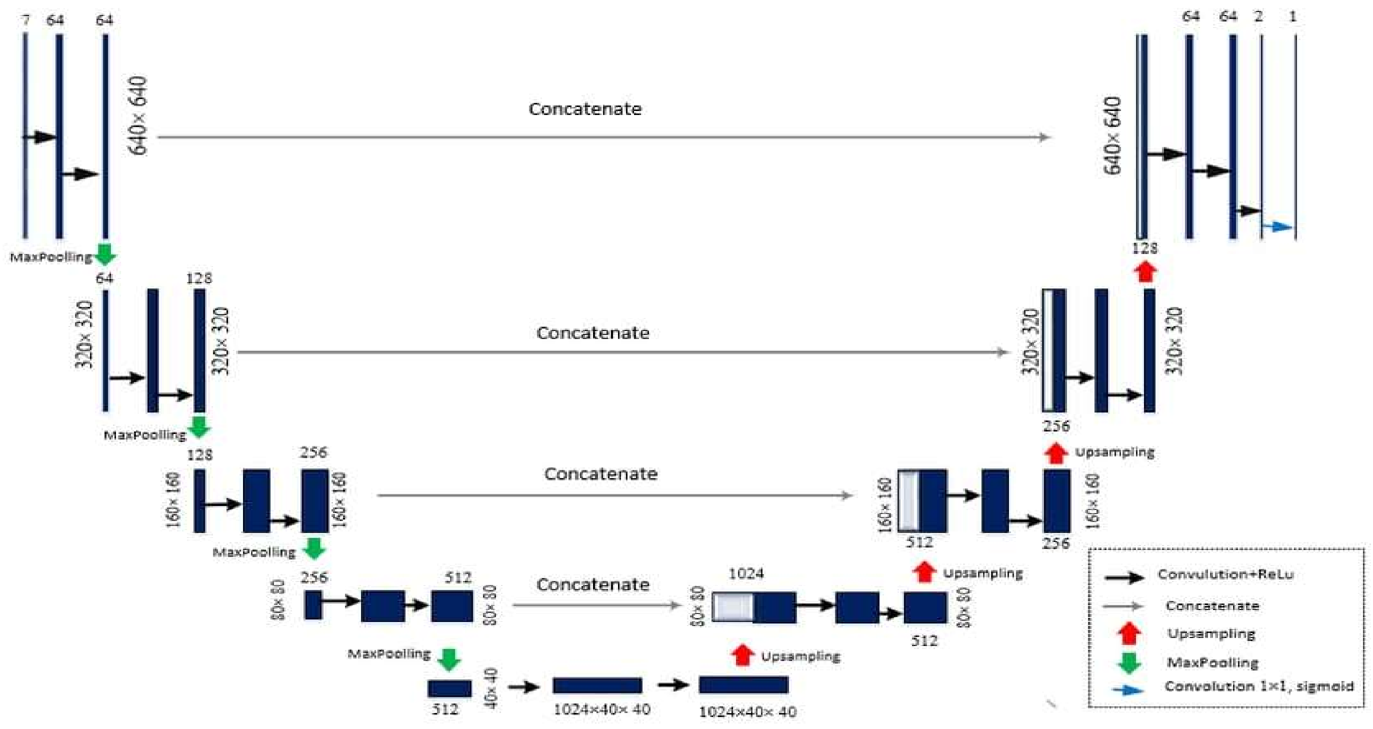
Computer Vision: Brain MRI Metastasis Segmentation

U-NET

U-Net is a convolutional neural network (CNN) architecture designed for biomedical image Segmentation.

ARCHITECTURE:



- Contracting Path (Encoder):
  - The contracting path involves convolutional layers with ReLU activations and max-pooling, reducing the spatial dimensions while increasing feature channels.
  - This process downscales the input, capturing context and learning abstract features crucial for distinguishing image regions.
- Bottleneck:
  - This is the middle layer of the U-Net, where the feature maps are at their most compressed. It captures the highest-level features of the input image before the network begins to expand.
- Expansive Path (Decoder):
  - The expansive path uses up-convolutions to increase spatial dimensions and includes skip connections to concatenate high-resolution features from the contracting path. This preserves spatial details for more accurate segmentation output.
- Skip connection:
  - Local Spatial Information:

        - Capture fine details from earlier layers, aiding precise segmentation.
    - Global Spatial Information:
        - Integrate high-level features, providing context for better overall predictions.

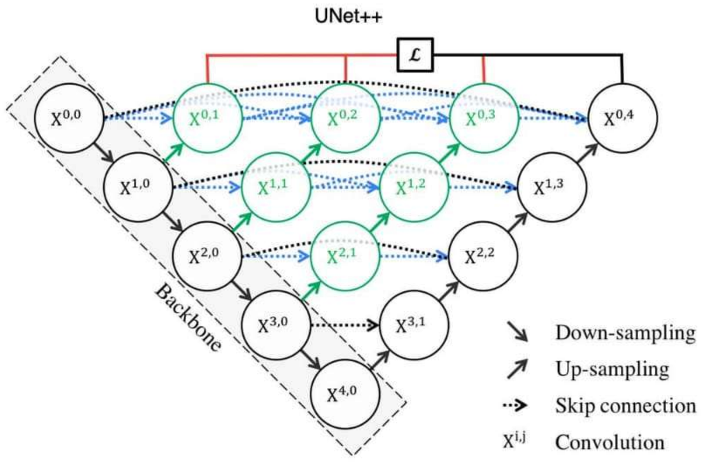
U-NET Variants:

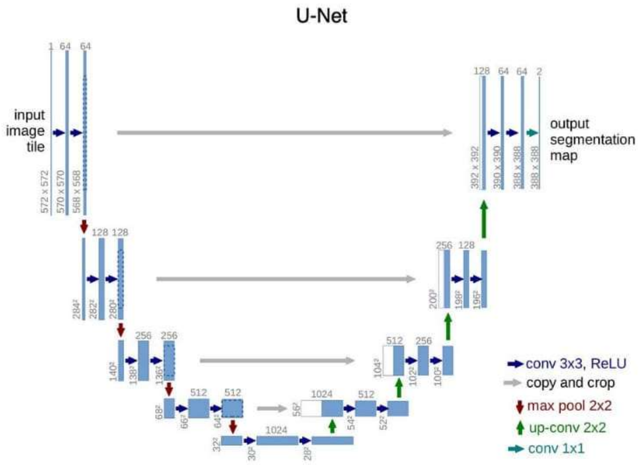
1. **U-net ++(nested)**
2. **Attention U-net**

U-NET ++ (NESTED) :

U-Net++ architecture is a semantic segmentation architecture based on U-Net. They introduced two main innovations in the traditional U-Net, architecture namely, nested dense skip connections and deep supervision.

- Nested Skip Connections:
    - U-Net++ architecture uses a nested skip connection network to aggregate the features while decoding the encoded data. By aggregating these features from different paths in the network, the model is able to improve the accuracy of the segmentation mask.
- Deep Supervision:
    - U-Net++ architecture uses deep supervision to enhance the model performance by providing a regularization to the network while training.





Architecture

```
# Basic Convolution Block
class ConvBlock(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(ConvBlock, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3, padding=1)
        self.relu = nn.ReLU(inplace=True)
    def forward(self, x):
        x = self.relu(self.conv1(x))
        x = self.relu(self.conv2(x))
        return x
```

```python
# Up-sampling block for decoding path
class UpBlock(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(UpBlock, self).__init__()
        self.up = nn.ConvTranspose2d(in_channels, out_channels, kernel_size=2, stride=2)
        self.conv = ConvBlock(in_channels, out_channels)
    def forward(self, x, skip):
        x = self.up(x)
        x = torch.cat((x, skip), dim=1)  # Concatenate with the skip connection
        x = self.conv(x)
        return x

# U-Net++ Architecture
class NestedUNet(nn.Module):
    def __init__(self, in_channels=1, num_classes=1):
        super(NestedUNet, self).__init__()

        # Encoder Path
        self.conv1_0 = ConvBlock(in_channels, 64)
        self.conv2_0 = ConvBlock(64, 128)
        self.conv3_0 = ConvBlock(128, 256)
        self.conv4_0 = ConvBlock(256, 512)
        self.conv5_0 = ConvBlock(512, 1024)


        # Decoder Path (with nested blocks)
        self.up4_1 = UpBlock(1024, 512)
        self.up3_2 = UpBlock(512, 256)
        self.up2_3 = UpBlock(256, 128)
        self.up1_4 = UpBlock(128, 64)
        self.up3_1 = UpBlock(512, 256)
        self.up2_2 = UpBlock(256, 128)
        self.up1_3 = UpBlock(128, 64)
        self.up2_1 = UpBlock(256, 128)
        self.up1_2 = UpBlock(128, 64)
        self.up1_1 = UpBlock(128, 64)

        # Final Convolution: Adjust output channels according to num_classes
        self.final_conv = nn.Conv2d(64, num_classes, kernel_size=1)

     def forward(self, x):
        # Encoder
        x1_0 = self.conv1_0(x)   # Encoder level 1
        x2_0 = self.conv2_0(F.max_pool2d(x1_0, 2))  # Encoder level 2
        x3_0 = self.conv3_0(F.max_pool2d(x2_0, 2))  # Encoder level 3
        x4_0 = self.conv4_0(F.max_pool2d(x3_0, 2))  # Encoder level 4
        x5_0 = self.conv5_0(F.max_pool2d(x4_0, 2))  # Encoder level 5

        # Decoder with Nested Skip Connections
        x4_1 = self.up4_1(x5_0, x4_0)  # First level decoding
        x3_2 = self.up3_2(x4_1, x3_0)  # Second level decoding
        x2_3 = self.up2_3(x3_2, x2_0)  # Third level decoding
        x1_4 = self.up1_4(x2_3, x1_0)  # Fourth level decoding
        x3_1 = self.up3_1(x4_0, x3_0)
        x2_2 = self.up2_2(x3_1, x2_0)
        x1_3 = self.up1_3(x2_2, x1_0)
        x2_1 = self.up2_1(x3_0, x2_0)
        x1_2 = self.up1_2(x2_1, x1_0)
        x1_1 = self.up1_1(x2_0, x1_0)

        # Output
        output = self.final_conv(x1_4)  # Apply final 1x1 convolution to get segmentation mask
```

. **Nested Design**: Combines multiple skip connections for better feature utilization between the encoder and decoder.

· **Convolution Blocks**: Uses two convolutional layers in each block to extract detailed features.

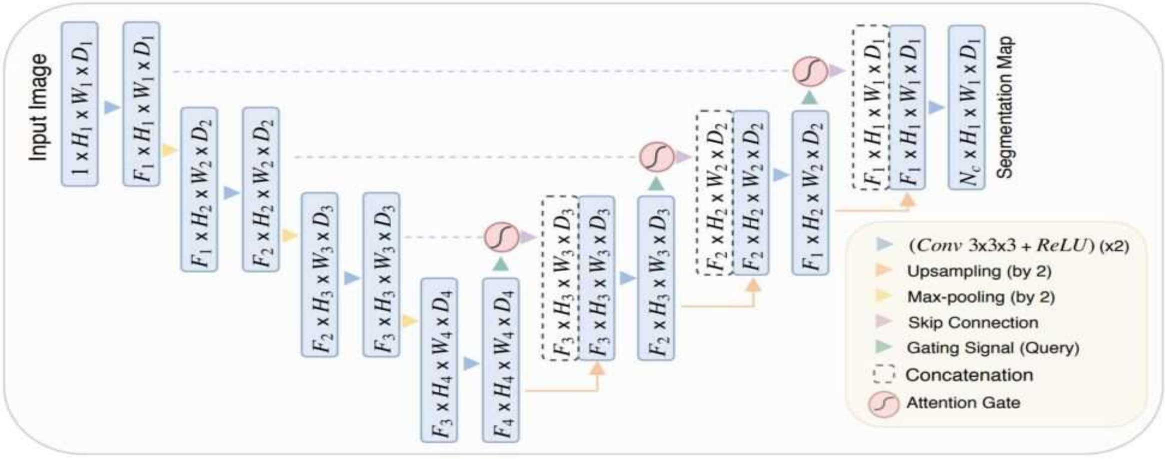
· **Up-Sampling**: Applies transposed convolutions to progressively reconstruct the segmentation map.

· **Final Output**: Ends with a 1×11 \times 11×1 convolution to produce a pixel-wise segmentation mask.

Attention U-net

· **Attention Mechanism**:

- Focuses on different parts of the input image to determine which areas are most relevant for the task
- Helps improve the model's ability to distinguish between relevant and irrelevant features.



Input Image
$1 \times H_1 \times W_1 \times D_1$
$F_1 \times H_1 \times W_1 \times D_1$
$F_1 \times H_2 \times W_2 \times D_2$
$F_2 \times H_2 \times W_2 \times D_2$
$F_2 \times H_3 \times W_3 \times D_3$
$F_3 \times H_3 \times W_3 \times D_3$
$F_3 \times H_4 \times W_4 \times D_4$
$F_4 \times H_4 \times W_4 \times D_4$
$F_3 \times H_3 \times W_3 \times D_3$
$F_3 \times H_3 \times W_3 \times D_3$
$F_2 \times H_3 \times W_3 \times D_3$
$F_2 \times H_2 \times W_2 \times D_2$
$F_2 \times H_2 \times W_2 \times D_2$
$F_1 \times H_2 \times W_2 \times D_2$
$F_1 \times H_1 \times W_1 \times D_1$
$F_1 \times H_1 \times W_1 \times D_1$
$N_c \times H_1 \times W_1 \times D_1$
Segmentation Map
$(Conv\ 3x3x3 + ReLU)$ (x2)
Upsampling (by 2)
Max-pooling (by 2)
Skip Connection
Gating Signal (Query)
Concatenation
Attention Gate

Attention Gates (AGs):

- **Functionality**: AGs are placed at the skip connections to filter features passed from the encoder to the decoder.
    - Each gate computes an attention coefficient based on the features from both the encoder and decoder paths.
    - The output is a spatial attention map that is used to scale the feature maps from the encoder.

Architeture:

```python
# Attention Gate
class AttentionGate(nn.Module):
   def __init__(self, F_g, F_l, F_int):
      super(AttentionGate, self).__init__()
      self.W_g = nn.Conv2d(F_g, F_int, kernel_size=1, padding=0)
      self.W_x = nn.Conv2d(F_l, F_int, kernel_size=1, padding=0)
      self.psi = nn.Conv2d(F_int, 1, kernel_size=1, padding=0)
      self.relu = nn.ReLU(inplace=True)
   def forward(self, g, x):
      g1 = self.W_g(g)
      x1 = self.W_x(x)
      psi = self.relu(g1 + x1)
      psi = self.psi(psi)
      attention = torch.sigmoid(psi)
      return x * attention

# Basic Convolution Block
class ConvBlock(nn.Module):
   def __init__(self, in_channels, out_channels):
      super(ConvBlock, self).__init__()
      self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1)
      self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3, padding=1)
      self.relu = nn.ReLU(inplace=True)
   def forward(self, x):
      x = self.relu(self.conv1(x))
      x = self.relu(self.conv2(x))
      return x

# Up-sampling block for decoding path
class UpBlock(nn.Module):
   def __init__(self, in_channels, out_channels):
      super(UpBlock, self).__init__()
      self.up = nn.ConvTranspose2d(in_channels, out_channels, kernel_size=2, stride=2)
      self.conv = ConvBlock(in_channels, out_channels)
   def forward(self, x, skip):
      x = self.up(x)
      x = torch.cat((x, skip), dim=1)  # Concatenate with the skip connection
      x = self.conv(x)
      return x

# Attention U-Net Architecture
class AttentionUNet(nn.Module):
   def __init__(self, in_channels=1, num_classes=1):
      super(AttentionUNet, self).__init__()
      # Encoder Path
      self.conv1 = ConvBlock(in_channels, 64)
      self.conv2 = ConvBlock(64, 128)

      self.conv3 = ConvBlock(128, 256)
      self.conv4 = ConvBlock(256, 512)
      self.conv5 = ConvBlock(512, 1024)
      # Decoder Path
      self.up4 = UpBlock(1024, 512)
      self.up3 = UpBlock(512, 256)
```

```
        self.up2 = UpBlock(256, 128)
        self.up1 = UpBlock(128, 64)
        # Attention Gates
        self.att4 = AttentionGate(F_g=512, F_l=512, F_int=256)
        self.att3 = AttentionGate(F_g=256, F_l=256, F_int=128)
        self.att2 = AttentionGate(F_g=128, F_l=128, F_int=64)
        self.att1 = AttentionGate(F_g=64, F_l=64, F_int=32)
        # Final Convolution
        self.final_conv = nn.Conv2d(64, num_classes, kernel_size=1)
    def forward(self, x):
        # Encoder
        x1 = self.conv1(x)   # Level 1
        x2 = self.conv2(F.max_pool2d(x1, 2))  # Level 2
        x3 = self.conv3(F.max_pool2d(x2, 2))  # Level 3
        x4 = self.conv4(F.max_pool2d(x3, 2))  # Level 4
        x5 = self.conv5(F.max_pool2d(x4, 2))  # Level 5
        # Decoder with Attention Gates
        x4_up = self.up4(x5, x4)
        x4_att = self.att4(x4_up, x4)
        x3_up = self.up3(x4_att, x3)
        x3_att = self.att3(x3_up, x3)
        x2_up = self.up2(x3_att, x2)
        x2_att = self.att2(x2_up, x2)
        x1_up = self.up1(x2_att, x1)
        x1_att = self.att1(x1_up, x1)
        # Output
        output = self.final_conv(x1_att)
```

· Downsampling (Encoder): This path extracts context information by progressively applying convolutions and max-pooling layers to reduce the spatial dimensions of the input while capturing abstract features.

· Attention Gate in Skip Connection: As the features are passed from the encoder to the decoder via skip connections, an attention gate is applied. This gate computes an attention coefficient that weighs the feature maps by comparing them to the output of the decoder. Only features deemed relevant by the attention mechanism are passed forward.

· Upsampling (Decoder): This path reconstructs the feature maps back to the original image resolution. The skip connections with attention help by feeding essential features from the downsampling layers to the upsampling layers.

IMPLEMENTATION:

Data Preprocessing

- CLAHE Preprocessing
- Normalization and Augmentation

Model Implementation

- Nested U-Net (U-Net++)
- Attention U-Net

Model Training and Evaluation

- Training
- DICE Score Evaluation

DATA PREPROCESSING:

Custom Dataset: The Metastasis3DDataset class loads and preprocesses 3D brain MRI images and masks from a directory.

CLAHE: The apply_clahe function enhances image contrast using CLAHE on grayscale images.

Data Augmentation: The get_transform function applies random flips for training and normalizes images and masks.

DataLoader: The dataset is instantiated, and a DataLoader is created for efficient batch loading during model training.

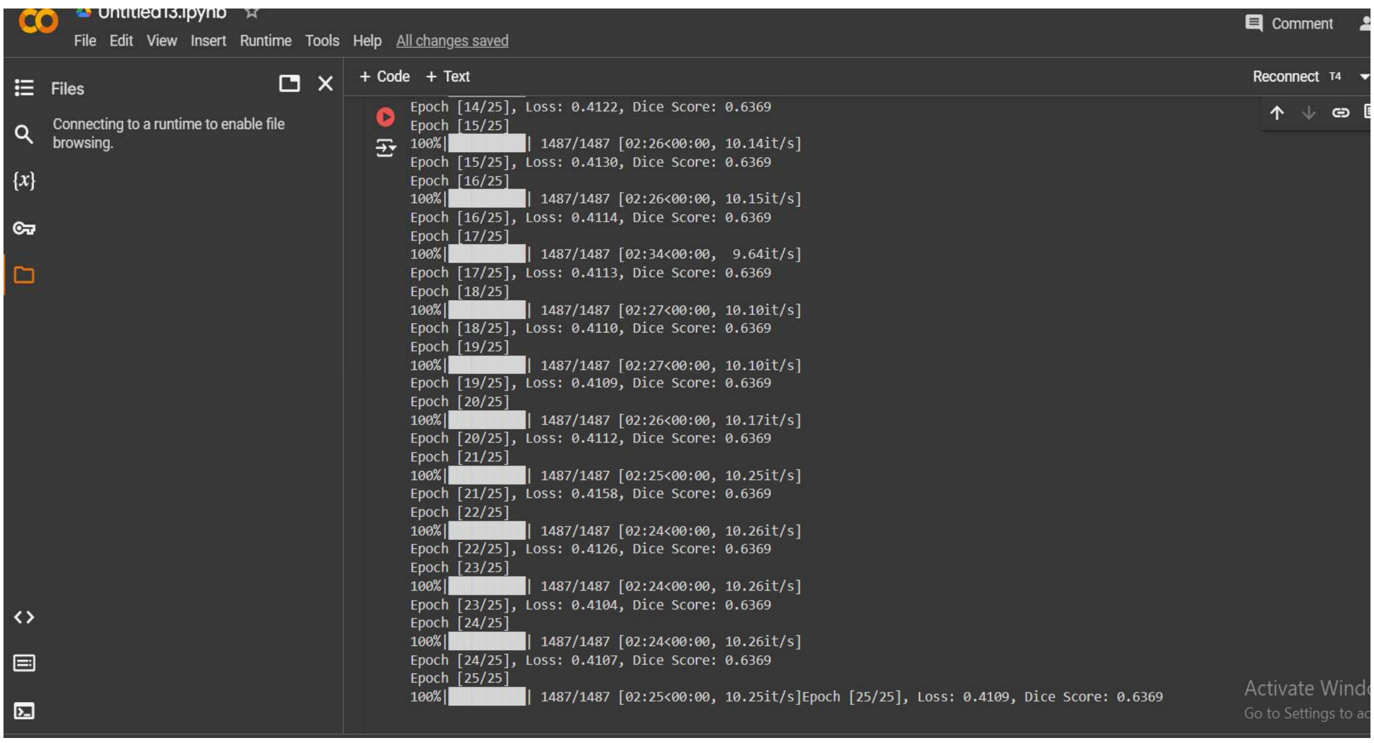
MODEL IMPLEMENTATION:

Initialize the model: The network architecture, like UNet++ or Attention UNet, is initialized with encoder, decoder, and attention components for feature extraction and segmentation.

Define the loss function and optimizer: Choose an appropriate loss function and an optimizer to minimize the loss during training, adjusting model weights accordingly.

MODAL TRAINING:

The train_model function trains the Nested U-Net and Attention matrix using a combination of BCE and Dice loss over a specified number of epochs, updating model weights based on the computed loss for each batch.


Untitled13.ipynb
File Edit View Insert Runtime Tools Help All changes saved
Comment
Files
Connecting to a runtime to enable file browsing.
+ Code + Text
Reconnect T4
Activate Windows
Go to Settings to activate

```
Epoch [14/25], Loss: 0.4122, Dice Score: 0.6369
Epoch [15/25]
100%|██████████| 1487/1487 [02:26<00:00, 10.14it/s]
Epoch [15/25], Loss: 0.4130, Dice Score: 0.6369
Epoch [16/25]
100%|██████████| 1487/1487 [02:26<00:00, 10.15it/s]
Epoch [16/25], Loss: 0.4114, Dice Score: 0.6369
Epoch [17/25]
100%|██████████| 1487/1487 [02:34<00:00,  9.64it/s]
Epoch [17/25], Loss: 0.4113, Dice Score: 0.6369
Epoch [18/25]
100%|██████████| 1487/1487 [02:27<00:00, 10.10it/s]
Epoch [18/25], Loss: 0.4110, Dice Score: 0.6369
Epoch [19/25]
100%|██████████| 1487/1487 [02:27<00:00, 10.10it/s]
Epoch [19/25], Loss: 0.4109, Dice Score: 0.6369
Epoch [20/25]
100%|██████████| 1487/1487 [02:26<00:00, 10.17it/s]
Epoch [20/25], Loss: 0.4112, Dice Score: 0.6369
Epoch [21/25]
100%|██████████| 1487/1487 [02:25<00:00, 10.25it/s]
Epoch [21/25], Loss: 0.4158, Dice Score: 0.6369
Epoch [22/25]
100%|██████████| 1487/1487 [02:24<00:00, 10.26it/s]
Epoch [22/25], Loss: 0.4126, Dice Score: 0.6369
Epoch [23/25]
100%|██████████| 1487/1487 [02:24<00:00, 10.26it/s]
Epoch [23/25], Loss: 0.4104, Dice Score: 0.6369
Epoch [24/25]
100%|██████████| 1487/1487 [02:24<00:00, 10.26it/s]
Epoch [24/25], Loss: 0.4107, Dice Score: 0.6369
Epoch [25/25]
100%|██████████| 1487/1487 [02:25<00:00, 10.25it/s]Epoch [25/25], Loss: 0.4109, Dice Score: 0.6369
```

DICE SCORE:

The DICE (or Dice) score is a common evaluation metric used in computer vision and medical imaging for measuring the similarity or overlap between two sets, typically used in the context of image segmentation and object detection. It quantifies how well the predicted region aligns with the ground truth region.

DICE Score = (2 * Intersection) / (Area of Set A + Area of Set B)

Intersection-- refers to the number of overlapping or common elements (pixels or regions) between the predicted segmentation (Set A) and the ground truth segmentation (Set B).

Area of Set A--represents the total number of elements (pixels or regions) in the predicted segmentation.

Area of Set B-- represents the total number of elements (pixels or regions) in the ground truth segmentation.