AI Project Report

AmirMohammad Ebrahiminasab 610301101

February 5, 2025

List Report

1. Preprocessing Step
2. Model Architect
3. Training and Validation Process
4. Test
5. Bonus
   (a) UNet++
   (b) TransNet
6. Comparison

1 Preprocessing Step

1.1 Dataset

- **Name:** LGG Segmentation Dataset
- **Size:** 749 MB
- **Content:** 3929 brain MRI images with corresponding ground truth masks
- **Format:** tif images

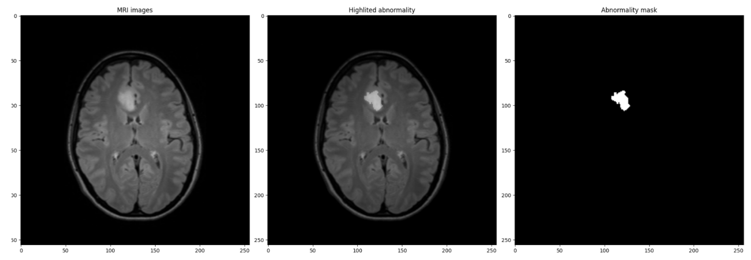

Figure 1: Sample Brain MRI Image with Mask

1.2 Preprocessing Steps

The images were resized as follows:

- Image: $128 \times 128 \times 3$
- Mask: $128 \times 128 \times 1$ (Binary Mask)
- Batch Size: 16

```
import pandas as pd
from sklearn.model_selection import train_test_split

df = pd.DataFrame({'image': image_filenames_train,
                   'mask': mask_files})

df_train, df_test = train_test_split(df, test_size=0.2,
                    random_state=42)
df_test, df_val = train_test_split(df_test, test_size=0.5,
                    random_state=42)
```

Listing 1: Splitting Dataset into Train, Validation, and Test Sets

1.3 Data Loading and Preprocessing

Below is the code for loading and preprocessing the dataset:

```python
def load_image(image_path, mask=False):
        img = tifffile.imread(image_path.numpy().decode('utf-8'
            ))

        if len(img.shape) == 2:
                img = np.expand_dims(img, axis=-1)

        img = tf.convert_to_tensor(img)
        img = tf.image.resize(img, [img_height, img_width])

        if not mask:
                img = tf.cast(img, tf.float32) / 255.0
                if img.shape[-1] == 1:
                        img = tf.repeat(img, 3, axis=-1)
        else:
                img = tf.cast(img > 0, tf.float32)
                if img.shape[-1] != 1:
                        img = tf.expand_dims(img, axis=-1)

        return img

def tf_load_image(image_path, mask_path):
        def _load_image(image_path, mask_path):
                image = load_image(image_path, mask=False)
                mask = load_image(mask_path, mask=True)
                return image, mask
        image, mask = tf.py_function(
                _load_image, [image_path, mask_path], [tf.
                    float32, tf.float32]
        )
        image.set_shape([img_height, img_width, 3])
        mask.set_shape([img_height, img_width, 1])
        return image, mask
```

Listing 2: Image and Mask Loading Function

This function reads '.tif' images, resizes them to the dimensions discussed earlier then normalize the images after that we make sure that masks value are binary, and prepares them for training process.

```
train_ds = tf.data.Dataset.from_tensor_slices(
(df_train['image'].values, df_train['mask'].values)
).shuffle(100).map(tf_load_image,
        num_parallel_calls=tf.data.AUTOTUNE).batch(BATCH_SIZE)
        .prefetch(tf.data.AUTOTUNE)

val_ds = tf.data.Dataset.from_tensor_slices(
(df_val['image'].values, df_val['mask'].values)
).map(tf_load_image).batch(BATCH_SIZE)
.prefetch(tf.data.AUTOTUNE)

test_ds = tf.data.Dataset.from_tensor_slices(
(df_test['image'].values, df_test['mask'].values)
).map(tf_load_image).batch(BATCH_SIZE)
.prefetch(tf.data.AUTOTUNE)
```

Listing 3: Creating TensorFlow Data Pipelines

This code defines training, validation, and test datasets using TensorFlow's 'tf.data' API, optimizing them for efficient loading, since I had some issues with gpu runtime and the overall time consuming training!

2 Model Architect

In this project, we utilize the UNet architecture, a widely adopted model in medical image segmentation tasks. UNet is particularly well-suited for such applications due to its encoder-decoder structure, which effectively captures both local and global contextual information. The skip connections between the encoder and decoder paths allow the network to preserve spatial details, enabling precise segmentation even with limited training data. This makes UNet an excellent choice for segmenting brain MRI images, where fine-grained details are crucial.

Below is the detailed architecture of the UNet model implemented in this project. The use of **Batch Normalization** after each convolutional layer significantly improves the training process by stabilizing the learning dynamics and accelerating convergence. Additionally, it helps mitigate issues related to vanishing/exploding gradients, leading to better overall performance.

The architecture leverages **Batch Normalization** after every convolutional operation to normalize the activations, ensuring that the network learns more efficiently. This not only speeds up convergence but also enhances the robustness of the model, as evidenced by the improved results achieved during training.

Table 1: UNet Model Architecture

Layer Type	**Configuration**	**Output Shape**
Input	$128 \times 128 \times 3$	$128 \times 128 \times 3$
Conv2D + BN + ReLU	Filters=64, Kernel=3×3, Padding=Same	$128 \times 128 \times 64$
Conv2D + BN + ReLU	Filters=64, Kernel=3×3, Padding=Same	$128 \times 128 \times 64$
MaxPooling2D	Pool Size=2×2	$64 \times 64 \times 64$
Conv2D + BN + ReLU	Filters=128, Kernel=3×3, Padding=Same	$64 \times 64 \times 128$
Conv2D + BN + ReLU	Filters=128, Kernel=3×3, Padding=Same	$64 \times 64 \times 128$
MaxPooling2D	Pool Size=2×2	$32 \times 32 \times 128$
Conv2D + BN + ReLU	Filters=256, Kernel=3×3, Padding=Same	$32 \times 32 \times 256$
Conv2D + BN + ReLU	Filters=256, Kernel=3×3, Padding=Same	$32 \times 32 \times 256$
MaxPooling2D	Pool Size=2×2	$16 \times 16 \times 256$
Conv2D + BN + ReLU	Filters=512, Kernel=3×3, Padding=Same	$16 \times 16 \times 512$
Conv2D + BN + ReLU	Filters=512, Kernel=3×3, Padding=Same	$16 \times 16 \times 512$
MaxPooling2D	Pool Size=2×2	$8 \times 8 \times 512$
Conv2D + BN + ReLU	Filters=1024, Kernel=3×3, Padding=Same	$8 \times 8 \times 1024$
Conv2D + BN + ReLU	Filters=1024, Kernel=3×3, Padding=Same	$8 \times 8 \times 1024$
Conv2DTranspose + Concatenate	Filters=512, Kernel=2×2, Stride=2×2	$16 \times 16 \times 1024$
Conv2D + BN + ReLU	Filters=512, Kernel=3×3, Padding=Same	$16 \times 16 \times 512$
Conv2D + BN + ReLU	Filters=512, Kernel=3×3, Padding=Same	$16 \times 16 \times 512$
Conv2DTranspose + Concatenate	Filters=256, Kernel=2×2, Stride=2×2	$32 \times 32 \times 512$
Conv2D + BN + ReLU	Filters=256, Kernel=3×3, Padding=Same	$32 \times 32 \times 256$
Conv2D + BN + ReLU	Filters=256, Kernel=3×3, Padding=Same	$32 \times 32 \times 256$
Conv2DTranspose + Concatenate	Filters=128, Kernel=2×2, Stride=2×2	$64 \times 64 \times 256$
Conv2D + BN + ReLU	Filters=128, Kernel=3×3, Padding=Same	$64 \times 64 \times 128$
Conv2D + BN + ReLU	Filters=128, Kernel=3×3, Padding=Same	$64 \times 64 \times 128$
Conv2DTranspose + Concatenate	Filters=64, Kernel=2×2, Stride=2×2	$128 \times 128 \times 128$
Conv2D + BN + ReLU	Filters=64, Kernel=3×3, Padding=Same	$128 \times 128 \times 64$
Conv2D + BN + ReLU	Filters=64, Kernel=3×3, Padding=Same	$128 \times 128 \times 64$
Conv2D (Output)	Filters=1, Kernel=1×1, Activation=Sigmoid	$128 \times 128 \times 1$

3 Training and Validation Process

In this section, we discuss the loss function and evaluation metrics used for training, along with the model checkpoint mechanism to save the best-performing model.

3.1 Loss Function and Metrics

To effectively train our UNet model for brain MRI segmentation, we employ the Dice Coefficient as our primary metric. The Dice Coefficient is widely used in medical image segmentation due to its ability to measure the overlap between predicted and ground truth masks. It is defined as:

$$\text{Dice Coefficient} = \frac{2|X \cap Y| + \epsilon}{|X| + |Y| + \epsilon} \tag{1}$$

where X and Y represent the predicted and actual masks, respectively, and ϵ is a small smoothing factor to prevent division by zero.

The corresponding Dice Loss is computed as:

$$\text{Dice Loss} = 1 - \text{Dice Coefficient} \tag{2}$$

which ensures that the model maximizes the overlap between the predicted and ground truth masks during training.

Below is the implementation of the Dice Coefficient and Dice Loss:

```python
from tensorflow.keras import backend as K

def dice_coefficient(y_true, y_pred, smooth=1e-5):
        y_true_f = K.flatten(y_true)
        y_pred_f = K.flatten(y_pred)
        intersection = K.sum(y_true_f * y_pred_f)
        return (2. * intersection + smooth) /
                        (K.sum(y_true_f) + K.sum(y_pred_f) +
                            smooth)

def dice_loss(y_true, y_pred):
        return 1 - dice_coefficient(y_true, y_pred)
```

Listing 4: Dice Coefficient and Dice Loss Functions

Additionally, we utilize Sensitivity (True Positive Rate) and Specificity (True Negative Rate) to evaluate the performance of our model:

```
def sensitivity(y_true, y_pred):
        true_positives = K.sum(K.round(
                K.clip(y_true * y_pred, 0, 1)))
        possible_positives = K.sum(K.round(
                K.clip(y_true, 0, 1)))
        return true_positives /
                (possible_positives + K.epsilon())

def specificity(y_true, y_pred):
        true_negatives = K.sum(K.round(K.clip((1 - y_true) *
                (1 - y_pred), 0, 1)))
        possible_negatives = K.sum(K.round(
                K.clip(1 - y_true, 0, 1)))
        return true_negatives /
                (possible_negatives + K.epsilon())
```

Listing 5: Sensitivity and Specificity Metrics

Sensitivity is crucial in medical image segmentation as it ensures that diseased regions are correctly identified, whereas specificity helps in minimizing false positives, ensuring a balanced and accurate segmentation.

3.2 Model Checkpoint and Training Strategy

To ensure we retain the best model, we employ a model checkpoint callback that saves the model with the highest validation Dice Coefficient. Training is conducted for 50 epochs, allowing sufficient time for the model to converge while preventing overfitting.

```
from tensorflow.keras.callbacks import ModelCheckpoint

checkpoint = ModelCheckpoint(
        'best_model.h5', monitor='val_dice_coefficient',
        save_best_only=True, mode='max', verbose=1
)
```

Listing 6: Model Checkpoint Callback

This ensures that the model with the best segmentation performance on the validation set is preserved, allowing for optimal deployment and further analysis.

Below, we present visual comparisons of the training progress. Figure 2 shows the accuracy trend over epochs, while Figure 3 illustrates the Dice Coefficient progression for both training and validation. Note that the loss function used in training is the Binary Cross-Entropy (BCE) loss.

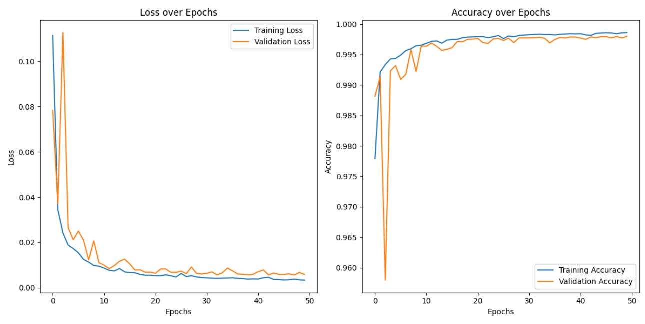

Figure 2: Accuracy/Loss over epochs for training and validation.

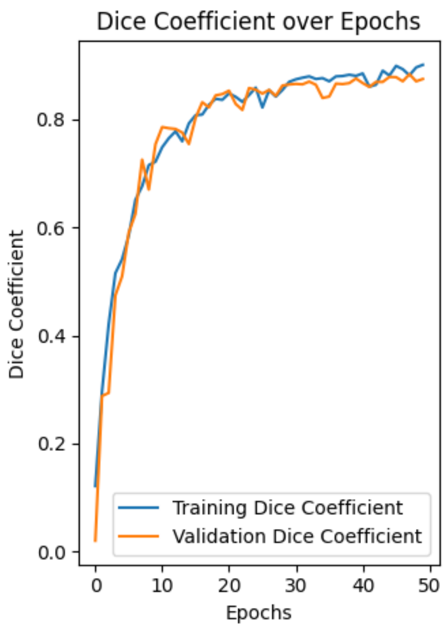

Figure 3: Dice Coefficient progression over epochs for training and validation.

4 Test

To evaluate the performance of our trained model on unseen data, we first load the best-performing model using the following code:

```
model = tf.keras.models.load_model('/content/drive/MyDrive
/unet_model_brain_mri.keras',
custom_objects={
        'dice_coefficient': dice_coefficient,
        'sensitivity': sensitivity,
        'specificity': specificity,
        'false_negative_proportion': false_negative_proportion
})

model.compile(
        optimizer='adam',
        loss='binary_crossentropy',
        metrics=['accuracy', BinaryIoU(threshold=0.5, name='iou
            '),
        dice_coefficient,
        sensitivity, specificity, false_negative_proportion]
)

predictions = model.predict(test_ds)
binary_preds = (predictions > 0.5).astype(np.float32)
```

Listing 7: Loading the Trained Model

We now present some predictions of the model on test images:

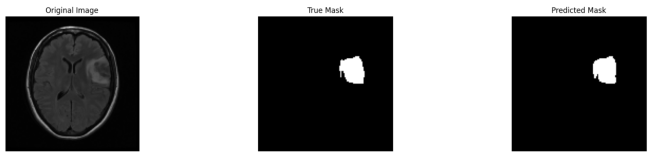

Figure 4: Example 1: Very Accurate.

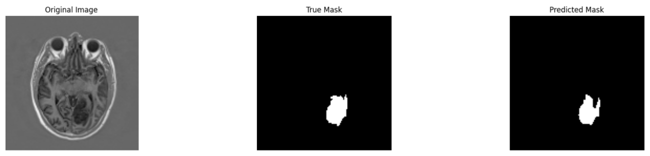

Figure 5: Example 2: Relatively Accurate.

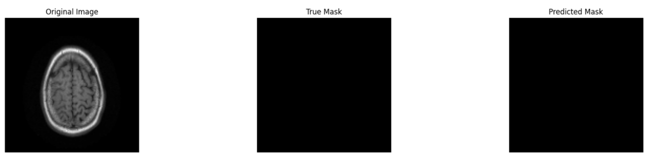

Figure 6: Example 3: Model shouldn't detect any abnormality.

The table below presents the overall performance metrics of our model across the training, validation, and test sets.

Metric	Train	Validation	Test
BCE	0.003349	0.005627	0.005497
Accuracy	0.998579	0.997951	0.998039
Dice Coefficient	0.898650	0.883070	0.879260
Sensitivity	0.935471	0.906238	0.904166
Specificity	0.999218	0.998995	0.999043
False Negative Proportion	0.064530	0.093762	0.095834

Table 2: Performance metrics of the model on train, validation, and test sets.

5 Bonus

In this section we explore two variations of UNet which are called UNet++ and TransUNet.

5.1 UNet++

UNet++ extends UNet by introducing nested and dense skip connections. These additional connections help bridge the semantic gap between encoder and decoder features, leading to improved segmentation accuracy. The architecture incorporates multiple intermediate layers that enhance feature propagation and gradient flow, making it more robust in medical image segmentation tasks.

```
def unet_plus_plus(input_size=(128, 128, 3)):
        inputs = Input(input_size)
        x00 = conv_block(inputs, 64)
        p0 = MaxPooling2D((2,2))(x00)
        x10 = conv_block(p0, 128)
        p1 = MaxPooling2D((2,2))(x10)
        x20 = conv_block(p1, 256)
        p2 = MaxPooling2D((2,2))(x20)
        x30 = conv_block(p2, 512)
        p3 = MaxPooling2D((2,2))(x30)
        x40 = conv_block(p3, 1024)

        x31 = Conv2DTranspose(512, (2,2), strides=(2,2),
            padding="same")(x40)
        x31 = concatenate([x31, x30])
        x31 = conv_block(x31, 512)
        x22 = Conv2DTranspose(256, (2,2), strides=(2,2),
            padding="same")(x31)
        x22 = concatenate([x22, x20])
        x22 = conv_block(x22, 256)
        x32 = Conv2DTranspose(256, (2,2), strides=(2,2),
            padding="same")(x31)
        x32 = concatenate([
        x32,
        Conv2DTranspose(512, (2,2), strides=(2,2), padding="
            same")(x30),
        x22
        ])
        x32 = conv_block(x32, 512)
        x13 = Conv2DTranspose(128, (2,2), strides=(2,2),
            padding="same")(x22)
        x13 = concatenate([x13, x10])
        x13 = conv_block(x13, 128)

        x23 = Conv2DTranspose(128, (2,2), strides=(2,2),
            padding="same")(x32)
        x23 = concatenate([
        x23,
        Conv2DTranspose(256, (2,2), strides=(2,2), padding="
            same")(x20),
        x13
        ])
        x23 = conv_block(x23, 256)
        x04 = Conv2DTranspose(64, (2,2), strides=(2,2), padding
            ="same")(x13)
        x04 = concatenate([x04, x00])
        x04 = conv_block(x04, 64)
        x14 = Conv2DTranspose(64, (2,2), strides=(2,2), padding
            ="same")(x23)
        x14 = concatenate([
        x14,
        Conv2DTranspose(128, (2,2), strides=(2,2), padding="
            same")(x10),
        x04
        ])
        x14 = conv_block(x14, 128)
        outputs = Conv2D(1, (1,1), activation="sigmoid")(x14)
        return Model(inputs=inputs, outputs=outputs)
```

Listing 8: UNet++ Architecture

and here is the code for ConvBlock:

```python
def conv_block(x, filters, kernel_size=(3,3), padding="same"):
        x = Conv2D(filters, kernel_size, padding=padding)(x)
        x = BatchNormalization()(x)
        x = Activation("relu")(x)
        x = Conv2D(filters, kernel_size, padding=padding)(x)
        x = BatchNormalization()(x)
        x = Activation("relu")(x)
        return x
```

Listing 9: ConvBlock Architecture

The table below presents the overall performance metrics of our model across the training, validation, and test sets.

Metric	Train	Validation	Test
BCE	0.003184	0.004255	0.005289
Accuracy	0.0.998647	0.998347	0.998296
Dice Coefficient	0.909443	0.894415	0.875419
Sensitivity	0.930191	0.903436	0.893269
Specificity	0.999388	0.999367	0.999245
False Negative Proportion	0.069809	0.096564	0.106731

Table 3: Performance metrics of the UNet++ model on train, validation, and test sets.

Next page is the Unet++ Architect (Please note that I used chatbots for the following architect and have made every effort to ensure that there is no misinformation. Nevertheless, please proceed cautiously.)

Table 4: UNet++ Model Architecture

Layer Type	**Configuration**	**Output Shape**
Input	$128 \times 128 \times 3$	$128 \times 128 \times 3$
Conv2D + BN + ReLU	Filters=64, Kernel=3×3, Padding=Same	$128 \times 128 \times 64$
MaxPooling2D	Pool Size=2×2	$64 \times 64 \times 64$
Conv2D + BN + ReLU	Filters=128, Kernel=3×3, Padding=Same	$64 \times 64 \times 128$
MaxPooling2D	Pool Size=2×2	$32 \times 32 \times 128$
Conv2D + BN + ReLU	Filters=256, Kernel=3×3, Padding=Same	$32 \times 32 \times 256$
MaxPooling2D	Pool Size=2×2	$16 \times 16 \times 256$
Conv2D + BN + ReLU	Filters=512, Kernel=3×3, Padding=Same	$16 \times 16 \times 512$
MaxPooling2D	Pool Size=2×2	$8 \times 8 \times 512$
Conv2D + BN + ReLU	Filters=1024, Kernel=3×3, Padding=Same	$8 \times 8 \times 1024$
Conv2DTranspose + Concatenate	Filters=512, Kernel=2×2, Stride=2×2	$16 \times 16 \times 1024$
Conv2D + BN + ReLU	Filters=512, Kernel=3×3, Padding=Same	$16 \times 16 \times 512$
Conv2DTranspose + Concatenate	Filters=256, Kernel=2×2, Stride=2×2	$32 \times 32 \times 512$
Conv2D + BN + ReLU	Filters=256, Kernel=3×3, Padding=Same	$32 \times 32 \times 256$
Conv2DTranspose + Concatenate	Filters=512, Kernel=2×2, Stride=2×2	$32 \times 32 \times 1024$
Conv2D + BN + ReLU	Filters=512, Kernel=3×3, Padding=Same	$32 \times 32 \times 512$
Conv2DTranspose + Concatenate	Filters=128, Kernel=2×2, Stride=2×2	$64 \times 64 \times 256$
Conv2D + BN + ReLU	Filters=128, Kernel=3×3, Padding=Same	$64 \times 64 \times 128$
Conv2DTranspose + Concatenate	Filters=256, Kernel=2×2, Stride=2×2	$64 \times 64 \times 512$
Conv2D + BN + ReLU	Filters=256, Kernel=3×3, Padding=Same	$64 \times 64 \times 256$
Conv2DTranspose + Concatenate	Filters=64, Kernel=2×2, Stride=2×2	$128 \times 128 \times 128$
Conv2D + BN + ReLU	Filters=64, Kernel=3×3, Padding=Same	$128 \times 128 \times 64$
Conv2DTranspose + Concatenate	Filters=128, Kernel=2×2, Stride=2×2	$128 \times 128 \times 256$
Conv2D + BN + ReLU	Filters=128, Kernel=3×3, Padding=Same	$128 \times 128 \times 128$
Conv2D (Output)	Filters=1, Kernel=1×1, Activation=Sigmoid	$128 \times 128 \times 1$

5.2 TransNet

TransUNet integrates Transformer modules into the UNet framework, leveraging self-attention mechanisms to capture long-range dependencies in medical images. Unlike traditional CNN-based architectures, TransUNet can better model global contextual information, making it particularly effective for complex segmentation tasks where spatial relationships play a crucial role.

I faced a challenge in training this model cause the lack of computational power and had to adjust the models architect to the point that I only used ViT as my encoder and didn't pass its output to each decoder label except the first one!

```
class TransUNet(nn.Module):
        def __init__(self, pretrained_model="google/vit-base-
            patch16-224", img_size=224, out_channels=1):
                super(TransUNet, self).__init__()

                self.encoder = AutoModel.from_pretrained(
                    pretrained_model)

                self.decoder = nn.Sequential(
                nn.ConvTranspose2d(768, 512, kernel_size=3,
                    stride=2, padding=1, output_padding=1),
                nn.BatchNorm2d(512),
                nn.ReLU(),
                nn.Conv2d(64, out_channels, kernel_size=1)
                )

                def forward(self, x):
                x = self.encoder(pixel_values=x).
                    last_hidden_state
                x = self.decoder(x)

                return x
```

Listing 10: TransUNet Architecture

Table 5: TransUNet Architecture Overview

Layer	Type	Output Shape
Encoder	Vision Transformer (ViT)	(Batch, 196, 768)
Feature Reshape	Reshape to 2D	(Batch, 768, 14, 14)
Decoder	Transposed Conv Layers	(Batch, 1, 224, 224)

The table below presents the overall performance metrics of the TransUNet model across the training, validation and test sets.

Metric	Train	Validation	Test
Dice Coefficient	0.921826	0.767283	0.771514
IOU	0.854989	0.622433	0.628020
Sensitivity	0.916524	0.724477	0.714994

Table 6: Performance metrics of TransUNet on train, validation, and test sets.

6 Comparison

Here are the tables of each model Result and the last table is the paper's result.

Metric	Train	Validation	Test
BCE	0.003349	0.005627	0.005497
Accuracy	0.998579	0.997951	0.998039
Dice Coefficient	0.898650	0.883070	0.879260
Sensitivity	0.935471	0.906238	0.904166
Specificity	0.999218	0.998995	0.999043
False Negative Proportion	0.064530	0.093762	0.095834

Table 7: Performance metrics of the UNet model on train, validation, and test sets.

Metric	Train	Validation	Test
BCE	0.003184	0.004255	0.005289
Accuracy	0.0.998647	0.998347	0.998296
Dice Coefficient	0.909443	0.894415	0.875419
Sensitivity	0.930191	0.903436	0.893269
Specificity	0.999388	0.999367	0.999245
False Negative Proportion	0.069809	0.096564	0.106731

Table 8: Performance metrics of the UNet++ model on train, validation, and test sets.

Metric	Train	Validation	Test
Dice Coefficient	0.921826	0.767283	0.771514
IOU	0.854989	0.622433	0.628020
Sensitivity	0.916524	0.724477	0.714994

Table 9: Performance metrics of TransUNet on train, validation, and test sets.

Dataset	**Loss**	**Accuracy**	**Mean IoU**	**Precision**	**Sensitivity**	**Specificity**	**Dice Score**
BraTS 2017	0.0056	0.9980	0.9637	0.9973	0.9970	0.9972	0.8453
BraTS 2018	0.0057	0.9979	0.8927	0.9972	0.9970	0.9940	0.8160
BraTS 2019	0.0054	0.9981	0.9130	0.9974	0.9971	0.9991	0.8409
BraTS 2020	0.0056	0.9980	0.8935	0.9973	0.9970	0.9983	0.8300

Table 10: Paper's Result on BraTS 2017-2020 dataset

As you can see, Our models except TransUNet are much better in performance!

Note that the reason why TransUNet isn't as great as others is because of the lack of computational power and the fact that we had to only use ViT as our encoder and not pass it to each decoder layer as TransUnet model is intended to do so!

Please Note that the paper's dataset is different to ours!