Brain Tumor Segmentation

Bharat Kashyap K

November 27, 2024

1 Background

1.1 Introduction

Tumor segmentation is a complex and time taking task for medical professionals. Automating this task using machine learning models can significantly aid in diagnosis. Many methods have been developed over the years with the UNet architecture achieving notable success and it continues to improve. Brain tumor segmentation is an important step to improve disease diagnosis and it is essential in identifying the tumor location and the severity of these tumors.

The challenge in particular in this segmentation task is that each tumor can vary in size shape and location. Further the contrast material used to differentiate between the tumor and healthy brain tissue may overlap. There are 3 main categories for segmentation tasks, the first is manual where a medical expert classifies each area of the MRI. The second is semi-automated and the third is fully automated. In this project the fully automated task is being tackled.

1.2 Data Description

The dataset of choice is the BraTS2020 Dataset, accessible on kaggle. It is an image dataset containing MRI images of 369 patients in the training set. Earlier versions of this dataset also contained synthesized data. Each MRI is a stack of 155 slices of size 240x240. Further each MRI has 4 modalities to address the issue of overlapping of the contrast material and healthy brain tissue. These modalities are T1-weighted MRI (T1), T1-weighted MRI with contrast (T1c), T2-weighted MRI (T2), and Fluid-Attenuated Inversion Recovery (FLAIR) MRI. They also help differentiate the different progressions within the tumor. The segmentation maps of the MRI datasets have 3 labels: "edema"; "non-enhancing (solid) core" and "necrotic (or fluid-filled) core"; and "enhancing core". These labels are considered as the ground truth for the problem of segmentation. The mean dice score for human annotations compared to the combined consensus of all the raters on the training set was 0.91, 0.86, and 0.85; setting an upper bound for human performance.

$$DSC = \frac{2 \cdot Y_{true} \cdot Y_{pred}}{Y_{true} + Y_{pred} + \epsilon}$$

Hausdorff distance is also used by the BraTS organizers to evaluate the distance between segmentation boundaries. This metric calculates the maximum value among the shortest least square distance d(p, t) of all points on the predicted label surface (p $\in \partial$P) to points on the ground truth surface (t $\in \partial$ T), and vice versa. It is calculated using the formula:

$$Haus(P,T) = \max\left[\sup_{p\in\partial P}\inf_{t\in\partial T} d(p,t), \sup_{t\in\partial T}\inf_{p\in\partial P} d(t,p)\right]$$

The dataset also contains data related to the number of days the patient survived. This data is used to predict the severity of the tumor and to give a timeline for doctors. This problem is not explored in this project.

1.3 Previous Work

This dataset is part of the BraTS challenge starting from 2012. Since then a variety of models have been submitted. The main categorization of the methods used is generative and discriminative. The generative models are largely based on feature selection through domain knowledge. These models did not achieve great success in the task and researchers have been using discriminative methods more. Discriminative methods are machine learning models where the features are learnt by the model itself. This makes the model less interpretable but more accurate. UNet is one such kind of discriminative model.

Initially a majority of the models used random forest classifiers. This method achieved a dice score of 0.6 for the entire tumor and 0.27 for the tumor core. The application of Convolutional networks for images was on the rise from 2014. These networks performed much better than the random forest classifiers. 3D CNNs were also used to obtain Dice scores of 0.87, 0.73 and 0.77 for whole tumor, tumor core, and enhancing tumor respectively.

The DeepMedic Network used 11 layers of 3D CNNs along with 2 parallel pathways with one using the normal MRI data and the other using the same MRI data but at a lower resolution. This method achieved Dice scores of 0.89, 0.76 and 0.72 for whole tumor, tumor core and enhancing tumor respectively. The UNet architecture was first introduced in 2015 and since then it has been modified and used in many image segmentation tasks. Most of this project is based on these 2 base models and some changes made to the architecture.

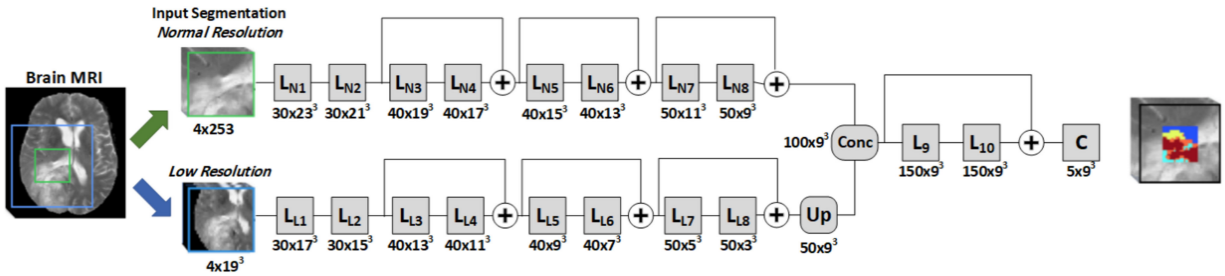


Figure 1: DeepMedic Architecture

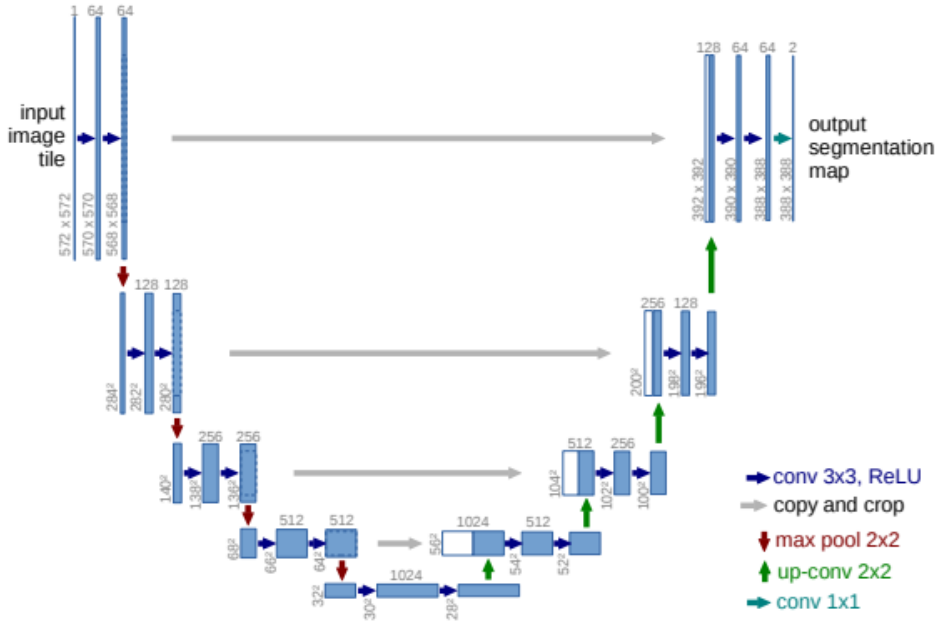


Figure 2: UNet Architecture

In the following years, the choice of loss function was being tweaked to improve the performance of the UNet models. Loss functions that were used are the weighted cross entropy and the dice score and so on. The choice of hyperparameters also plays an important role in the convergence and the optimal performance of such models. However this choice is partly arbitrary and obtained mostly by trial and error. To overcome this, cross validation is one approach but the more effective way was to use ensemble methods to combine many machine learning

algorithms together. The approach that obtained the most success was an ensemble of multiple models and architectures (EMMA). This method had an ensemble of 2 different DeepMedics, 2 fully convolution networks and a UNet. They were trained separately and the final output is the average of the confidence maps. This methodology achieved Dice scores of 0.90, 0.82 and 0.75 for the whole tumor, tumor core and enhancing tumor respectively.

Other works build on the UNet model by adding specific architecture changes. More recent successes have been found by transformer architecture tuned specifically for this dataset. Vision transformer that was effective on the ImageNet dataset was modified for image segmentation such that it is included within the UNet-Architecture.

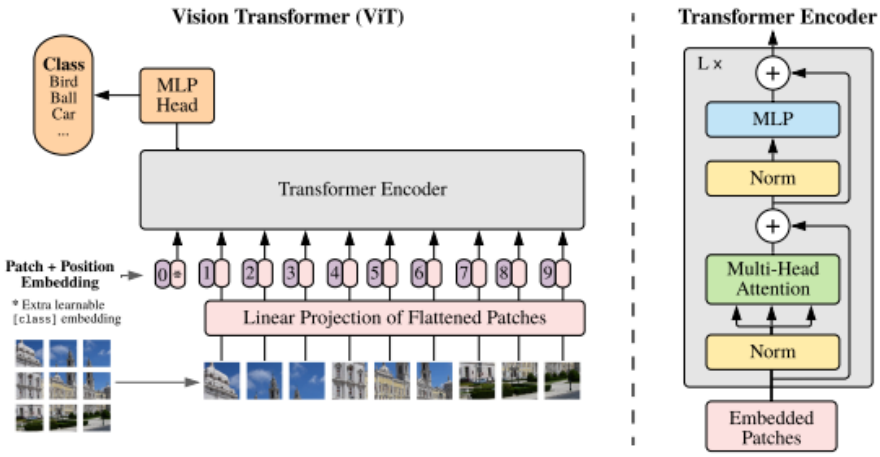


Figure 3: Vision Transformer

Other notable extensions to UNets is the combination of using residual connections as well as attention blocks with the existing architecture. Residual connections allow for the combination of both coarse and fine details learnt by the network. They also let the model be a lot more complex without issues with training such as exploding or vanishing gradient. The choice of activation function to be ReLU or LeakyReLU is another way to avoid that problem. Attention modules were also an addition to the UNet architecture within the bottleneck that achieved significant results. And finally looking at a unique way to combine different modalities, the Dense Multi-path U-Net architecture merges early feature maps and the feature maps of the later CNN outputs are also merged.

2 Methods and Experiments

2.1 Pre-processing

A few different approaches to preprocessing these volumes of MRI were explored amongst which only the last one was used. The computation resources used were the T4 GPUs available on Kaggle.

First the the aim was to preserve the 3D structure of the MRI and build a 3D UNet out of it. Despite using a small 3D convolution network, the training process was significantly time-taking. The initial explanation that was thought of was that the 3D convolution involved too many parameters and therefore the training was taking too long to be considered for using it with the UNet architecture. To counter this, the 3D volume was reshaped to a matrix with all the slices stacked together. This way the kernel that will be used for the 2D convolution will still be able to capture relations between different slices of the MRI. While using a 3D convolution lets the kernel learn patterns in 3 dimensions, the aim was to replicate that using large 2D kernels that would be able to capture different features between MRI scans.

In addition to the time it took, the memory taken for each batch from the training set was too large to be stored on the RAM of the CPU or GPU. Even when the batch size was 4, the computation resources were not enough. One way to counter this is to downsample the volume to a manageable size and build a model based on that. Another method is to use high dimensional statistics such as Tucker Decomposition to compress the volume to

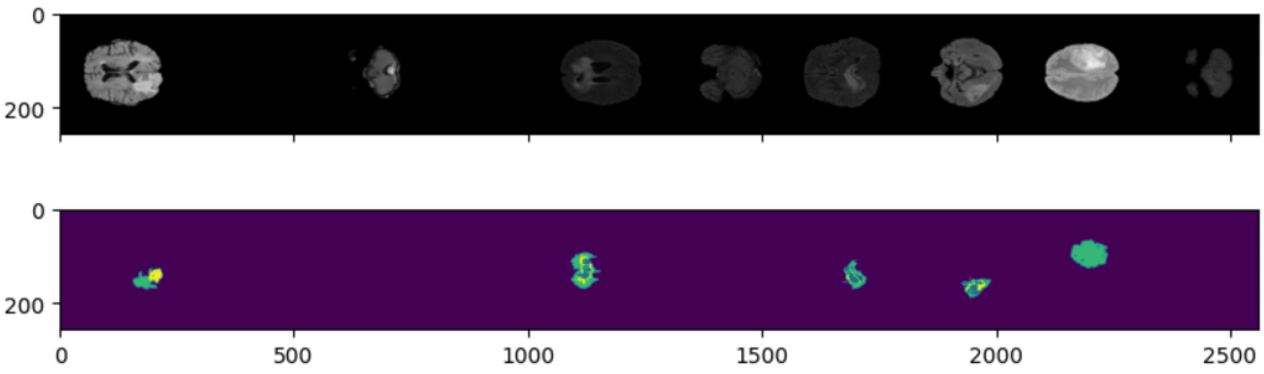


Figure 4: Random FLAIR samples from the training dataset and their corresponding segmentation maps.

sparse matrices and a core tensor. The drawbacks of this method is that it is once again very computationally expensive to calculate and non-uniqueness among others.

Therefore for the rest of the modeling, each slice was segmented individually converting this 3D segmentation problem to a 2D segmentation problem. This is expected to perform worse than 3D UNet implementations but the general trends amongst the models should transfer to 3D. A total of 4 models were implemented with results detailed below. Another constraint to control the computations was to only use the FLAIR channel.

Pre-processing of each of the slices consisted of min-max normalization to limit all the values between 0 and 1 using the maximum value in all the MRI scans. The images were then resized to 256x256 to follow more standard principles. The segmentation labels in the dataset are 0 for background, 1 for non-enhancing (solid) core and necrotic (or fluid-filled) core and 2 for Peritumoral Edema and 4 for Enhancing Tumor. This is relabeled to labels from 0-3.

Short note on hyperparamter choice of batch size - Smaller batch sizes than usual were chosen to make sure the memory required did not exceed the memory of the CPU and GPU.

2.2 Models

Comparison of all the models is given at the end of discussing the models used.

2.2.1 3D CNN

First model used was a shallow 3D CNN. As mentioned earlier the platform did not allow for a larger network to be built. This can also be considered as a baseline model that will be compared to with the rest of the models. The architecture is as follows. The argmax of the output is then taken to find the label for each pixel.

```
class Conv3D(nn.Module):
    def __init__(self):
        super(Conv3D, self).__init__()
        self.conv1 = nn.Conv3d(1, 16, kernel_size=3, padding=1)
        self.conv2 = nn.Conv3d(16, 32, kernel_size=3, padding=1)
        self.conv3 = nn.Conv3d(32, 16, kernel_size=3, padding=1)
        self.conv4 = nn.Conv3d(16, 4, kernel_size=3, padding=1)
        self.relu = nn.ReLU()

    def forward(self,x):
        x = self.relu(self.conv1(x))
        x = self.relu(self.conv2(x))
        x = self.relu(self.conv3(x))
        x = self.conv4(x)
        return x
```

```
Total trainable parameters: 29876
```

2.2.2 UNet

This is the most used algorithm for segmentation upon which changes are made to improve its performance. The following architecture was used.

```
class UNet(nn.Module):
    def __init__(self):
        super(UNet, self).__init__()
        self.conv2_1 = nn.Conv2d(1, 16, kernel_size=3, padding=1)
        self.conv2_2 = nn.Conv2d(16, 16, kernel_size=3, padding=1)
        self.pool2 = nn.MaxPool2d(2)

        self.conv3_1 = nn.Conv2d(16, 32, kernel_size=3, padding=1)
        self.conv3_2 = nn.Conv2d(32, 16, kernel_size=3, padding=1)

        self.up4 = nn.ConvTranspose2d(16, 16, stride=2, kernel_size=2)
        self.conv4_1 = nn.Conv2d(32, 16, kernel_size=3, padding=1)
        self.conv4_2 = nn.Conv2d(16, 4, kernel_size=3, padding=1)

    def forward(self, x):
        encoder2_1 = torch.relu(self.conv2_1(x))
        encoder2_2 = torch.relu(self.conv2_2(encoder2_1))
        encoder2_pool = self.pool2(encoder2_2)

        bottleneck_1 = torch.relu(self.conv3_1(encoder2_pool))
        bottleneck_2 = torch.relu(self.conv3_2(bottleneck_1))

        decode2_up = self.up4(bottleneck_2)
        decode2_concat = torch.cat([decode2_up, encoder2_2], dim=1)
        decode2_1 = torch.relu(self.conv4_1(decode2_concat))
        decode2_2 = torch.relu(self.conv4_2(decode2_1))

        return decode2_2
Total trainable parameters: 17988
```

2.2.3 UNet (Larger Model)

The larger UNet model uses another encoder decoder layer. There are also some differences in terms of the number of channels in each of the convolutions.

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

        self.conv1_1 = nn.Conv2d(1, 16, kernel_size=3, padding=1)
        self.conv1_2 = nn.Conv2d(16, 16, kernel_size=3, padding=1)
        self.pool1 = nn.MaxPool2d(2)

        self.conv2_1 = nn.Conv2d(16, 16, kernel_size=3, padding=1)
        self.conv2_2 = nn.Conv2d(16, 32, kernel_size=3, padding=1)
        self.pool2 = nn.MaxPool2d(2)

        self.conv3_1 = nn.Conv2d(32, 32, kernel_size=3, padding=1)
        self.conv3_2 = nn.Conv2d(32, 16, kernel_size=3, padding=1)
```

```
        self.up4 = nn.ConvTranspose2d(16, 32, kernel_size=2, stride=2)
        self.conv4_1 = nn.Conv2d(64, 32, kernel_size=3, padding=1)
        self.conv4_2 = nn.Conv2d(32, 16, kernel_size=3, padding=1)

        self.up5 = nn.ConvTranspose2d(16, 16, kernel_size=2, stride=2)
        self.conv5_1 = nn.Conv2d(32, 16, kernel_size=3, padding=1)
        self.conv5_2 = nn.Conv2d(16, 4, kernel_size=3, padding=1)

    def forward(self, x):
        encoder1_1 = torch.relu(self.conv1_1(x))
        encoder1_2 = torch.relu(self.conv1_2(encoder1_1))
        encoder1_pool = self.pool1(encoder1_2)

        encoder2_1 = torch.relu(self.conv2_1(encoder1_pool))
        encoder2_2 = torch.relu(self.conv2_2(encoder2_1))
        encoder2_pool = self.pool2(encoder2_2)

        bottleneck_1 = torch.relu(self.conv3_1(encoder2_pool))
        bottleneck_2 = torch.relu(self.conv3_2(bottleneck_1))

        decode2_up = self.up4(bottleneck_2)
        decode2_concat = torch.cat([decode2_up, encoder2_2], dim=1)
        decode2_1 = torch.relu(self.conv4_1(decode2_concat))
        decode2_2 = torch.relu(self.conv4_2(decode2_1))

        decode1_up = self.up5(decode2_2)
        decode1_concat = torch.cat([decode1_up, encoder1_2], dim=1)
        decode1_2 = torch.relu(self.conv5_1(decode1_concat))
        decode1_3 = self.conv5_2(decode1_2)

        return decode1_3

Total trainable parameters: 54724
```

2.2.4 Residual UNet

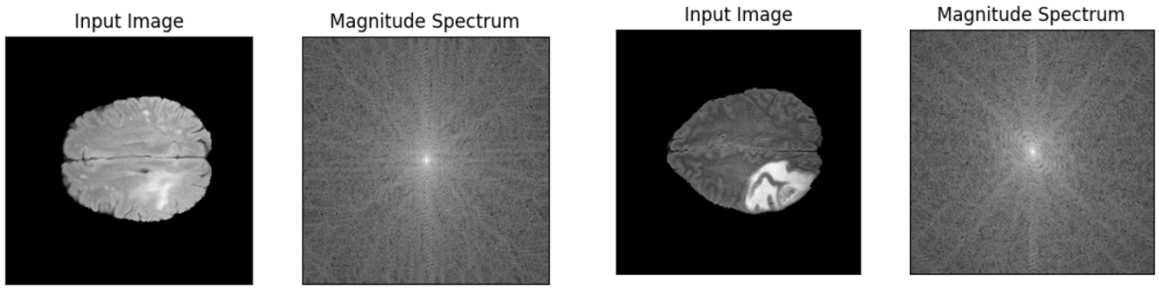
This is an extension to the traditional UNet to include residual connections within each block. This allows for better gradient propagation and thus faster learning.

```
class ResBlock(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(ResBlock, 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.skip = nn.Conv2d(in_channels, out_channels, kernel_size=1, padding=0)
        # Match dimensions

    def forward(self, x):
        identity = self.skip(x)
        out = torch.relu(self.conv1(x))
        out = self.conv2(out)
        out += identity # Reisdual connection
        out = torch.relu(out)
        return out

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

```
        self.enc1 = ResBlock(1, 16)
        self.pool1 = nn.MaxPool2d(2)

        self.enc2 = ResBlock(16, 32)
        self.pool2 = nn.MaxPool2d(2)

        self.bottleneck = ResBlock(32, 32)

        self.up2 = nn.ConvTranspose2d(32, 32, kernel_size=2, stride=2)
        self.dec2 = ResBlock(64, 32)

        self.up1 = nn.ConvTranspose2d(32, 16, kernel_size=2, stride=2)
        self.dec1 = ResBlock(32, 16)

        self.final_conv = nn.Conv2d(16, 4, kernel_size=1)

    def forward(self, x):
        # Encoder
        enc1 = self.enc1(x)
        enc1_pool = self.pool1(enc1)

        enc2 = self.enc2(enc1_pool)
        enc2_pool = self.pool2(enc2)

        bottleneck = self.bottleneck(enc2_pool)

        up2 = self.up2(bottleneck)
        dec2 = self.dec2(torch.cat([up2, enc2], dim=1))

        up1 = self.up1(dec2)
        dec1 = self.dec1(torch.cat([up1, enc1], dim=1))

        out = self.final_conv(dec1)
        return out

Total trainable parameters: 80020
```

2.2.5 PreFiltering then UNet (Original Work)

Preprocessing of the images using fft as part of the Prefiltering network was the initial idea. However, on observation of the frequency transforms, they are not similar even for the same types of brain scans. This is due to the difference in the conditions present during the process of obtaining the MRI image. It leads to different brightness values to the non-tumor regions of the brain. For this reason, a statistical approach was not preferred and instead a learnt feature transform was used. To learn these features, a shallow ConvNet was used on a low resolution version of the images as segmentation. While training the UNet, this convnet output is upscaled and given as an additional channel to the network. The objective of doing this is to bias the UNet towards areas of the tumor that the filtering network believes the tumor is present in.



[h]

FFT of one of the MRI images

FFT of another MRI image at the same slice

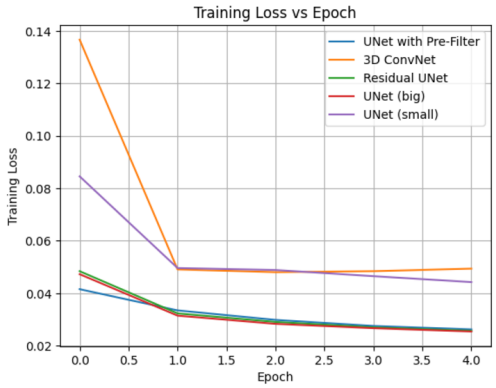
It is concatenated as a channel instead of using as a mask for 2 reasons. The first reason is simply the unreliability of the ConvNet. The ConvNet is trained on low resolution images and is meant to only be a crude filter. Over-reliance of the results from that model will lead to poor performance. The second reason is to allow the network to still use the entire image and come to its own conclusion instead of completely depending on the ConvNet output.

```
class PreFilter(nn.Module):
    def __init__(self):
        super(PreFilter, self).__init__()
        self.conv1 = nn.Conv2d(1, 8, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(8, 16, kernel_size=3, padding=1)
        self.conv3 = nn.Conv2d(16, 16, kernel_size=3, padding=1)
        self.conv4 = nn.Conv2d(16, 4, kernel_size=3, padding=1)

    def forward(self, x):
        x = torch.relu(self.conv1(x))
        x = torch.relu(self.conv2(x))
        x = torch.relu(self.conv3(x))
        out = self.conv4(x)
        return out

class ResBlock(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(ResBlock, 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.skip = nn.Conv2d(in_channels, out_channels, kernel_size=1, padding=0)
        # Match dimensions

    def forward(self, x):
        identity = self.skip(x)
        out = torch.relu(self.conv1(x))
        out = self.conv2(out)
        out += identity # Reisdual connection
        out = torch.relu(out)
        return out


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

        self.enc1 = ResBlock(2, 16)
        self.pool1 = nn.MaxPool2d(2)

        self.enc2 = ResBlock(16, 32)
        self.pool2 = nn.MaxPool2d(2)

        self.bottleneck = ResBlock(32, 32)

        self.up2 = nn.ConvTranspose2d(32, 32, kernel_size=2, stride=2)
        self.dec2 = ResBlock(64, 32)

        self.up1 = nn.ConvTranspose2d(32, 16, kernel_size=2, stride=2)
        self.dec1 = ResBlock(32, 16)

        self.final_conv = nn.Conv2d(16, 4, kernel_size=1)

    def forward(self, x, seg_mask):
        # Encoder
        x = torch.cat([x,seg_mask], dim=1)
```

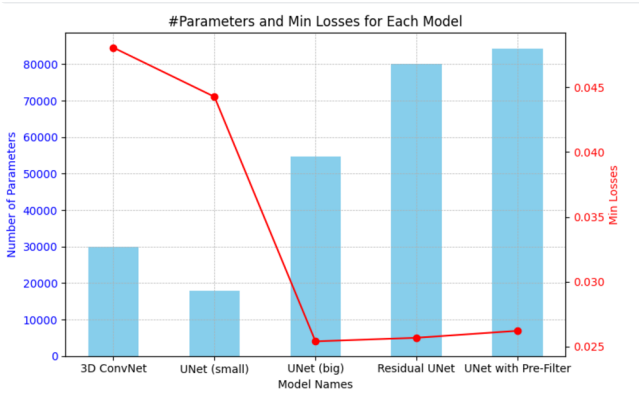
```
        enc1 = self.enc1(x)
        enc1_pool = self.pool1(enc1)

        enc2 = self.enc2(enc1_pool)
        enc2_pool = self.pool2(enc2)

        bottleneck = self.bottleneck(enc2_pool)

        up2 = self.up2(bottleneck)
        dec2 = self.dec2(torch.cat([up2, enc2], dim=1))

        up1 = self.up1(dec2)
        dec1 = self.dec1(torch.cat([up1, enc1], dim=1))

        out = self.final_conv(dec1)
        return out

Total trainable parameters: 4148 + 80180
```

3 Results and Insights

The comparisons are made between Loss and number of parameters.



Loss with increasing epoch for each model



Results summary

The bigger models are expected to work better and the results reflect that. The 3D Conv network is a bigger network than the UNet but performs worse. This shows the importance of choosing the correct architecture for the ML task. The training was faster for the Resnets. The inclusion of a pre filter aimed to decrease the training time and improve the model accuracy and during the experiment it slightly increased computation and slightly improved performance. One advantage of using the prefilter network is the interpretability factor of the inputs to the UNet architecture. The output from the ConvNet returns a heat map of important regions of the MRI the UNet needs to focus on and it also lets a medical expert the area they should be looking at.

A common mistake the UNet makes is to use non-important features such as a label of the MRI source or certain movement artifacts to classify the MRI image. Using the CNN on a low res image and using that output and the high res image together prevents that from happening.

Looking for future improvements, all these architectures can be modified to include 3D convolutions instead of 2D functions. A more sophisticated filter can be designed with interpretability in mind to preprocess the segmentation task before moving it to the main model.

This report compares a few of the segmentation models while accounting for the difference in their complexity. Challenges in this project include managing memory, gpu resources and tensor manipulations. While not included in the code, exploration on variable precision for floating numbers was also done. Scheduling the change in learning rate is another good addition for future projects.

4 Citations

1. D. Zikic et al., "Context-sensitive classification forests for segmentation of brain tumor tissues," in *Proc. MICCAI-BRATS*, 2012, pp. 1–9.

2. G. Urban, M. Bendszus, F. Hamprecht, and J. Kleesiek, "Multi-modal brain tumor segmentation using deep convolutional neural networks," in *Proc. MICCAI-BRATS*, 2014, pp. 31–35.

3. K. Kamnitsas, W. Bai, E. Ferrante, N. Scientific, and M. Sinclair, "Ensembles of multiple models and architectures for robust brain tumour segmentation," in *Proc. 6th MICCAI BraTS Challenge*, 2017, pp. 135–146.

4. O. Ronneberger, P. Fischer, and B. Thomas, "U-Net: Convolutional networks for biomedical image segmentation," in *Proc. Med. Image Comput. Comput.-Assisted Intervention*, 2015, vol. 9351, pp. 234–241.

5. A. Dosovitskiy et al., "An image is worth 16x16 words: Transformers for image recognition at scale," 2020, *arXiv: 2010.11929*.

6. J. Long, E. Shelhamer, and T. Darrell, "Fully convolutional networks for semantic segmentation," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2015, pp. 3431–3440. (First use of residual connections in image segmentation tasks).

7. T. Fan, G. Wang, Y. Li, and H. Wang, "MA-Net: A multi-scale attention network for liver and tumor segmentation," *IEEE Access*, vol. 8, pp. 179 656–179 665, 2020. (Attention in U-Net segmentation).

8. J. Dolz, I. Ben Ayed, and C. Desrosiers, "Dense multi-path U-Net for ischemic stroke lesion segmentation in multiple image modalities," in *Proc. Int. MICCAI Brainlesion Workshop*, Springer, 2018, pp. 271–282.

9. J. Kleesiek et al., "Deep MRI brain extraction: A 3D convolutional neural network for skull stripping," *NeuroImage*, 2016.

10. Ö. Çiçek et al., "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, Springer International Publishing, 2016.

11. L. R. Tucker, "Some mathematical notes on three-mode factor analysis," *Psychometrika*, vol. 31, no. 3, pp. 279–311, 1966. DOI: 10.1007/BF02289464.

12. S. Bakas, H. Akbari, and A. Sotiras, "Advancing The Cancer Genome Atlas glioma MRI collections with expert segmentation labels and radiomic features," *Sci. Data*, vol. 4, no. 1, Sept. 2017, p. 170117.

13. S. Bakas, M. Reyes, and A. Jakab, "Identifying the best machine learning algorithms for brain tumor segmentation, progression assessment, and overall survival prediction in the BRATS Challenge," 2019, *arXiv:1811.02629*.

14. M. Ghaffari, A. Sowmya, and R. Oliver, "Automated brain tumor segmentation using multimodal brain scans: A survey based on models submitted to the BraTS 2012–2018 challenges," *IEEE Reviews in Biomedical Engineering*, vol. 13, pp. 156–168, 2020. DOI: 10.1109/RBME.2019.2946868.

15. T. Henry et al., "Brain tumor segmentation with self-ensembled, deeply-supervised 3D U-Net neural networks: A BraTS 2020 Challenge solution," in *Brainlesion: Glioma, Multiple Sclerosis, Stroke and Traumatic Brain Injuries*, A. Crimi and S. Bakas, Eds., vol. 12658, Lecture Notes in Computer Science, Springer, Cham, 2021. DOI: 10.1007/978-3-030-72084-1_30.

16. B. H. Menze, A. Jakab, and S. Bauer, "The Multimodal Brain Tumor Image Segmentation Benchmark (BRATS)," *IEEE Trans. Med. Imaging*, vol. 34, no. 10, pp. 1993–2024, Oct. 2015.

5 Code

Github Link: https://github.com/Bharat1252003/MachineLearningProject

This concludes my project report. Thank you for your time and for reading this!