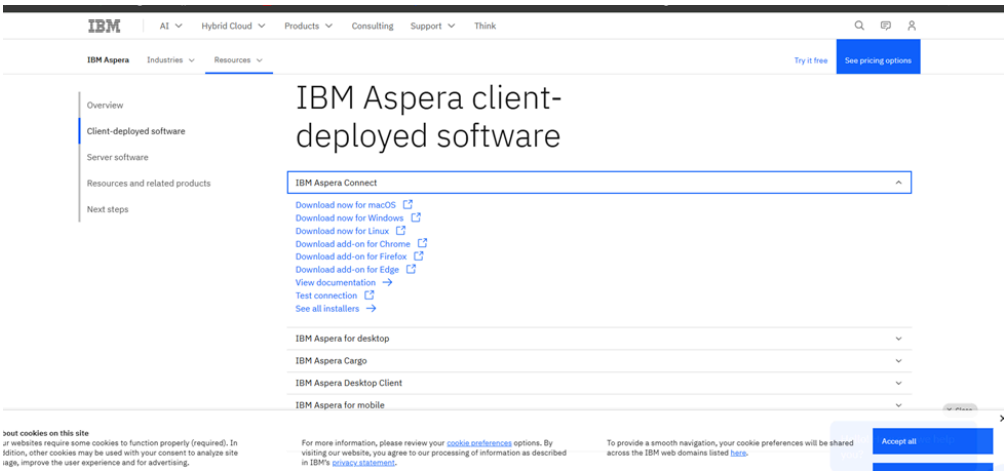
3.2 B. Download the Dataset

To download the dataset, navigate to the **TCIA BraTS-Africa website** and scroll to *Data Access.*

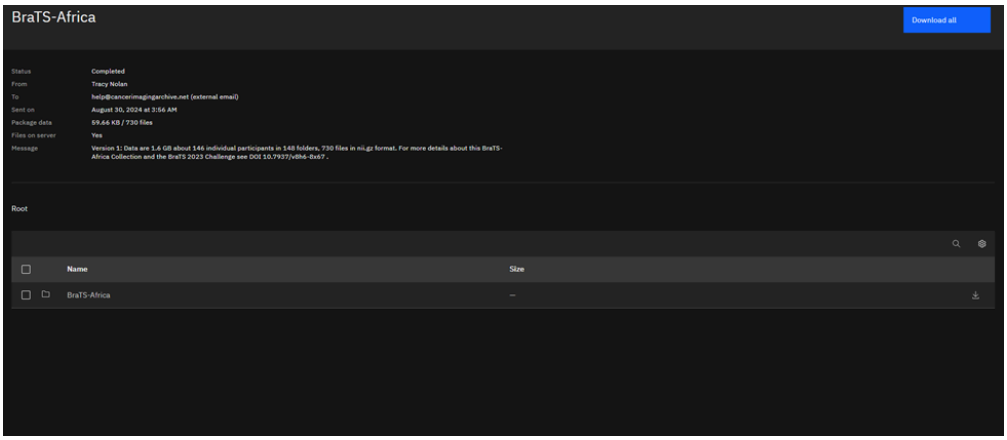
Version 1: Updated 2024/09/04

Title	Data Type	Format	Access Points	Subjects
Radiology Images and Segmentations - BraTS 2023 Challenge	MR, Segmentation	NIFTI	DOWNLOAD (1.6GB) Download requires IBM-Aspera-Connect plugin	146

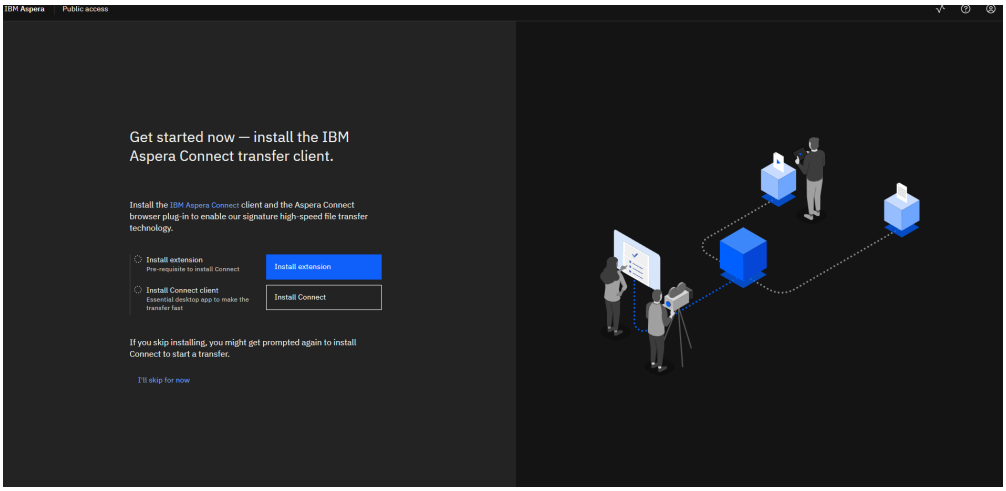
Before you click on the "*DOWNLOAD(1.6GB)*" button, you need to download IBM Aspera plugin by clicking on the *IBM-Aspera-Connect plugin*. This redirects you to the page below:



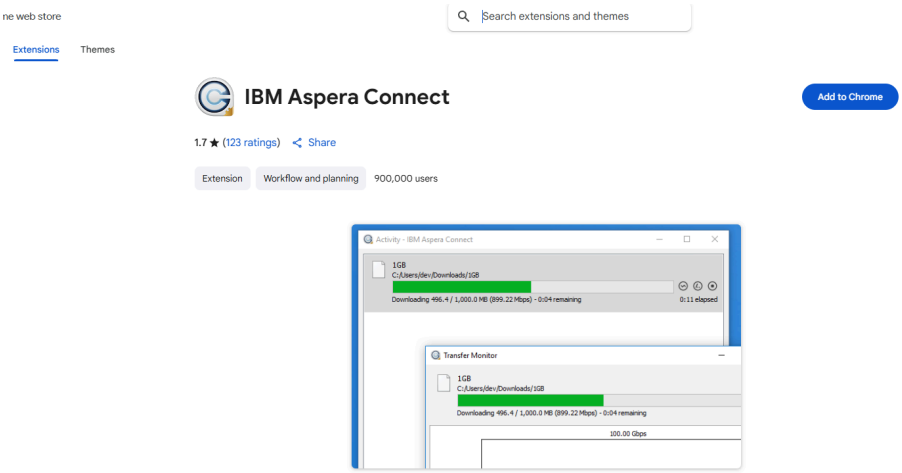
On this path, select the *Download now.* link with respect to your PC operating system. For, Windows, *Download now for Windows*. Then install the IBM Aspera connect application. Navigate back to the TCIA page and click on the *DOWNLOAD (1.6GB)* button,which will redirect you the page below (if a pop-up that asks you to **enable High-speed Transfer** appears, click on it).



When you click on the Download button, it will redirect you to the page below:



Click on **Install extension,** which will open up a new tab as shown below:



Click on the **Add to Chrome**, this will automatically reload to the page where you can download the data. Then click on the **Download icon**, immediately the download will start and be saved on your local computer as a 1.6 GB zipped folder. Right-click on the folder to extract (or decompress) the folder. The folder downloaded will later be opened in your VS code application.

1. Locate the downloaded data from TCIA "BraTS-Africa" which is inside "PKG BraTS-Africa" folder on your computer.
2. Move the entire "BraTS-Africa" folder into the "BT Segmentation" folder you just created.
3. Your folder structure should look like this:

```
BT Segmentation/
├── BraTS-Africa/
└── bt_processing.ipynb
|___bt_segmentation.ipynb
```

3.3 C. Data Collection and Pre-Processing

The first step is to download the dataset from TCIA. The dataset is organized into patient-specific folders, each containing multiple `. nii.gz` files for different MRI modalities and segmentation masks (See above).

The data will be later split into three folders of train, validation (val), and test data in a “glioma split data” folder; The data split: Train = 66 cases Validation = 13 cases , test = 9 cases.

Loading and Converting NIfTI Files

NIfTI files are loaded using the `nibabel` library, which provides tools for reading and writing neuroimaging data. The data is then converted into NumPy arrays for easier processing and manipulation. Now open your "**bt_processing.ipynb**" file and paste this code on the first code block.

```
import numpy as np
import nibabel as nib

#Define the path to your dataset
TRAIN_DATASET_PATH = 'BraTS-Africa/95_Glioma'

#Load sample images and visualize
image_t1n = nib.load(TRAIN_DATASET_PATH + '/BraTS-SSA-00008-
000/BraTS-SSA-00008-000-t1n.nii.gz').get_fdata()
image_t1c = nib.load(TRAIN_DATASET_PATH + '/BraTS-SSA-00008-
000/BraTS-SSA-00008-000-t1c.nii.gz').get_fdata()
image_t2f = nib.load(TRAIN_DATASET_PATH + '/BraTS-SSA-00008-
000/BraTS-SSA-00008-000-t2f.nii.gz').get_fdata()
image_t2w = nib.load(TRAIN_DATASET_PATH + '/BraTS-SSA-00008-
000/BraTS-SSA-00008-000-t2w.nii.gz').get_fdata()
mask = nib.load(TRAIN_DATASET_PATH + '/BraTS-SSA-00008-
000/BraTS-SSA-00008-000-seg.nii.gz').get_fdata()
```

3.4 ***Scaling the Images***

Medical images often have varying intensity ranges. To standardize the data, we use `MinMaxScaler` from `sklearn` to scale the pixel values to a range of [0,1]. Note that the reshape was used to convert the 3D images to a 2D shape so the scaler function can operate on the images. The .reshape (-1, image_t1n.shape[-1]) flattens the first two dimensions while keeping the last one unchanged, converting (H, W, D) into a 2D shape (H∗W, D).

```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()

image_t1n =
scaler.fit_transform(image_t1n.reshape(-1,image_t1n.shape[-1]))
.reshape(image_t1n.shape)
image_t1c =
scaler.fit_transform(image_t1c.reshape(-1,image_t1c.shape[-1]))
.reshape(image_t1c.shape)
image_t2f=scaler.fit_transform(image_t2f.reshape(-1,image_t2f.s
hape[-1])).reshape(image_t2f.shape)
image_t2w =
scaler.fit_transform(image_t2w.reshape(-1,image_t2w.shape[-1]))
.reshape(image_t2w.shape)
```

3.5 ***Handling Segmentation Masks***

The segmentation masks are labelled with integer values representing different tumour regions. For consistency, we convert the mask to `uint8` and reassign label `4`to `3`.

```
mask = mask.astype(np.uint8)
mask[mask== 4] = 3 # Reassign mask values 4 to 3
```

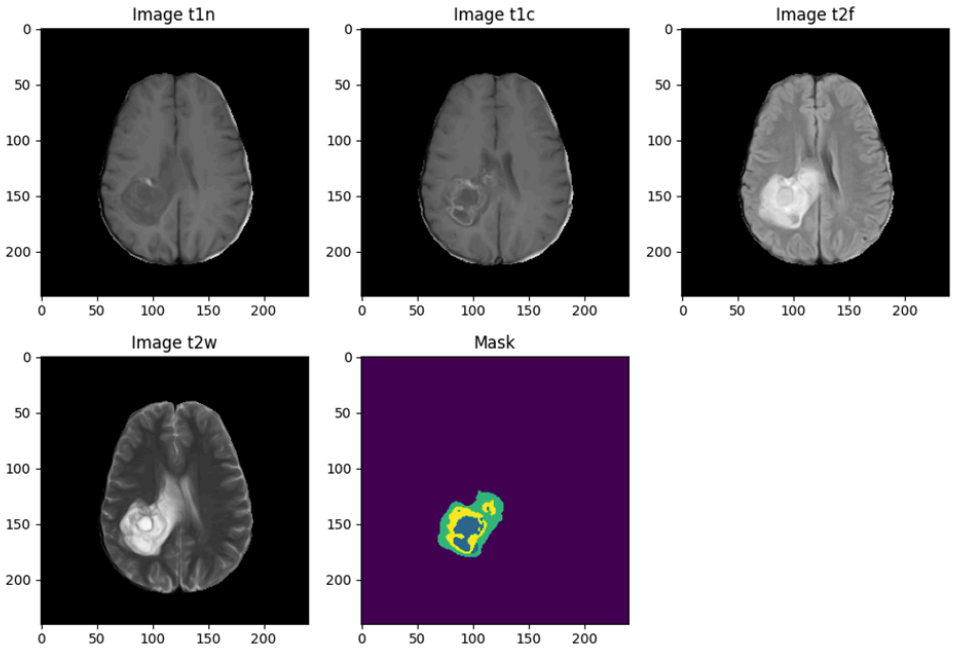
Note that:
0 → Background (Non-brain regions or healthy tissue)
1 → Tumor Core (Necrotic/Cancerous region)
2 → Edema (Swelling or fluid accumulation around the tumor)
3 → Enhancing Tumor (Active tumor region that enhances with contrast agents in MRI)

3.6 ***Visualizing the Data***

To ensure the data is correctly loaded and processed, we visualize a random slice from the MRI scan and its corresponding mask.

```
import numpy as np
import matplotlib.pyplot as plt
import random

n_slice = random.randint(0, mask.shape[2]) #this picks random
slice for viewing

plt.figure(figsize=(12,8))

plt.subplot(231)
plt.imshow(np.rot90(image_t1n[:,:, n_slice]), cmap='gray')
plt.title('Imaget1n')

plt.subplot(232)
plt.imshow(np.rot90(image_t1c[:,:, n_slice]), cmap='gray')
plt.title('Image t1c')

plt.subplot(233)
plt.imshow(np.rot90(image_t2f[:,:, n_slice]), cmap='gray')
plt.title('Image t2f')

plt.subplot(234)
plt.imshow(np.rot90(image_t2w[:,:, n_slice]), cmap='gray')
plt.title('Imaget2w')

plt.subplot(235)
plt.imshow(np.rot90(mask[:,:, n_slice]))
plt.title('Mask')

plt.show()
```

Visual Output:



Note: This output is from a random slice picked by the code. You should run this code multiple times to go through different slices of the image.

3.7 ***Combining Multi-Modality Images***

Brain MRI scans often include multiple modalities (e.g., T1, T2, FLAIR), each providing complementary information. We combine these modalities into a single multi-channel NumPy array for input to the model.

```
combined_x = np.stack([image_t1n, image_t1c, image_t2f,
image_t2w], axis=3)
```

Cropping the Images

To reduce computational load, we crop the images to a smaller size (from 256×256×256 to 128×128×128). This step ensures the data fits into memory and is compatible with the model architecture.

```
combined_x = combined_x[56:184, 56:184, 13:141] #Crop to
128x128x128
mask = mask[56:184, 56:184, 13:141]
```

3.8 ***Processing the Entire Dataset***

The above steps are repeated for all images in the dataset. We use `glob` to locate all `.nii.gz` files and process them in a loop. A 1% threshold was applied to screen out images with less tumor-related images (1, 2, or 3), if too much background label (0) is found, it is removed from the images. The images are finally saved into a folder “glioma”.

```
import glob
import os
import nibabel as nib
import numpy as np
from sklearn.preprocessing import MinMaxScaler
from tensorflow.keras.utils import to_categorical

# Initialize scaler
scaler = MinMaxScaler()

# Create output folders
os.makedirs('glioma/images', exist_ok=True)
os.makedirs('glioma/masks', exist_ok=True)

# List all patient folders
patient_folders = glob.glob(os.path.join(TRAIN_DATASET_PATH, '*'))

# Loop over each patient folder
for img, folder in enumerate(patient_folders):
    # Construct paths
    t1n_path = glob.glob(os.path.join(folder, '*t1n.nii.gz'))
    t1c_path = glob.glob(os.path.join(folder, '*t1c.nii.gz'))
    t2f_path = glob.glob(os.path.join(folder, '*t2f.nii.gz'))
    t2w_path = glob.glob(os.path.join(folder, '*t2w.nii.gz'))
    mask_path = glob.glob(os.path.join(folder, '*seg.nii.gz'))

    # Check if all required files are present
    if not (t1n_path and t1c_path and t2f_path and t2w_path and
mask_path):
        print(f"[SKIP] Missing modalities or mask in folder:
{folder}")
        continue

    print("Now preparing image and masks number:", img)

    # Load and scale modalities
    def load_and_scale(img_path):
        img_data = nib.load(img_path[0]).get_fdata()
        img_data = scaler.fit_transform(img_data.reshape(-1,
img_data.shape[-1])).reshape(img_data.shape)
        return img_data

    try:
        temp_image_t1n = load_and_scale(t1n_path)
```

```
        temp_image_t1c = load_and_scale(t1c_path)
        temp_image_t2f = load_and_scale(t2f_path)
        temp_image_t2w = load_and_scale(t2w_path)

        temp_mask = nib.load(mask_path[0]).get_fdata()
        temp_mask = temp_mask.astype(np.uint8)
        temp_mask[temp_mask == 4] = 3 # Reassign mask value 4 to 3

        # Combine modalities
        temp_combined_images = np.stack([temp_image_t1n,
temp_image_t1c, temp_image_t2f, temp_image_t2w], axis=3)

        # Crop to desired shape
        temp_combined_images = temp_combined_images[56:184,
56:184, 13:141]
        temp_mask = temp_mask[56:184, 56:184, 13:141]

        val, counts = np.unique(temp_mask, return_counts=True)

        if len(counts) > 1 and (1 - (counts[0] / counts.sum())) >
0.01:
            print("Save Me")
            temp_mask = to_categorical(temp_mask, num_classes=4)
            np.save(f'glioma/images/image_{img}.npy',
temp_combined_images)
            np.save(f'glioma/masks/mask_{img}.npy', temp_mask)
        else:
            print("I am not a good addition to the model")
    except Exception as e:
        print(f"[ERROR] Failed to process {folder}: {e}")
        continue
```

3.9 ***Data Splitting***

The dataset is split into training, validation, and test sets using the `splitfolders` library. This ensures a balanced distribution of data across the sets. The split datasets were saved in a folder “glioma split data”.

```
import splitfolders
import os

os.makedirs('glioma split data', exist_ok=True)

input_folder = 'glioma/'
output_folder = 'glioma split data/'
splitfolders.ratio(input_folder, output=output_folder, seed=42,
ratio=(.75, .15, .10), group_prefix=None)
```

This concludes the end of **Phase 1: Data Preparation and Preprocessing.** The dataset is now ready for training a lightweight brain tumor segmentation model. In the next phase, we will build and train a 3D U-Net model using the pre-processed data. Here is the ***link*** to the complete code for Phase 1.

PHASE 2: BUILDING THE 3D U-NET MODEL

4 **Objective**: To design a lightweight 3D U-Net model suitable for low-resource systems.

(Learn visually: Watch the tutorial video for this section.)

4.1 **A. Introduction to 3D U-Net**

The **3D U-Net** architecture is a powerful model for medical image segmentation tasks, particularly for volumetric data like brain MRI scans. It extends the traditional 2D U-Net by incorporating a third spatial dimension, enabling the model to capture 3D contextual information. This is especially important for
tasks like brain tumour segmentation, where tumours can span multiple slices in a 3D volume.

However, 3D U-Net models can be computationally expensive, making them challenging to run on low-resource systems (e.g., normal CPUs). To address this, we make several modifications to create a **lightweight 3D U-Net**:
- **Fewer Layers**: Reducing the number of convolutional layers to decrease model complexity.
- **Smaller Filters**: Using fewer filters in each convolutional layer to reduce memory usage.
- **Efficient Patch Extraction**: Processing smaller 3D patches instead of the entire volume to fit within memory constraints.

4.2 **B. Model Implementation**

The code for the implementation of the 3D U-Net model was referenced from Sreenivas [Bhattiprolu, n.d.] who converted his 2D U-Net to a simple **3D U-Net model.** The 3D U-Net uses 3D convolutions to process volumetric data, includes dropout

layers to prevent overfitting, uses a contracting path (encoder) to extract features and an expanding path (decoder) to reconstruct segmentation maps, and outputs a multi-class segmentation mask. (Now open your "**bt_segmentation.ipynb**" file)

```
from keras.layers import (
    Input,
    Conv3D,
    MaxPooling3D,
    concatenate,
    Conv3DTranspose,
    Dropout,
    Activation
)
from keras.models import Model


kernel_initializer =  'he_uniform' #Try others if you want

################################################################
def simple_unet_model(IMG_HEIGHT, IMG_WIDTH, IMG_DEPTH,
IMG_CHANNELS,
num_classes):
#Build the model
    inputs = Input((IMG_HEIGHT, IMG_WIDTH, IMG_DEPTH,
IMG_CHANNELS))
    #s = Lambda(lambda x: x / 255)(inputs)   #No needfor this if
we normalize our inputs beforehand
    s = inputs

    #Contraction path
    c1 = Conv3D(16, (3, 3, 3),
activation='relu',kernel_initializer=kernel_initializer,
padding='same')(s)
    c1 = Dropout(0.1)(c1)
    c1 = Conv3D(16, (3, 3, 3), activation='relu',
kernel_initializer=kernel_initializer, padding='same')(c1)
    p1 = MaxPooling3D((2, 2, 2))(c1)

    c2 = Conv3D(32, (3, 3, 3), activation='relu',
kernel_initializer=kernel_initializer, padding='same')(p1)
    c2 = Dropout(0.1)(c2)
    c2 = Conv3D(32, (3, 3, 3), activation='relu',
kernel_initializer=kernel_initializer, padding='same')(c2)
    p2 = MaxPooling3D((2, 2, 2))(c2)

    c3 = Conv3D(64, (3, 3, 3), activation='relu',
kernel_initializer=kernel_initializer, padding='same')(p2)
    c3 = Dropout(0.2)(c3)
```

```
    c3 = Conv3D(64, (3, 3, 3), activation='relu',
kernel_initializer=kernel_initializer, padding='same')(c3)
    p3 = MaxPooling3D((2, 2, 2))(c3)

    c4 = Conv3D(128, (3, 3, 3), activation='relu',
kernel_initializer=kernel_initializer, padding='same')(p3)
    c4 = Dropout(0.2)(c4)
    c4 = Conv3D(128, (3, 3, 3), activation='relu',
kernel_initializer=kernel_initializer, padding='same')(c4)
    p4 = MaxPooling3D(pool_size=(2, 2, 2))(c4)

    c5 = Conv3D(256, (3, 3, 3), activation='relu',
kernel_initializer=kernel_initializer, padding='same')(p4)
    c5 = Dropout(0.3)(c5)
    c5 = Conv3D(256, (3, 3, 3), activation='relu',
kernel_initializer=kernel_initializer, padding='same')(c5)

    #Expansive path
    u6 = Conv3DTranspose(128, (2, 2, 2), strides=(2, 2, 2),
padding='same')(c5)
    u6 = concatenate([u6, c4])
    c6 = Conv3D(128, (3, 3, 3), activation='relu',
kernel_initializer=kernel_initializer, padding='same')(u6)
    c6 = Dropout(0.2)(c6)
    c6 = Conv3D(128, (3, 3, 3), activation='relu',
kernel_initializer=kernel_initializer, padding='same')(c6)

    u7 = Conv3DTranspose(64, (2, 2, 2), strides=(2, 2, 2),
padding='same')(c6)
    u7 = concatenate([u7, c3])
    c7 = Conv3D(64, (3, 3, 3), activation='relu',
kernel_initializer=kernel_initializer, padding='same')(u7)
    c7 = Dropout(0.2)(c7)
    c7 = Conv3D(64, (3, 3, 3), activation='relu',
kernel_initializer=kernel_initializer, padding='same')(c7)

    u8 = Conv3DTranspose(32, (2, 2, 2), strides=(2, 2, 2),
padding='same')(c7)
    u8 = concatenate([u8, c2])
    c8 = Conv3D(32, (3, 3, 3), activation='relu',
kernel_initializer=kernel_initializer, padding='same')(u8)
    c8 = Dropout(0.1)(c8)
    c8 = Conv3D(32, (3, 3, 3), activation='relu',
kernel_initializer=kernel_initializer, padding='same')(c8)
```

```
    u9 = Conv3DTranspose(16, (2, 2, 2), strides=(2, 2, 2),
padding='same')(c8)
    u9 = concatenate([u9, c1])
    c9 = Conv3D(16, (3, 3, 3), activation='relu',
kernel_initializer=kernel_initializer, padding='same')(u9)
    c9 = Dropout(0.1)(c9)
    c9 = Conv3D(16, (3, 3, 3), activation='relu',
kernel_initializer=kernel_initializer, padding='same')(c9)

    outputs = Conv3D(num_classes, (1, 1, 1),
activation='softmax')(c9)

    model = Model(inputs=[inputs], outputs=[outputs])
    #compile model outside of this function to make it flexible.
    model.summary()

    return model

#Test if everything is working ok.
model = simple_unet_model(128, 128, 128, 4, 4)
print(model.input_shape)
print(model.output_shape)
```

4.3 **C. Patch Extraction and Data Augmentation**

Patch Extraction

Processing entire 3D volumes can be memory-intensive, especially on low-resource systems. To address this, we extract smaller 3D patches from the input volumes. This reduces memory usage while still providing sufficient context for segmentation.

```
def extract_patch(image, mask, patch_size):
    img_shape = image.shape[:3]
    patch_x = np.random.randint(0, max(img_shape[0] -
patch_size[0], 1))
    patch_y = np.random.randint(0, max(img_shape[1] -
patch_size[1], 1))
    patch_z = np.random.randint(0, max(img_shape[2] -
patch_size[2], 1))

    return (
        image[patch_x:patch_x + patch_size[0], patch_y:patch_y +
patch_size[1], patch_z:patch_z + patch_size[2], :],
        mask[patch_x:patch_x + patch_size[0], patch_y:patch_y +
patch_size[1], patch_z:patch_z + patch_size[2]])
```

The `extract_patch` function randomly selects a patch of a specified size from the input volume and mask.

4.4 ***Data Augmentation Techniques***

Data augmentation improves model robustness by introducing variability into the training data. The `augment_image` function applies random rotations, flips, brightness adjustments, noise addition, and gamma correction.

```
def gamma_correction(image, gamma):
    return np.clip(image ** gamma, 0, 1)

def augment_image(image, mask, is_training=True):
    if is_training:
        # Rotation
        angle = np.random.uniform(-15, 15)
        image = rotate(image, angle, axes=(0, 1), reshape=False,
mode='reflect')
        mask = rotate(mask, angle, axes=(0, 1), reshape=False,
mode='reflect')

        # Flipping
        if np.random.rand() > 0.5:
            image, mask = np.flip(image, axis=0), np.flip(mask,
axis=0)
        if np.random.rand() > 0.5:
            image, mask = np.flip(image, axis=1), np.flip(mask,
axis=1)

        # Brightness Adjustment
        brightness = np.random.uniform(0.9, 1.1)
        image = np.clip(image * brightness, 0, 1)

        # Noise Addition (Gaussian noise)
        if np.random.rand() > 0.5:
            noise = np.random.normal(0, 0.02, image.shape)
            image = np.clip(image + noise, 0, 1)

        # Gamma Correction
        if np.random.rand() > 0.5:
            gamma = np.random.uniform(0.8, 1.2)
            image = gamma_correction(image, gamma)

    return image, mask
```

4.5 ***Image Loader Function***

The `imageLoader` function generates batches of augmented patches for training. The code below combines the augment and extract patch function into a function that loads the images for training and evaluation “imageLoader”.

```
import numpy as np
import os
from scipy.ndimage import rotate

def load_img(img_dir, img_list):
    images = []
    for image_name in img_list:
        if image_name.endswith('.npy'):
            try:
                image = np.load(os.path.join(img_dir, image_name),
allow_pickle=True).astype(np.float32)
                images.append(image)
            except Exception as e:
                print(f"Error loading file {image_name}: {e}")
    return np.array(images) if images else np.array([])

def imageLoader(img_dir, img_list, mask_dir, mask_list,
batch_size, patch_size, is_training=True):
    L = len(img_list)
    while True:
        batch_start = 0
        while batch_start < L:
            batch_end = min(batch_start + batch_size, L)
            X = load_img(img_dir, img_list[batch_start:batch_end])
            Y = load_img(mask_dir,
mask_list[batch_start:batch_end])

            if len(X) == 0 or len(Y) == 0:
                batch_start = batch_end
                continue

            X_patches, Y_patches = zip(*
[augment_image(*extract_patch(img, mask, patch_size), is_training)
                                        for img, mask in zip(X,
Y)])

            yield np.stack(X_patches, axis=0), np.stack(Y_patches,
axis=0)
            batch_start = batch_end
```

This concludes the end of **Phase 2: Building the 3D U-Net Model.** The model is now ready for training on the preprocessed dataset. In the next phase, we will cover the

training process and evaluation metrics. Here is the **link** to the complete code for Phase 2.

PHASE 3: TRAINING AND EVALUATION

5 **Objective**: To train the model and evaluate its performance.

(Learn visually: Watch the tutorial video for this section.)

5.1 **A. Training Setup**

Defining the Dice Score Function

The Dice Score function was custom defined as a class. The class implements the **Dice Score metric**, commonly used in segmentation tasks to evaluate the overlap between predicted and ground truth masks. The Dice Score is particularly useful in medical image segmentation, such as brain image segmentation, where precise boundary delineation is crucial.

```
import tensorflow as tf

class DiceScore(tf.keras.metrics.Metric):
    def __init__(self, num_classes, class_weights=None, smooth=1e-
6, **kwargs):
        super(DiceScore, self).__init__(**kwargs)
        self.num_classes = num_classes
        self.smooth = smooth
        self.class_weights = class_weights if class_weights is not
None else tf.ones(num_classes) # Default to equal weights
        self.dice_scores = self.add_weight(name='dice_scores',
shape=(self.num_classes,), initializer='zeros')

    def update_state(self, y_true, y_pred, sample_weight=None):
        # Flatten the tensors to ensure computations are class-
wise
        y_true = tf.reshape(y_true, [-1]) # Flatten ground truth
        y_pred = tf.reshape(y_pred, [-1]) # Flatten predictions

        # Initialize the dice scores for each class
        dice_scores = []

        for i in range(self.num_classes):
            # Create binary masks for class i
            y_true_class = tf.cast(tf.equal(y_true, i), 'float32')
            y_pred_class = tf.cast(tf.equal(tf.round(y_pred), i),
'float32')

            # Calculate intersection and union
            intersection = tf.reduce_sum(y_true_class *
y_pred_class)
            union = tf.reduce_sum(y_true_class) +
tf.reduce_sum(y_pred_class)

            # Compute Dice score for the current class
            dice_class = (2. * intersection + self.smooth) /
(union + self.smooth)

            # Apply class weight to the Dice score
            weighted_dice_class = dice_class *
self.class_weights[i]
            dice_scores.append(weighted_dice_class)
```

```
        # Update state by averaging weighted dice scores across
all classes
        dice_scores = tf.stack(dice_scores)
        self.dice_scores.assign(dice_scores)

    def result(self):
        # Return the mean of the weighted dice scores
        return tf.reduce_mean(self.dice_scores)

    def reset_states(self):
        # Reset the dice scores at the start of each batch
        self.dice_scores.assign(tf.zeros(self.num_classes))
```

5.2 ***Model Training***

The model is trained using a loss function that combines **Dice Loss** and **Categorical Focal Loss**, both of which are well-suited for segmentation tasks. **Dice Loss** emphasizes the overlap between predictions and ground truth, improving segmentation accuracy, while **Focal Loss** addresses class imbalance by down-weighting easy examples and focusing on hard-to-classify pixels.

Model evaluation is conducted using **accuracy**, **Intersection over Union (IoU)**, and a **custom Dice Score metric**. To enhance convergence, a **learning rate scheduler** is implemented to reduce the learning rate when validation loss plateaus. The model is optimized using the **Nadam optimizer** with a
fixed learning rate of 0.001, and **gradient clipping** is applied to prevent exploding gradients.

For efficient training, **data generators** are used to load and preprocesstraining and validation data in batches. Additionally, **callbacks** are incorporated to save the best model and implement **early stopping**, terminating training if validation loss ceases to improve.

```
# Set seeds for reproducibility
import random
SEED = 42
os.environ['PYTHONHASHSEED'] = str(SEED)
random.seed(SEED)
np.random.seed(SEED)
tf.random.set_seed(SEED)

import time
import segmentation_models_3D as sm
import numpy as np
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.callbacks import EarlyStopping,
ModelCheckpoint, ReduceLROnPlateau
import os
import pandas as pd



start = time.time()
# Set up loss function with class weights
wt0, wt1, wt2, wt3 = 0.25, 0.25, 0.25, 0.25
class_weights = np.array([wt0, wt1, wt2, wt3], dtype=np.float32)

dice_loss = sm.losses.DiceLoss(class_weights=class_weights)
focal_loss = sm.losses.CategoricalFocalLoss()
total_loss = 0.2 * dice_loss + 0.2 * focal_loss # focal loss of
0.2 (perf best now) becomes 0.1

# Define metrics
metrics = ['accuracy', sm.metrics.IOUScore(threshold=0.5),
DiceScore(num_classes=4)]

# Learning rate scheduler (Cosine Decay)
lr_scheduler = tf.keras.callbacks.ReduceLROnPlateau(
    monitor='val_loss', factor=0.5, patience=5, min_lr=1e-5 # from
6 to 4(perf better) to 3
)

# Optimizer (use a fixed learning rate)
optimizer = keras.optimizers.Nadam(learning_rate=0.001,
clipnorm=1.0) #increase from 0.001(perf best) to 0.01
```

```
# Data paths
DATA_ROOT = "glioma split data"
train_img_dir, train_mask_dir = f"{DATA_ROOT}/train/images/", f"
{DATA_ROOT}/train/masks/"
val_img_dir, val_mask_dir = f"{DATA_ROOT}/val/images/", f"
{DATA_ROOT}/val/masks/"

# Load data
train_img_list, train_mask_list = os.listdir(train_img_dir),
os.listdir(train_mask_dir)
val_img_list, val_mask_list = os.listdir(val_img_dir),
os.listdir(val_mask_dir)

# Data generators
batch_size = 2
patch_size = (64, 64, 64)
train_data = imageLoader(train_img_dir, train_img_list,
train_mask_dir, train_mask_list, batch_size, patch_size)
val_data = imageLoader(val_img_dir, val_img_list, val_mask_dir,
val_mask_list, batch_size, patch_size)

# Training parameters
steps_per_epoch = len(train_img_list) // batch_size
val_steps_per_epoch = len(val_img_list) // batch_size
epochs = 100

# Initialize and compile model
model = simple_unet_model(IMG_HEIGHT=64, IMG_WIDTH=64,
IMG_DEPTH=64, IMG_CHANNELS=4, num_classes=4)
model.compile(optimizer=optimizer, loss=total_loss,
metrics=metrics)
model.summary()

# Define callbacks
checkpoint_path =
f"saved_model/3D_unet_{epochs}_epochs_{batch_size}_batch_patch_tra
ining.keras"
callbacks = [
    EarlyStopping(monitor='val_loss', patience=10,
restore_best_weights=True), #increase patience from 10 to 20
    ModelCheckpoint(checkpoint_path, monitor='val_loss',
save_best_only=True, mode='min'),
    lr_scheduler # Learning rate decay callback
]
```

```
# Train model
history = model.fit(
    train_data,
    steps_per_epoch=steps_per_epoch,
    epochs=epochs,
    validation_data=val_data,
    validation_steps=val_steps_per_epoch,
    callbacks=callbacks,
    verbose=1
)

# Save training history
history_df = pd.DataFrame(history.history)
os.makedirs("model_history", exist_ok=True)
history_df.to_csv("model_history/training_history.csv",
index=False)
print("Training history and best model saved successfully.")

end = time.time()
exec_time = (end - start)/60
print(f'execution time is - {exec_time} minutes or {exec_time/60}
hour')
```

The model training ran on my system for 1 hour and 5 minutes.

5.3 **B. Model Evaluation**

Loading the Trained Model

The trained model is loaded for inference without recompiling.

```
import os
import numpy as np
from keras.models import load_model
from keras.metrics import MeanIoU
from matplotlib import pyplot as plt


# -------------------- MODEL LOADING --------------------
# Path to trained model
model_path =
'saved_model/3D_unet_100_epochs_2_batch_patch_training.keras'

# Load the trained model without recompiling (for inference only)
try:
    my_model = load_model(model_path, compile=False)
except Exception as e:
    raise ValueError(f"Error loading model from {model_path}:
{str(e)}")

# -------------------- DATA LOADING --------------------
# Path to dataset (Modify as needed)
DATA_PATH = "glioma split data"

# Validate directories exist
def validate_dir(path):
    if not os.path.exists(path):
        raise FileNotFoundError(f"Directory not found: {path}")

# Train and Validation Directories
train_img_dir = os.path.join(DATA_PATH, "train/images/")
train_mask_dir = os.path.join(DATA_PATH, "train/masks/")
val_img_dir = os.path.join(DATA_PATH, "val/images/")
val_mask_dir = os.path.join(DATA_PATH, "val/masks/")
test_img_dir = os.path.join(DATA_PATH, "test/images/")
test_mask_dir = os.path.join(DATA_PATH, "test/masks/")

# Validate all directories
for dir_path in [train_img_dir, train_mask_dir, val_img_dir,
val_mask_dir, test_img_dir, test_mask_dir]:
    validate_dir(dir_path)

# Get sorted list of images and masks to ensure alignment
def get_sorted_files(directory):
```

```
    return sorted([f for f in os.listdir(directory) if
f.endswith('.npy')])

train_img_list = get_sorted_files(train_img_dir)
train_mask_list = get_sorted_files(train_mask_dir)
val_img_list = get_sorted_files(val_img_dir)
val_mask_list = get_sorted_files(val_mask_dir)
test_img_list = get_sorted_files(test_img_dir)
test_mask_list = get_sorted_files(test_mask_dir)

# Define patch and batch size
patch_size = (64, 64, 64)
batch_size = 2

# -------------------- EVALUATION UTILITIES --------------------
def evaluate_stochastic(model, data_loader, n_batches=10):
    iou_metric = MeanIoU(num_classes=4)
    dice_metric = DiceScore(num_classes=4)

    for _ in range(n_batches):
        try:
            X, Y = next(data_loader)
            pred = model.predict(X, verbose=0)  # Disable
prediction logging
            pred_argmax = np.argmax(pred, axis=-1)
            mask_argmax = np.argmax(Y, axis=-1)

            iou_metric.update_state(mask_argmax, pred_argmax)
            dice_metric.update_state(mask_argmax, pred_argmax)
        except StopIteration:
            break  # In case we have fewer than n_batches

    return iou_metric.result().numpy(),
dice_metric.result().numpy()

# -------------------- VALIDATION DATA EVALUATION ----------------
----
# Create Data Generator for validation set (with
is_training=False)
val_img_loader = imageLoader(val_img_dir, val_img_list,
val_mask_dir, val_mask_list,
                             batch_size, patch_size,
is_training=False)

# Evaluate multiple batches for more reliable metrics
```

```
val_iou, val_dice = evaluate_stochastic(my_model, val_img_loader,
n_batches=10)
print(f"Validation Metrics - Mean IoU: {val_iou:.4f}, Dice Score:
{val_dice:.4f}")

# -------------------- TEST DATA EVALUATION --------------------
# Create Data Generator for test set (with is_training=False)
test_img_loader = imageLoader(test_img_dir, test_img_list,
test_mask_dir, test_mask_list,
                              batch_size, patch_size,
is_training=False)

# Evaluate multiple batches for more reliable metrics
test_iou, test_dice = evaluate_stochastic(my_model,
test_img_loader, n_batches=10)
print(f"Test Metrics - Mean IoU: {test_iou:.4f}, Dice Score:
{test_dice:.4f}")
```

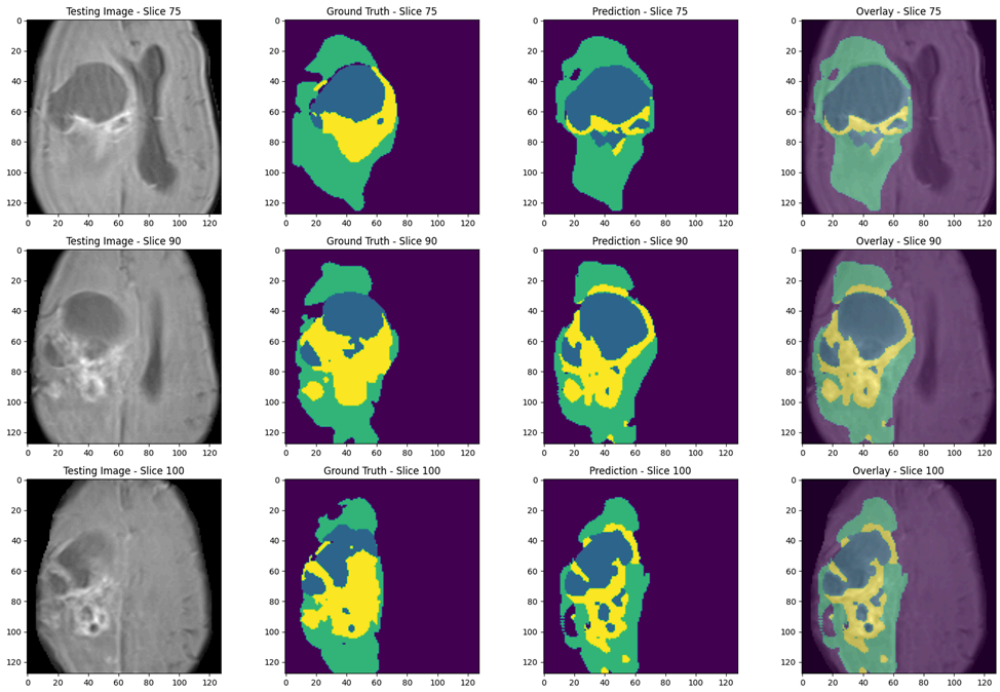
5.4 ***Visualizing Results***

A random test image is selected, and the model's prediction is visualized alongside the ground truth mask.

```
#-------------------- VISUALIZATION --------------------
# Select a random test image for visualization
img_num = 21  # Change index as needed
test_img = np.load(os.path.join(test_img_dir,
f"image_{img_num}.npy"))
test_mask = np.load(os.path.join(test_mask_dir,
f"mask_{img_num}.npy"))
test_mask_argmax = np.argmax(test_mask, axis=3)

# Expand dimensions for model prediction
test_img_input = np.expand_dims(test_img, axis=0)
test_prediction = my_model.predict(test_img_input)
test_prediction_argmax = np.argmax(test_prediction, axis=4)[0, :,
:, :]

# Select slice indices for visualization
slice_indices = [75, 90, 100]  # Change slice indices as needed

# Plotting Results
plt.figure(figsize=(18,12))

for i, n_slice in enumerate(slice_indices):

  # Rotate images to correct orientation

  test_img_rotated = np.rot90(test_img[:, :, n_slice, 1])
#Rotating 90 degrees

  test_mask_rotated = np.rot90(test_mask_argmax[:, :, n_slice])

  test_prediction_rotated = np.rot90(test_prediction_argmax[:, :,
n_slice])


  # Plotting Results

  plt.subplot(3, 4, i*4 + 1)

  plt.title(f'Testing Image - Slice {n_slice}')

  plt.imshow(test_img_rotated, cmap='gray')


  plt.subplot(3, 4, i*4 + 2)
```

```
  plt.title(f'Ground Truth - Slice {n_slice}')

  plt.imshow(test_mask_rotated)


  plt.subplot(3, 4, i*4 + 3)

  plt.title(f'Prediction - Slice {n_slice}')

  plt.imshow(test_prediction_rotated)


  plt.subplot(3, 4, i*4 + 4)

  plt.title(f'Overlay - Slice {n_slice}')

  plt.imshow(test_img_rotated, cmap='gray')

  plt.imshow(test_prediction_rotated, alpha=0.5)  # Overlay
prediction mask

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

Visual Output:



This concludes the end of **Phase 3: Training and Evaluation**. The model has been trained, evaluated, and visualized, demonstrating its performance on brain tumour segmentation tasks. In the next phase, we will cover deployment and practical usage of the model. Here is the **link** to the complete code for Phase 3.

PHASE 4: DEPLOYMENT AND PRACTICAL USAGE

6 **Objective**: To make the model usable for real-world segmentation tasks.

The segmentation model can be deployed locally on your computer or done globally through frameworks such as Streamlit. **Streamlit** is a platform that allows you to deploy and share your data science and machine learning apps.

6.1 **A. Script Creation for Local Usage**

Create a python file called "deploy.py" on your VS Code, and paste the code below in the python file.

```
import os
import numpy as np
import nibabel as nib
from sklearn.preprocessing import MinMaxScaler

# Function to preprocess a new NIfTI file
def preprocess_nifti(file_path, patch_size=(64, 64, 64)):
    # Load the NIfTI file
    image = nib.load(file_path).get_fdata()

    # Normalize the image
    scaler = MinMaxScaler()
    image = scaler.fit_transform(image.reshape(-1,
image.shape[-1])).reshape(image.shape)

    # Pad or crop the image to match the patch size
    if image.shape != patch_size:
      # Example: Center cropping
      start_x = (image.shape[0] - patch_size[0]) // 2
      start_y = (image.shape[1] - patch_size[1]) // 2
      start_z = (image.shape[2] - patch_size[2]) // 2
      image = image[start_x:start_x + patch_size[0],
start_y:start_y +
patch_size[1], start_z:start_z + patch_size[2]]

    # Expand dimensions for model input
    image = np.expand_dims(image, axis=0)  # Add batch dimension
    image = np.expand_dims(image, axis=-1)  # Add channel
dimension
    return image

# Function to run segmentation and save results
def run_segmentation(model, input_file, output_folder):
    # Preprocess the input file
    input_image = preprocess_nifti(input_file)

    # Run segmentation
    prediction = model.predict(input_image)
    prediction_argmax = np.argmax(prediction, axis=4)[0, :, :, :]

    # Save the segmentation result
    output_file = os.path.join(output_folder,
os.path.basename(input_file).replace('.nii.gz',
'_segmentation.nii.gz'))
```

```
    nib.save(nib.Nifti1Image(prediction_argmax, np.eye(4)),
output_file)
    print(f"Segmentation saved to {output_file}")

# Example usage
input_file = "path/to/new_image.nii.gz"  #replace with the correct
path to your image
output_folder = "path/to/output_folder" #replace with the correct
part to where you want to save the output
run_segmentation(my_model, input_file, output_folder)
```

6.2 **B. Deployment on Streamlit**

First, we create a python script called “app.py” to host the code to load the model and the scan image. This script will show the images and the predictions, and also allow users to download the segmentation output.

Prerequisites

1. Streamlit account: Create a Streamlit account.
2. Streamlit app code: Prepare your Python script (e.g., app.py) with Streamlit imports.

Deployment Steps

1. Create a GitHub repository: Store your app code in a GitHub repository.
2. Sign in to Streamlit: Log in to your Streamlit account.
3. New app: Click "New app" and select your GitHub repository.
4. Configure app: Choose the branch, main file (e.g., app.py), and other settings.
5. Deploy: Click "Deploy" to launch your app.

Requirements.txt
Ensure you have a requirements.txt file listing your app's dependencies.

Watch this **video** learn more about deploying your Machine Learning codes on Streamlit.

6.3 ***Creation of a app.py***

Paste the code below into a created app.py file.

```
import streamlit as st
import numpy as np
import nibabel as nib
from tensorflow.keras.models import load_model
from sklearn.preprocessing import MinMaxScaler
import os
import matplotlib.pyplot as plt
import gdown  # For downloading files from Google Drive
import zipfile  # To handle folder uploads
import tempfile  # To handle temporary files
from tensorflow.keras.utils import to_categorical

# Title of the app
st.title("Brain Tumor Segmentation using 3D U-Net - (Lightweight
Architecture on Normal CPUs)")

# Function to download the default model from Google Drive
def download_default_model():
    # Google Drive file ID for the
default model
    file_id = "1lV1SgafomQKwgv1NW2cjlpyb4LwZXFwX"
# Replace with your file ID
    output_path = "default_model.keras"

    # Download the file if it doesn't already exist
    if not
os.path.exists(output_path):
        url =
f"https://drive.google.com/uc?id={file_id}"
        gdown.download(url,
output_path, quiet=False)

    return output_path

# Load the default model
@st.cache_resource  # Cache the
model to avoid reloading on every interaction
def load_default_model():
    # Download the model from
Google Drive
    model_path = download_default_model()
    model = load_model(model_path,
compile=False)
    return model
```

```
default_model = load_default_model()

# Function to preprocess a NIfTI file
def preprocess_nifti(file_path):
    # Load the NIfTI file
    image = nib.load(file_path).get_fdata()

    # Normalize the image
    scaler = MinMaxScaler()
              image    =    scaler.fit_transform(image.reshape(-1,
image.shape[-1])).reshape(image.shape)

    return image

# Function to combine 4 channels into a single 4-channel numpy
array
def combine_channels(t1n, t1c, t2f, t2w):
    # Stack the 4 channels along the last axis
    combined_image = np.stack([t1n, t1c, t2f, t2w], axis=3)
    combined_image = combined_image[56:184, 56:184, 13:141]
    return combined_image

# Function to run segmentation
def run_segmentation(model, input_image):
    # Add batch and channel dimensions
    input_image =np.expand_dims(input_image, axis=0)  #Add batch
dimension
     #input_image  =  np.expand_dims(input_image,  axis=-1)   #Add
channel dimension


    # Ensure the input_image has the correct shape
    if len(input_image.shape) != 5:
        st.error(f"Unexpected
shape  for  input_image:  {input_image.shape}.  Expected  shape:
(batch_size,
height, width, depth, channels).")
        return None

    # Run prediction
    prediction =
model.predict(input_image)
    prediction_argmax =
np.argmax(prediction, axis=4)[0, :, :, :]
```

```
    return prediction_argmax

# Sidebar for model upload
st.sidebar.header("Upload Your Own Model")
uploaded_model = st.sidebar.file_uploader("Upload a Keras model
(.keras)", type=["keras"])

# Load the model (default or uploaded)
if uploaded_model is not None:
    # Save the uploaded model
temporarily
    with
open("temp_model.keras", "wb") as f:

f.write(uploaded_model.getbuffer())

    # Load the uploaded model
    try:
        model =
load_model("temp_model.keras", compile=False)

st.sidebar.success("Custom model loaded successfully!")
    except Exception as e:

st.sidebar.error(f"Error loading custom model: {e}")

st.sidebar.info("Using the default model instead.")
        model = default_model
else:
    model = default_model
    st.sidebar.info("Using
the default model.")

# Main app: Upload a folder containing NIfTI files
st.header("Upload a Folder Containing NIfTI Files")
uploaded_folder = st.file_uploader("Upload a folder (as a zip
file)
containing T1n, T1c, T2f, T2w NIfTI files", type=["zip"])

if uploaded_folder is not None:
    # Create a temporary directory to extract the zip file
    with
tempfile.TemporaryDirectory() as temp_dir:
        # Save the uploaded zip
file
```

```
        zip_path =
os.path.join(temp_dir, "uploaded_folder.zip")
        with open(zip_path,
"wb") as f:

f.write(uploaded_folder.getbuffer())

        # Extract the zip file
        with
zipfile.ZipFile(zip_path, "r") as zip_ref:

zip_ref.extractall(temp_dir)

        # Find the NIfTI files in
the extracted folder
        t1n_path = None
        t1c_path = None
        t2f_path = None
        t2w_path = None
        mask_path = None

        for root, _, files in
os.walk(temp_dir):
            for file in files:
                if
file.endswith("t1n.nii.gz"):
                    t1n_path =
os.path.join(root, file)
                elif
file.endswith("t1c.nii.gz"):
                    t1c_path =
os.path.join(root, file)
                elif
file.endswith("t2f.nii.gz"):
                    t2f_path =
os.path.join(root, file)
                elif
file.endswith("t2w.nii.gz"):
                    t2w_path =
os.path.join(root, file)
                elif
file.endswith("seg.nii.gz"):
                    mask_path =
os.path.join(root, file)
```

```
        # Check if all required
files are found
        if t1n_path and t1c_path
and t2f_path and t2w_path:
            # Preprocess each
channel
            t1n =
preprocess_nifti(t1n_path)
            t1c =
preprocess_nifti(t1c_path)
            t2f =
preprocess_nifti(t2f_path)
            t2w =
preprocess_nifti(t2w_path)

            # Combine the 4
channels
            combined_image =
combine_channels(t1n, t1c, t2f, t2w)

            # Print the shape of
combined_image for debugging
            st.write(f"Shape
of combined_image: {combined_image.shape}")

            # Ensure the
combined_image has the correct shape
            if
len(combined_image.shape) != 4:

st.error(f"Unexpected shape for combined_image:
{combined_image.shape}. Expected shape: (height, width, depth,
channels).")
            else:
                # Run segmentation

st.write("Running segmentation...")

segmentation_result = run_segmentation(model, combined_image)

                # Display the
segmentation result

st.write("Segmentation completed! Displaying results...")
```

```
                # Load the ground
truth mask if available
                if mask_path:
                    mask =
nib.load(mask_path).get_fdata()
                    mask =
mask.astype(np.uint8)
                    mask[mask ==
4] = 3  # Reassign mask values 4 to 3
                    mask_argmax =
np.argmax(to_categorical(mask, num_classes=4), axis=3)
                else:
                    mask_argmax =
None

                # Select slice
indices for visualization
                slice_indices =
[75, 90, 100]  # Example: 25%, 50%, 75%
of depth


                # Plotting Results
                fig, ax =
plt.subplots(3, 4, figsize=(18, 12))

                for i, n_slice in
enumerate(slice_indices):
                    # Rotate
images to correct orientation

test_img_rotated = np.rot90(combined_image[:, :, n_slice, 0])  #
Rotating 90 degrees

test_prediction_rotated  =  np.rot90(segmentation_result[:,  :,
n_slice])

                    # Plotting
Results
                    ax[i,
0].imshow(test_img_rotated, cmap='gray')
                    ax[i,
0].set_title(f'Testing Image - Slice {n_slice}')

                    if mask_path:
```

```
test_mask_rotated = np.rot90(mask_argmax[:, :, n_slice])
                        ax[i,
1].imshow(test_mask_rotated)
                        ax[i,
1].set_title(f'Ground Truth - Slice {n_slice}')
                    else:
                        ax[i,
1].axis('off')  # Hide the ground truth
subplot if no mask is available

                    ax[i,
2].imshow(test_prediction_rotated)
                    ax[i,
2].set_title(f'Prediction - Slice {n_slice}')

                    ax[i,
3].imshow(test_img_rotated, cmap='gray')
                    ax[i,
3].imshow(test_prediction_rotated, alpha=0.5)
# Overlay prediction mask
                    ax[i,
3].set_title(f'Overlay - Slice {n_slice}')

                plt.tight_layout()
                st.pyplot(fig)

                # Save the
segmentation result
                output_file =
"segmentation_result.nii.gz"

nib.save(nib.Nifti1Image(segmentation_result.astype(np.float32),
np.eye(4)), output_file)

                # Provide a
download link for the segmentation result
                with
open(output_file, "rb") as f:

st.download_button(

label="Download Segmentation Result",
                        data=f,
```

```
file_name=output_file,

mime="application/octet-stream"
                    )

                # Clean up
temporary files

os.remove(output_file)
        else:
             st.error("The uploaded folder does not contain all
required NIfTI files (T1n, T1c, T2f,
T2w).")
Save
the script in a “app.py” file.
```

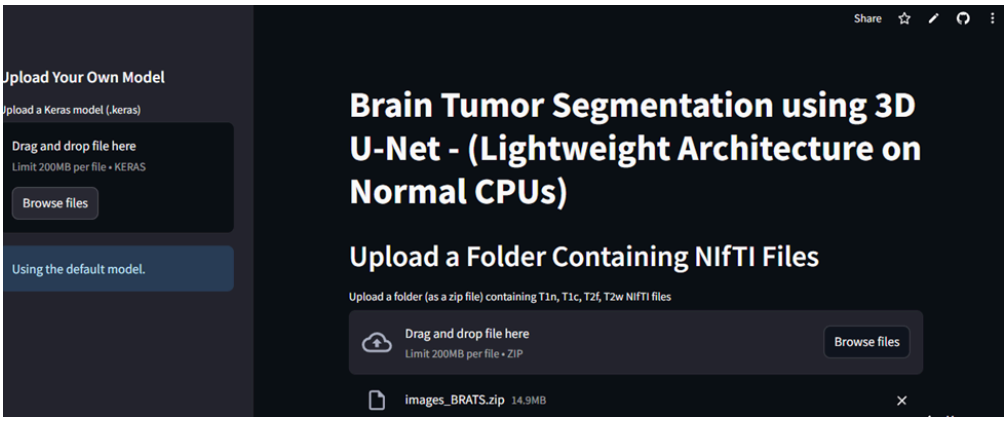
6.4 ***Creation of a requirements.txt***

A requirements.txt file is necessary for deploying your Streamlit app because it specifies the dependencies and libraries required to run your application. Create a **Requirements.txt** file and paste the code below.

```
streamlit
numpy
nibabel
tensorflow
scikit-learn
matplotlib
scipy
gdown
```

6.5 ***Deploy on Streamlit***

The application was successfully deployed on Streamlit by hosting the necessary files on a GitHub repository. The GitHub account was seamlessly integrated with Streamlit through their website, enabling a smooth deployment process. The deployed model can be accessed at the following link **HERE**.



Here is the **link** to the complete code for Phase 4.

Thank you!!. That brings us to the end of this tutorial.

Additional Resources

7

- **Link** to the full code repository on GitHub.

Protocol references

Adewole, M., Rudie, J.D., Gbadamosi, A.,Zhang, D., Raymond, C., Ajigbotoshso, J., Toyobo, O., Aguh, K., Omidiji, O., Akinola R., Suwaid, M.A., Emegoakor, A., Ojo, N., Kalaiwo, C., Babatunde, G., Ogunleye, A., Gbadamosi, Y., Iorpagher, K., Onuwaje M., Betiku B., Saluja, R., Menze, B., Baid, U., Bakas, S., Dako, F., Fatade A., Anazodo, U.C. (2024) Expanding the Brain Tumor Segmentation (BraTS) data to include African Populations (BraTS-Africa) (version 1) [Dataset]. The Cancer Imaging Archive. https://doi.org/10.7937/v8h6-8×67.

Adewole, M., Rudie, J. D., Gbadamosi, A., Zhang, D., Raymond, C., Ajigbotoshso, J., Toyobo, O., Aguh, K., Omidiji, O., Akinola, R., Suwaid, M. A., Emegoakor, A., Ojo, N., Kalaiwo, C., Babatunde, G., Ogunleye, A., Gbadamosi, Y., Iorpagher, K., Onuwaje, M., Betiku, B., … Anazodo, U. C. (2025). The BraTS-Africa Dataset: Expanding the Brain Tumor Segmentation (BraTS) Data to Capture African Populations. *Radiology. Artificial intelligence*, e240528. Advance online publication. https://doi.org/10.1148/ryai.240528

Bhattiprolu, S. (n.d.). *BraTa2020 Unet segmentation* . GitHub. Retrieved [January, 2025], from https://github.com/bnsreenu/python_for_microscopists/tree/master/231_234_BraTa2020_Unet_segmentation