gz4sbx8wf

Lightweight Brain Tumor Segmentation on Low-Resource Systems: A Step-by-Step Guide with 3D U-Net V.(gz4sbx8wf)

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

Protocol Info: Ayomide Oladele, Raymond Confidence, Dong Zhang, Charity Umoren, Aondana M Iorumbur, Anu Gbadamosi, Farouk Dako, Maruf Adewole, Udunna Anazodo . Lightweight Brain Tumor Segmentation on Low-Resource Systems: A Step-by-Step Guide with 3D U-Net. **protocols.io** **https://protocols.io/view/lightweight-brain-tumor-segmentation-on-low-resour-gz4sbx8wf**Version created by **MAI Lab**

Created: April 10, 2025

Last Modified: July 07, 2025

Protocol Integer ID: 218994

Keywords: Deep learning, MRI Scans, GPUs, CPUs, Brain Tumor Segmentation, Resource-Constrained Settings, Segmentation, Validation, 3D U-Net Architecture, Deployment

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.

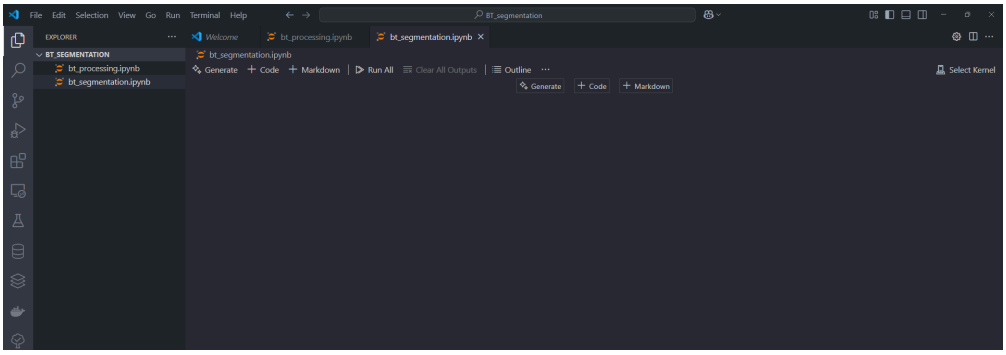
Step 3: Create a New Python Notebook

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**

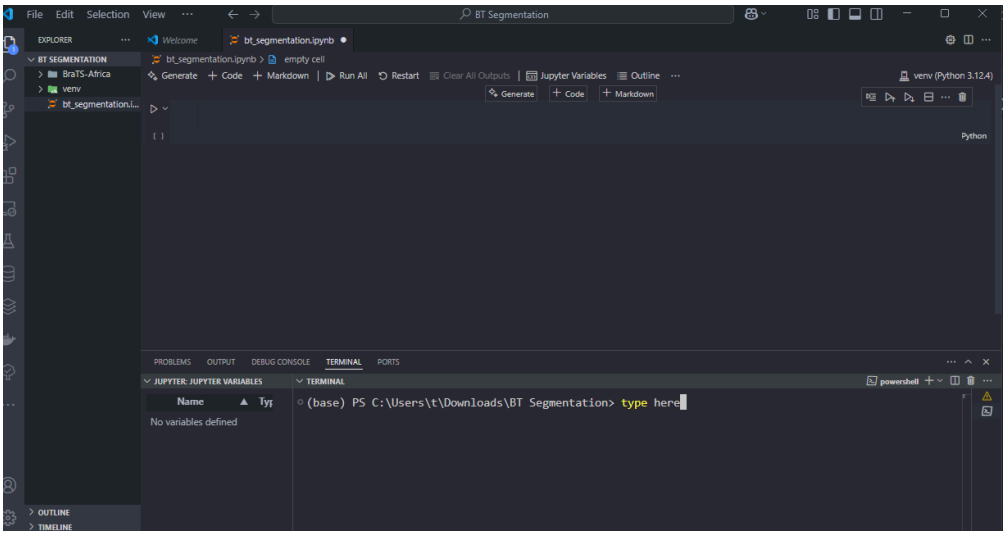
Follow the steps below to create a virtual environment and install the required dependencies using the following steps in **Visual Studio Code (VS Code)**:

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)

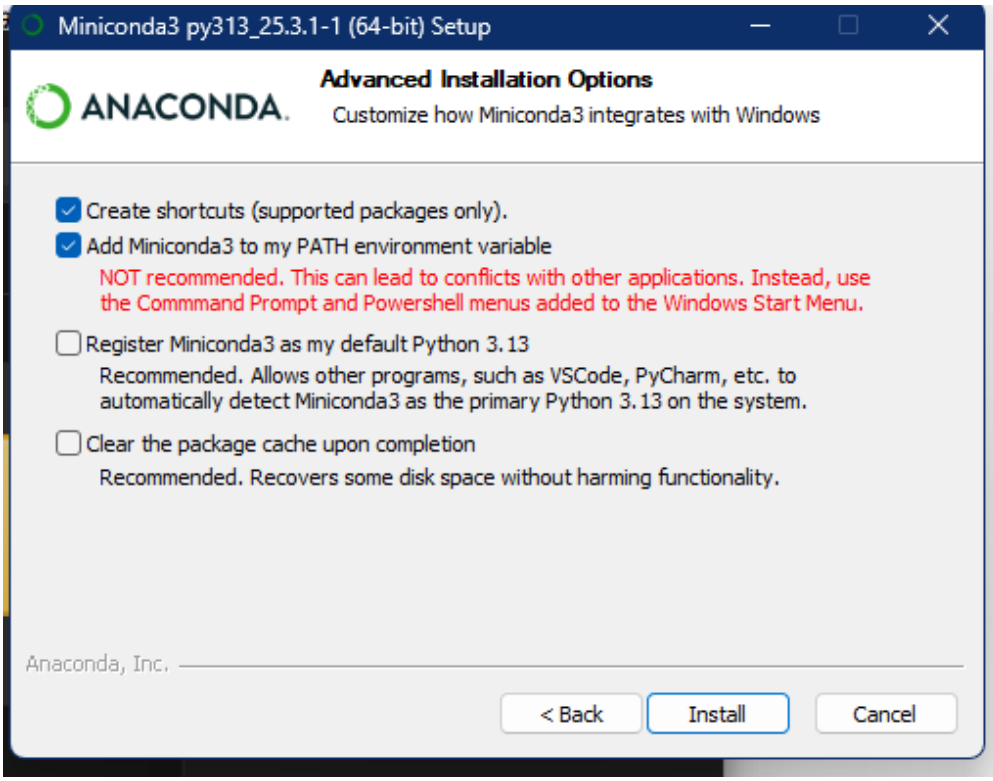
Go to this website to install Miniconda on your computer.

Warning: You must always add Miniconda to your system path.

For Windows Users:

During installation, ensure you check the option:

"Add Miniconda3 to my PATH environment variable"



For Mac Users:

After installation, you'll need to add Miniconda to your system's PATH environment variable manually. You can do this by:

1. Opening your shell configuration file: ~/.bashrc (for Bash) or ~/.zshrc (for Zsh)
2. Adding the following line: export PATH="/Users/your_username/miniconda3/bin:$PATH" (replace your_username with your actual username)
3. Saving and restarting your terminal: or running source ~/.bashrc (or source ~/.zshrc) to apply the changes

By adding Miniconda to your system path, you'll be able to use Conda commands in your terminal

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 psutil
```

then,

```
pip install numpy tensorflow keras segmentation_models_3D split-
folders monai medpy scipy
```

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*

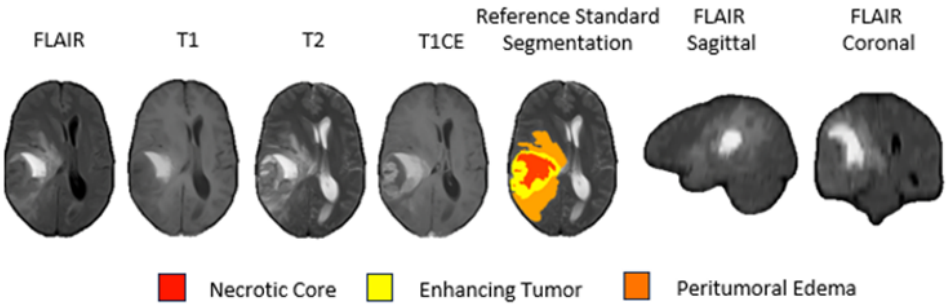


Illustration of the BraTS-Africa Dataset showing brain slices of the four MRI image contrasts and Segmentation masks of the three tumor sub-regions in one representative patie*nt (Adewole, et al., 2025).*

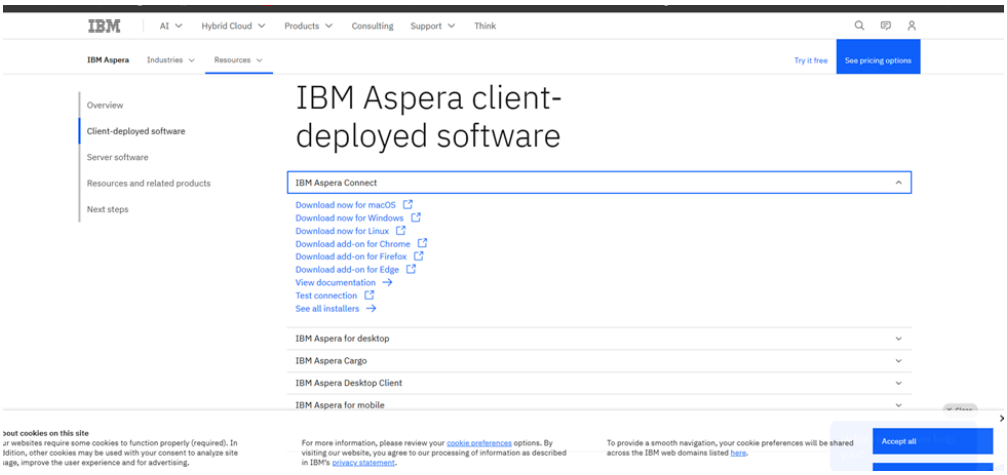
3.2 **B. Download the Dataset**

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

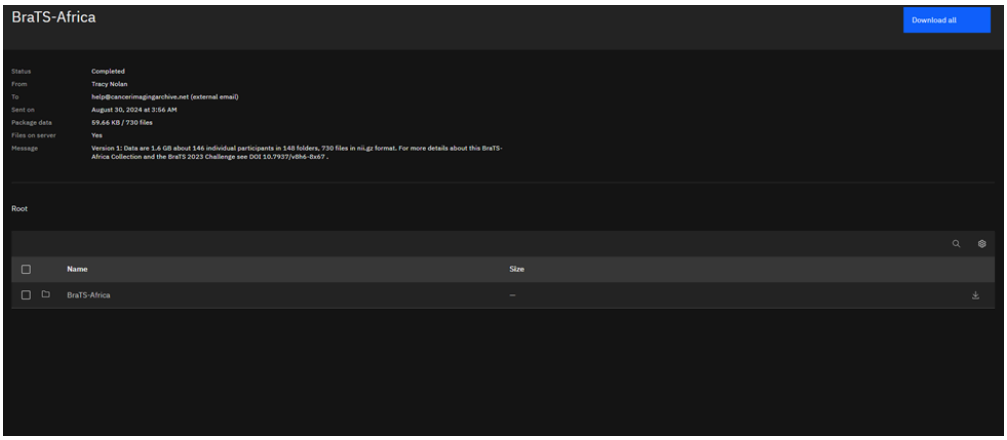
Version 1: Updated 2024/09/04

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

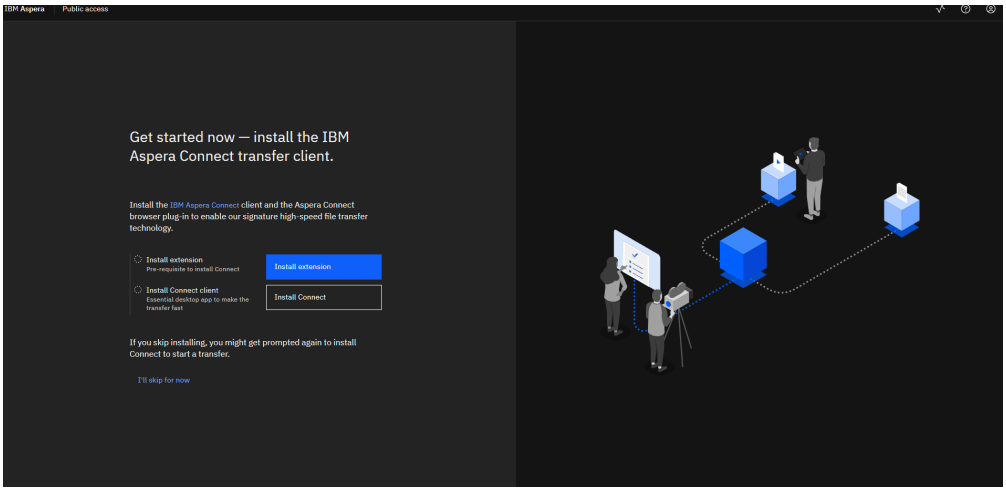
Before you click on the "*DOWNLOAD(1.6GB)"* button, you need to download IBM Aspera plugin by clicking on the *IBM-Aspera-Connect plugin*. This redirects you to the page below:



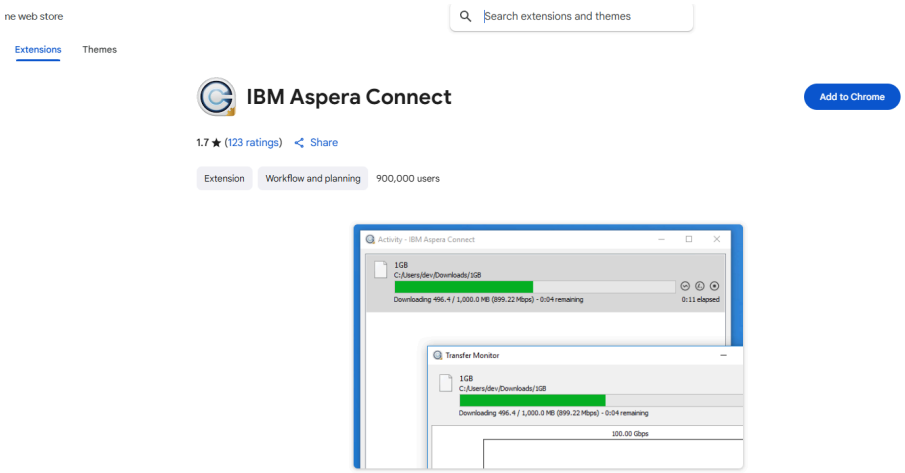
On this path, select the *Download now.* link with respect to your PC operating system. For, Windows, *Download now for Windows*. Then install the IBM Aspera connect application. Navigate back to the TCIA page and click on the *DOWNLOAD (1.6GB)* button,which will redirect you the page below (if a pop-up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

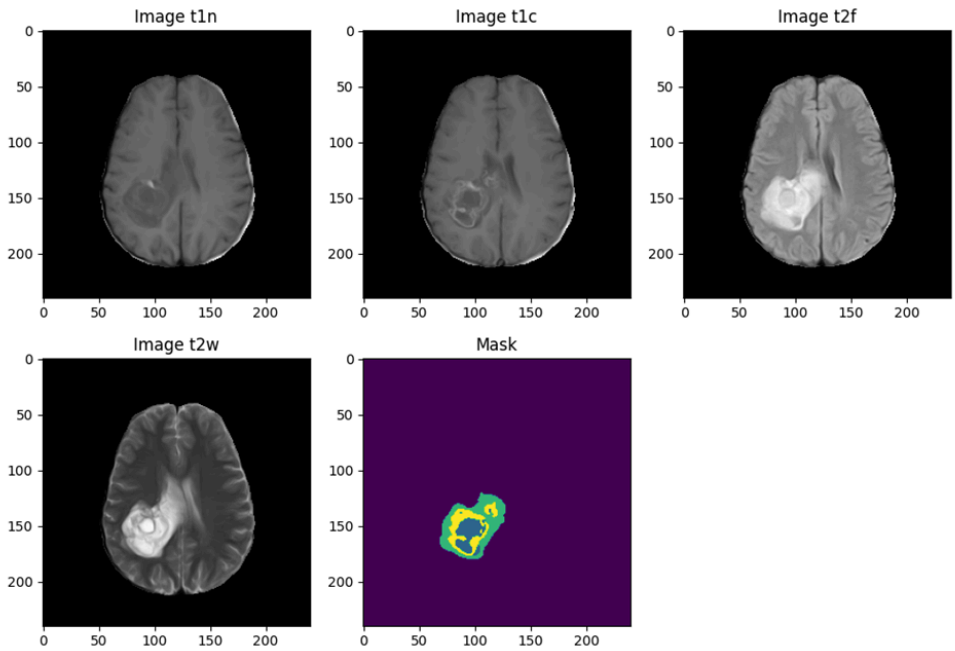
Note that:
0 → Background (Non-brain regions or healthy tissue)
1 → Tumor Core (Necrotic/Cancerous region)
2 → Edema (Swelling or fluid accumulation around the tumor)
3 → Enhancing Tumor (Active tumor region that enhances with contrast agents in MRI)

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)

```
import os
import numpy as np
import tensorflow as tf
from tensorflow.keras.layers import (
    Input, Conv3D, MaxPooling3D, concatenate,
    Conv3DTranspose, Dropout, Activation
)
from tensorflow.keras.models import Model
from tensorflow.keras.metrics import MeanIoU
from tensorflow.keras.callbacks import EarlyStopping,
ModelCheckpoint, ReduceLROnPlateau
from scipy.ndimage import rotate
from tensorflow import keras
import segmentation_models_3D as sm
import pandas as pd
import time
from medpy.metric.binary import hd95
import psutil
from datetime import datetime



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

#################################################################
# Model Architecture
#################################################################

def simple_unet_model(IMG_HEIGHT, IMG_WIDTH, IMG_DEPTH,
IMG_CHANNELS, num_classes):
    """3D U-Net model for brain tumor segmentation"""
    kernel_initializer = 'he_uniform'

    inputs = Input((IMG_HEIGHT, IMG_WIDTH, IMG_DEPTH,
IMG_CHANNELS))
    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])
    return model
```

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):
    """Extract random patch from image and mask"""
    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):
    """Apply gamma correction to image"""
    return np.clip(image ** gamma, 0, 1)

def augment_image(image, mask, is_training=True):
    """Apply random augmentations to image and mask"""
    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”.

```
def load_img(img_dir, img_list):
    """Load numpy images from directory"""
    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):
    """Generator for loading and augmenting image patches"""
    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

            # Convert masks to one-hot if they aren't already
            if len(Y.shape) == 4:  # Assuming shape [batch, H, W,
D]
                Y = tf.one_hot(Y.astype(np.int32), depth=4)  #
Convert to one-hot

            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 and HD95 score Function

This code defines two custom metrics for evaluating medical image segmentation models:
1. Dice Score: Calculates the Dice Score, a measure of overlap between predicted and ground truth masks. It tracks per-class scores but returns a scalar value for training.
2. HD95 Metric: Calculates the 95th percentile Hausdorff Distance (HD95), a measure of boundary accuracy between predicted and ground truth masks.
Both metrics are implemented as TensorFlow Keras metrics, allowing them to be used during model training and evaluation. The code includes error handling and configuration methods for serialization.
These metrics are particularly useful in medical image segmentation tasks, such as brain tumor segmentation, where accurate boundary delineation is crucial.

```
##################################################################
# Custom Metrics
##################################################################

class DiceScore(tf.keras.metrics.Metric):
    def __init__(self, num_classes, class_weights=None, smooth=1e-
6, name='dice_score', **kwargs):
        super(DiceScore, self).__init__(name=name, **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)
        self.total_dice = self.add_weight(name='total_dice',
initializer='zeros')
        self.count = self.add_weight(name='count',
initializer='zeros')

    def update_state(self, y_true, y_pred, sample_weight=None):
        # Convert predictions to class indices
        y_pred = tf.argmax(y_pred, axis=-1)

        # Convert one-hot y_true to class indices
        y_true = tf.argmax(y_true, axis=-1)

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

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

            dice = (2. * intersection + self.smooth) / (union +
self.smooth)
            weighted_dice = dice * self.class_weights[i]
            dice_scores.append(weighted_dice)

        mean_dice = tf.reduce_mean(dice_scores)
        self.total_dice.assign_add(mean_dice)
        self.count.assign_add(1.)
```

```
    def result(self):
        return self.total_dice / self.count

    def reset_states(self):
        self.total_dice.assign(0.)
        self.count.assign(0.)

def compute_hd95(pred, gt):
    """Compute 95th percentile Hausdorff Distance for binary
masks"""
    pred = pred.astype(bool)
    gt = gt.astype(bool)

    if not np.any(pred) or not np.any(gt):
        return np.nan  # Return nan for empty predictions or
ground truth

    return hd95(pred, gt)

def evaluate_per_region(model, data_loader, n_batches=10):
    """Evaluate model performance per tumor region"""
    regions = {
        'ET': {'label': 3, 'iou': MeanIoU(num_classes=2), 'dice':
DiceScore(num_classes=2), 'hd95': []},
        'NETC': {'label': 1, 'iou': MeanIoU(num_classes=2),
'dice': DiceScore(num_classes=2), 'hd95': []},
        'SNFH': {'label': 2, 'iou': MeanIoU(num_classes=2),
'dice': DiceScore(num_classes=2), 'hd95': []},
        'WT': {'label': [1,2,3], 'iou': MeanIoU(num_classes=2),
'dice': DiceScore(num_classes=2), 'hd95': []}  # Whole Tumor
    }

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

            for region_name, region_data in regions.items():
                if region_name == 'WT':
                    # Whole Tumor combines all regions
                    pred_bin = np.isin(pred_argmax,
region_data['label']).astype(np.float32)
                    true_bin = np.isin(mask_argmax,
region_data['label']).astype(np.float32)
                else:
```

```
                    # Individual regions
                    pred_bin = (pred_argmax ==
region_data['label']).astype(np.float32)
                    true_bin = (mask_argmax ==
region_data['label']).astype(np.float32)

                # Update metrics
                region_data['iou'].update_state(true_bin,
pred_bin)
                region_data['dice'].update_state(true_bin,
pred_bin)

                # Compute HD95 for each sample in batch
                for i in range(pred_bin.shape[0]):
                    hd95_value = compute_hd95(pred_bin[i],
true_bin[i])
                    if not np.isnan(hd95_value):
                        region_data['hd95'].append(hd95_value)

        except StopIteration:
            break

    # Compile results
    results = {}
    for region_name, region_data in regions.items():
        results[region_name] = {
            'IoU': region_data['iou'].result().numpy(),
            'Dice': region_data['dice'].result().numpy(),
            'HD95': np.mean(region_data['hd95']) if
region_data['hd95'] else np.nan,
            'HD95_std': np.std(region_data['hd95']) if
region_data['hd95'] else np.nan
        }

    return results
```

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.

```
def train_and_evaluate():
    # Initialize tracking variables
    epoch_times = []
    total_start = time.time()
    memory_usage = []
    cpu_usage = []

    # Function to get current memory usage
    def get_memory_usage():
        process = psutil.Process(os.getpid())
        return process.memory_info().rss / (1024 ** 2)  # Return
in MB

    # Function to get CPU usage
    def get_cpu_usage():
        return psutil.cpu_percent(interval=None)
    # Data paths
    DATA_ROOT = "glioma split data"
    train_img_dir = os.path.join(DATA_ROOT, "train/images/")
    train_mask_dir = os.path.join(DATA_ROOT, "train/masks/")
    val_img_dir = os.path.join(DATA_ROOT, "val/images/")
    val_mask_dir = os.path.join(DATA_ROOT, "val/masks/")
    test_img_dir = os.path.join(DATA_ROOT, "test/images/")
    test_mask_dir = os.path.join(DATA_ROOT, "test/masks/")

    # Get sorted file lists
    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)

    # Training parameters
    batch_size = 2
    patch_size = (96, 96, 96)
    epochs = 100

    # Data generators
    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,
is_training=False)

    # Model compilation
    model = simple_unet_model(IMG_HEIGHT=96, IMG_WIDTH=96,
IMG_DEPTH=96, IMG_CHANNELS=4, num_classes=4)

    # Loss and metrics
    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

    metrics = ['accuracy', sm.metrics.IOUScore(threshold=0.5),
DiceScore(num_classes=4)]
    optimizer = tf.keras.optimizers.Nadam(learning_rate=0.001,
clipnorm=1.0)

    model.compile(optimizer=optimizer, loss=total_loss,
metrics=metrics)

    # Callbacks
    checkpoint_path =
f"saved_model/3D_unet_{epochs}_epochs_{batch_size}_batch_patch_tra
ining.keras"

        # Custom callback to track epoch times and resource usage
    class ResourceTracker(keras.callbacks.Callback):
        def on_epoch_begin(self, epoch, logs=None):
            self.epoch_start_time = time.time()
            self.epoch_start_mem = get_memory_usage()
            self.epoch_start_cpu = get_cpu_usage()

        def on_epoch_end(self, epoch, logs=None):
            epoch_time = time.time() - self.epoch_start_time
            epoch_mem = get_memory_usage() - self.epoch_start_mem
            epoch_cpu = get_cpu_usage()

            epoch_times.append(epoch_time)
            memory_usage.append(epoch_mem)
            cpu_usage.append(epoch_cpu)

            logs['epoch_time'] = epoch_time
            logs['memory_usage'] = epoch_mem
```

```
            logs['cpu_usage'] = epoch_cpu

            print(f"\nEpoch {epoch+1} resource usage:")
            print(f" - Time: {epoch_time:.2f} seconds")
            print(f" - Memory delta: {epoch_mem:.2f} MB")
            print(f" - CPU usage: {epoch_cpu:.2f}%")

    callbacks = [
    EarlyStopping(monitor='val_loss', patience=10,
restore_best_weights=True),
    ModelCheckpoint(checkpoint_path, monitor='val_loss',
save_best_only=True, mode='min'),
    ReduceLROnPlateau(monitor='val_loss', factor=0.5, patience=5,
min_lr=1e-5),
    ResourceTracker()
    ]
    # Training
    start_time = time.time()
    history = model.fit(
        train_data,
        steps_per_epoch=len(train_img_list) // batch_size,
        epochs=epochs,
        validation_data=val_data,
        validation_steps=len(val_img_list) // batch_size,
        callbacks=callbacks,
        verbose=1
    )

    # Save training history
    history_df = pd.DataFrame(history.history)
    os.makedirs("model_history", exist_ok=True)



    # Calculate total training time
    total_end = time.time()
    total_time = total_end - total_start
    total_time_min = total_time / 60
    total_time_hours = total_time / 3600

    # Save training history with additional metrics
    history_df = pd.DataFrame(history.history)
    history_df['epoch_time'] = epoch_times[:len(history_df)]  #
Ensure lengths match
    history_df['memory_usage'] = memory_usage[:len(history_df)]
    history_df['cpu_usage'] = cpu_usage[:len(history_df)]
```

```
    os.makedirs("model_history", exist_ok=True)
    history_df.to_csv("model_history/training_history.csv",
index=False)

    # Print summary statistics
    print("\nTraining summary:")
    print(f"- Total training time: {total_time:.2f} seconds
({total_time_min:.2f} minutes, {total_time_hours:.2f} hours)")
    print(f"- Average epoch time: {np.mean(epoch_times):.2f} ±
{np.std(epoch_times):.2f} seconds")
    print(f"- Peak memory usage: {max(memory_usage):.2f} MB")
    print(f"- Average CPU usage: {np.mean(cpu_usage):.2f}% ±
{np.std(cpu_usage):.2f}%")
    print("\nTraining history and best model saved successfully.")



if __name__ == "__main__":
    train_and_evaluate()
```

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 numpy as np
import matplotlib.pyplot as plt
import os
import tensorflow as tf
from tensorflow.keras.models import load_model
from matplotlib.colors import ListedColormap

# Load the trained model
model_path =
"saved_model/3D_unet_100_epochs_2_batch_patch_training.keras"
model = load_model(model_path, compile=False)

# Data paths
DATA_ROOT = "glioma split data"
val_img_dir = os.path.join(DATA_ROOT, "val/images/")
val_mask_dir = os.path.join(DATA_ROOT, "val/masks/")

test_img_dir = os.path.join(DATA_ROOT, "test/images/")
test_mask_dir = os.path.join(DATA_ROOT, "test/masks/")

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

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 size (must match model input size)
patch_size = (96, 96, 96)
batch_size = 2

val_data = imageLoader(val_img_dir, val_img_list, val_mask_dir,
val_mask_list, batch_size, patch_size, is_training=False)


# Evaluation
print("\n=== Validation Set Evaluation ===")
val_results = evaluate_per_region(model, val_data, n_batches=20)
for region, metrics in val_results.items():
    print(f"{region}:")
    print(f"  IoU: {metrics['IoU']:.4f}")
    print(f"  Dice: {metrics['Dice']:.4f}")
```

```
    print(f"  HD95: {metrics['HD95']:.2f} ±
{metrics['HD95_std']:.2f}")

print("\n=== Test Set Evaluation ===")
test_data = imageLoader(test_img_dir, test_img_list,
test_mask_dir, test_mask_list,
                        batch_size, patch_size, is_training=False)
test_results = evaluate_per_region(model, test_data, n_batches=20)
for region, metrics in test_results.items():
    print(f"{region}:")
    print(f"  IoU: {metrics['IoU']:.4f}")
    print(f"  Dice: {metrics['Dice']:.4f}")
    print(f"  HD95: {metrics['HD95']:.2f} ±
{metrics['HD95_std']:.2f}")
```

5.4 ***Visualizing Results***

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

```
import numpy as np
import matplotlib.pyplot as plt
import os
import tensorflow as tf
from tensorflow.keras.models import load_model
from matplotlib.colors import ListedColormap

# Load the trained model
model_path =
"saved_model/3D_unet_100_epochs_2_batch_patch_training.keras"
model = load_model(model_path, compile=False)

# Data paths
DATA_ROOT = "glioma split data"
test_img_dir = os.path.join(DATA_ROOT, "test/images/")
test_mask_dir = os.path.join(DATA_ROOT, "test/masks/")

# Get a test sample
test_img_list = sorted([f for f in os.listdir(test_img_dir) if
f.endswith('.npy')])
test_mask_list = sorted([f for f in os.listdir(test_mask_dir) if
f.endswith('.npy')])

# Load one test sample
sample_idx = 1  # Change this to visualize different samples
test_image = np.load(os.path.join(test_img_dir,
test_img_list[sample_idx]))
test_mask = np.load(os.path.join(test_mask_dir,
test_mask_list[sample_idx]))

# Define patch size (must match model input size)
patch_size = (96, 96, 96)

# Function to extract center patch
def extract_center_patch(volume, patch_size):
    """Extract center patch from a 3D volume"""
    start_x = (volume.shape[0] - patch_size[0]) // 2
    start_y = (volume.shape[1] - patch_size[1]) // 2
    start_z = (volume.shape[2] - patch_size[2]) // 2

    return volume[
        start_x:start_x + patch_size[0],
        start_y:start_y + patch_size[1],
        start_z:start_z + patch_size[2]
    ]
```

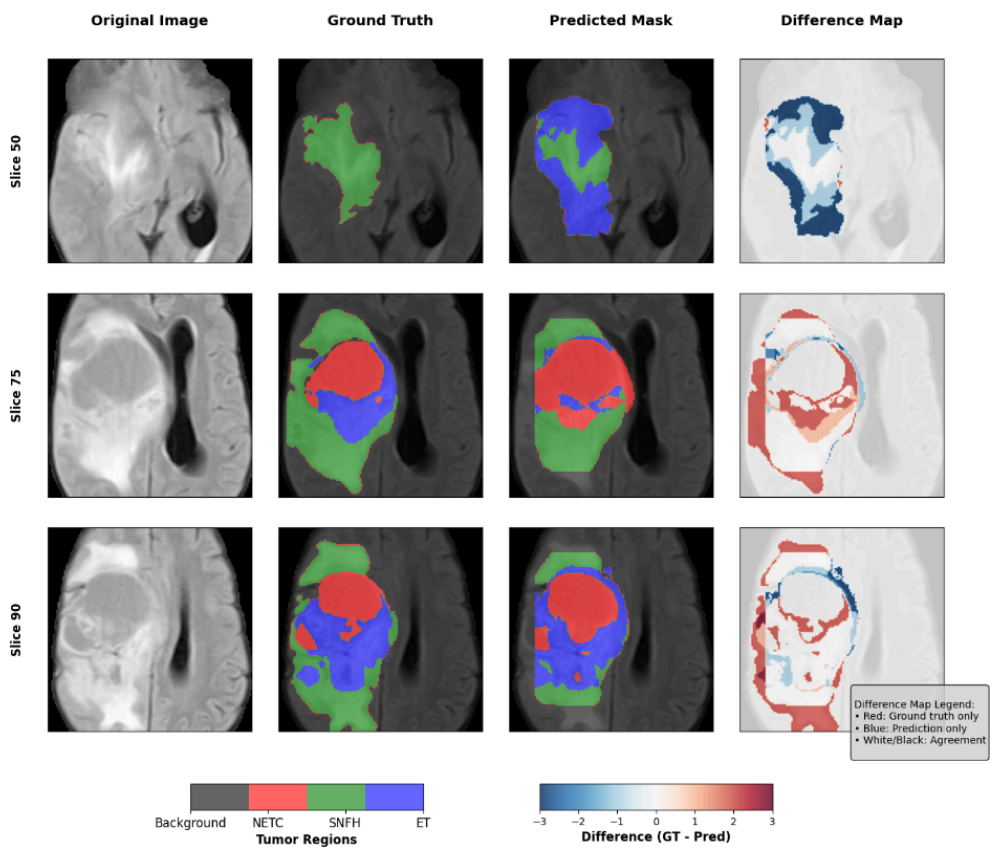
```
# Extract center patches - use full brain instead of patches for
better visualization
# Only use patches for model prediction
image_patch = extract_center_patch(test_image, patch_size)
mask_patch = extract_center_patch(test_mask, patch_size)

# For visualization, use the full brain volumes
viz_image = test_image
viz_mask = test_mask

input_image = np.expand_dims(image_patch, axis=0)
prediction = model.predict(input_image)
predicted_mask = np.argmax(prediction, axis=-1)[0]

# Convert ground truth to class indices if it's one-hot encoded
if len(mask_patch.shape) == 4:
    mask_patch = np.argmax(mask_patch, axis=-1)

if len(viz_mask.shape) == 4:
    viz_mask = np.argmax(viz_mask, axis=-1)

# Select specific slices to visualize
slice_numbers = [50, 75, 90]

# Create a colormap for the masks
colors = ['black', 'red', 'green', 'blue']  # 0: background, 1:
NETC, 2: SNFH, 3: ET
cmap = ListedColormap(colors)

# Create the figure - adjusted for better text visibility
fig, axes = plt.subplots(3, 4, figsize=(16, 12),
                        gridspec_kw={'wspace': 0.1, 'hspace':
0.15})
fig.suptitle(f'Brain Tumor Segmentation Results - Sample:
{test_img_list[sample_idx]}',
             fontsize=16, y=0.98, weight='bold')

# Define column titles
column_titles = ['Original Image', 'Ground Truth', 'Predicted
Mask', 'Difference Map']

# Add column titles
for j, title in enumerate(column_titles):
    axes[0, j].text(0.5, 1.15, title, transform=axes[0,
j].transAxes,
                    fontsize=14, weight='bold', ha='center',
va='bottom')
```

```
for i, slice_num in enumerate(slice_numbers):
    # Ensure slice number is within valid range
    if slice_num >= viz_image.shape[2] or slice_num < 0:
        slice_num = max(0, min(slice_num, viz_image.shape[2] - 1))

    # Get the slices from full brain
    img_slice = viz_image[:, :, slice_num, 0] if
len(viz_image.shape) == 4 else viz_image[:, :, slice_num]
    gt_slice = viz_mask[:, :, slice_num]

    # For predicted mask, we need to map patch coordinates to full
brain coordinates
    # Create a full-size prediction mask filled with zeros
    full_pred_mask = np.zeros(viz_image.shape[:3])

    # Calculate patch position in full brain
    start_x = (viz_image.shape[0] - patch_size[0]) // 2
    start_y = (viz_image.shape[1] - patch_size[1]) // 2
    start_z = (viz_image.shape[2] - patch_size[2]) // 2

    # Place the predicted patch in the full brain
    full_pred_mask[
        start_x:start_x + patch_size[0],
        start_y:start_y + patch_size[1],
        start_z:start_z + patch_size[2]
    ] = predicted_mask

    pred_slice = full_pred_mask[:, :, slice_num]

    # Rotate for proper orientation
    img_slice = np.rot90(img_slice)
    gt_slice = np.rot90(gt_slice)
    pred_slice = np.rot90(pred_slice)

    # Normalize image for better contrast
    img_slice = (img_slice - img_slice.min()) / (img_slice.max() -
img_slice.min())

    # Apply contrast enhancement
    img_slice = np.power(img_slice, 0.7)  # Gamma correction for
better visibility

    # Create difference map (ground truth - predicted)
    diff_slice = gt_slice.astype(float) - pred_slice.astype(float)

    # Row label
```

```
    axes[i, 0].text(-0.15, 0.5, f'Slice {slice_num}',
transform=axes[i, 0].transAxes,
                    fontsize=12, weight='bold', rotation=90,
va='center', ha='center')

    # Column 1: Original Image with enhanced contrast
    axes[i, 0].imshow(img_slice, cmap='gray', vmin=0, vmax=1)
    axes[i, 0].set_xticks([])
    axes[i, 0].set_yticks([])
    axes[i, 0].set_aspect('equal')

    # Column 2: Ground Truth overlay
    axes[i, 1].imshow(img_slice, cmap='gray', vmin=0, vmax=1)
    gt_display = axes[i, 1].imshow(gt_slice, cmap=cmap, vmin=0,
vmax=3, alpha=0.6)
    axes[i, 1].set_xticks([])
    axes[i, 1].set_yticks([])
    axes[i, 1].set_aspect('equal')

    # Column 3: Predicted Mask overlay
    axes[i, 2].imshow(img_slice, cmap='gray', vmin=0, vmax=1)
    pred_display = axes[i, 2].imshow(pred_slice, cmap=cmap,
vmin=0, vmax=3, alpha=0.6)
    axes[i, 2].set_xticks([])
    axes[i, 2].set_yticks([])
    axes[i, 2].set_aspect('equal')

    # Column 4: Difference Map (NEW)
    diff_cmap = ListedColormap(['blue', 'black', 'white', 'red'])
# -3 to +3 range
    axes[i, 3].imshow(img_slice, cmap='gray', vmin=0, vmax=1)
    diff_display = axes[i, 3].imshow(diff_slice, cmap='RdBu_r',
vmin=-3, vmax=3, alpha=0.8)
    axes[i, 3].set_xticks([])
    axes[i, 3].set_yticks([])
    axes[i, 3].set_aspect('equal')

# Add colorbars with better positioning and larger text
# Main colorbar for GT and Pred
cbar_ax1 = fig.add_axes([0.25, 0.02, 0.2, 0.03])  # Horizontal
colorbar
cbar1 = fig.colorbar(gt_display, cax=cbar_ax1,
orientation='horizontal', ticks=[0, 1, 2, 3])
cbar1.ax.set_xticklabels(['Background', 'NETC', 'SNFH', 'ET'],
fontsize=12)
cbar1.set_label('Tumor Regions', fontsize=12, weight='bold')
```

```
# Difference map colorbar
cbar_ax2 = fig.add_axes([0.55, 0.02, 0.2, 0.03])  # Horizontal
colorbar
cbar2 = fig.colorbar(diff_display, cax=cbar_ax2,
orientation='horizontal')
cbar2.set_label('Difference (GT - Pred)', fontsize=12,
weight='bold')
cbar2.ax.tick_params(labelsize=10)

# Add text annotations for better understanding
fig.text(0.02, 0.5, 'Axial Slices', rotation=90, fontsize=14,
weight='bold',
         va='center', ha='center')

# Add legend for difference map
legend_text = """
Difference Map Legend:
• Red: Ground truth only
• Blue: Prediction only
• White/Black: Agreement
"""
fig.text(0.82, 0.08, legend_text, fontsize=10, va='bottom',
ha='left',
         bbox=dict(boxstyle="round,pad=0.3",
facecolor="lightgray", alpha=0.8))

plt.tight_layout(rect=[0.05, 0.08, 0.95, 0.95])
plt.show()

# Print some statistics
print(f"\nVisualization Statistics:")
print(f"Original brain shape: {viz_image.shape}")
print(f"Patch size used for prediction: {patch_size}")
print(f"Slices displayed: {slice_numbers}")
print(f"Ground truth classes present: {np.unique(viz_mask)}")
print(f"Predicted classes present: {np.unique(predicted_mask)}")
```

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=(96, 96, 96)):
    # 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 nibabel as nib
import numpy as np
from sklearn.preprocessing import MinMaxScaler
import os
import tempfile
from tensorflow.keras.models import load_model
from matplotlib import pyplot as plt
import time
import gdown
from pathlib import Path
import tensorflow as tf
from scipy.ndimage import zoom

# Force CPU-only operation to avoid CUDA errors
os.environ['CUDA_VISIBLE_DEVICES'] = '-1'

# Set page config
st.set_page_config(page_title="Glioma Segmentation",
layout="wide")

# Initialize scaler
scaler = MinMaxScaler()
# Constants
MODEL_URL = "https://drive.google.com/uc?id=15DvYjyBHo-OgI-
oVPrruocNr_WUauQEk"
MODEL_DIR = "saved_model"
MODEL_PATH = os.path.join(MODEL_DIR,
"3D_unet_100_epochs_2_batch_patch_training.keras")

# Model expects (96, 96, 96, 4) input
TARGET_SHAPE = (96, 96, 96, 4)

# Ensure model directory exists
os.makedirs(MODEL_DIR, exist_ok=True)

# Download model from Google Drive (cache this to avoid repeated
downloads)
@st.cache_resource
def download_and_load_model():
    # Check if model already exists
    if not os.path.exists(MODEL_PATH):
        st.info("Downloading model from Google Drive... (This may
take a few minutes)")
        try:
            # Download using gdown
```

```
            gdown.download(MODEL_URL, MODEL_PATH, quiet=False)

            # Verify download
            if not os.path.exists(MODEL_PATH):
                raise FileNotFoundError("Model download failed")

        except Exception as e:
            st.error(f"Failed to download model: {str(e)}")
            return None

    # Load the model
    try:
        # Disable TensorFlow logging
        tf.get_logger().setLevel('ERROR')

        model = load_model(MODEL_PATH, compile=False)
        return model
    except Exception as e:
        st.error(f"Error loading model: {str(e)}")
        return None

# Function to process uploaded files
def process_uploaded_files(uploaded_files):
    modalities = {}

    for uploaded_file in uploaded_files:
        file_name = uploaded_file.name.lower()

        # Save to temporary file
        with tempfile.NamedTemporaryFile(delete=False,
suffix='.nii.gz') as tmp_file:
            tmp_file.write(uploaded_file.getbuffer())
            tmp_path = tmp_file.name

        try:
            # Load NIfTI file
            img = nib.load(tmp_path)
            img_data = img.get_fdata()

            # Scale the data
            img_data = scaler.fit_transform(img_data.reshape(-1,
img_data.shape[-1])).reshape(img_data.shape)

            # Determine modality
            if 't1n' in file_name:
                modalities['t1n'] = img_data
            elif 't1c' in file_name:
```

```
                modalities['t1c'] = img_data
            elif 't2f' in file_name:
                modalities['t2f'] = img_data
            elif 't2w' in file_name:
                modalities['t2w'] = img_data
            elif 'seg' in file_name:
                modalities['mask'] = img_data.astype(np.uint8)

        except Exception as e:
            st.error(f"Error processing file {uploaded_file.name}:
{str(e)}")
        finally:
            # Clean up temporary file
            if os.path.exists(tmp_path):
                os.unlink(tmp_path)

    return modalities

# Function to prepare input for model
def prepare_input(modalities):
    # Check we have all required modalities
    required = ['t1n', 't1c', 't2f', 't2w']
    if not all(m in modalities for m in required):
        return None, None

    # Combine modalities
    combined = np.stack([
        modalities['t1n'],
        modalities['t1c'],
        modalities['t2f'],
        modalities['t2w']
    ], axis=3)

    # First crop to original expected size (128, 128, 128, 4)
    combined = combined[56:184, 56:184, 13:141, :]
    original_shape = combined.shape

    # Then resize to model's expected input shape (64, 64, 64, 4)
    # Using simple downsampling for CPU compatibility
    downsampled = combined[::2, ::2, ::2, :]

    return downsampled, original_shape, combined

# Function to make prediction
def make_prediction(model, input_data):
    # Add batch dimension
    input_data = np.expand_dims(input_data, axis=0)
```

```
    # Make prediction
    prediction = model.predict(input_data, verbose=0)
    prediction_argmax = np.argmax(prediction, axis=4)[0, :, :, :]

    return prediction_argmax

# Function to upsample prediction to original size
def upsample_prediction(prediction, target_shape):
    # Simple nearest-neighbor upsampling for CPU compatibility
    zoom_factors = (
        target_shape[0] / prediction.shape[0],
        target_shape[1] / prediction.shape[1],
        target_shape[2] / prediction.shape[2]
    )
    return zoom(prediction, zoom_factors, order=0)  # order=0 for
nearest-neighbor

# Function to visualize results
def visualize_results(original_data, prediction,
ground_truth=None):
    # Select a modality to display (using T1c here)
    image_data = original_data[:, :, :, 1]  # T1c is the second
channel

    # Select some slices to display
    slice_indices = [50, 75, 90]

    # Create figure
    fig, axes = plt.subplots(3, 3 if ground_truth is not None else
2,
                             figsize=(10, 6))

    for i, slice_idx in enumerate(slice_indices):
        # Rotate images for better visualization
        img_slice = np.rot90(image_data[:, :, slice_idx])
        pred_slice = np.rot90(prediction[:, :, slice_idx])

        # Plot input image
        axes[i, 0].imshow(img_slice, cmap='gray')
        axes[i, 0].set_title(f'Input Image - Slice {slice_idx}')
        axes[i, 0].axis('off')

        # Plot prediction
        axes[i, 1].imshow(pred_slice)
        axes[i, 1].set_title(f'Prediction - Slice {slice_idx}')
        axes[i, 1].axis('off')
```

```
        # Plot ground truth if available
        if ground_truth is not None:
            gt_slice = np.rot90(ground_truth[:, :, slice_idx])
            axes[i, 2].imshow(gt_slice)
            axes[i, 2].set_title(f'Ground Truth - Slice
{slice_idx}')
            axes[i, 2].axis('off')

    plt.tight_layout()
    return fig

# Main app
def main():
    st.title("3D Glioma Segmentation with U-Net")
    st.write("Upload MRI scans in NIfTI format for glioma
segmentation")

    with st.expander("How to use this app"):
        st.markdown("""
        1. Upload **all four MRI modalities** (T1n, T1c, T2f, T2w)
as NIfTI files (.nii.gz)
        2. Optionally upload a segmentation mask for comparison
(must contain 'seg' in filename)
        3. Click 'Process and Predict' button
        4. View the segmentation results

        **Note:**
        - The first run will download the model (~100MB) which may
take a few minutes.
        - This version runs on CPU and may be slower than GPU-
accelerated versions.
        """)

    # Load model (this will trigger download if needed)
    model = download_and_load_model()

    if model is None:
        st.error("Failed to load model. Please check the error
message above.")
        return

    # File uploader
    uploaded_files = st.file_uploader(
        "Upload MRI scans (NIfTI format)",
        type=['nii', 'nii.gz'],
        accept_multiple_files=True
```

```
    )

    if uploaded_files and len(uploaded_files) >= 4:
        if st.button("Process and Predict"):
            with st.spinner("Processing files..."):
                # Process uploaded files
                modalities =
process_uploaded_files(uploaded_files)

                # Prepare input (returns downsampled, original
shape, and original data)
                input_data, original_shape, original_data =
prepare_input(modalities)

                if input_data is None:
                    st.error("Could not prepare input data. Please
ensure you've uploaded all required modalities.")
                    return

                # Get ground truth if available
                ground_truth = modalities.get('mask', None)
                if ground_truth is not None:
                    ground_truth = ground_truth[56:184, 56:184,
13:141]
                    ground_truth[ground_truth == 4] = 3  #
Reassign label 4 to 3

                # Make prediction
                with st.spinner("Making prediction (this may take
a few minutes on CPU)..."):
                    start_time = time.time()
                    prediction = make_prediction(model,
input_data)

                    # Upsample prediction to original size
                    prediction = upsample_prediction(prediction,
original_shape[:3])

                    # Convert prediction to int32 for NIfTI
compatibility
                    prediction = prediction.astype(np.int32)

                    elapsed_time = time.time() - start_time

                st.success(f"Prediction completed in
{elapsed_time:.2f} seconds")
```

```
                # Visualize results using original size data
                fig = visualize_results(original_data, prediction,
ground_truth)
                st.pyplot(fig)

                # Provide download option for prediction
                st.subheader("Download Prediction")

                # Create a temporary NIfTI file for download
                with tempfile.NamedTemporaryFile(suffix='.nii.gz',
delete=False) as tmp_file:
                    # Convert prediction to NIfTI with explicit
dtype
                    pred_img = nib.Nifti1Image(prediction,
affine=np.eye(4), dtype=np.int32)
                    nib.save(pred_img, tmp_file.name)

                    # Read back the file data
                    with open(tmp_file.name, 'rb') as f:
                        pred_data = f.read()

                    # Clean up
                    os.unlink(tmp_file.name)

                st.download_button(
                    label="Download Segmentation (NIfTI)",
                    data=pred_data,
                    file_name="glioma_segmentation.nii.gz",
                    mime="application/octet-stream"
                )
    elif uploaded_files and len(uploaded_files) < 4:
        st.warning("Please upload all four modalities (T1n, T1c,
T2f, T2w)")

if __name__ == "__main__":
    main()
```

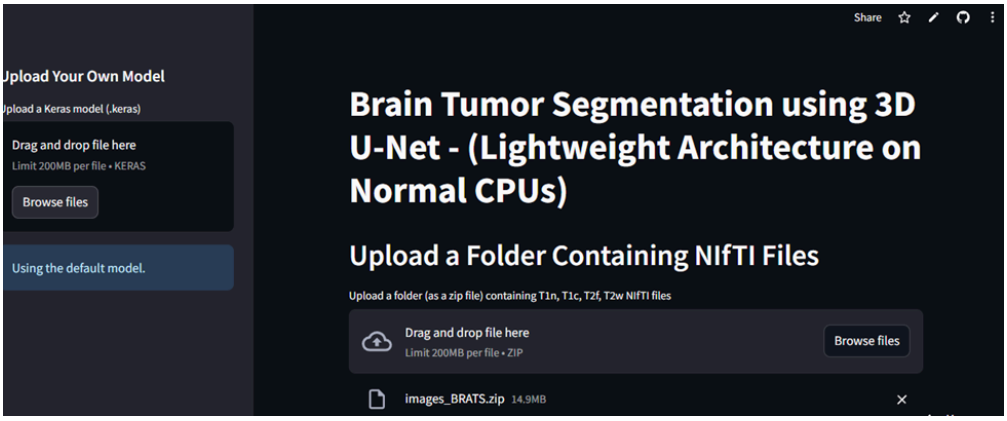
6.4 ***Creation of a requirements.txt***

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