OOP Lab Project Report

Brain Tumor Segmentation

Group Members:

AWAIS SADDIQUI (21PWCSE1993)

AIMAL KHAN (21PWCSE1996)

MOEEN KHAN (21PWCSE2069)

Submitted to: Ma’m Sumayyea Salahuddin

Department of Computer Systems Engineering University of Engineering and Technology, Peshawar

Brain Tumor Segmentation

Introduction:

Brain tumor segmentation is the process of identifying and separating the brain tumor tissue from healthy tissue in a medical image. This project uses a lot of OOP concepts. This is important for accurately diagnosing the type and extent of the tumor, and for planning and evaluating the effectiveness of treatment. Brain tumor segmentation can be performed manually by a radiologist, or using automated methods such as deep learning algorithms.

Tools Used:

- OOP Concepts
- Python
- Pytorch (Machine Learning Library)
- UNET Model

Source code: (/Brain-Tumor-Segmentation)

```
                    #Library Imports
import os import warnings import
pickle
warnings.filterwarnings('ignore')
 import torch from torch.utils.data import
SubsetRandomSampler
 import numpy as
np %matplotlib
inline
 import bts.dataset as dataset
import bts.model as model import
bts.classifier as classifier import
bts.plot as plot
 device = torch.device('cuda' if torch.cuda.is_available() else
'cpu')
 print('Computation Details')
print(f'\tDevice Used: ({device})
{torch.cuda.get_device_name(torch.cuda.current_device())}\n')


print('Packages Used Versions:-') print(f'\tPytorch
Version: {torch.__version__}')
                    #Hyperparameters Tuning
# Dataset part used for testing
```

```
TEST_SPLIT = 0.2
# Batch size for training. Limited by GPU memory
BATCH_SIZE = 6
# Dataset folder used
DATASET_USED = 'png_dataset'
# Full Dataset path
DATASET_PATH = os.path.join('dataset',DATASET_USED)
# Training Epochs
EPOCHS = 100
# Filters used in UNet Model
FILTER_LIST = [16,32,64,128,256]
# Flag to train the model
TRAIN = False
# Flag to load saved model
LOAD_MODEL = True
# Flag to save model trained
SAVE_MODEL = False
# Model name to save or load.
MODEL_NAME = f"UNet-{FILTER_LIST}.pt"


print(f"Model Name :   {MODEL_NAME}")
Model Name :   UNet-[16, 32, 64, 128, 256].pt
```

#Dataset Loading

```
def get_indices(length, new=False):     #
Pickle file location of the indices.
    file_path =
os.path.join('dataset',f'split_indices_{DATASET_USED}.p')     data =
dict()     if os.path.isfile(file_path) and not new:
        # File found.         with
open(file_path,'rb') as file :
            data = pickle.load(file)             return
data['train_indices'], data['test_indices']     else:
        # File not found or fresh copy is required.
indices = list(range(length))
np.random.shuffle(indices)
        split = int(np.floor(TEST_SPLIT * len(tumor_dataset)))
train_indices , test_indices = indices[split:], indices[:split]         #
Indices are saved with pickle.
        data['train_indices'] = train_indices
data['test_indices'] = test_indices         with
open(file_path,'wb') as file:
            pickle.dump(data,file)     return
train_indices, test_indices In [4]: tumor_dataset
= dataset.TumorDataset(DATASET_PATH)
 train_indices, test_indices = get_indices(len(tumor_dataset))
train_sampler, test_sampler =
SubsetRandomSampler(train_indices),
SubsetRandomSampler(test_indices)
 trainloader = torch.utils.data.DataLoader(tumor_dataset, BATCH_SIZE,
sampler=train_sampler) testloader = torch.utils.data.DataLoader(tumor_dataset, 1,
sampler=test_sampler)
```

#Model Declaration

```
unet_model = None unet_classifier
= None
```

```
if not LOAD_MODEL:
    # New model is created.     unet_model =
model.DynamicUNet(FILTER_LIST).to(device)     unet_classifier =
classifier.BrainTumorClassifier(unet_model,device) else:
    # Saved model is loaded on memory.
    unet_model = model.DynamicUNet(FILTER_LIST)     unet_classifier =
classifier.BrainTumorClassifier(unet_model,device)
unet_classifier.restore_model(os.path.join('saved_models',MODEL_NAME))
print('Saved model loaded')
                                    #Model Training
if TRAIN:
    unet_model.train()     path = os.path.join('saved_models',MODEL_NAME)
if SAVE_MODEL else None     unet_train_history =
unet_classifier.train(EPOCHS,trainloader,mini_batch=100,save_best=path)
print(f'Training Finished after {EPOCHS} epoches')


# Testing process on test data.
unet_model.eval() unet_score =
unet_classifier.test(testloader)
print(f'\n\nDice Score {unet_score}')
```

Explanation:

The code is a python script that trains and uses a UNet neural network for classifying brain tumors. The code starts by importing required libraries, including torch for deep learning, numpy for numerical computations, and bts, a custom library for loading the dataset, creating the UNet model, and training the classifier.

The device used for computation is either a GPU or a CPU, depending on the availability. The code then sets hyperparameters such as the size of the test split, batch size, the number of epochs, and the filters used in the UNet model.

The next section loads the dataset and splits it into train and test sets. A saved version of the indices used to split the dataset is loaded if it exists, and if it doesn't, a new set of indices is generated and saved.

The UNet model is then created or loaded from a saved state, and if the training flag is set, the model is trained for the specified number of epochs, and if desired, the best model is saved.

Finally, the trained model is used to classify brain tumors in the test set.

Source code: (bts/model.py)

```
import torch import
torch.nn as nn
```

```
import torch.nn.functional as F from
torchsummary import summary   class
DynamicUNet(nn.module):
        Shape Format :  (Channel, Width, Height)     def
__init__(self, filters, input_channels=1, output_channels=1):
        super(DynamicUNet, self).__init__()
if len(filters) != 5:
            raise Exception(f"Filter list size {len(filters)}, expected
5!")         padding = 1         ks = 3
        # Encoding Part of Network.          self.conv1_1 = nn.Conv2d(input_channels,
filters[0], kernel_size=ks, padding=padding)         self.conv1_2 = nn.Conv2d(filters[0],
filters[0], kernel_size=ks, padding=padding)         self.maxpool1 = nn.MaxPool2d(2)
        #   Block 2         self.conv2_1 = nn.Conv2d(filters[0], filters[1],
kernel_size=ks, padding=padding)         self.conv2_2 = nn.Conv2d(filters[1], filters[1],
kernel_size=ks, padding=padding)         self.maxpool2 = nn.MaxPool2d(2)


        #   Block 3         self.conv3_1 = nn.Conv2d(filters[1], filters[2],
kernel_size=ks, padding=padding)         self.conv3_2 = nn.Conv2d(filters[2], filters[2],
kernel_size=ks, padding=padding)         self.maxpool3 = nn.MaxPool2d(2)
        #   Block 4         self.conv4_1 = nn.Conv2d(filters[2], filters[3],
kernel_size=ks, padding=padding)         self.conv4_2 = nn.Conv2d(filters[3],
filters[3], kernel_size=ks, padding=padding)         self.maxpool4 = nn.MaxPool2d(2)


        # Bottleneck Part of Network.
        self.conv5_1 = nn.Conv2d(filters[3], filters[4], kernel_size=ks,
padding=padding)         self.conv5_2 = nn.Conv2d(filters[4], filters[4],
kernel_size=ks, padding=padding)         self.conv5_t = nn.ConvTranspose2d(filters[4],
filters[3], 2, stride=2)
        # Decoding Part of Network.
        #   Block 4         self.conv6_1 = nn.Conv2d(filters[4], filters[3],
kernel_size=ks, padding=padding)         self.conv6_2 = nn.Conv2d(filters[3],
filters[3], kernel_size=ks, padding=padding)         self.conv6_t =
nn.ConvTranspose2d(filters[3], filters[2], 2, stride=2)
        #   Block 3         self.conv7_1 = nn.Conv2d(filters[3], filters[2],
kernel_size=ks, padding=padding)
```

```
        self.conv7_2  =  nn.Conv2d(filters[2],  filters[2],  kernel_size=ks,  padding=padding)
self.conv7_t = nn.ConvTranspose2d(filters[2], filters[1], 2, stride=2)
        #   Block 2       self.conv8_1 = nn.Conv2d(filters[2], filters[1], kernel_size=ks,
padding=padding)         self.conv8_2 = nn.Conv2d(filters[1], filters[1], kernel_size=ks,
padding=padding)            self.conv8_t  =  nn.ConvTranspose2d(filters[1],  filters[0],  2,
stride=2)
        #   Block 1       self.conv9_1 = nn.Conv2d(filters[1], filters[0], kernel_size=ks,
padding=padding)         self.conv9_2 = nn.Conv2d(filters[0], filters[0], kernel_size=ks,
padding=padding)
        # Output Part of Network.
        self.conv10 = nn.Conv2d(filters[0], output_channels, kernel_size=ks, padding=padding)
```

Explanation:

The UNet is a type of convolutional neural network that is commonly used for image segmentation problems.The code creates a class called DynamicUNet that is a inherited from the nn.Module class in the PyTorch library. The DynamicUNet class takes three parameters: filters, input_channels and output_channels.

The filters parameter is a list of 5 integers that represent the number of filters for each of the 5 encoding/decoding blocks in the UNet. The input_channels parameter is an integer that specifies the number of channels in the input image, and the output_channels parameter is an integer that specifies the number of channels in the output image.

The DynamicUNet class implements the architecture of the UNet network using the nn module of the PyTorch library. The network consists of a series of encoding and decoding blocks, each of which includes two nn.Conv2d layers and one nn.MaxPool2d layer in the encoding part, and two nn.Conv2d layers and one nn.ConvTranspose2d layer in the decoding part. The final block of the network includes a single nn.Conv2d layer that produces the output image.

The nn.Conv2d layers are convolutional layers that apply filters to the input image to extract features. The nn.MaxPool2d layers perform max-pooling operations to reduce the size of the input image. The nn.ConvTranspose2d layers perform transposed convolutions to increase the size of the feature maps and produce the output image.

Source code: (bts/classifier.py)

```
import torch import bts.loss as
loss import torch.optim as optim
from torch.autograd import
Variable
```

```
 from tensorboardX import
SummaryWriter import numpy as np from
datetime import datetime from time
import time  class
BrainTumorClassifier():     def
__init__(self, model, device):
        # Tensorboard Writter         self.tb_writer =
SummaryWriter(log_dir=f'logs/{self.log_path}')
        # Training session history data.
history = {'train_loss': list()}
        # For save best feature. Initial loss taken a very high value.
last_loss = 1000
        # Optimizer used for training process. Adam Optimizer.
        self.optimizer = optim.Adam(self.model.parameters(), lr=learning_rate)
        # Reducing LR on plateau feature to improve training.
self.scheduler = optim.lr_scheduler.ReduceLROnPlateau(
self.optimizer, factor=0.85, patience=2, verbose=True)
print('Starting Training Process')
        # Epoch Loop         for epoch in range(epochs):
start_time = time()             # Training a single epoch
epoch_loss = self._train_epoch(trainloader, mini_batch)             #
Collecting all epoch loss values for future visualization.
history['train_loss'].append(epoch_loss)
            # Logging to Tensorboard
self.tb_writer.add_scalar('Train Loss', epoch_loss, epoch)
self.tb_writer.add_scalar(
                'Learning Rate', self.optimizer.param_groups[0]['lr'], epoch)
            # Reduce LR On Plateau
self.scheduler.step(epoch_loss)


            # Plotting some sample output on TensorBoard for visualization purpose.
if plot_image:
                self.model.eval()
self._plot_image(epoch, plot_image)
```

```
                self.model.train()
             time_taken = time()-
start_time
            # Training Logs printed.
            print(f'Epoch: {epoch+1:03d},   ', end='')
print(f'Loss:{epoch_loss:.7f},           ',     end='')
print(f'Time:{time_taken:.2f}secs', end='')
            # Save the best model with lowest epoch loss feature.
if save_best != None and last_loss > epoch_loss:
                self.save_model(save_best)
last_loss = epoch_loss                 print(f'\tSaved at
loss: {epoch_loss:.10f}')             else:
                print()
return history
     def save_model(self,
path):
        torch.save(self.model.state_dict(), path)
     def restore_model(self,
path):         if self.device ==
'cpu':
            self.model.load_state_dict(torch.load(path, map_location=device))
else:
            self.model.load_state_dict(torch.load(path))
self.model.to(self.device)
```

Explanation:

This code defines a class named "BrainTumorClassifier" for training a neural network model. The class uses the PyTorch library for training the model and TensorboardX for logging and visualizing the training process. The class contains several methods:

- __init__: A constructor that initializes the training process and performs a loop over a set number of epochs. In each epoch, it trains the model using the _train_epoch method, records the training loss and learning rate in Tensorboard, reduces the learning rate if the loss plateaus, plots some sample outputs, and saves the model if the current epoch loss is lower than the previous best loss.
- save_model: Saves the model's state dictionary to a specified file path.

- restore_model: Restores the model from a saved state dictionary.

Explanation: (bts/plot.py)

The code defines two functions result and loss_graph using the Matplotlib library.

The result function creates a 2x3 plot to visualize the comparison of original mask and constructed mask of a MRI scan and their differences, original and constructed segments, and the original image. The function accepts 5 parameters:

- image: a numpy array containing the original image of MRI scan.
- mask: a numpy array containing the original mask of tumor.
- output: a numpy array containing the model-constructed mask from the input image.
- title: a string representing the title of the plot.
- transparency: a float representing the transparency level of the mask on the images (default value is 0.38).
- save_path: a string representing the location where the plot should be saved, if provided.

The loss_graph function creates a plot to visualize the loss function over epoch during training of a model. The function accepts 2 parameters:

- loss_list: a list of loss values at each epoch of training time.
- save_plot: a string representing the location where the plot should be saved, if provided.

Both functions display the plots using the plt.show() method and the result function can save the plot to the specified location using plt.savefig().

OOP Concepts Used

- **Classes and Constructors**: Models and some of other codes used classes and constructors for their smooth workflow and ease provided by these concepts. For example:

```
class BrainTumorClassifier():
    def __init__(self, model, device):
    #Some Code
```

- **Inheritance**: Models are inherited from existing models to create new models with additional functionality.

```
class DynamicUNet(nn.module):
        #Some Code
```

- **Encapsulation**: This model is self-contained and hide their internal workings from other parts of the code. This makes it easier to maintain and modify the models. For example:

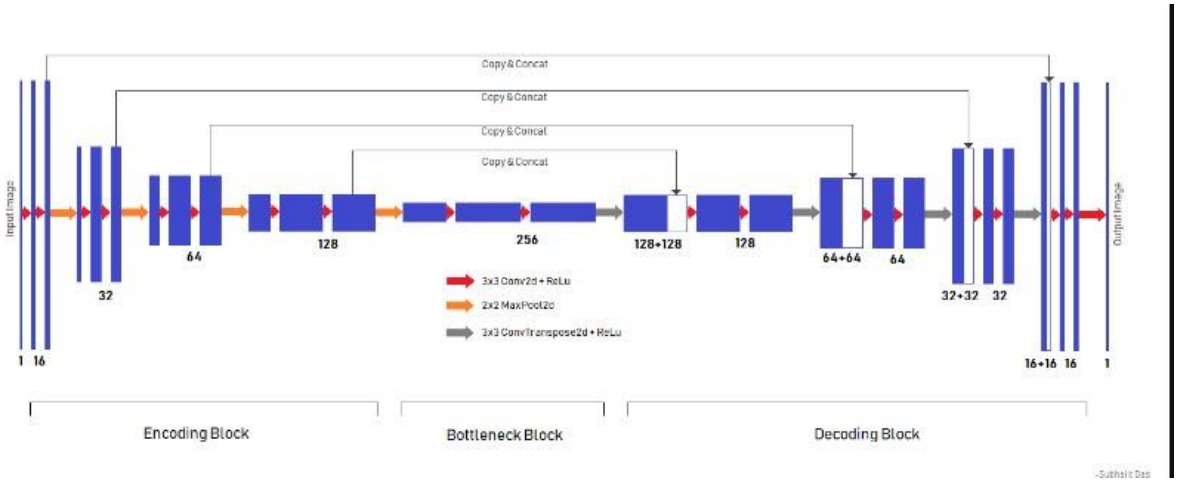
```
self.tb_writer          #Data member of BrainTumorClassifier.
self.conv1_1            #Data member of class DynamicUNet(nn.module)
```

.

- **Abstraction**: This models provide a high-level interface to the underlying computations, making it easier for developers to use and extend the models without having to understand the underlying implementation details.
- **Objects**: This code uses different objects of defined classes. For example;

```
tumor_dataset = dataset.TumorDataset(DATASET_PATH)
unet_model = model.DynamicUNet(FILTER_LIST).to(device)
unet_classifier = classifier.BrainTumorClassifier(unet_model,device)
```

UNET Model Architecture



Dataset Used

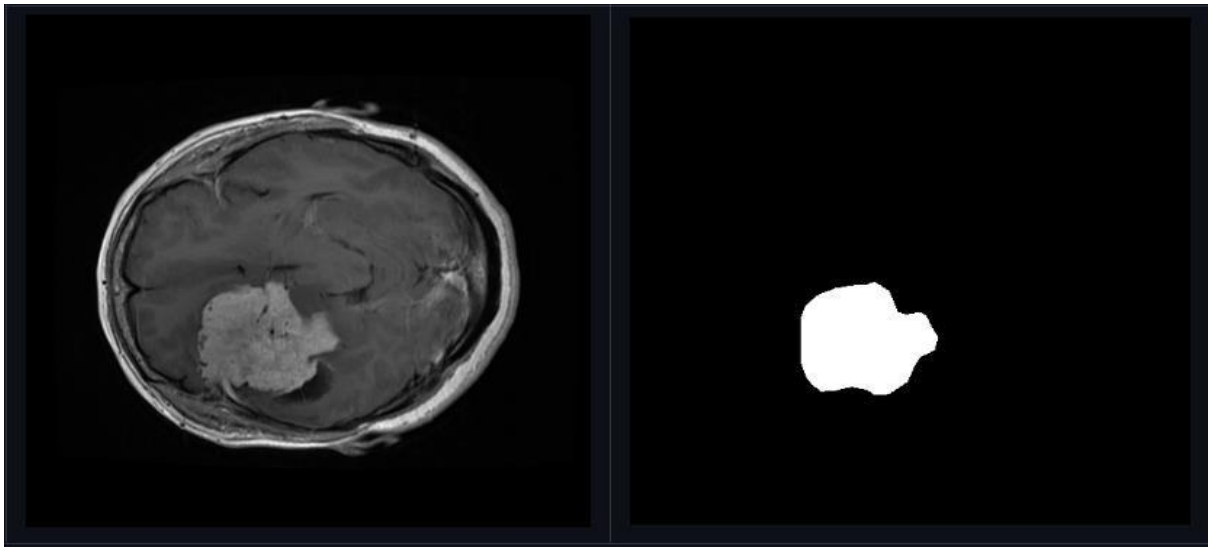
Dataset used in this project was provided by Jun Cheng.
This dataset contains 3064 T1-weighted contrast-enhanced images with three kinds of brain tumor. For a detailed information about the dataset please refer to this site. Version 5 of this dataset is used in this project. Each image is of dimension 512 x 512 x 1 , these are black and white images thus having a single channel.

Data Augmentation

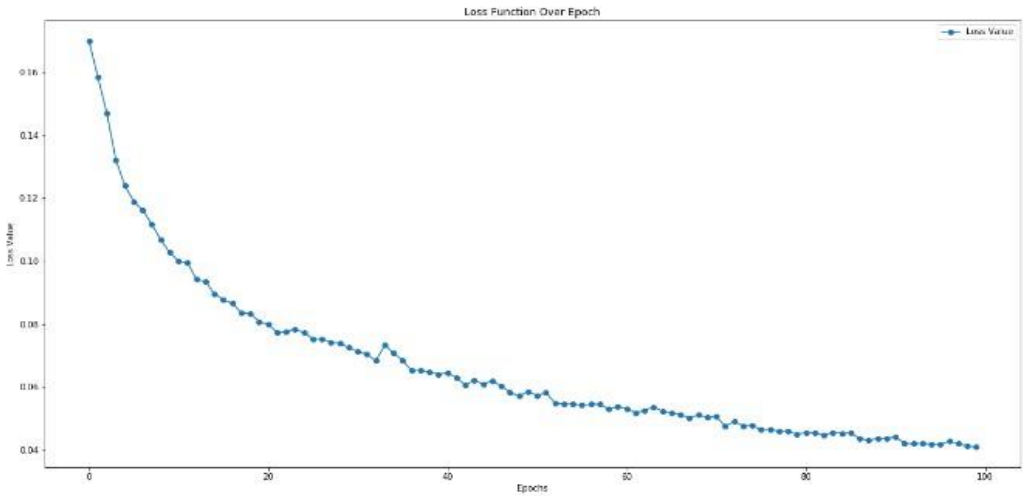
Data augmentation is a technique used in deep learning to artificially increase the size of a training dataset. This is done by applying various transformations to the existing data to generate new, similar data samples. Common transformations include rotation, flipping, scaling, and shifting of images. This can help reduce overfitting and increase the robustness of the model by training it on a wider range of data. The goal of data augmentation is to increase the diversity of the training data and make the model more robust to variations in the input data.

Training Results

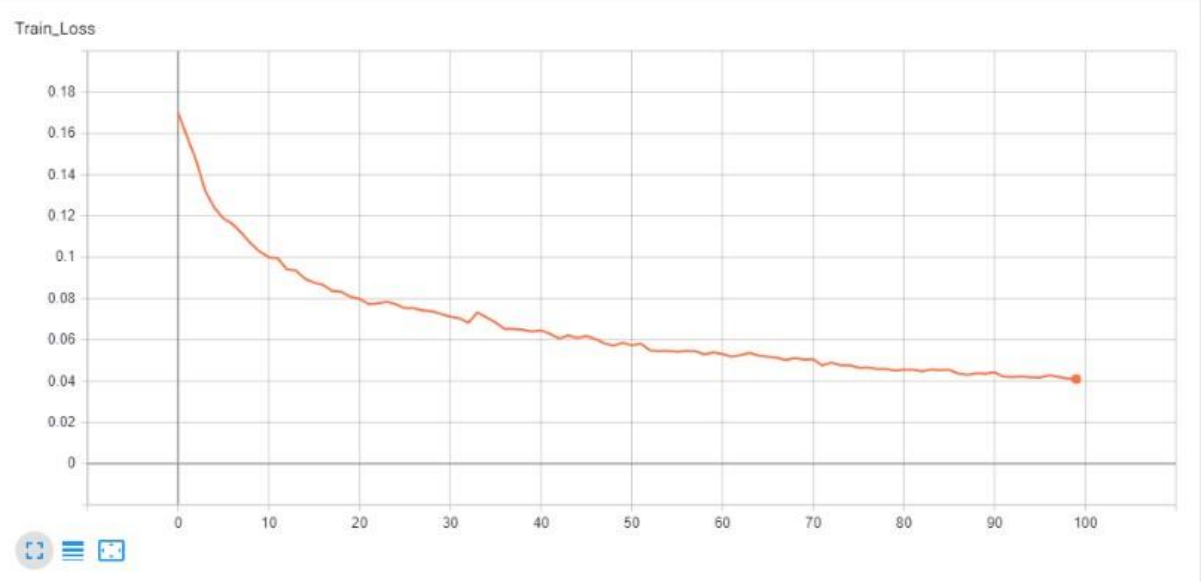
```
# Run this cell repeatedly to see some results. image_index =
test_indices[i] sample = tumor_dataset[image_index] image, mask,
output, d_score = unet_classifier.predict(sample,0.65) title =
f'Name: {image_index}.png   Dice Score: {d_score:.5f}'
# save_path =
os.path.join('images',f'{d_score:.5f}_{image_index}.png')
plot.result(image,mask,output,title,save_path=None) i += 1 if i >=
len(test_indices):
    i = 0
```



Rate Graph in Tensorboard



Loss Graph in Matplotlib:



Application

- Diagnosis and treatment planning: Segmentation of brain tumors can help medical practitioners accurately diagnose the type and stage of the tumor, as well as plan effective treatments.
- Monitoring disease progression: By regularly segmenting brain tumors, medical practitioners can monitor the progression of the disease and adjust treatment accordingly.
- Research and drug development: Accurate segmentation of brain tumors can aid in the development of new treatments and drugs by providing a more precise understanding of the location and size of tumors.
- Improved patient outcomes: By accurately identifying and segmenting brain tumors, medical practitioners can provide more effective treatments and improve patient outcomes.
- Computer-aided diagnosis: Deep learning algorithms can assist medical practitioners in the diagnosis of brain tumors by providing accurate and consistent segmentation results.

Conclusion:

This code is totally based on Object Oriented Programming. Convolutional neural networks (CNNs) have proven to be effective in the task of brain tumor segmentation. These techniques have the advantage of being able to automatically learn complex and high-level features from large amounts of medical imaging data, leading to high accuracy and reliability in tumor segmentation. Additionally, with advances in hardware and computational resources, deep learning algorithms can be efficiently trained and deployed in clinical settings for assisting in diagnosis and treatment planning for brain tumor patients. However, it is important to note that deep learning algorithms still require high-quality annotated medical imaging data for training and validation, and further research is necessary to

address remaining challenges such as generalization to unseen imaging modalities and improving robustness to variability in medical images.

References:

- ***https://github.com/sdsubhajitdas/Brain-Tumor-Segmentation***
- https://arxiv.org/abs/1505.04597
- https://chat.openai.com/
- https://youtu.be/v5cngxo4mIg
- https://medium.com/mlearning-ai/brain-tumor-segmentation-using-deep-learning-models
- https://developer.nvidia.com/blog/automatically-segmenting-brain-tumors-with-ai/
- https://www.youtube.com/watch?v=CVYCmB1e8J8&list=PLZHnYvH1qtOaI5QZVi5OwPAqq15AMEoMl&t=3s

---