BRAIN TUMOR MRI IMAGE CLASSIFICATION AND SEGMENTATION

Chenhao Wu
Shanghai Jiao Tong University

Yubing Zhao
Tianjin University of Technology

Tongyu Wu
Shandong University

Yuntian Zhang
Chinese University of Hongkong, Shenzhen

ABSTRACT

Non-invasive imaging techniques, particularly Magnetic Resonance Imaging (MRI), play a pivotal role in diagnosing brain tumors by providing detailed anatomical and functional insights. In the classification task, we employed the VGG16 neural network architecture and achieved an accuracy of 96% on the test set. For the segmentation task, we referred to the U-Net architecture, designed our own loss function, and incorporated extensive data augmentation techniques. Ultimately, we developed a U-Net3D network architecture that directly processes 3D files. The Dice coefficient reached 83.0% on the training set and 83.2% on the test set.

***K*eywords** MRI · Brain tumor · Image classification · VGG16 · Image segmentation · Unet3D

1 Literature Review

1.1 Overview of Non-invasive Imaging Techniques

Non-invasive techniques like CT, PET, and MRI are widely used for diagnosing internal organs. CT uses X-rays, and PET involves radioactive isotopes, while MRI, being harmless, provides clear imaging and is often chosen for diagnosing organ abnormalities. Despite advancements in deep learning, confirming the malignancy of brain tumors non-invasively remains a challenge.

1.2 MRI Imaging Principle and Image Types

MRI Imaging Principle and Image Types MRI works by using a magnetic field to excite hydrogen atoms in the body, which emit radio frequency (RF) pulses when the field is removed. These atoms return to their original state, and the time required for this process is called the relaxation time, which is divided into longitudinal (T1) and transverse (T2) relaxation times [1][2]. Based on that, three main MRI sequences are obtained: T1-weighted (T1w), T2-weighted (T2w), and FLAIR. T1w images show bone structures, T2w highlights tumor cores, and FLAIR suppresses normal CSF, enhancing the visibility of abnormal tissues, crucial for tumor detection.

1.3 Tumor MRI Image Classification

Tumor MRI Image Classification Various methods have been developed for classifying brain tumors in MRI images. CNNs are frequently used in tumor classification. Jun Cheng et al. [3] achieved 94.68 accuracy without neural networks, while Gawande and Mendre [4] employed DNNs and autoencoders. Pereira et al. [5] used small 3x3 kernels to reduce overfitting. Other architectures like AlexNet [6], VGG16 [5], and ResNet [7]were tested, showing the flexibility of CNNs for brain tumor classification.

1.4 Tumor MRI Image Segmentation

Traditional segmentation methods rely on techniques like edge detection and region growing. However, CNNs have revolutionized this task, eliminating manual feature engineering. U-Net, proposed by Ronneberger et al. [8], is a deep learning network designed for medical image segmentation. Its key feature is the use of skip connections, allowing for precise localization while preserving spatial information. U-Net's architecture has been enhanced through modifications such as skip connection improvements, backbone design changes, and transformer integration [9]. For 3D data, Cicek et al. [10] introduced 3D U-Net, which reduces manual annotation efforts by utilizing adjacent slice information, improving segmentation performance in volumetric datasets. [11]In 2024, Paul Jaeger et al. reassessed the performance of the nnU-Net framework in 3D medical image segmentation, providing empirical evidence supporting the continued competitiveness of 3D U-Net. [12]In the same year, Tianrun Chen et al. proposed using Vision-LSTM as the backbone of U-Net, combining the local feature extraction capability of convolution with the long-range dependency modeling of LSTM. Recent models like the Segment Anything Model (SAM) [13] and its adaptation MedSAM [14] have further advanced segmentation. SAM uses prompt-based learning, allowing for zero-shot segmentation across diverse tasks. MedSAM, optimized for high-resolution medical images, improves segmentation accuracy by leveraging expert annotations and adapting to 3D CT and MRI data. These advancements, especially with U-Net and its variants, have significantly enhanced segmentation efficiency and accuracy in medical imaging, particularly for brain tumor detection.

2 Classification

2.1 Design of a Transfer Learning Classification Model Based on VGG16

2.1.1 DataGenerator

Use `keras.utils.Sequence` to implement a custom data generator (DataGenerator) for batch loading of .npy format image data, performing data preprocessing, augmentation, and binary classification based on segmentation data.

2.1.2 VGG16 Transfer Learning

Using VGG16 shown in Figure 1 as a feature extractor by loading ImageNet pre-trained weights, removing the original fully connected layers while retaining only the convolutional part. The last 12 convolutional layers are unfrozen, allowing the model to adapt to a new classification task while reducing computational cost.

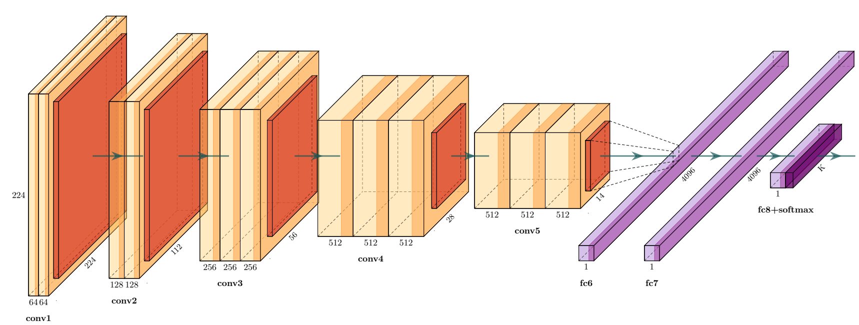


Figure 1: VGG16 network architecture

2.1.3 Training Process and Early Stopping Strategy

A well-designed early stopping strategy can effectively prevent the model from overfitting. If the validation accuracy (val_accuracy) does not improve within 15 epochs, training will be halted, and the model will revert to the best weights. The learning rate is dynamically adjusted if the validation loss (val_loss) does not show improvement within 6 epochs; the learning rate is reduced by 20%, with a lower bound of 1e-9 to prevent the learning rate from becoming too small. In the model training phase, the optimal model was obtained by adjusting the early stopping condition, modifying the number of trainable layers, and modifying the dynamic adjustment of the learning rate parameter.

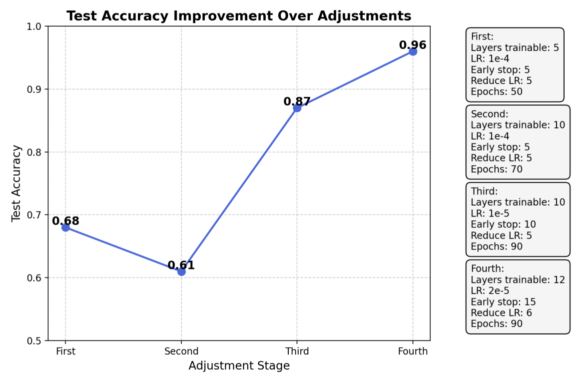


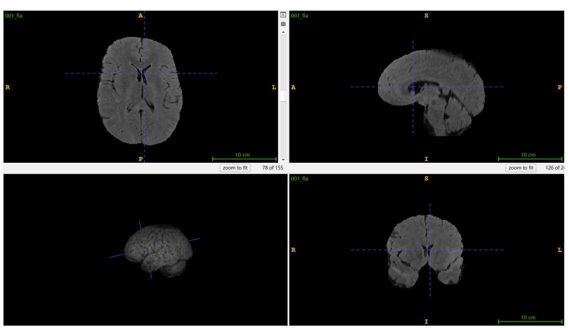
Figure 2: Test Accuracy Improvement Over Adjustments

3 Segmentation

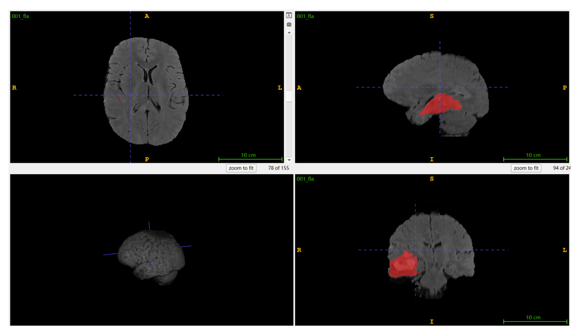
3.1 Data Visualization

The provided dataset consists of 210 3D brain MRI images with a resolution of 240×240×155. Each entry includes two files: `xxx_fla.nii.gz` and `xxx_seg.nii.gz`, which represent the 3D brain MRI image and the tumor mask data, respectively. `.nii` is a medical imaging data format that stores 3D MRI image data of the brain. Each data entry comprises 155 axial slices. There are two ways to visualize the data.

The most straightforward method is to utilize the ITK-SNAP software. By importing the two files as visualization data and mask data respectively, one can observe the brain MRI images and the corresponding tumor locations from various sectional views, as depicted in Figure 3.



(a) Brain MRI image slice



(b) Brain MRI image slice with tumor mask

Figure 3: Visualization using ITK-SNAP software. The tumor region is marked in red in Figure 3b.

Alternatively, the Python library Nibabel can be employed to read and load the data as three-dimensional arrays. We have also conducted some visualization demonstrations, as shown in Figure 4.

3.2 Data Argumentation

In the initial training phase, we did not incorporate data augmentation. However, this led to overfitting, as evidenced by the high accuracy on the training set and the significantly lower accuracy on the validation set. By introducing data augmentation, we were able to reduce the model's sensitivity to specific image features, thereby enhancing its generalization capability.

In Figure 5 we present five different types of data augmentations, including random flipping, random rotation, random scaling, adding noise and random cropping.

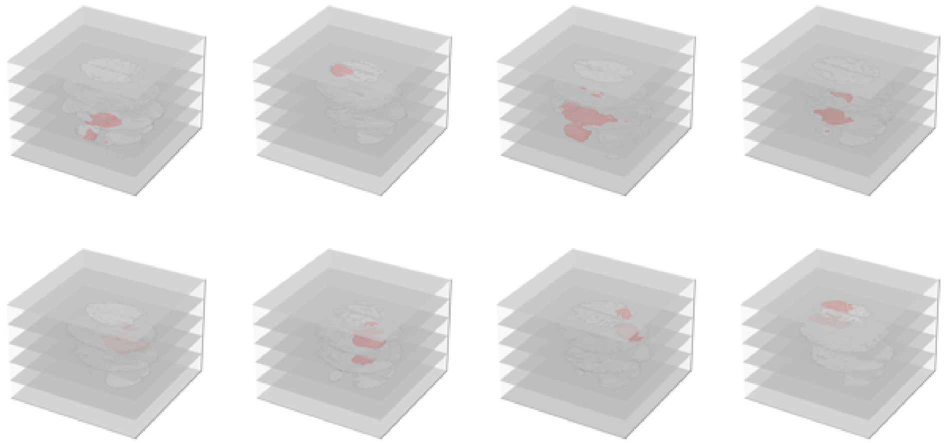


Figure 4: Visualization using Nibabel library in Python.The brain slices are visualized by overlaying multiple axial views, with the tumor regions marked in red.

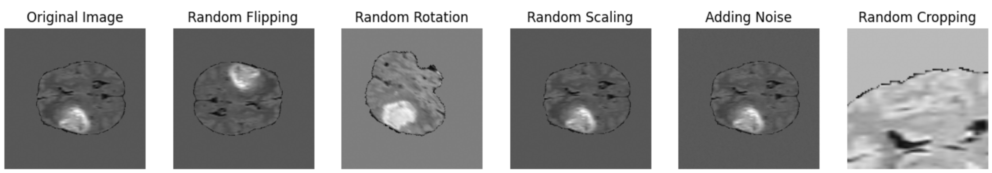


Figure 5: Several data augmentation methods

3.3 Methodology

3.3.1 Loss Function

For medical image segmentation, the IoU loss and Dice loss are commonly used as loss functions. They are defined as below.

$$\text{IoU loss} = 1 - \frac{Area\ of\ Overlap}{Area\ of\ Union}$$

$$\text{Dice loss} = 1 - \frac{2 \times Area\ of\ Overlap}{Total\ Area}$$

Here we combine the binary cross-entropy loss with the dice loss [15]. The overall loss function is calculated as below:

$$L_{total} = L_{BCE} - D$$

$$L_{BCE} = -\frac{1}{N}\sum_{i=1}^{N}[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)]$$

$$D = \frac{2\sum_{i=1}^{N} y_i \hat{y}_i}{\sum_{i=1}^{N} y_i + \sum_{i=1}^{N} \hat{y}_i}$$

3.3.2 Optimizer

We chose Adam optimizer as our optimizer. It is widely used in deep learning due to its efficiency and adaptability. Adam combines the advantages of both RMSprop and Momentum methods, allowing for adaptive learning rates and faster convergence.

3.3.3 Network Architecture

We have replicated the 3D U-Net architecture. The network consists of an encoder path and a decoder path, each with four resolution steps.

As shown in Figure 6, in the encoder path, each layer comprises two $3 \times 3 \times 3$ convolutions followed by ReLU activation and a $2 \times 2 \times 2$ max pooling operation for downsampling. In the decoder path, upconvolutions (transposed convolutions) are used instead of regular convolutions to achieve upsampling. Each upconvolution is followed by two $3 \times 3 \times 3$ convolutions and ReLU activation. The final layer employs a $1 \times 1 \times 1$ convolution to produce the segmentation output. To address the issue of overfitting, we experimented with several regularization techniques: 1:L2 regularization was applied to the network weights. 2:Batch normalization layers were added after each convolution to stabilize training. 3:Dropout layers (with a dropout rate of 50%) were incorporated into each layer to further reduce overfitting.

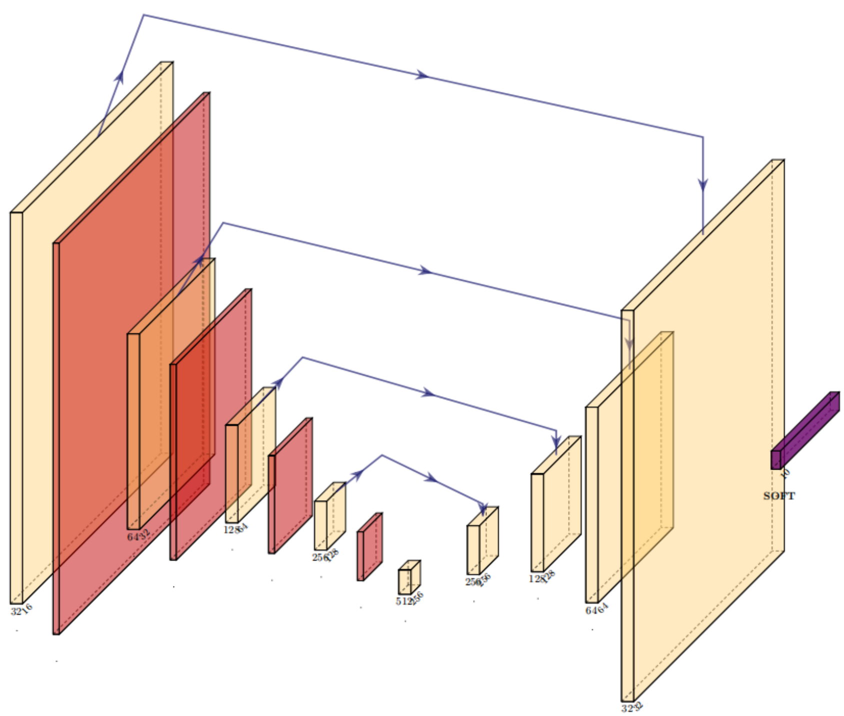


Figure 6: 3D U-Net architecture

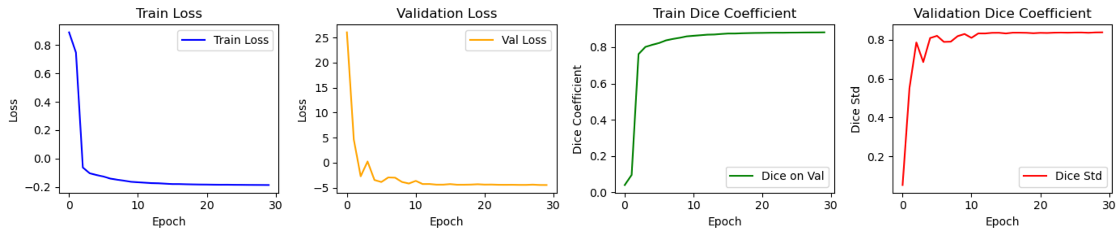
3.4 Experiment Design

All experiments were conducted on a NVIDIA GeForce RTX 4090 with 24GB of memory.

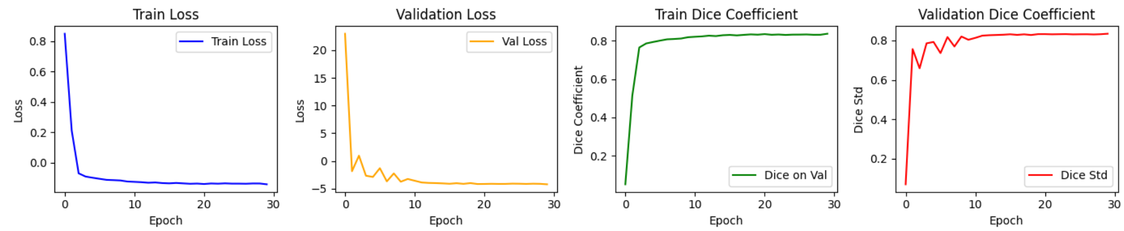
Data Preprocessing Dataset is randomly divided into an 80% training set and a 20% validation set. Before training, the non-zero regions of the 3D images in the entire dataset are normalized. Subsequently, data augmentation is performed on the training set, including random flipping, random rotation, random scaling, adding noise, and random cropping.

Training Due to limitations of memory, we set the batch size of the model at 1 and the learning rate at 3×10^{-4}. We used StepLR module to dynamically decrease the learning rate in the training process to achieve higher accuracy.

Results During training, we initially refrained from using extensive data augmentation techniques and only applied simple normalization. The results are shown in Figure 7a. The model was trained for a total of 30 epochs, achieving a Dice coefficient of 80% on the training set and 83% on the validation set. After employing data augmentation techniques, the results are shown in Figure 7b. The model was trained for 30 epochs, achieving a Dice coefficient of 83.0% on the training set and 83.2% on the validation set. It can be observed that the accuracy on both the training and validation sets is more balanced, which reduces the overfitting of the model and enhances its robustness.



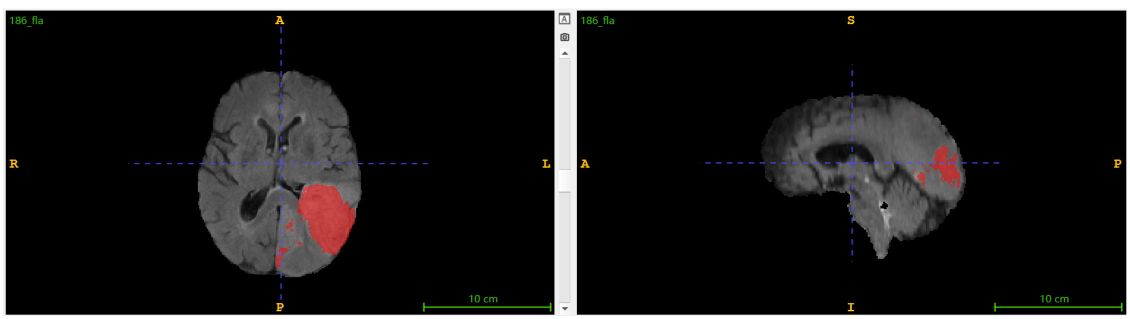
(a) Training Result I with data augmentation



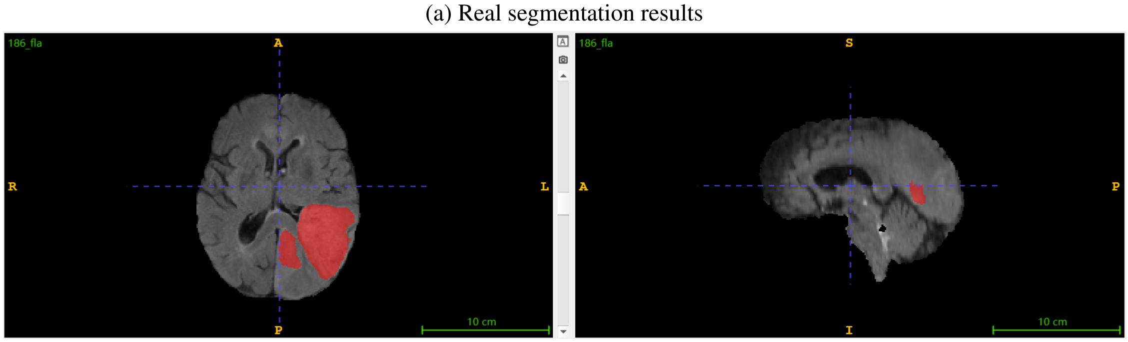
(b) Training Result II without data augmentation

Figure 7: Training Results

Finally, a series of data was randomly selected, and the prediction results along with the ground truth are shown in Figure 8. It can be observed that most regions of the prediction are consistent with the ground truth, with only a few areas of significant color change failing to be accurately predicted.



(a) Real segmentation results



(b) Predicted segmentation Results

Figure 8: Training Results shown with ITK-SNAP

References

[1] Stuart Clare. Functional mri: methods and applications. *University of Nottingham*, page 155, 1997.

[2] Lakshmaiah C Kuntegowdenahalli, Linu Abraham Jacob, Ashok S Komaranchath, and Usha Amirtham. A rare case of primary anaplastic large cell lymphoma of the central nervous system. *Journal of Cancer Research and Therapeutics*, 11(4):943–945, 2015.

[3] Jun Cheng, Wei Huang, Shuangliang Cao, Ru Yang, Wei Yang, Zhaoqiang Yun, Zhijian Wang, and Qianjin Feng. Enhanced performance of brain tumor classification via tumor region augmentation and partition. *PloS one*, 10(10):e0140381, 2015.

[4] SS Gawande and V Mendre. Brain tumor diagnosis using deep neural network (dnn). *International Journal of Advanced Research in Electrical, Electronics and Instrumentation Engineering*, 5(5):10196–10203, 2017.

[5] Sérgio Pereira, Adriano Pinto, Victor Alves, and Carlos A Silva. Brain tumor segmentation using convolutional neural networks in mri images. *IEEE transactions on medical imaging*, 35(5):1240–1251, 2016.

[6] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25, 2012.

[7] Karen Simonyan. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[8] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *Medical image computing and computer-assisted intervention–MICCAI 2015: 18th international conference, Munich, Germany, October 5-9, 2015, proceedings, part III 18*, pages 234–241. Springer, 2015.

[9] Reza Azad, Ehsan Khodapanah Aghdam, Amelie Rauland, Yiwei Jia, Atlas Haddadi Avval, Afshin Bozorgpour, Sanaz Karimijafarbigloo, Joseph Paul Cohen, Ehsan Adeli, and Dorit Merhof. Medical image segmentation review: The success of u-net. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2024.

[10] Özgün Çiçek, Ahmed Abdulkadir, Soeren S Lienkamp, Thomas Brox, and Olaf Ronneberger. 3d u-net: learning dense volumetric segmentation from sparse annotation. In *Medical Image Computing and Computer-Assisted Intervention–MICCAI 2016: 19th International Conference, Athens, Greece, October 17-21, 2016, Proceedings, Part II 19*, pages 424–432. Springer, 2016.

[11] Fabian Isensee, Tassilo Wald, Constantin Ulrich, Michael Baumgartner, Saikat Roy, Klaus Maier-Hein, and Paul F Jaeger. nnu-net revisited: A call for rigorous validation in 3d medical image segmentation. In *International Conference on Medical Image Computing and Computer-Assisted Intervention*, pages 488–498. Springer, 2024.

[12] Tianrun Chen, Chaotao Ding, Lanyun Zhu, Tao Xu, Deyi Ji, Yan Wang, Ying Zang, and Zejian Li. xlstm-unet can be an effective 2d & 3d medical image segmentation backbone with vision-lstm (vil) better than its mamba counterpart. *arXiv preprint arXiv:2407.01530*, 2024.

[13] Alexander Kirillov, Eric Mintun, Nikhila Ravi, Hanzi Mao, Chloe Rolland, Laura Gustafson, Tete Xiao, Spencer Whitehead, Alexander C Berg, Wan-Yen Lo, et al. Segment anything. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 4015–4026, 2023.

[14] Jun Ma, Yuting He, Feifei Li, Lin Han, Chenyu You, and Bo Wang. Segment anything in medical images. *Nature Communications*, 15(1):654, 2024.

[15] Fabian Isensee, Jens Petersen, Andre Klein, David Zimmerer, Paul F. Jaeger, Simon Kohl, Jakob Wasserthal, Gregor Koehler, Tobias Norajitra, Sebastian Wirkert, and Klaus H. Maier-Hein. nnu-net: Self-adapting framework for u-net-based medical image segmentation, 2018.

A Appendix: Segmentation Code

Unet3D_dataset.py

```python
import os
import numpy as np
import torch
from torch.utils.data import Dataset, DataLoader
import nibabel as nib
import monai.transforms as mt

transforms_ex1 = mt.Compose([
        mt.RandFlipd(keys=["image", "seg"], prob=0.5, spatial_axis=0),
        mt.RandFlipd(keys=["image", "seg"], prob=0.5, spatial_axis=1),
        mt.RandFlipd(keys=["image", "seg"], prob=0.5, spatial_axis=2),

        mt.RandRotated(keys=["image", "seg"], prob=0.5, range_x=(-15, 15), range_y=(-15, 15), range_z=(-15, 15), mode="nearest", padding_mode="zeros"),

        mt.RandZoomd(keys=["image", "seg"], prob=0.5, min_zoom=0.9, max_zoom=1.1, mode="nearest"),

        mt.RandGaussianNoised(keys=["image"], prob=0.2, mean=0.0, std=0.1),

        mt.RandShiftIntensityd(keys=["image"], prob=0.2, offsets=0.1),

        mt.RandSpatialCropd(keys=["image", "seg"], roi_size=(96, 96, 96), random_size=False),

        mt.EnsureTyped(keys=["image", "seg"], dtype="float32")
    ])

class MRIDataset(Dataset):
    def __init__(self, root_dir, mode="test", transform=None):
        self.root_dir = root_dir
        self.transform = transform
        self.mode = mode
        self.file_list = self.load_file()

    def load_file(self):
        file_list = []
        for file_name in os.listdir(self.root_dir):
            file_path = os.path.join(self.root_dir, file_name)
            if os.path.isfile(file_path) and file_name.endswith("_fla.nii.gz"):
                seg_file_name = file_name.replace("_fla.nii.gz", "_seg.nii.gz")
                seg_file_path = os.path.join(self.root_dir, seg_file_name)
                if os.path.exists(seg_file_path):
                    file_list.append((file_path, seg_file_path))
        return file_list

    def __len__(self):
        return len(self.file_list)

    def __getitem__(self, idx):
        img_path, seg_path = self.file_list[idx]

        img_nifti = nib.load(img_path)
        seg_nifti = nib.load(seg_path)
        img_data = img_nifti.get_fdata()
```

```
        seg_data = seg_nifti.get_fdata()

        non_zero_mask = img_data != 0
        mean = np.mean(img_data[non_zero_mask])
        std = np.std(img_data[non_zero_mask])
        img_data[non_zero_mask] = (img_data[non_zero_mask] - mean) / std

        img_tensor = torch.from_numpy(img_data).float().unsqueeze(0)  # Shape: [1, H, W, D]
        seg_tensor = torch.from_numpy(seg_data).long().unsqueeze(0)   # Shape: [1, H, W, D]

        if self.transform:
            data_dict = {
                "image": img_tensor,
                "seg": seg_tensor
            }
            data_dict = self.transform(data_dict)
            img_tensor = data_dict["image"]
            seg_tensor = data_dict["seg"]

        img_tensor = img_tensor.permute(0, 3, 1, 2)
        seg_tensor = seg_tensor.permute(0, 3, 1, 2)

        return img_tensor, seg_tensor
```

Unet3D_model.py

```
import torch
import torch.nn as nn
from torch.nn import Module, Sequential
from torch.nn import Conv3d, ConvTranspose3d, BatchNorm3d, AvgPool1d
from torch.nn import ReLU

class Conv3D_Block(Module):
    def __init__(self, inp_feat, out_feat, kernel=3, stride=1, padding=1, residual=None):
        super(Conv3D_Block, self).__init__()

        self.conv1 = Sequential(
            Conv3d(inp_feat, out_feat, kernel_size=kernel, stride=stride, padding=padding, bias=True),
            BatchNorm3d(out_feat),
            ReLU()
        )

        self.conv2 = Sequential(
            Conv3d(out_feat, out_feat, kernel_size=kernel, stride=stride, padding=padding, bias=True),
            BatchNorm3d(out_feat),
            ReLU()
        )

        self.residual = residual

        if self.residual is not None:
            self.residual_upsampler = Conv3d(inp_feat, out_feat, kernel_size=1, bias=False)

    def forward(self, x):
        res = x
```

```python

        if not self.residual:
            return self.conv2(self.conv1(x))
        else:
            return self.conv2(self.conv1(x)) + self.residual_upsampler(res)


class Deconv3D_Block(Module):
    def __init__(self, inp_feat, out_feat, kernel=4, stride=2, padding=1):
        super(Deconv3D_Block, self).__init__()

        self.deconv = Sequential(
            ConvTranspose3d(inp_feat, out_feat, kernel_size=(1, kernel, kernel),
                            stride=(1, stride, stride), padding=(0, padding, padding),
                            output_padding=0, bias=True),
            ReLU()
        )

    def forward(self, x):
        return self.deconv(x)


class ChannelPool3d(AvgPool1d):
    def __init__(self, kernel_size, stride, padding):
        super(ChannelPool3d, self).__init__(kernel_size, stride, padding)
        self.pool_1d = AvgPool1d(self.kernel_size, self.stride, self.padding, self.ceil_mode)

    def forward(self, inp):
        n, c, d, w, h = inp.size()
        inp = inp.view(n, c, d * w * h).permute(0, 2, 1)
        pooled = self.pool_1d(inp)
        c = int(c / self.kernel_size[0])
        return inp.view(n, c, d, w, h)


class UNet3D(Module):
    def __init__(self, num_channels=32, feat_channels=[16, 32, 64, 128, 256], residual='conv'):
        super(UNet3D, self).__init__()

        self.pool1 = nn.MaxPool3d((1, 2, 2))
        self.pool2 = nn.MaxPool3d((1, 2, 2))
        self.pool3 = nn.MaxPool3d((1, 2, 2))
        self.pool4 = nn.MaxPool3d((1, 2, 2))

        self.conv_blk1 = Conv3D_Block(num_channels, feat_channels[0], residual=residual)
        self.conv_blk2 = Conv3D_Block(feat_channels[0], feat_channels[1], residual=residual)
        self.conv_blk3 = Conv3D_Block(feat_channels[1], feat_channels[2], residual=residual)
        self.conv_blk4 = Conv3D_Block(feat_channels[2], feat_channels[3], residual=residual)
        self.conv_blk5 = Conv3D_Block(feat_channels[3], feat_channels[4], residual=residual)

        self.dec_conv_blk4 = Conv3D_Block(2 * feat_channels[3], feat_channels[3], residual=residual)
```

```
        self.dec_conv_blk3 = Conv3D_Block(2 * feat_channels[2], feat_channels[2], residual=residual)
        self.dec_conv_blk2 = Conv3D_Block(2 * feat_channels[1], feat_channels[1], residual=residual)
        self.dec_conv_blk1 = Conv3D_Block(2 * feat_channels[0], feat_channels[0], residual=residual)

        self.deconv_blk4 = Deconv3D_Block(feat_channels[4], feat_channels[3])
        self.deconv_blk3 = Deconv3D_Block(feat_channels[3], feat_channels[2])
        self.deconv_blk2 = Deconv3D_Block(feat_channels[2], feat_channels[1])
        self.deconv_blk1 = Deconv3D_Block(feat_channels[1], feat_channels[0])

        self.bn1 = nn.BatchNorm3d(feat_channels[0])
        self.bn2 = nn.BatchNorm3d(feat_channels[1])
        self.bn3 = nn.BatchNorm3d(feat_channels[2])
        self.bn4 = nn.BatchNorm3d(feat_channels[3])
        self.bn5 = nn.BatchNorm3d(feat_channels[4])

        self.bn_d4 = nn.BatchNorm3d(2 * feat_channels[3])
        self.bn_d3 = nn.BatchNorm3d(2 * feat_channels[2])
        self.bn_d2 = nn.BatchNorm3d(2 * feat_channels[1])
        self.bn_d1 = nn.BatchNorm3d(2 * feat_channels[0])

        self.dropout = nn.Dropout(0.5)

        self.one_conv = nn.Conv3d(feat_channels[0], num_channels, kernel_size=1, stride=1, padding=0, bias=True)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        x1 = self.conv_blk1(x)
        x_low1 = self.pool1(x1)

        x2 = self.conv_blk2(x_low1)
        x_low2 = self.pool2(x2)

        x3 = self.conv_blk3(x_low2)
        x_low3 = self.pool3(x3)

        x4 = self.conv_blk4(x_low3)
        x_low4 = self.pool4(x4)

        base = self.conv_blk5(x_low4)

        d4 = torch.cat([self.deconv_blk4(base), x4], dim=1)
        d_high4 = self.dec_conv_blk4(d4)

        d3 = torch.cat([self.deconv_blk3(d_high4), x3], dim=1)
        d_high3 = self.dec_conv_blk3(d3)

        d2 = torch.cat([self.deconv_blk2(d_high3), x2], dim=1)
        d_high2 = self.dec_conv_blk2(d2)

        d1 = torch.cat([self.deconv_blk1(d_high2), x1], dim=1)
        d_high1 = self.dec_conv_blk1(d1)

        seg = self.sigmoid(self.one_conv(d_high1))

        return seg
```

```python

def dice(inputs, targets):
    smooth = 1e-6
    intersection = ((inputs > 0.5).float() * targets).sum()

    return (2 * intersection + smooth) / (inputs.sum() + targets.sum() + smooth)

def loss_dice(inputs, targets):
    smooth = 1e-6
    inputs = inputs.view(-1)
    targets = targets.view(-1)
    intersection = ((inputs > 0.5).float() * targets).sum()

    dice = (2 * intersection + smooth) / (inputs.sum() + targets.sum() + smooth)

    return 1 - dice


class Loss(nn.Module):
    def __init__(self, type):
        super(Loss, self).__init__()
        self.type = type

    def forward(self, inputs, targets):
        if self.type == 'dice':
            return loss_dice(inputs, targets)
        elif self.type == 'mixed dice':
            smooth = 1.
            inputs_flat = inputs.reshape(-1)
            targets_flat = targets.reshape(-1)

            intersection = (inputs_flat * targets_flat).sum()

            ce = nn.BCEWithLogitsLoss()(inputs, targets.float())

            return ce - (2 * intersection + smooth) / (inputs_flat.sum() +
            ↪  targets_flat.sum() + smooth)
        elif self.type == 'mixed dice2':
            smooth = 1.
            inputs_flat = inputs.reshape(-1)
            targets_flat = targets.reshape(-1)

            intersection = (inputs_flat * targets_flat).sum()

            ce = nn.BCEWithLogitsLoss()(inputs, targets.float())

            return 0.3*ce - (1-0.3)*(2 * intersection + smooth) / (inputs_flat.sum() +
            ↪  targets_flat.sum() + smooth)
```

Unet3D_train.py

```python
import torch
from torch.utils.data import Dataset, DataLoader
from torch.optim.lr_scheduler import StepLR
from torch.optim import Adam
from tqdm import tqdm
import numpy as np

from Unet3D_dataset import MRIDataset, transforms_ex1
from Unet3D_model import UNet3D, Loss, dice
```

```python

# Dataset and DataLoader
train_dataset = MRIDataset(root_dir='./dataset_segmentation/train', mode="train", transform=transforms_ex1)
val_dataset = MRIDataset(root_dir='./dataset_segmentation/val', mode="test")
train_loader = DataLoader(train_dataset, batch_size=2, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=1, shuffle=False)

# Device
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(device)

# Model, Loss, Optimizer
model = UNet3D(num_channels=1, feat_channels=[16, 32, 64, 128, 256]).to(device)
criterion = Loss()
optimizer = Adam(model.parameters(), lr=3e-4)

# Training
num_epochs = 30
train_loss_list = []
val_loss_list = []
scheduler = StepLR(optimizer, step_size=1, gamma=0.8)

for epoch in range(num_epochs):
    train_loader_iter = tqdm(train_loader, desc=f'Epoch {epoch + 1}/{num_epochs}', leave=True)
    model.train()
    train_loss = []
    for data in train_loader_iter:
        inputs, targets = data
        inputs, targets = inputs.to(device), targets.to(device)
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, targets)
        loss.backward()
        optimizer.step()
        train_loader_iter.set_postfix({'Train Loss': loss.item(), 'Dice': dice(outputs, targets).item()})
        train_loss.append(loss.item())
    if epoch >= 4 and epoch <= 20:
        scheduler.step()

    model.eval()
    val_loss_total = 0.0
    val_dices = []
    with torch.no_grad():
        for data in val_loader:
            inputs, targets = data
            inputs, targets = inputs.to(device), targets.to(device)
            outputs = model(inputs)
            val_loss = criterion(outputs, targets)
            val_dice = dice(outputs, targets)
            val_loss_total += val_loss.item()
            val_dices.append(val_dice.item())

    avg_val_loss = val_loss_total / len(val_loader)
    avg_val_dice = np.mean(np.array(val_dices))
    std_val_dice = np.std(np.array(val_dices))
```

```
    print(f'Epoch {epoch + 1}/{num_epochs}, Train Loss: {round(sum(train_loss) / len(train_loss), 5)}, '
          f'Val Loss: {round(avg_val_loss, 5)}, Dice on Val: {round(avg_val_dice, 5)}, '
          f'Dice Std: {round(std_val_dice, 5)}')

    train_loss_list.append(sum(train_loss) / len(train_loss))
    val_loss_list.append(val_loss_total)

    if avg_val_dice > 0.8:
        torch.save(model.state_dict(), './model_epoch.pth')

torch.save(model.state_dict(), './model.pth')
```