# Seminar paper
**Brain Tumor Segmentation**

## Table of Contents

## 1. INTRODUCTION

### IMAGE SEGMENTATION

Image segmentation is a fundamental process in image processing, involving the partitioning of an image into multiple segments to simplify or change its representation for easier analysis. This technique is widely applied in various domains, including medical imaging, computer vision, and pattern recognition. The process of image segmentation has been the subject of extensive research, leading to the development of diverse techniques and algorithms to address its challenges. Brain tumor segmentation is a critical task in medical imaging for accurate diagnosis and treatment planning. Various techniques have been explored for this purpose, including deep learning models, image processing approaches, and segmentation algorithms. Deep learning methods, such as deep neural networks and deep generative models, have shown success in brain disease diagnosis [1]. Additionally, automated segmentation methods have been developed, allowing rapid identification of brain and tumor tissue with accuracy comparable to manual segmentation, making them practical for clinical use [2]. Furthermore, the effectiveness of mean shift clustering and content-based active segmentation algorithms in brain tumor segmentation has been elaborated [3].

The use of advanced technologies like deformable registration, diffusion reaction modeling, and supervised classification using support vector machines has been investigated for glioma image segmentation [4]. Additionally, the application of deep interactive geodesic frameworks for 3D brain tumor segmentation from FLAIR images has been validated [5]. Furthermore, the use of deep feature fusion for segmentation of human motion injury ultrasound medical images has been explored, demonstrating a maximum segmentation error rate of 2.68% [6].

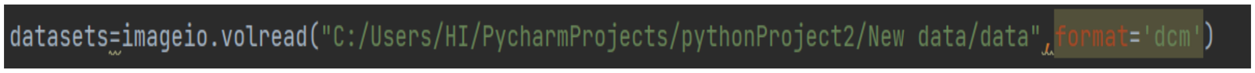
In the context of medical image segmentation, the selection of appropriate kernel functions for support vector machine classifiers has been shown to yield high correct rates for segmentation of MR brain images [7]. Furthermore, the use of deep learning-based medical image segmentation methods has been reviewed, emphasizing the importance of accurate segmentation results to meet clinical requirements [8]. Additionally, the segmentation of 3-D tumor structures from MRI has been addressed as a challenging problem due to the variability of tumor geometry and intensity patterns [9]. This paper focuses on performing manual segmentation of the brain tumors by performing automatic segmentation using traditional image processing with OPENCV; and performing automatic segmentation using Deep learning method.

## 2. Automatic Segmentation Using Traditional Image Processing

### 2.1 Methodology

The following methods were used to carry out automatic segmentation using traditional image processing methods to obtain segmented images of the brain tumor in python.

- **Reading the image**: The dataset was read in DICOM format using the "imageio" library as shown in figure 1.



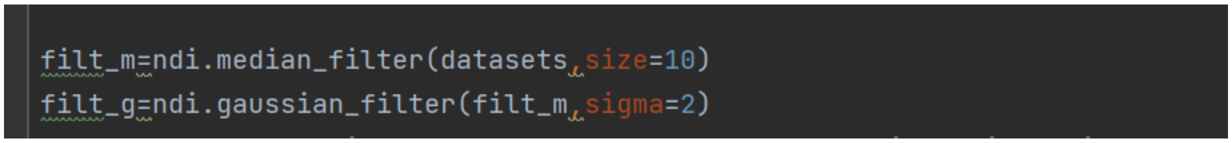
```
datasets=imageio.volread("C:/Users/HI/PycharmProjects/pythonProject2/New data/data",format='dcm')
```

Fig 1: Reading The Brain Dataset

- **Applying filters**: In image processing, filters are mathematical operations applied to images to enhance or suppress certain features, information, or noise. Filters are widely used for tasks such as smoothing, sharpening, edge detection, and noise reduction. They can be linear or nonlinear, and they play a crucial role in various image processing applications. Here are some of the filters used in the segmentation brain tumor.

Gaussian Filter**:** The Gaussian filter is commonly used for smoothing and blurring. It is effective in reducing noise and removing high-frequency details from an image.

Median Filter**:** The median filter is a nonlinear filter effective in removing salt-and-pepper noise. It replaces each pixel with the median value in its local neighborhood, making it robust to outliers. The application of the filters is shown in figure 2.



```
filt_m=ndi.median_filter(datasets,size=10)
filt_g=ndi.gaussian_filter(filt_m,sigma=2)
```

Fig 2 : Applying Filters

- **Plotting Histograms:** A histogram is a graphical representation of the distribution of pixel intensities in an image. It provides a visual summary of the tonal distribution of an image, helping to analyze the image's contrast, brightness, and overall characteristics. The histogram plot of the distribution of intensities in the datasets as well as the code implementation is shown in figure 3 and figure 4 respectively.

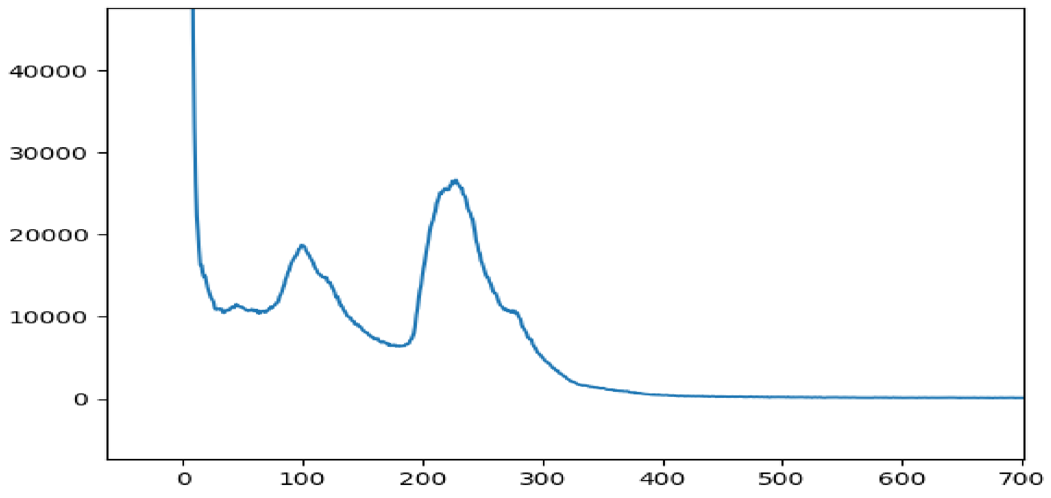
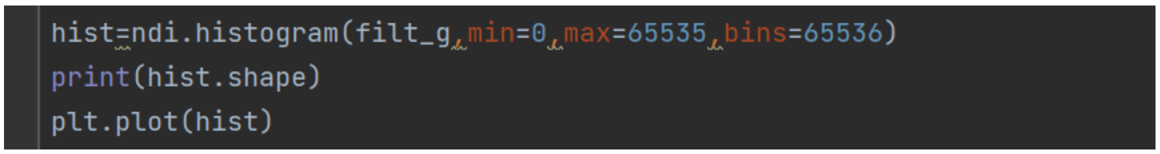


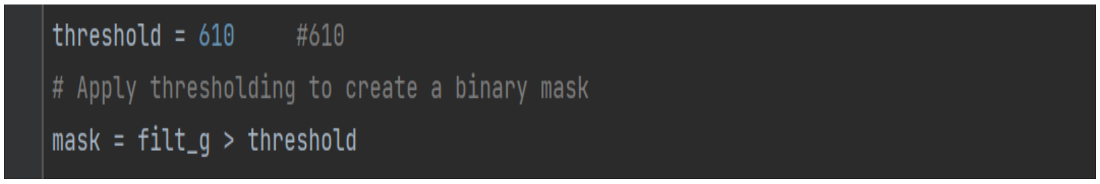
Fig 3 : Histogram Plot



```
hist=ndi.histogram(filt_g,min=0,max=65535,bins=65536)
print(hist.shape)
plt.plot(hist)
```

Fig 4 : Plotting The Histogram

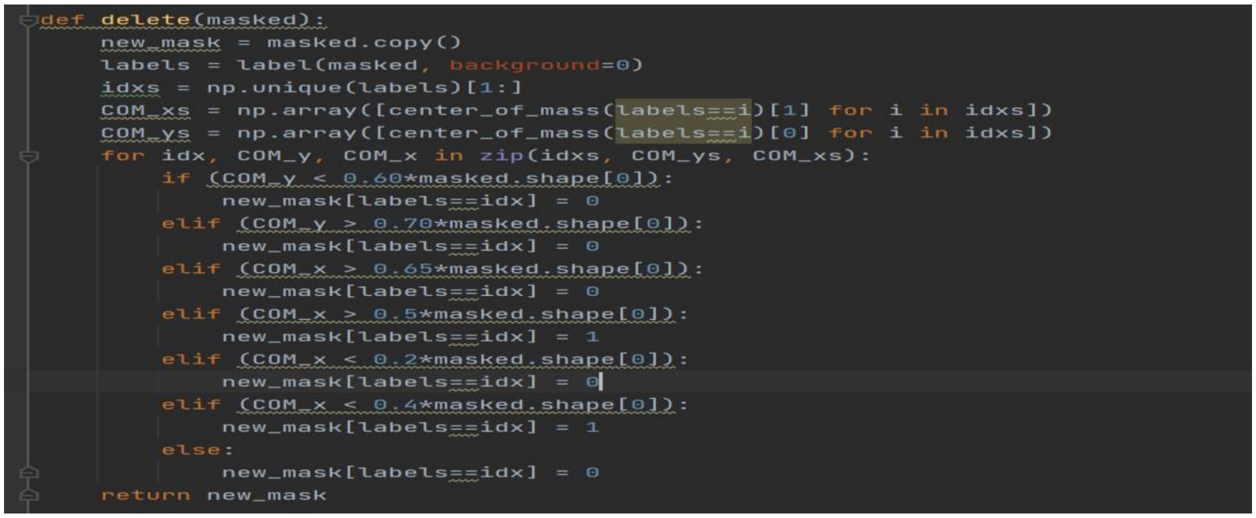
- **Applying Thresholding**: It involves dividing an image into two or more regions based on intensity values. The goal is to simplify the representation of an image and highlight or separate objects or regions of interest. The code implementation for the application of threshold is shown in figure 5.



```
threshold = 610     #610
# Apply thresholding to create a binary mask
mask = filt_g > threshold
```

Fig 5: Applying Thresholding

- **Defining a Delete Function:** The delete function is defined to calculate the center of mass of an image using unique labels assigned to individual areas in the masked image and then delete the unwanted brain regions displaying in the masked image leaving behind only the masked tumors. The delete function is shown in figure 6.



```
def delete(masked):
    new_mask = masked.copy()
    labels = label(masked, background=0)
    idxs = np.unique(labels)[1:]
    COM_xs = np.array([center_of_mass(labels==i)[1] for i in idxs])
    COM_ys = np.array([center_of_mass(labels==i)[0] for i in idxs])
    for idx, COM_y, COM_x in zip(idxs, COM_ys, COM_xs):
        if (COM_y < 0.60*masked.shape[0]):
            new_mask[labels==idx] = 0
        elif (COM_y > 0.70*masked.shape[0]):
            new_mask[labels==idx] = 0
        elif (COM_x > 0.65*masked.shape[0]):
            new_mask[labels==idx] = 0
        elif (COM_x > 0.5*masked.shape[0]):
            new_mask[labels==idx] = 1
        elif (COM_x < 0.2*masked.shape[0]):
            new_mask[labels==idx] = 0
        elif (COM_x < 0.4*masked.shape[0]):
            new_mask[labels==idx] = 1
        else:
            new_mask[labels==idx] = 0
    return new_mask
```

Fig. 6 : Delete function

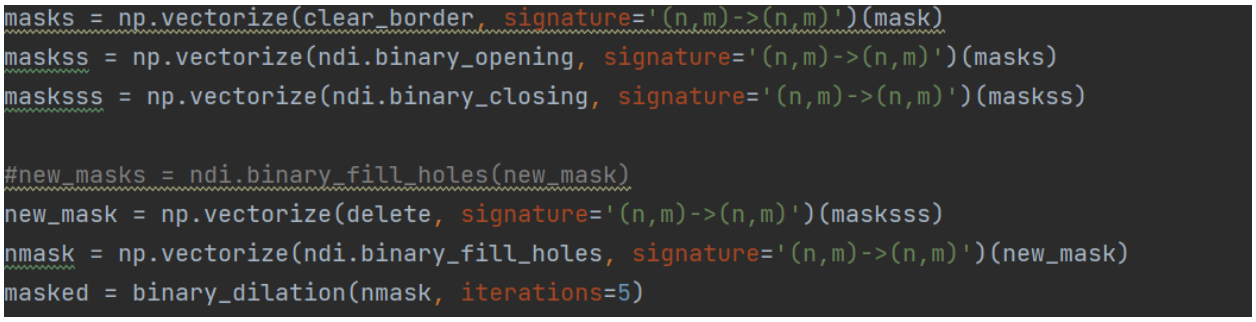
- **Applying Morphological Operations:** Morphological operations in image processing involve the manipulation of the shapes of objects within an image. These operations are commonly used for tasks such as noise removal, object detection, and image segmentation. The basic morphological operations include dilation, erosion, opening, and closing. The following operations were used:

  Binary Fill Hole: The binary fill holes operation is used to fill internal holes within connected components in a binary image. It can be achieved by applying morphological operations like binary dilation followed by binary erosion

  Binary Closing: Binary closing is a morphological operation that combines dilation followed by erosion. It is useful for smoothing object boundaries and closing small gaps in the foreground.

  Clear Borders: Clearing borders is a process of removing connected components that touch the image borders. This is often done to eliminate artifacts or incomplete objects near the image boundaries.

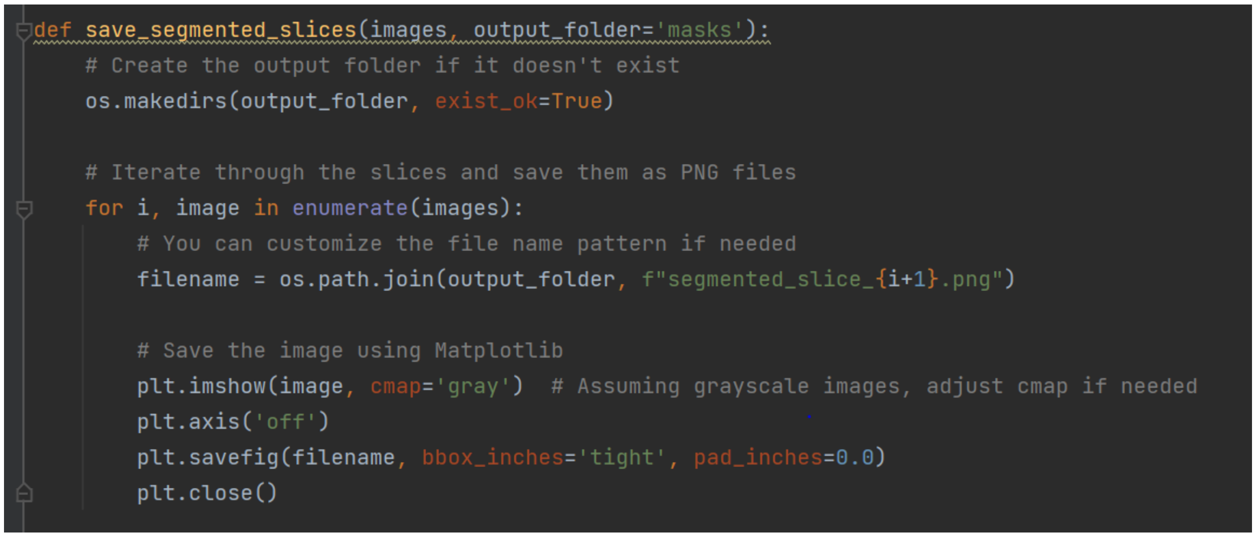
  Binary Opening: Binary opening is a morphological operation in image processing that involves performing an erosion operation followed by a dilation operation on a binary image. Like binary closing, binary opening is a fundamental operation in morphological processing and is useful for various tasks, including removing small objects and smoothing the boundaries of larger objects. The application of morphological functions using the “np.vectorize” function to apply the operations to each slice in the masked image is shown in figure 7.



```
masks = np.vectorize(clear_border, signature='(n,m)->(n,m)')(mask)
maskss = np.vectorize(ndi.binary_opening, signature='(n,m)->(n,m)')(masks)
masksss = np.vectorize(ndi.binary_closing, signature='(n,m)->(n,m)')(maskss)

#new_masks = ndi.binary_fill_holes(new_mask)
new_mask = np.vectorize(delete, signature='(n,m)->(n,m)')(masksss)
nmask = np.vectorize(ndi.binary_fill_holes, signature='(n,m)->(n,m)')(new_mask)
masked = binary_dilation(nmask, iterations=5)
```

Fig 7 : Applying Morphological Operations

- **Saving Segmented Brain Tumor:** A function was defined to save the segmented brain tumor slices to an output folder. This code implementation for saving segmented brain tumor slices as well as the results of the segmentation in axial view is shown in figure 8 and 9.



```
def save_segmented_slices(images, output_folder='masks'):
    # Create the output folder if it doesn't exist
    os.makedirs(output_folder, exist_ok=True)

    # Iterate through the slices and save them as PNG files
    for i, image in enumerate(images):
        # You can customize the file name pattern if needed
        filename = os.path.join(output_folder, f"segmented_slice_{i+1}.png")

        # Save the image using Matplotlib
        plt.imshow(image, cmap='gray')  # Assuming grayscale images, adjust cmap if needed
        plt.axis('off')
        plt.savefig(filename, bbox_inches='tight', pad_inches=0.0)
        plt.close()
```

Fig. 8 : Saving the function

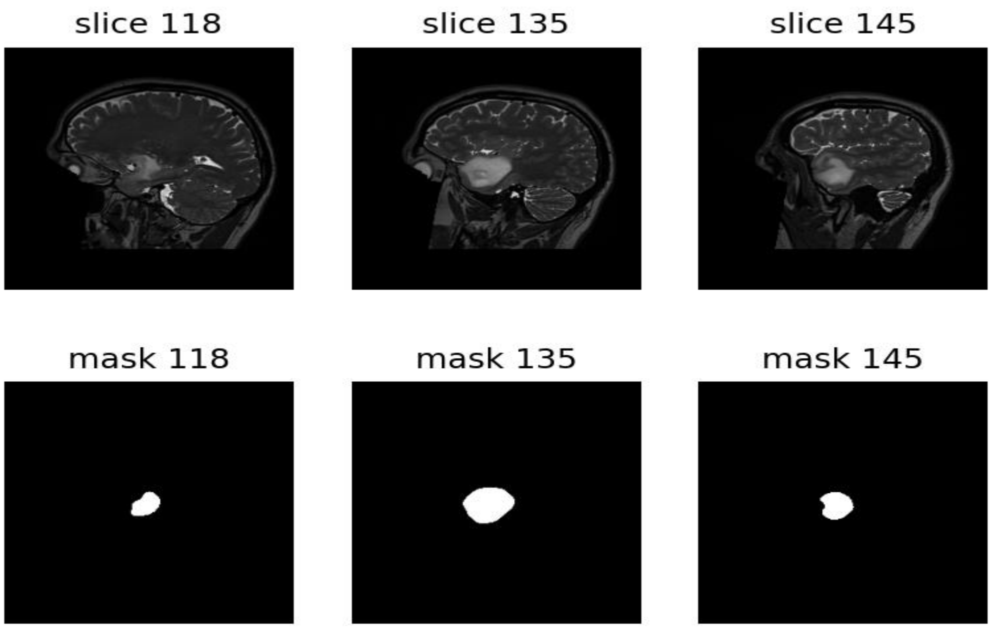
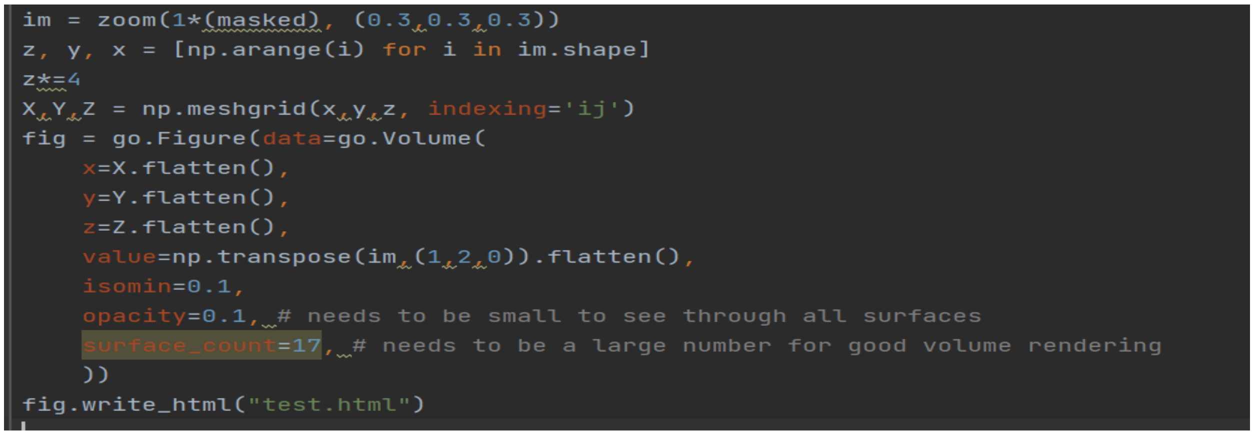


Fig 9: Segmented Tumor Results

- **Plotting 3D Mesh:** The resulting 3d mesh of the segmented brain tumor was plotted in the axial view using the code implementation shown in figure 10 as well as the result shown in figure 11.



```
im = zoom(1*(masked), (0.3,0.3,0.3))
z, y, x = [np.arange(i) for i in im.shape]
z*=4
X,Y,Z = np.meshgrid(x,y,z, indexing='ij')
fig = go.Figure(data=go.Volume(
    x=X.flatten(),
    y=Y.flatten(),
    z=Z.flatten(),
    value=np.transpose(im,(1,2,0)).flatten(),
    isomin=0.1,
    opacity=0.1, # needs to be small to see through all surfaces
    surface_count=17, # needs to be a large number for good volume rendering
    ))
fig.write_html("test.html")
```

Fig 10: Plotting 3d mesh

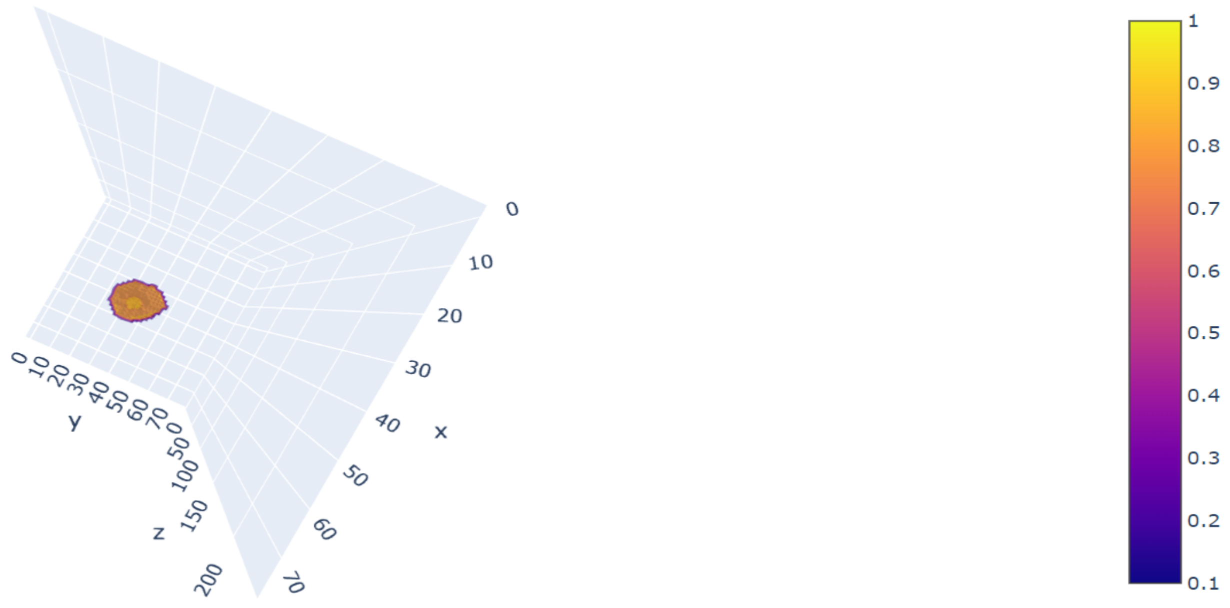


Fig 11: 3D mesh

# 3. AUTOMATIC SEGMENTATION USING DEEP LEARNING

## 3.1 Methodology

The following methods were used to carry out automatic segmentation using deep learning methods to obtain segmented images of the brain tumors in python.

### UNET ARCHITECTURE MODEL

The UNet architecture has gained significant attention in the field of image segmentation due to its effectiveness in various applications. UNet, a convolutional neural network (CNN) architecture, is widely utilized for semantic segmentation tasks, particularly in medical imaging, remote sensing, and computer vision. The UNet architecture consists of a contracting path to capture context and a symmetric expanding path for precise localization. The architecture's ability to capture fine-grained details while maintaining spatial information has made it a popular choice for segmentation tasks. The study by [10] introduces UNet++ as a redesign of skip connections to exploit multiscale features in image segmentation. The research demonstrates the consistent outperformance of UNet++ over baseline models across different datasets and backbone architectures, particularly in medical image segmentation. Additionally, the study by [11] highlights the application of UNet-inspired architectures for multi-planar 3D knee MRI segmentation, showcasing the adaptability of UNet variants in diverse imaging modalities. Furthermore, the work by [12] introduces MST-UNet, a modified Swin Transformer for water bodies' mapping using Sentinel-2 images, emphasizing the significance of UNet-based architectures in remote sensing applications. The study by [13] underscores the state-of-the-art performance of UNet and its variations in medical image segmentation, particularly in breast mass segmentation using deep learning. The U-Net architecture is characterized by its U-shaped structure, which includes a contracting path, a bottleneck, and an expansive path. Here is a high-level overview of the U-Net architecture:

- **Contracting Path (Encoder):**
    - The top part of the U-shaped network consists of a series of convolutional and max-pooling layers that reduce the spatial resolution of the input image while increasing the number of feature channels.
    - Each block in the contracting path typically includes two 3x3 convolutional layers followed by rectified linear unit (ReLU) activation functions and a 2x2 max-pooling layer for downsampling.
    - Downsampling helps capture high-level features at a coarser scale.
- **Bottleneck:**
    - The bottleneck is located at the bottom of the U-shape and represents the transition from the contracting path to the expansive path.
    - It typically consists of a set of convolutional layers that maintain a high-dimensional representation of the input, capturing the most essential features.

- **Expansive Path (Decoder):**
    - The bottom part of the U-shape is the expansive path, which involves up sampling the feature maps to gradually reconstruct the spatial resolution of the original input image.
    - Each block in the expansive path usually includes two 3x3 convolutional layers followed by ReLU activation functions and a 2x2 up-convolution (transposed convolution or up sampling) to increase spatial resolution.
    - Skip connections are introduced, connecting corresponding layers from the contracting path to the expansive path. These skip connections help preserve fine details during up sampling.
- **Final Layer:**
    - The final layer of the network typically consists of a 1x1 convolutional layer with a sigmoid activation function, producing a binary segmentation map for each pixel in the input image.

The architecture's design allows it to capture both local and global context, making it well-suited for image segmentation tasks where precise localization of objects is crucial. The structure of the UNet architecture as well as the code implementation is shown in figure 12, 13, 14, 15, and 16 respectively.

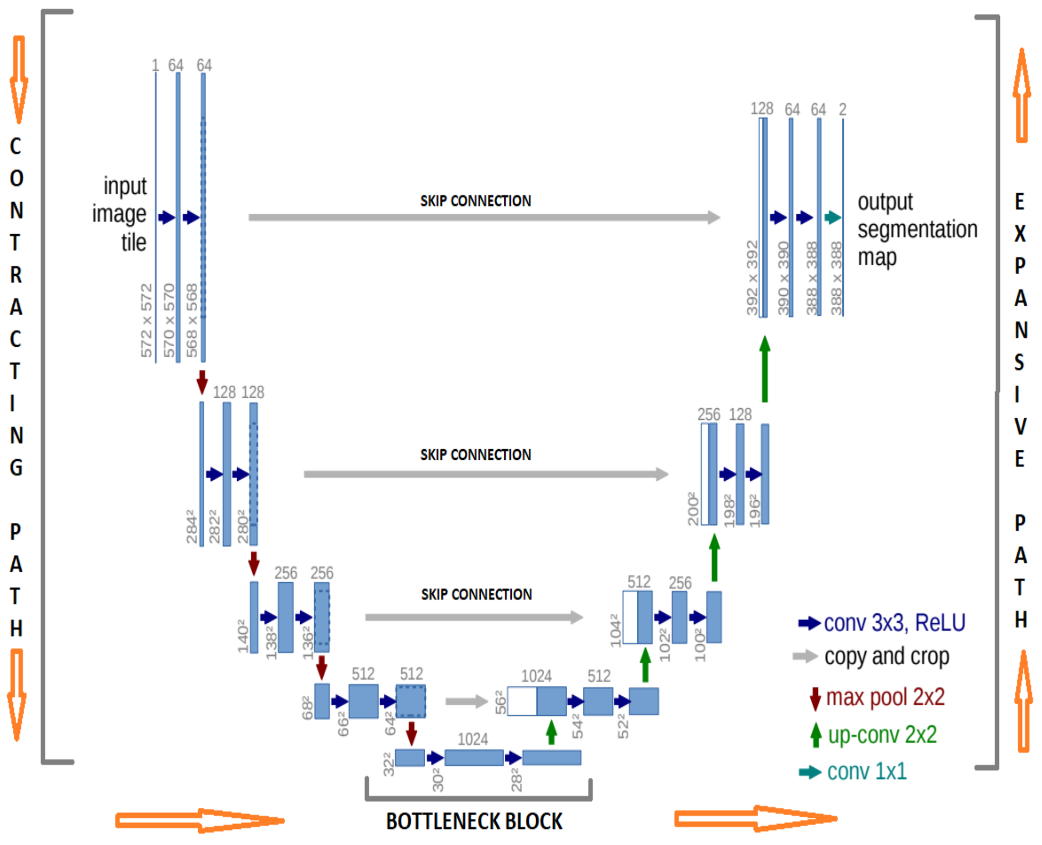
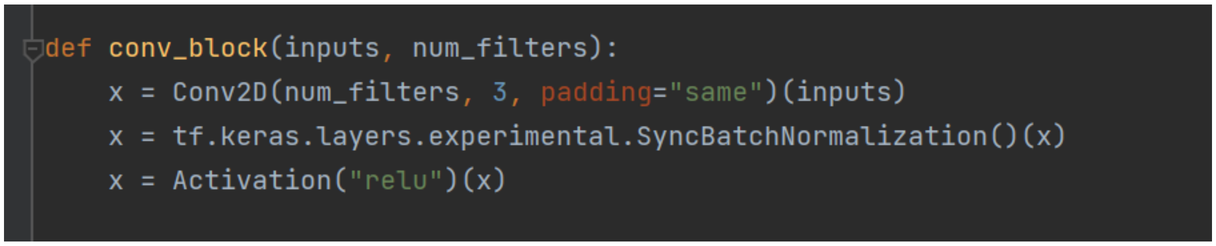
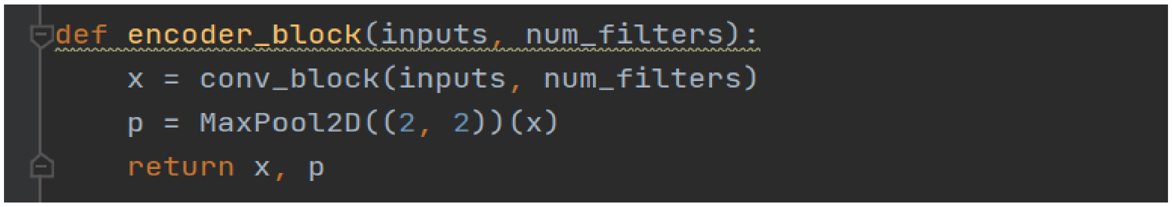


Fig 12: UNet Architecture Structure



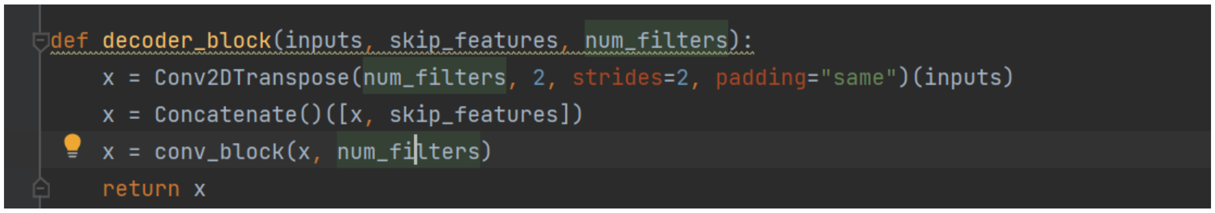
```
def conv_block(inputs, num_filters):
    x = Conv2D(num_filters, 3, padding="same")(inputs)
    x = tf.keras.layers.experimental.SyncBatchNormalization()(x)
    x = Activation("relu")(x)
```

Fig 13: Convolution Block



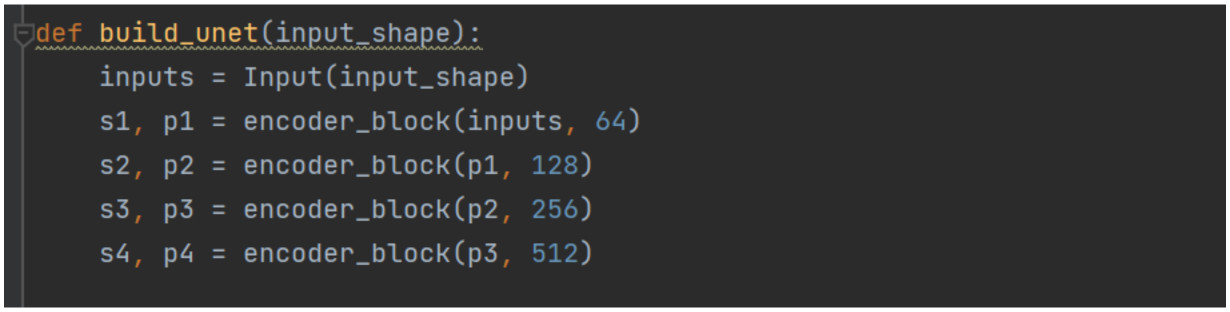
```
def encoder_block(inputs, num_filters):
    x = conv_block(inputs, num_filters)
    p = MaxPool2D((2, 2))(x)
    return x, p
```

Fig 14: Encoder Block



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

Fig 15: Decoder Block



```
def build_unet(input_shape):
    inputs = Input(input_shape)
    s1, p1 = encoder_block(inputs, 64)
    s2, p2 = encoder_block(p1, 128)
    s3, p3 = encoder_block(p2, 256)
    s4, p4 = encoder_block(p3, 512)
```

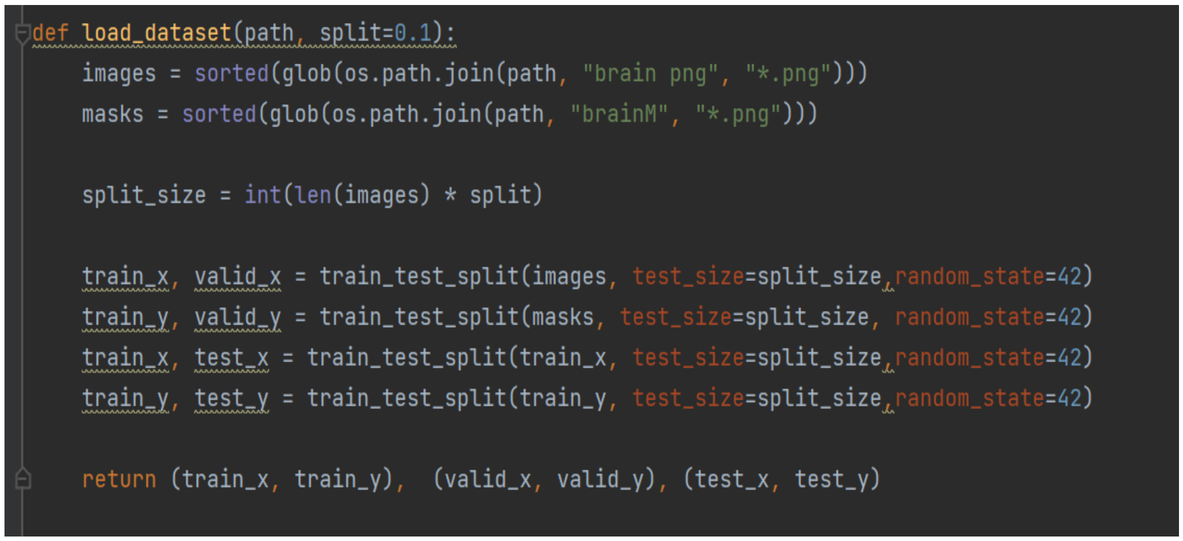
Fig 16: Building UNet

**3.2 Training the Model:** Training a U-Net architecture involves preparing your data, defining the model, compiling it, and then training it using your dataset. The following steps were implanted to train the UNet architecture.

The code begins by importing necessary libraries, including Tensor Flow for deep learning, OpenCV for image processing, NumPy for numerical operations, and other utilities.

- **Global Parameters:**
  - Sets global parameters H and W to define the height and width of the images after resizing.

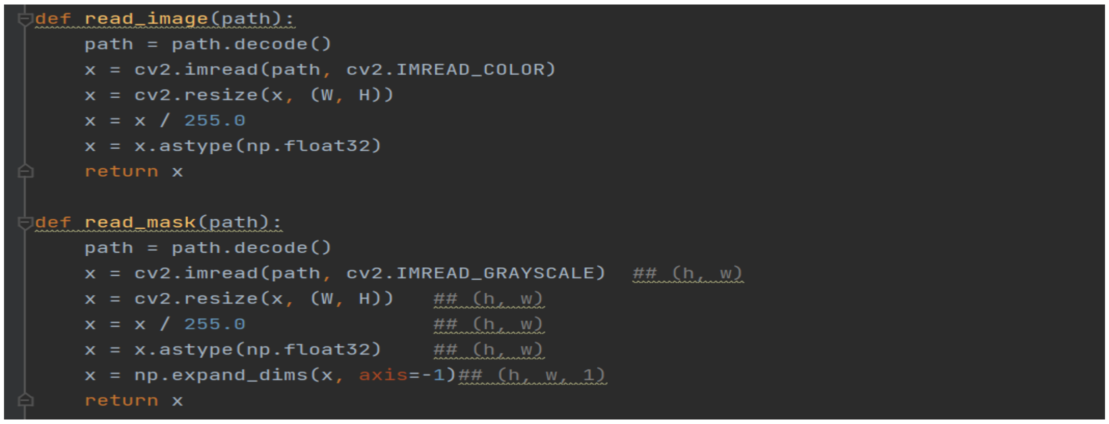
- **Utility Functions:**
  - create_dir function: A function that creates a directory if it doesn't exist, useful for organizing files.
  - load_dataset function: A function to load the dataset, which includes original images and corresponding binary masks. It splits the dataset into training, validation, and test sets. The code implementation is shown in figure 17.



```python
def load_dataset(path, split=0.1):
    images = sorted(glob(os.path.join(path, "brain png", "*.png")))
    masks = sorted(glob(os.path.join(path, "brainM", "*.png")))

    split_size = int(len(images) * split)

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

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

Fig 17: Loading Dataset

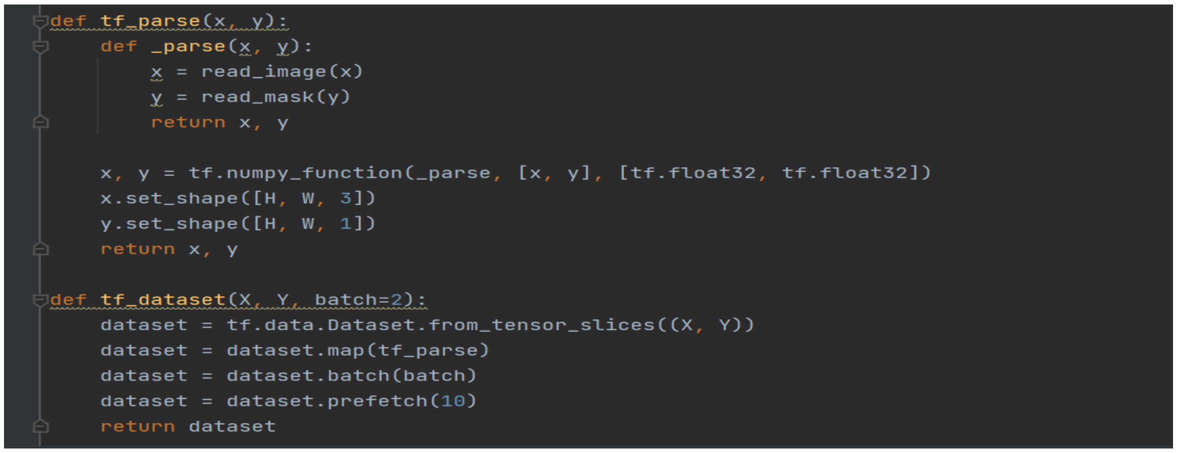
  - read_image and read_mask functions: Functions to read and preprocess images and masks is shown in figure 18.



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

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

Fig 18: Reading Image and Masks

- tf_parse function: A TensorFlow function used to parse the dataset, applying the read_image and read_mask functions.
- tf_dataset function: Creates a TensorFlow dataset from the parsed dataset, batching the data. The code implementation is shown in figure 19.



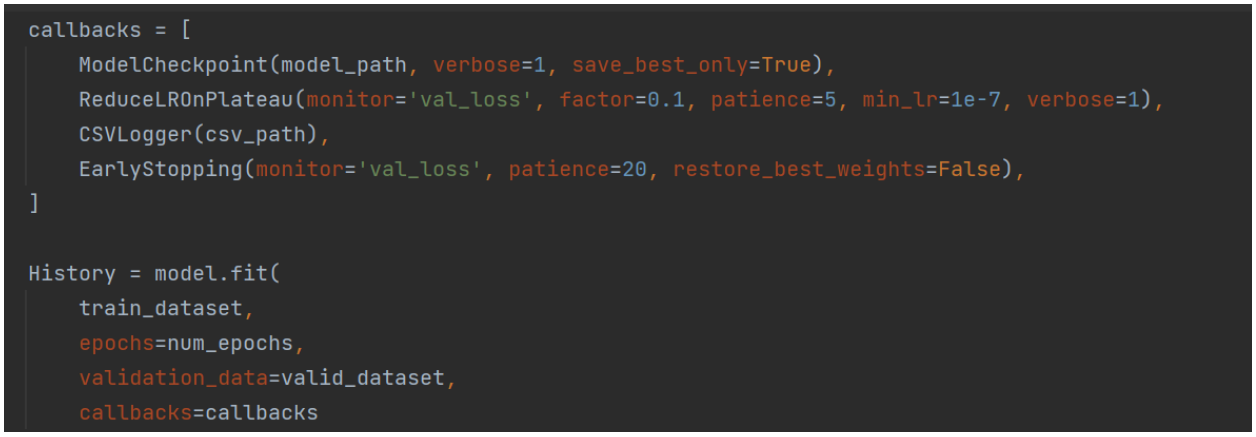
```python
def tf_parse(x, y):
    def _parse(x, y):
        x = read_image(x)
        y = read_mask(y)
        return x, y

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

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

Fig 19: Tf Parse and Tf Dataset function

- **Main Section:**
  - **Seeding:**
    - Sets random seeds for NumPy and TensorFlow to ensure reproducibility.
  - **Directories:**
    - Creates directories "files" for storing model checkpoints and logs.

  - **Hyperparameters:**

  - Defines hyperparameters such as batch size, learning rate, number of epochs, and file paths for the model (model.h5) and the csv file.
- **Dataset Loading:**
  - Loads the dataset using load_dataset. The dataset includes paths to original images and binary masks.
- **Dataset Preparation:**
  - Creates TensorFlow datasets (train_dataset) from the training data using tf_dataset.
- **Model Creation:**
  - Builds a U-Net model using the build_unet function and compiles it with a specific loss function (dice_loss), optimizer (Adam with a specified learning rate), and metrics (custom metrics like dice_coef, iou, Recall, Precision).
- **Callbacks:**
  - Defines a list of callbacks, including saving the best model (ModelCheckpoint), reducing learning rate on plateau (ReduceLROnPlateau), logging CSV files (CSVLogger), and stopping early if validation loss does not improve (EarlyStopping).
- **Model Training:**
  - Uses the fit method to train the model on the training dataset for a specified number of epochs. The training progress is monitored using the defined callbacks. The code implementation for the callbacks and model training is shown in figure 20.



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

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

Fig 20:  Callbacks and Model Training

## 3.3. Testing and Prediction:

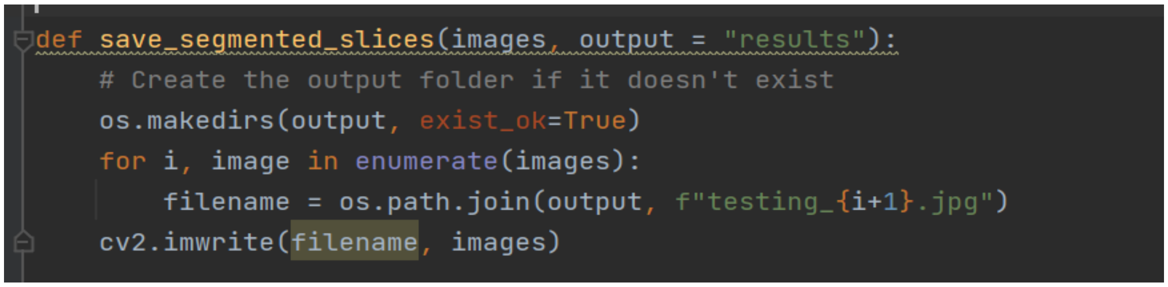
Testing and prediction was implemented using the following steps:

- **Importing Libraries and Modules:**

  The code starts by importing necessary libraries and modules, including TensorFlow for deep learning, tqdm for progress bars, and various metrics from scikit-learn.

- **Global Parameters and Functions:**

  - H and W are global parameters representing the height and width of the images after resizing.
  - create_dir: A function to create a directory if it doesn't exist.
  - save_segmented_slices: A function for saving segmented images. The code implantation is shown in figure 21.



```
def save_segmented_slices(images, output = "results"):
    # Create the output folder if it doesn't exist
    os.makedirs(output, exist_ok=True)
    for i, image in enumerate(images):
        filename = os.path.join(output, f"testing_{i+1}.jpg")
    cv2.imwrite(filename, images)
```

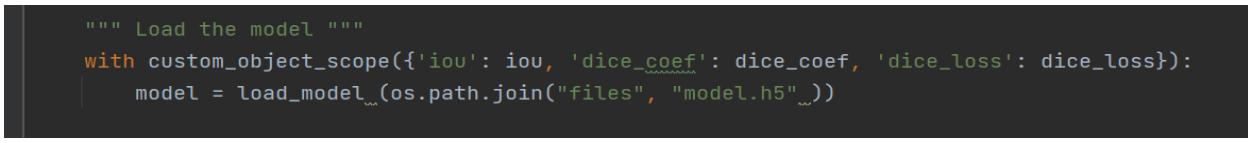
Fig 21: Saving Segmented Slices

- **Seeding and Directories:**

  - Seeding is done to set random seeds for NumPy and TensorFlow to ensure reproducibility.
  - A directory named "results" is created to store output images.

- **Load Pre-trained U-Net Model:**

  - The pre-trained U-Net model is loaded using load_model from TensorFlow's Keras. Custom metrics (iou, dice_coef, dice_loss) are specified using custom_object_scope as shown is figure 22.



```
""" Load the model """
with custom_object_scope({'iou': iou, 'dice_coef': dice_coef, 'dice_loss': dice_loss}):
    model = load_model (os.path.join("files", "model.h5" ))
```

Fig 22: Loading Model

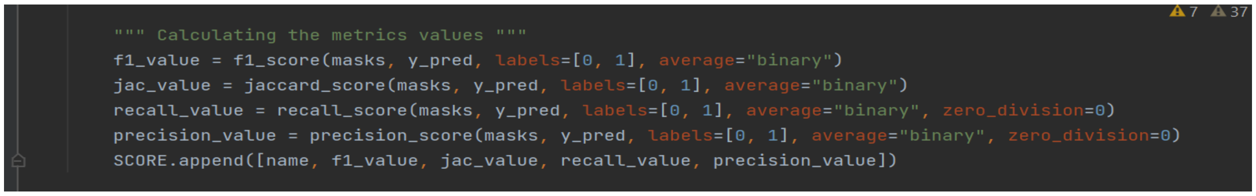
- **Load Test Dataset:**
    - The test dataset is loaded using the load_dataset function from a presumed train.py module. It appears to split the dataset into training and testing sets.

- **Prediction and Evaluation Loop:**

  For each test image and mask pair:

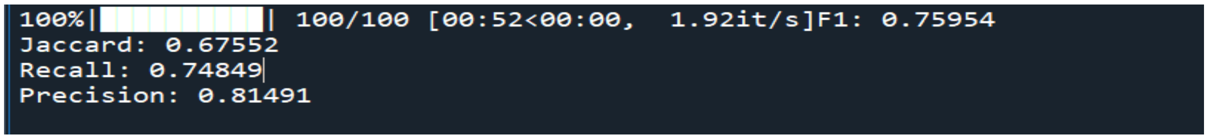
    - Image and mask paths are read, and the image is resized.
    - The image is normalized and expanded for prediction.
    - The U-Net model predicts the segmentation mask.
    - The predicted mask, original image, and ground truth mask are saved.
    - Evaluation metrics (F1 score, Jaccard score, recall, precision) are calculated.
    - Results are stored in a list named SCORE.

- **Metrics Calculation:**
    - The code calculates the average of F1, Jaccard, recall, and precision scores across all test images. The F1 score, Jaccard score (Intersection over Union, IoU), recall, and precision are common metrics used to evaluate the performance of binary or multi-class classification models. These metrics are especially relevant in tasks like image segmentation, where the goal is to identify and delineate objects or regions of interest in an image. The code implementation as well as the results is shown in figure 23 and 24.



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

Fig 23: Calculating Metrics

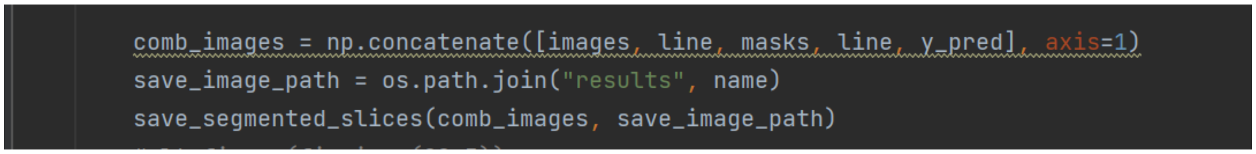


```
100%|██████████| 100/100 [00:52<00:00,  1.92it/s]F1: 0.75954
Jaccard: 0.67552
Recall: 0.74849
Precision: 0.81491
```

Fig 24: Metrics scores

- **Saving Metrics to CSV:**
    - Individual scores for each image and the overall averages are saved to a CSV file named "files/score.csv".

- **Final Output Image Path:**
  - The final output image path is saved in the variable save_image_path, but this is done outside the loop and refers to the last image processed. The code implementation for saving the predicted masks of the segmentation as well as the final results is shown in fig shown in figure 25, 26, 27 and 28.



```
comb_images = np.concatenate([images, line, masks, line, y_pred], axis=1)
save_image_path = os.path.join("results", name)
save_segmented_slices(comb_images, save_image_path)
```

Fig 25: Saving Prediction

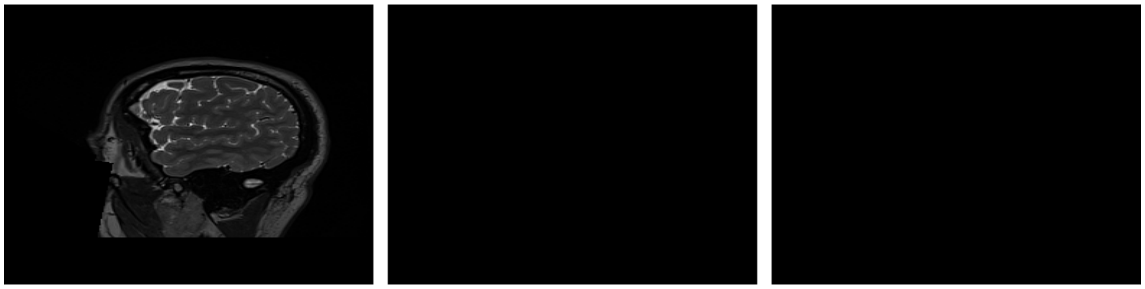


Fig 26: predicted mask 1

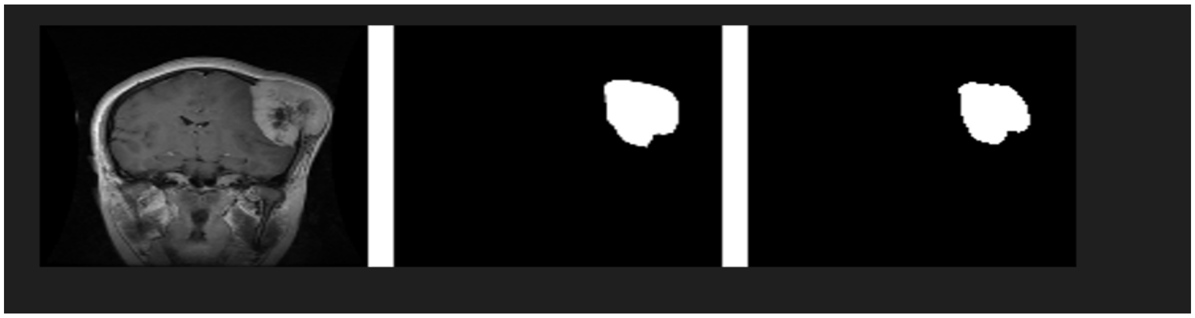


Fig 27: predicted mask 2

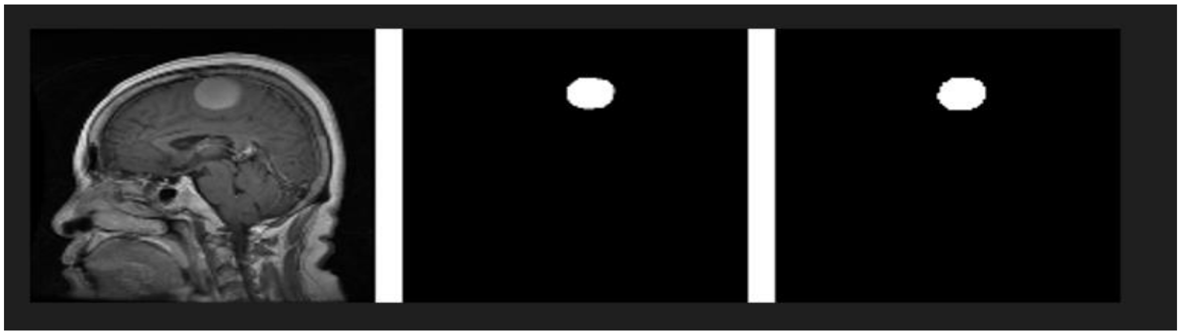


Fig 28: predicted mask 3

## 4. Conclusion

Image segmentation is a crucial process in image processing, playing a fundamental role in various domains such as medical imaging, computer vision, and pattern recognition. The segmentation of brain tumors, in particular, is vital for accurate diagnosis and treatment planning. The paper explores different techniques and methodologies for image segmentation, focusing on both traditional image processing methods using OpenCV and deep learning methods employing the UNet architecture.

The traditional image processing approach involves several steps, including reading DICOM images, applying filters, plotting histograms, thresholding, defining a delete function, and applying morphological operations. These methods leverage techniques such as Gaussian filtering, median filtering, thresholding, and morphological operations to segment brain tumors. The results include saving segmented tumor slices, plotting 3D meshes, and visualizing the segmentation outcomes.

On the other hand, the deep learning approach employs the UNet architecture, a widely used convolutional neural network structure for semantic segmentation tasks. The UNet architecture consists of a contracting path, a bottleneck, and an expansive path, enabling it to capture both local and global context for precise localization of objects. The methodology involves training the UNet model using a dataset, defining hyperparameters, creating a U-Net model, and employing callbacks for model training. The testing and prediction phase involves loading a pre-trained UNet model, predicting segmentation masks for test images, calculating evaluation metrics, and saving the results.

Overall, both traditional image processing and deep learning methods have their strengths and applications in the context of brain tumor segmentation. The traditional methods, leveraging filters and morphological operations, offer interpretability and simplicity. In contrast, deep learning methods, particularly the UNet architecture, provide a powerful tool for capturing intricate details and achieving state-of-the-art performance. The choice between these approaches depends on the specific requirements of the task and the available resources.

## REFERENCES

[1] D. Shen, G. Wu, & H. Suk, "Deep learning in medical image analysis," *Annual Review of Biomedical Engineering,* vol. 19, no. 1, pp. 221-248, 2017.

[2] M. Kaus, S. Warfield, A. Nabavi, P. Black, F. Jólesz, & R. Kikinis, "Automated segmentation of mr images of brain tumors," *Radiology,* vol. 218, no. 2, pp. 586-591, 2001.

[3] D. Mahalakshmi and S. Sumathi, "Brain tumour segmentation strategies utilizing mean shift clustering and content based active contour segmentation," *ICTACT Journal on Image and Video Processing,* vol. 9, no. 4, pp. 2002-2008, 2019.

[4] A. Gooya, G. Biros, & C. Davatzikos, "Deformable registration of glioma images using em algorithm and diffusion reaction modeling," *IEEETransactions on Medical Imaging,* vol. 30, no. 2, pp. 375-390, 2011.

[5] Wang, G. and Zuluaga, M. A. and Li, W. and Pratt, R. and Patel, P. A. and Aertsen, M. and Doel, T. and David, A. L. and Deprest, J. and Ourselin, S. and Vercauteren, T., "Deepigeos: a deep interactive geodesic framework for medical image segmentation," *IEEE Transactions on Pattern Analysis and Machine Intelligence,* vol. 41, no. 7, pp. 1559-1572, 2019.

[6] J. Sun and Y. Liu, "Segmentation for human motion injury ultrasound medical images using deep feature fusion," *Mathematical Problems in Engineering,* vol. 2022, pp. 1-9, 2022.

[7] "The application of support vector machines in medical image segmentation," in *Machine Learning Theory and Practice*, 2022.

[8] X. Liu, L. Song, S. Liu, & Y. Zhang, "A review of deep-learning-based medical image segmentation methods," *Sustainability,* vol. 13, no. 3, pp. 1224, 2021, 2021.

[9] E. B. &. G. G. S. Ho, "Level-set evolution with region competition: automatic 3-d segmentation of brain tumors".*Object Recognition Supported by User Interaction for Service Robots.*

[10] Zhou, Z., Siddiquee, M. R., Tajbakhsh, N., & Liang, J, "Unet++: redesigning skip connections to exploit multiscale features in image segmentation," *IEEE Transactions on Medical Imaging,* vol. 39, no. 6, pp. 1856-1867, 2020.

[11] Sengar, S. S., Meulengracht, C., Boesen, M., Overgaard, A., Gudbergsen, H., Nybing, J. D., … & Dam, E. B., "Multi-planar 3d knee mri segmentation via unet inspired architectures.," *International Journal of Imaging Systems and Technology,* vol. 33, no. 3, pp. 985-998, 2022.

[12] Li, J., Xie, T., & Wu, Z., "Mst-unet: a modified swin transformer for water bodies’ mapping using

sentinel-2 images," *Journal of Applied Remote Sensing,* vol. 17, no. 2, 2023.

[13] Baccouche, A., Garcia-Zapirain, B., Cristian, C., & Elmaghraby, A., "Connected-unets: a deep learning architecture for breast mass segmentation," *NPJ Breast Cancer,* vol. 7, no. 1, 2021.

[14] Wang, Z., You, J., Liu, W., & Wang, X. (2023). Transformer assisted dual u-net for seismic fault detection. Frontiers in Earth Science, 11. https://doi.org/10.3389/feart.2023.1047626

[15] Buscombe, D. and Goldstein, E. B. (2022). A reproducible and reusable pipeline for segmentation of geoscientific imagery.. https://doi.org/10.31223/x5hs81

[15] https://nbia.cancerimagingarchive.net/nbia