BRAIN TUMOR MRI IMAGE SEGMENTATION BY 3D U-NET — LOW-RESOLUTION SMALL-SCALE DATASET
2024 Imperial Data Science Winter School Best CV Project

Guan Zhengkang* Xiamen University

Liu Jiayin Tsinghua University

Zhou Yizhi Nanjing University

ABSTRACT

The rapid development of deep learning techniques has highlighted the capabilities of the AI to tackle the classification and segmentation tasks for medical images in an efficient and accurate manner. Using neural network to segment gliomas from MRI brain scan images is a challenging task, especially when dealing with relatively scarce datasets. In our project, we utilized a 3D-Unet network to perform tumor segmentation on a small-sized, low-resolution 3D MRI brain image dataset. Various common methods were experimented with to enhance the model's capabilities, including normalization, different data augmentation schemes, L2 regularization, batch normalization, dropout methods, and the exploration of different loss functions and learning rates. Despite achieving relatively good segmentation accuracy given the small dataset, the experiments revealed that various common techniques aimed at improving the model's generalization did not yield significant improvements to the model, possibly due to the dataset's limited size.

1 Introduction

In recent years, with the rapid development of deep learning and the surge in medical imaging data [1], the segmentation task for medical images, such as CT and MRI, has gradually become a focal point for many researchers. Among numerous methods addressing this issue, U-Net [2] stands out for its considerable accuracy and reasonable training time, thus almost dominating the field in recent years. However, since diagnostic medical imagery is usually volumetric, being able to perform segmentation of 3D volumes [3, 4] is promising by considering the whole image rather than tediously analyzing 2D slides with high relevance.

Our project aims to segment gliomas from MRI brain scan images. We adopted the architectural structure of the traditional 3D U-Net model [3] while simultaneously experimented with and incorporated some practical non-architectural methods suggested by the nnU-Net model [5], particularly in the reprocessing and training section.

Throughout the project, we made several attempts to adapt the 3D U-Net model to a relatively small dataset. We explored z-score normalization, various data augmentation schemes, L2 regularization, batch normalization, dropout methods, and considered different loss functions and learning ratesall of which have been proven effective in large datasets. However, some of these methods were found to be less suitable for small datasets. Nonetheless, the terminal accuracy of our 3D U-Net model continues to demonstrate its robust capability.

*School of Economics, Xiamen University, Xiamen 361000, China, Email: guanzhengkang@stu.xmu.edu.cn

2 Literature Review

2.1 Network Architectures

The fundamental structure of U-Net consists of two symmetric pathways. The first is the en coder(contracting) pathway, similar to a typical convolutional network, employed to provide classification information. The second is the decoder(expanding) pathway, incorporating up-sampling operations and operations for connecting features with the contracting pathway. The entire network exhibits an U-shape. U-Net possesses three main advantages: strong generalization ability, fast training speed, and accurate segmentation results. With a relatively limited dataset of medical images, U-Net can achieve detailed segmentation even under conditions of insufficient samples. This is a crucial factor that makes it highly suitable for medical imaging.

V-Net [3] is one of the earliest structures to apply U-Net to 3D segmentation. By utilizing convolutional kernels for downsampling instead of traditional pooling methods, it aims to retain higher resolution information required for accurate segmentation in 3D medical images, especially in applications such as prostate MRI.

3D U-Net [4] is also an early 3D segmentation U-Net-based model that addresses the issue of sparse annotations in 3D image segmentation. By introducing data augmentation techniques such as rotation, scale transformation, and intensity enhancement, the model enhances its ability to learn from limited annotated data. The use of softmax and weighted cross-entropy losses further balances the importance of different regions, ensuring more accurate and robust segmentation of dense volumes.

H-DenseUNet [6] addresses the limitation of 2D and 3D DenseUNet models in capturing spatial information along the z-axis. By combining the advantages of dense connections and UNet-like connections through a hybrid approach, it provides an effective solution for liver tumor segmentation in CT volumes. The proposed H-DenseUNet shows significant progress in lesion segmentation and performs well in liver segmentation, as demonstrated in the LiTS benchmark. The concept of Cascaded U-Net [7] is explored in the context of glioma segmentation, addressing issues related to model confidence and processing different modalities of information. Through deep cascades with shared topological structures and advanced data augmentation strategies, the proposed method achieves improved segmentation quality and better handling of multi-modal information

The innovative ConvNet [8] architecture combines spatial representation learning from ViT [9] and Swin Trans formers [10] with convolutional inductive biases, enabling the expansion of network width without being constrained by the size of convolutional kernels. In the context of medical image segmentation, this idea is inspiring for learning long-distance semantic correlations through large convolutional kernels and achieving multi-level networks scalabil ity simultaneously. Based on this concept, the MedNeXt [11] network is proposed, consisting of a pure ConvNet architecture that maximizes the advantages of ConvNet design. It maintains rich context through Residual Inverted Bottlenecks and prevents performance saturation from large convolutional kernels using UpKern technology. Addi tionally, through a composite scaling network structure, MedNeXt achieves simultaneous scaling in width, receptive f ield, and depth, providing a powerful and flexible solution for medical image segmentation tasks

2.2 Non-architectural Aspects

nnU-Net [5] is an adaptive U-Net segmentation framework that argues that adjustments to the U-Net network structure do not necessarily improve segmentation performance or advance the state-of-the-art (SOTA). Instead, it emphasizes the importance of non-structural factors in segmentation performance.

nnU-Net simplifies adjustments to the U-Net network structure and focuses more on optimizing critical aspects of the entire segmentation process to improve overall segmentation performance. In the preprocessing stage, it involves cropping non-zero areas for all data and applying cubic spline and nearest neighbor interpolation to learn voxel spacing. For CT images, data truncation and z-score normalization are performed, while MRIimages undergo direct z-score normalization. In terms of data augmentation, techniques such as random rotation, scaling, elastic deformation, gamma correction, and mirroring are employed. The use of image blocks and test-time augmentation (TTA) contributes to improving the models robustness. During training, 5-fold cross-validation is adopted, and Dice and cross-entropy losses are used as training objectives. To dynamically adjust the learning rate, learning rate adjustments are made based on the exponential moving average loss of the training and validation sets.Finally, in the post-processing stage, considering that each class is generally believed to exist within a single connected domain, a method is used to retain the largest connected domain in each category while removing other smaller connected domains. nnU-Net provides an end-to-end solution for medical image (2D or 3D) segmentation, and many subsequent U-Net improvement schemes are based on nnU-Net.

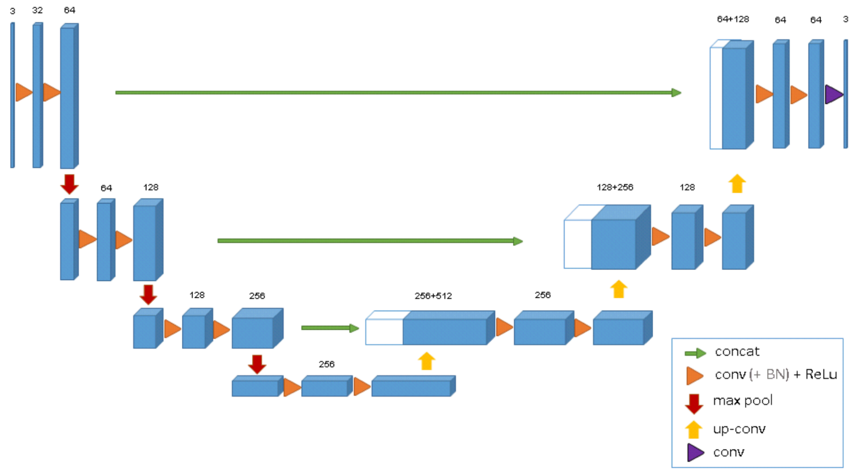

Figure 1: The architecture of 3D U-Net [3].

3 Methodology

3.1 Network Architecture

We have replicated the 3D U-Net [3] neural network architecture(see in Figure 1). It consists of an encoder and a decoder path with four resolution steps. Each layer in the analysis path comprises two $3 \times 3 \times 3$ convolutions and a ReLU activation, followed by a $2 \times 2 \times 2$ max pooling operation. Conversely, in the synthesis path, convolutions are replaced by upconvolutions. A $1 \times 1 \times 1$ convolution is employed in the final layer for segmentation.

Additionally, when addressing the issue of overfitting, we experimented with adding L2 regularization layers. We also explored the effects of adding batch normalization and dropout (50%) layers to each layer separately.

3.2 Training method

We utilize the Adam optimizer in our training process to optimize model parameters. The optimizer computes gradients in each epoch and updates variables accordingly using momentum estimation. To enhance model stability and avoid unsatisfactory convergence to local optima, we adopt the StepLR scheduler to adjust the learning rate on a predefined schedule, particularly effective for smaller datasets.

3.3 Loss function

For the loss function, we combine the binary cross-entropy loss method with the Dice function [5]. The validation loss is calculated as follows:

$$\mathcal{L}_{total} = \mathcal{L}_{BCE} - \mathcal{D}$$

Where $\mathcal{D}$ is the dice value of the segmentation tesults, calculated as follows:

$$\mathcal{D} = \frac{1}{2} \frac{\sum_{i \in I} u_i v_i + \epsilon}{\sum_{i \in I} u_i + \sum_{i \in I} v_i + \epsilon}$$

where u is the softmax output of the network and v is a one hot encoding of the ground truth segmentation map. ϵ is a smoothing factor.

The binary cross-entropy loss $\mathcal{L}_{BCE}$ is calculated as follows:

$$\mathcal{L}_{BCE} = -\frac{1}{N} \sum_{i=1}^{N} (y_i \log(p_i) + (1 - y_i) \log(1 - p_i))$$

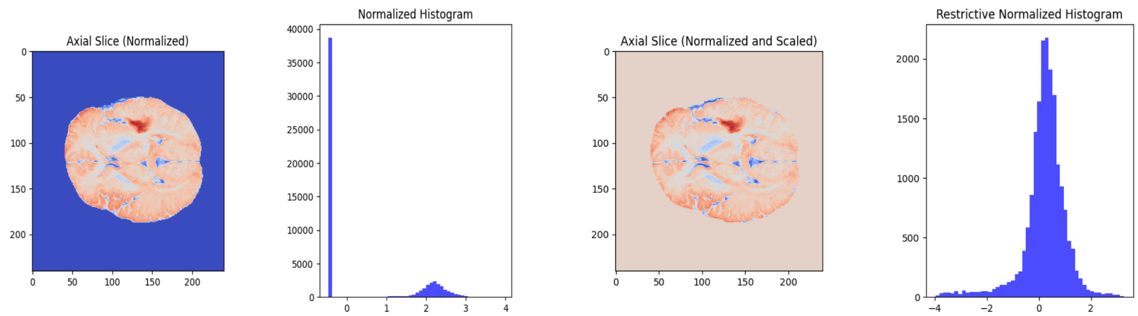

Figure 2: Different way to do standardization: left subgraph is regular standardization, while the right subgraph is standardization only applied to the ROI, ensuring the background of the image is consistent (encoded as 0), and make the distribution of the image closer to a normal distribution with a mean of 0.

Where $\hat{y}_i$ represents the predicted probability of the positive class of sample i, and y_i symbolizes the true label. This combination enables the model to focus on both spatial overlap and probability distribution of classes, thus enabling a more comprehensive study of data features.

4 Experiment and Results

The primary objective of the experimental model is to devise robust models for segmenting gliomas with the small dataset comprising 3D MRI images. We explored z-score normalization, various data augmentation schemes, L2 regularization, batch normalization, dropout methods, and considered different loss functions. The dataset consists of 210 3D brain MRI images with a resolution of 240×240×155, along with their corresponding annotations indicating the location of brain tumors in one-hot encoding. All experiments were conducted on a 40GB A100 GPU rentedå on Google Colaboratory.

Preprocessing We initially attempted standardizing the dataset using the z-score method [5]. The conventional z-score method significantly improved overall accuracy on the validation set by approximately 20%.

However, we found that directly applying z-score normalization to images resulted in varying shades of gray in the background of each target image due to the different overall brightness of each MRI image. This issue could potentially lead to poor model performance. Therefore, we restricted the region of z-score manipulation to the ROI (Regions of Interest), specifically the areas with actual brain images. This approach ensured the standardization was applied only to the brain information, mitigating adverse effects, as shown in Figure 2. Consequently, this additional step resulted in an approximately 5% increase in accuracy on the validation set, building upon the regular standardization.

Data Augmentation We experimented with various data augmentation techniques, including combinations of rotations and mirroring. However, these did not significantly improve the final training outcomes, likely due to the small dataset size and high image resolution. Consequently, we opted for simple augmentation by mirroring images along the Sagittal plane.

Regularization Techniques We experimented with L2 regularization, batch normalization, and dropout methods separately. However, these did not enhance the final training performance, possibly due to the small dataset size and high image resolution. Considering computational burden, we ultimately did not incorporate regularization layers into the model.

Loss Function Compared to using Dice as the sole loss, incorporating binary cross-entropy loss did not improve model performance. However, it did make the training process more efficient, with a more stable decline in loss.

Training Due to limitations in the training equipment, we set the model's batch size to 2 and the learning rate to 3×10^{-4}. We utilized the Adam optimizer for training and decreased the learning rate by 20% every epoch after the first 4 epochs to ensure the model's training depth. The final model achieved an average Dice value of 82.3% on the validation and 85.3% on the test set.

5 Conclusion

On the given small-sized, low-resolution 3D brain MRI images, we employed a 3D-Unet network for the challenging task of tumor segmentation. Various common techniques aimed at improving the model's generalization were experimented with, including z-score normalization, different data augmentation schemes, L2 regularization, batch normalization, dropout methods, and the exploration of different loss functions and learning rates. Ultimately, we adopted preprocessing with image brain region normalization, applied a single mirroring data augmentation, omitted other regularization methods, and trained the 3D-Unet network using a combined loss function of binary cross-entropy and Dice coefficient on the test set, achieving a Dice value of 85.3%. While this value may not be high, it is considered acceptable given the small dataset.

Surprisingly, the experiments revealed that different data augmentation schemes, L2 regularization, batch normalization, and dropout methods did not apparent enhance the model's performance or generalization on the small dataset. This might be attributed to the limited number of images, lower resolution, and challenges in improving the model's fitting ability due to these constraints.

References

[1] Rebecca Smith-Bindman, Marilyn L Kwan, Emily C Marlow, Mary Kay Theis, Wesley Bolch, Stephanie Y Cheng, Erin JA Bowles, James R Duncan, Robert T Greenlee, Lawrence H Kushi, et al. Trends in use of medical imaging in us health care systems and in ontario, canada, 2000-2016. *Jama*, 322(9):843–856, 2019.

[2] 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.

[3] 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.

[4] Fausto Milletari, Nassir Navab, and Seyed-Ahmad Ahmadi. V-net: Fully convolutional neural networks for volumetric medical image segmentation. In *2016 fourth international conference on 3D vision (3DV)*, pages 565–571. Ieee, 2016.

[5] Fabian Isensee, Jens Petersen, Andre Klein, David Zimmerer, Paul F Jaeger, Simon Kohl, Jakob Wasserthal, Gregor Koehler, Tobias Norajitra, Sebastian Wirkert, et al. nnu-net: Self-adapting framework for u-net-based medical image segmentation. *arXiv preprint arXiv:1809.10486*, 2018.

[6] Xiaomeng Li, Hao Chen, Xiaojuan Qi, Qi Dou, Chi-Wing Fu, and Pheng-Ann Heng. H-denseunet: hybrid densely connected unet for liver and tumor segmentation from ct volumes. *IEEE transactions on medical imaging*, 37(12):2663–2674, 2018.

[7] Dmitry Lachinov, Evgeny Vasiliev, and Vadim Turlapov. Glioma segmentation with cascaded unet. In *International MICCAI Brainlesion Workshop*, pages 189–198. Springer, 2018.

[8] Zhuang Liu, Hanzi Mao, Chao-Yuan Wu, Christoph Feichtenhofer, Trevor Darrell, and Saining Xie. A convnet for the 2020s. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 11976–11986, 2022.

[9] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

[10] Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, Zheng Zhang, Stephen Lin, and Baining Guo. Swin transformer: Hierarchical vision transformer using shifted windows. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 10012–10022, 2021.

[11] Saikat Roy, Gregor Koehler, Constantin Ulrich, Michael Baumgartner, Jens Petersen, Fabian Isensee, Paul F Jaeger, and Klaus H Maier-Hein. Mednext: transformer-driven scaling of convnets for medical image segmentation. In *International Conference on Medical Image Computing and Computer-Assisted Intervention*, pages 405–415. Springer, 2023.

Appendix: Python Code

```
import nibabel as nib
from torch.nn import Module, Sequential
from torch.nn import Conv3d, ConvTranspose3d, BatchNorm3d, MaxPool3d, AvgPool1d
from torch.nn import ReLU, Sigmoid
import torch.nn.functional as F
from torch.utils.data import Dataset, DataLoader
from torch.optim.lr_scheduler import StepLR
import torch.nn as nn
import torch
from torch.optim import Adam
from tqdm import tqdm
import os
import numpy as np
from scipy.ndimage import zoom
from scipy.ndimage import rotate
import matplotlib.pyplot as plt

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

        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__()
```

```python

        self.deconv = Sequential(
                            #3D
                            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):
    # __                          __
    #  1|__   ________________   __|1
    #     2|__  ____________  __|2
    #        3|__  ______  __|3
    #           4|__ __ __|4
    # The convolution operations on either side are residual subject to 1*1 Convolution for channel homogeneity
    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])

        # Add Batch Normalization after each Conv3D_Block and Deconv3D_Block
        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)
        #x1 = self.dropout(x1)
        #x1 = self.bn1(x1)  # Batch Normalization
        x_low1 = self.pool1(x1)

        x2 = self.conv_blk2(x_low1)
        #x2 = self.dropout(x2)
        #x2 = self.bn2(x2)  # Batch Normalization
        x_low2 = self.pool2(x2)

        x3 = self.conv_blk3(x_low2)
        #x3 = self.dropout(x3)
        #x3 = self.bn3(x3)  # Batch Normalization
        x_low3 = self.pool3(x3)

        x4 = self.conv_blk4(x_low3)
        #x4 = self.dropout(x4)
        #x4 = self.bn4(x4)  # Batch Normalization
        x_low4 = self.pool4(x4)

        base = self.conv_blk5(x_low4)
        #base = self.dropout(base)
        #base = self.bn5(base)  # Batch Normalization

        d4 = torch.cat([self.deconv_blk4(base), x4], dim=1)
        #d4 = self.bn_d4(d4)  # Batch Normalization
```

```
        d_high4 = self.dec_conv_blk4(d4)

        d3 = torch.cat([self.deconv_blk3(d_high4), x3], dim=1)
        #d3 = self.bn_d3(d3)  # Batch Normalization
        d_high3 = self.dec_conv_blk3(d3)

        d2 = torch.cat([self.deconv_blk2(d_high3), x2], dim=1)
        #d2 = self.bn_d2(d2)  # Batch Normalization
        d_high2 = self.dec_conv_blk2(d2)

        d1 = torch.cat([self.deconv_blk1(d_high2), x1], dim=1)
        #d1 = self.bn_d1(d1)  # Batch Normalization
        d_high1 = self.dec_conv_blk1(d1)

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

        return seg



def dice(inputs, targets):
    smooth = 0.

    #intersection = (inputs * targets).sum()
    intersection = ((inputs>0.5).float() * targets).sum()

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



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

    def forward(self, inputs, targets):
        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)


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):
        if self.mode == "test":
            file_list = []
            for patient_folder in os.listdir(self.root_dir):
                patient_path = os.path.join(self.root_dir, patient_folder)
                if os.path.isdir(patient_path):
                    img_path = os.path.join(patient_path, f"{patient_folder}_fla.nii.gz")
                    seg_path = os.path.join(patient_path, f"{patient_folder}_seg.nii.gz")
```

```
                    if os.path.exists(img_path) and os.path.exists(seg_path):
                        file_list.append((img_path,seg_path))
            return file_list
        else:
            file_list = []
            for i in range(1,400):
                img_path = os.path.join(self.root_dir, f"{i}_fla.nii.gz")
                seg_path = os.path.join(self.root_dir, f"{i}_seg.nii.gz")
                file_list.append((img_path,seg_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_values = img_data[img_data != 0]
        mean = np.mean(non_zero_values)
        std = np.std(non_zero_values)
        img_data[img_data != 0] = (img_data[img_data != 0] - mean) / std

        ###
        if self.transform:
            img_data, seg_data = self.transform(img_data, seg_data)


        img_tensor = torch.from_numpy(img_data).float()
        seg_tensor = torch.from_numpy(seg_data).long()

        img_tensor = img_tensor.unsqueeze(0)
        seg_tensor = seg_tensor.unsqueeze(0)


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

        return img_tensor, seg_tensor


        train_dataset = MRIDataset(root_dir='/content/drive/MyDrive/Colab Notebooks/dataset_segmentation/re', mode="train")
val_dataset = MRIDataset(root_dir='/content/drive/MyDrive/Colab Notebooks/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 = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(device)

model = UNet3D(num_channels=1, feat_channels=[16, 32, 64, 128, 256]).to(device)
```

```
criterion = Loss()
optimizer = Adam(model.parameters(), lr=3e-4)

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=False)
    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)}, Val Loss: {round(avg_val_loss, 5)}, Dice on Val: {round(avg_val_dice, 5)}, 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(), '/content/drive/MyDrive/Colab Notebooks/dataset_segmentation/model_epoch.pth')

torch.save(model.state_dict(), '/content/drive/MyDrive/Colab Notebooks/dataset_segmentation/model.pth')
```