Automatic Glioma Segmentation Pipeline with U-Net Model

Zaitsev Anton
anton.zaitsev.001@student.uni.lu

Papadimitriou Emmanouil Filippos
emmanouil.papadimitriou.001@student.uni.lu

Lonsie Zanmene Marie
marie.lontsie.001@student.uni.lu

Abstract

This report provides a high-level overview of the code implemented for the glioma segmentation task on the BraTS dataset using a U-Net model and model evaluation results on the testing set. The work was carried out as part of the final project for the "Introduction to Imaging AI with Applications in Medical Imaging" course at the University of Luxembourg. Although we describe the key functions and steps involved for each task, not all functions or methods are fully described here. For more detailed information, including function descriptions, specific function parameters, classes, and utilities, refer directly to the source code. The code contains in-line comments, doc strings, and references that offer clarification and details.

1 Data Handling and Preprocessing

1.a Loading T1-Weighted Images and Labels

```python
from scripts.model.train import trainingPipeline, trainingPipelineKFold
from scripts.model.test import testingPipeline
from scripts.utils.helpers import setGlobalSeed

def run():
    setGlobalSeed() # set python, random and pytorch seeds for reproducibility
    modalities = ["t1"] # loads "t1" modality and segmentation mask
    available_models = ["UNet", "AttentionUNet", "UNETR", "SwinUNETR", "
        HighResNet"] # available models to train/test
    chosen_model = "UNet"
    kfold = False # not kfold training
    test = False # if to test or not
    if test: # start testing pipeline
        testingPipeline(modalities=modalities, model_name=chosen_model,
            ensemble=kfold)
    else: # start training pipeline
        if not kfold:
            trainingPipeline(modalities=modalities, model_name=chosen_model)
        else:
            trainingPipelineKFold(modalities=modalities, model_name=
                chosen_model)
if __name__ == "__main__":
    run()
```

Figure 1: main.py

1.b DataLoader

```python
def createDataloader(patient_ids: list, modalities: list, batch_size: int =
    BATCH_SIZE, train: bool = True, model_name: str = "UNet"):
    # prepare data in MONAI format
    data_dicts = [
        {
            **{
                f"modality_{i}": f"{DATA_PATH}/{PATIENT_FOLDER_NAME}{pid}/{
                    PATIENT_FOLDER_NAME}{pid}_{modality}.nii.gz"
                for i, modality in enumerate(modalities)
            },
            "mask": f"{DATA_PATH}/{PATIENT_FOLDER_NAME}{pid}/{
                PATIENT_FOLDER_NAME}{pid}_seg.nii.gz"
        }
        for pid in patient_ids
    ]
    # define data augmentation and transforms
    swin = True if model_name == "SwinUNETR" else False
    transforms = defineTransforms(modalities=modalities, train=train, swin=
        swin)
    # create MONAI dataset and DataLoader, return DataLoader
    monai_dataset = MonaiDataset(data=data_dicts, transform=transforms)
    dataloader = DataLoader(monai_dataset, batch_size=batch_size, shuffle=
        train)
    return dataloader
```

Figure 2: `scripts/data/dataloader.py`

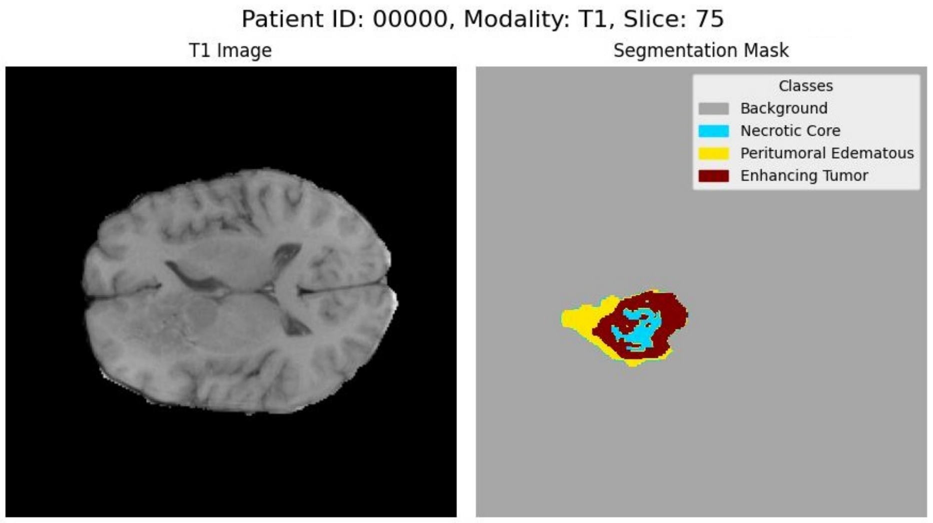
1.c Data Augmentations

For data transformation, we start by cropping the dataset to remove empty regions in both modality images and the segmentation masks, thus reducing the space usage and improving computational efficiency (see Figure 3b).

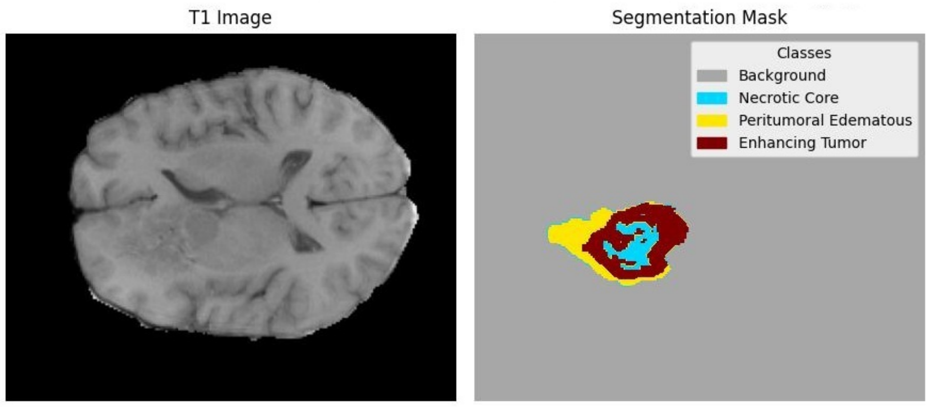
Next, we define a `transformations` object, which is then used by the `MonaiDataset` object. The transformations we apply depend on the current partition, that is, training, validation, or test set. Specifically, augmentations such as random flipping along spatial axes, random bias field addition, and random Gaussian noise are only applied during the training phase (see Figure 3c).

Transformations that are consistently applied across all partitions aim to preprocess the data in a model-compatible format. These transformations include ensuring the data has channels in the first dimension, normalizing the intensity values of the data to a $[0, 1]$ range, and standardizing the orientation of the data to a common format. Additional preprocessing steps involve resampling the data to 1 mm isotropic spacing and concatenating the input modalities into a single tensor for further processing.

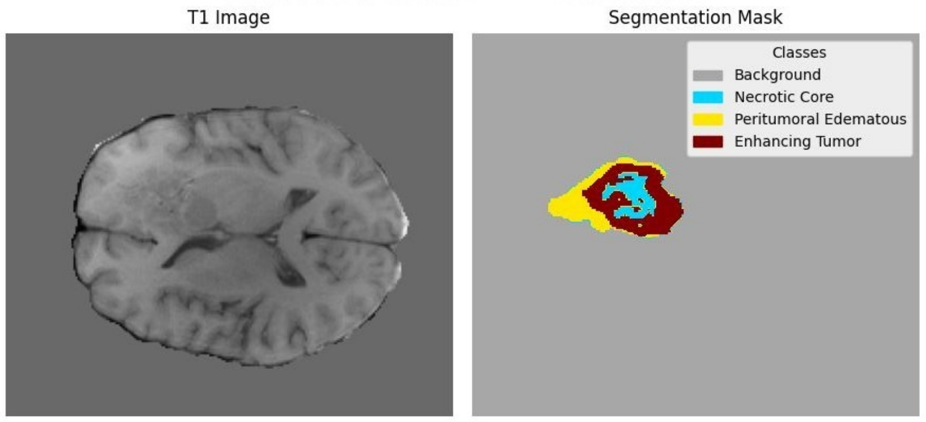
To ensure compatibility with specific models, the final transformations include padding the spatial dimensions of the data to sizes divisible by 2^5 for the `UNet` model. Similarly, for the `SwinUNet` model, additional spatial padding is applied to match its required input dimensions.



(a) Original T1 modality and segmentation mask.



(b) Cropped T1 modality and segmentation mask.



(c) Augmented T1 modality and segmentation mask.

Figure 3: Visualization for T1 modality and segmentation mask (top: original, middle: cropped, bottom: augmented) for patient with ID 00000 and depth dimension 75.

2 Data Splitting

2.a Train/Validation/Test Splits

Having separate sets ensures the avoidance of data leakage and an unbiased evaluation. Data leakage leads to overly optimistic performance metrics and invalid evaluation.

The training set is what the model is going to be trained on. Ideally, it should cover the whole space for the task at hand. Otherwise, the model will not be able to generalize well for the data instances not covered by the training data.

The validation set is used to evaluate the model training. It does not make sense to evaluate the model in the training set because that is what the model is trained on. Naturally, it will perform the best on the training set, so we use another set, validation, to evaluate the training process and guide the model towards the most optimal solution using training callbacks. For example, we use early stopping, which tracks the validation metric and stops the training when the metric does not improve for some epochs. Using training metrics for early stopping does not work, because training metrics constantly improve as the training continues.

The test set is usually used to evaluate model performance outside of the training loop. It provides an unbiased estimate of how well the model generalizes to unseen data. Ideally, this set should also represent the entire data space so that our evaluation is valid.

We use split ratios of 0.75 for training, 0.15 for validation, and 0.1 for testing.

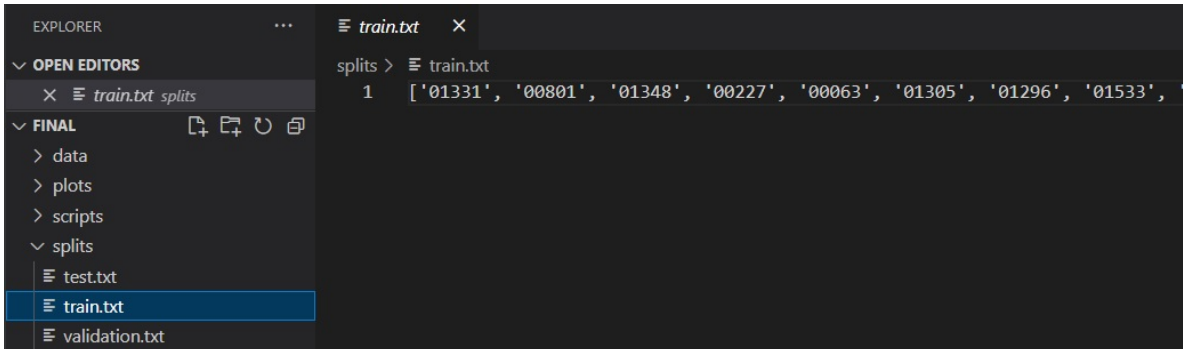
```python
def createSplits():
    # retrieve patient IDs
    patient_ids = getPatientIDs()
    # shuffle IDs
    random.shuffle(patient_ids)
    # get length for train, validation and test sets
    len_train = int(len(patient_ids)*TRAIN_PERCENTAGE)
    len_val = int(len(patient_ids)*VAL_PERCENTAGE)
    # retrieve patient IDs for train, validation and test sets
    train_ids = patient_ids[:len_train]
    val_ids = patient_ids[len_train:len_train + len_val]
    test_ids = patient_ids[len_train + len_val:]
    # store IDs to txt files and return them
    writeTXT(train_ids, TRAIN_IDS_PATH)
    writeTXT(val_ids, VAL_IDS_PATH)
    writeTXT(test_ids, TEST_IDS_PATH)
    return train_ids, val_ids, test_ids
```

Figure 4: `scripts/data/split.py`

2.b Defining Patient IDs for Each Split

```python
def getSplits():
    try:
        with open(TRAIN_IDS_PATH, "r") as train_file:
            # convert string to list
            train_ids = literal_eval(train_file.read())
        with open(VAL_IDS_PATH, "r") as val_file:
            # convert string to list
            val_ids = literal_eval(val_file.read())
        with open(TEST_IDS_PATH, "r") as test_file:
            # convert string to list
            test_ids = literal_eval(test_file.read())
    except FileNotFoundError:
        train_ids, val_ids, test_ids = createSplits()
    return train_ids, val_ids, test_ids
```

Figure 5: `scripts/data/split.py`



```
['01331', '00801', '01348', '00227', '00063', '01305', '01296', '01533',
```

Figure 6: Content of `train.txt` file holding training patient IDs. The result of running `createSplits()` function.

3 Model Selection and Training

3.a Model Selection

We attempted to run the entire pipeline on four models: `UNet`, `UNETR`, `SwinUNet`, and `HighResNet`. However, due to memory limitations, we were only able to train the `UNet` model. The other models required more memory and we could not define a deep enough network that trains efficiently on the `BRaTS` dataset.

3.b Training Pipeline

The training pipeline involves defining training and validation datasets, creating corresponding dataloaders, and initializing the neural network (see `trainingPipeline()` function in `scripts/model/train.py` file). We use Dice loss function and Dice metric for evaluating the model performance, and `AdamW` optimizer with a cosine annealing warm restart scheduler for learning rate adjustments.

Training iteratively computes losses, updates model weights, and logs metrics via `TensorBoard`. Validation evaluates model performance and updates the best model if improvements are observed. Early stopping is triggered after consecutive epochs without improvement.

When the script is interrupted during training and later resumed, it automatically loads the latest checkpoint to continue the pipeline from the most recent saved state, ensuring that model weights, optimizer configurations, and training progress are preserved.

```python
def train(model, dataloader, loss_fn, optimizer, epoch, writer):
    # set model to train state (for proper gradients calculation)
    model.train()
    epoch_loss = 0
    step = 0
    # train
    for batch_data in dataloader:
        step += 1
        # retrieve data from dataloader for current batch
        inputs = batch_data["modalities"].to(DEVICE)
        labels = batch_data["mask"].to(DEVICE).long()
        # forward pass
        optimizer.zero_grad()
        outputs = model(inputs)
        # backward pass
        loss = loss_fn(outputs, labels)
        loss.backward()
        optimizer.step()
        # accumulate loss
        epoch_loss += loss.item()
        # write results
        if step % 5 == 0:
            print(f"          Epoch {epoch + 1}, Step {step}, Loss: {loss.item
                ():.4f}")
            writer.add_scalar("Train/BatchLoss", loss.item(), epoch * len(
                dataloader) + step)
    # compute average loss for the epoch
    epoch_loss /= step
    # write the results
    writer.add_scalar("Train/EpochLoss", epoch_loss, epoch)
    return epoch_loss
```

Figure 7: `scripts/model/train.py`

3.c Validation

```python
def validate(model, dataloader, post_pred, loss_fn, metric_fn, metric_fn_batch
    , epoch, writer):
    model.eval() # set model to evaluation mode
    val_loss = 0
    step = 0
    # reset metrics at the beginning of validation
    metric_fn.reset()
    metric_fn_batch.reset()
    # validate
    with torch.no_grad():
        for batch_data in dataloader:
            step += 1
            # retrieve data from the dataloader
            inputs = batch_data["modalities"].to(DEVICE)
            labels = batch_data["mask"].to(DEVICE).long()
            # forward pass
            outputs = model(inputs)
            loss = loss_fn(outputs, labels)
            val_loss += loss.item()
            # apply post-processing to outputs (sigmoid, discrete with
                threshold)
            outputs_post_pred = post_pred(outputs)
            # metric score
            metric_fn(y_pred=outputs_post_pred, y=labels)
            metric_fn_batch(y_pred=outputs_post_pred, y=labels)
    # total loss after processing all batches
    val_loss /= step
    # aggregate the metrics over all batches for the entire epoch
    epoch_metric = metric_fn.aggregate().item() # single scalar metric (mean
        over batch and classes)
    metric_batch = metric_fn_batch.aggregate() # vector of metrics per class
    metric_fn.reset()
    metric_fn_batch.reset()
    # extract per-class metrics from metric_batch
    metric_nc = metric_batch[0].item()
    metric_edema = metric_batch[1].item()
    metric_et = metric_batch[2].item()
    # log results
    writer.add_scalar("Validation/Loss/TotalLoss", val_loss, epoch)
    writer.add_scalar("Validation/Metric/TotalMetric", epoch_metric, epoch)
    writer.add_scalar("Validation/Metric/MetricNC", metric_nc, epoch)
    writer.add_scalar("Validation/Metric/MetricEDEMA", metric_edema, epoch)
    writer.add_scalar("Validation/Metric/MetricET", metric_et, epoch)
    return val_loss, epoch_metric
```

Figure 8: scripts/model/train.py

3.d Early Stopping

```
...
        if epochs_no_improve >= PATIENCE:
            print("  Early stopping triggered")
            break
...
```

Figure 9: `scripts/model/train.py`, in `trainingPipeline()` function.

3.e Loss Function

We use MONAI's Dice loss. It is well suited for segmentation tasks due to its focus on overlap between predicted and ground-truth regions.

$$\text{DiceLoss}(P, G) = 1 - \frac{2\sum_i p_i g_i}{\sum_i p_i + \sum_i g_i}$$

where P represents the predicted segmentation and G the ground truth segmentation. Each p_i and g_i is a voxel-wise prediction or a ground-truth label, respectively.

```
...
    # loss function
    loss_fn = DiceLoss(
        to_onehot_y=False, sigmoid=True, # raw outputs -> sigmoid
        include_background=True, reduction="mean",
        smooth_nr=1e-5, smooth_dr=1e-5, squared_pred=False)
...
```

Figure 10: `scripts/model/train.py`, in `trainingPipeline()` function.

3.f Saving Model Weights

We save the model with the lowest validation loss as the best model. At the end of each epoch, we also save a checkpoint with the model, optimizer, scheduler states, and the epoch number. This allows us to continue the training process from any checkpoint.

```
...
        # save best model
        if val_loss < best_metric:
            best_metric = val_loss
            best_metric_epoch = epoch + 1
            epochs_no_improve = 0
            torch.save(model.state_dict(), best_model_path_modalities)
        else:
            epochs_no_improve += 1
        # save checkpoint
        saveCheckpoint(
            model, optimizer, scheduler,
            epoch, best_metric, best_metric_epoch,
            epochs_no_improve, modalities
        )
...
```

Figure 11: scripts/model/train.py, in trainingPipeline() function.

4 Testing and Performance Evaluation

4.a Evaluation Metrics

To evaluate the performance of our trained segmentation model, we use the Dice metric implemented by MONAI. Specifically, we employ two variants of the Dice metric for evaluation purposes:

1. **Dice Metric with Mean Reduction:** This metric computes the Dice score between classes and batches and then takes the mean. It is defined as:

$$\text{Dice}(P, G) = \frac{2\sum_i p_i g_i}{\sum_i p_i + \sum_i g_i}$$

Here, p_i denotes the predicted value for voxel i and g_i denotes the corresponding ground-truth label. The mean reduction aggregates results over classes and the entire batch.

2. **Dice Metric with Mean Batch Reduction:** In this variant, the Dice metric is calculated for each batch independently, and then the results are averaged over the batches. The formula for each batch remains the same as above and a final mean is taken over all batches.

Both metrics use the Dice coefficient, which reflects the spatial overlap between prediction and ground truth. By using the batch metric, we obtain a vector of per-class scores, enabling us to understand the model's performance on each individual class separately.

```
    # metric functions
    metric_fn = DiceMetric(
        include_background=True,
        reduction="mean", # reduction over classes and batch
        get_not_nans=False # ignore NaNs, recommended
    )
    metric_fn_batch = DiceMetric(
        include_background=True,
        reduction="mean_batch", # reduction over batch
        get_not_nans=False # ignore NaNs, recommended
    )
    metric_fn.name = "Dice Score"
    metric_fn_batch.name = "Dice Score Batch"
```

Figure 12: `scripts/model/test.py`, in `testingPipeline()` function.

4.b Model to Evaluate

Since we save the best-performing model, we only need to evaluate that single model.

```python
def loadTrainedUNet(modalities: list, model_name: str):
    best_model_path_modalities = getBestModelPath(modalities=modalities,
        model_name=model_name)
    # define model architecture
    model = defineMONAIModel(modalities=modalities, model_name=model_name)
    # load the state_dict from the file
    state_dict = torch.load(best_model_path_modalities, map_location=torch.
        device(DEVICE), weights_only=True)
    # load the state_dict into the model
    model.load_state_dict(state_dict)
    print(f"Model weights loaded successfully from {best_model_path_modalities
        }")
    return model
```

Figure 13: `scripts/model/test.py`

4.c Test Pipeline

The testing pipeline evaluates a trained model (or an ensemble of models) on the test dataset using the Dice Score metric. It begins by retrieving test patient IDs and defining a DataLoader. The trained model(s) are then loaded, followed by metric calculations on the test set. For ensemble testing, predictions are aggregated across models. The results are recorded using `TensorBoard`.

```python
def testingPipeline(modalities:list, model_name:str, ensemble:bool = False):
    _, _, test_ids = getSplits()
    testloader = createDataloader(patient_ids=test_ids, modalities=modalities,
         train=False, model_name=model_name)
    # load trained model(s)
    if not ensemble:
        model = loadTrainedUNet(modalities=modalities, model_name=model_name)
    else:
        models = loadKFoldUNets(modalities=modalities, model_name=model_name)
    # metric functions
    metric_fn = DiceMetric(include_background=True, reduction="mean",
        get_not_nans=False)
    metric_fn_batch = DiceMetric(include_background=True, reduction="
        mean_batch", get_not_nans=False)
    metric_fn.name = "Dice Score"
    metric_fn_batch.name = "Dice Score Batch"
    # TensorBoard writer
    writer = SummaryWriter()
    # calculate metric on the test set
    if not ensemble:
        metric = test(model=model, dataloader=testloader, metric_fn=metric_fn,
             metric_fn_batch=metric_fn_batch, writer=writer)
    else:
        metric = testEnsemble(models=models, dataloader=testloader, metric_fn=
            metric_fn, metric_fn_batch=metric_fn_batch, writer=writer)
    writer.close()
    print(f"Results for metric {metric_fn.name} on the test set: {metric}")
```

Figure 14: `scripts/model/test.py`

```python
def test(model, dataloader, metric_fn, metric_fn_batch, writer):
    model.eval() # set model to evaluation mode
    step = 0
    # reset metrics at the beginning of validation
    metric_fn.reset()
    metric_fn_batch.reset()
    # post-processing transforms for predictions
    post_pred = Compose([Activations(sigmoid=True), AsDiscrete(threshold=0.5)
        ])
    # start testing
    with torch.no_grad():
        for batch_data in dataloader:
            step += 1
            # retrieve data from the dataloader
            inputs = batch_data["modalities"].to(DEVICE)
            labels = batch_data["mask"].to(DEVICE).long()
            # forward pass
            outputs = model(inputs)
            # apply post-processing to outputs (sigmoid, discrete with
                threshold)
            outputs_post_pred = post_pred(outputs)
            # metric score
            metric_fn(y_pred=outputs_post_pred, y=labels)
            metric_fn_batch(y_pred=outputs_post_pred, y=labels)
            # log predictions for visualization
            if step % 5 == 0:
                middle_depth = inputs.shape[-1] // 2
                writer.add_images("Test/Imags/Inputs", inputs[0:1, 0:1, :, :,
                    middle_depth], step)
                writer.add_images("Test/Imags/Labels", labels[0:1, :, :, :,
                    middle_depth], step)
                writer.add_images("Test/Imags/Predictions", outputs_post_pred
                    [0:1, :, :, :, middle_depth], step)
    # aggregate the metrics over all batches for the entire epoch
    epoch_metric = metric_fn.aggregate().item() # single scalar metric (mean
        over batch and classes)
    metric_batch = metric_fn_batch.aggregate() # vector of metrics per class
    metric_fn.reset()
    metric_fn_batch.reset()
    # extract per-class metrics from metric_batch
    metric_nc = metric_batch[0].item()
    metric_edema = metric_batch[1].item()
    metric_et = metric_batch[2].item()
    # write metric results to TensorBoard and return it
    writer.add_scalar("Test/Metric/TotalMetric", epoch_metric, global_step=
        step)
    writer.add_scalar("Test/Metric/MetricNC", metric_nc, global_step=step)
    writer.add_scalar("Test/Metric/MetricEDEMA", metric_edema, global_step=
        step)
    writer.add_scalar("Test/Metric/MetricET", metric_et, global_step=step)
    return epoch_metric
```

Figure 15: scripts/model/test.py

5 Extension to New Modalities

5.a Include Multiple Modalities

After training the model only on the T1 modality, we extend the modality set. We include T1, T2, and FLAIR modalities. T1-weighted images highlight anatomical structures and tissue boundaries, useful for identifying solid tumor regions. The T2-weighted images are highly sensitive to fluid content, making them the best for revealing edema and other fluid-rich abnormalities surrounding the tumor. FLAIR images improve the visibility of subtle lesions and edema that may be less apparent on T1 or T2 alone.

5.b Training and Evaluating on Multiple Modalities

```python
from scripts.model.train import trainingPipeline, trainingPipelineKFold
from scripts.model.test import testingPipeline
from scripts.utils.helpers import setGlobalSeed

def run():
    setGlobalSeed() # set python, random and pytorch seeds for reproducibility
    modalities = ["t1", "t2", "flair"] # load "t1", "t2", and "flair" modalities
    available_models = ["UNet", "AttentionUNet", "UNETR", "SwinUNETR", "HighResNet"]
    chosen_model = "UNet"
    kfold = False # not kfold training
    test = False # if to test or not
    if test: # start testing pipeline
        testingPipeline(modalities=modalities, model_name=chosen_model, ensemble=kfold)
    else: # start training pipeline
        if not kfold:
            trainingPipeline(modalities=modalities, model_name=chosen_model)
        else:
            trainingPipelineKFold(modalities=modalities, model_name=chosen_model)
if __name__ == "__main__":
    run()
```

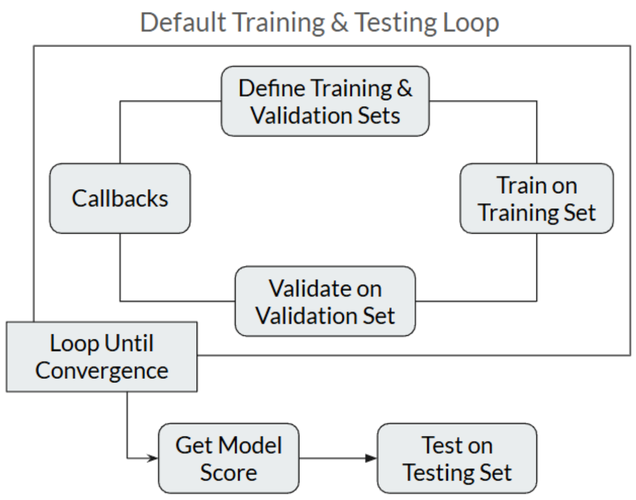
Figure 16: `main.py`

5.c Cross-validation

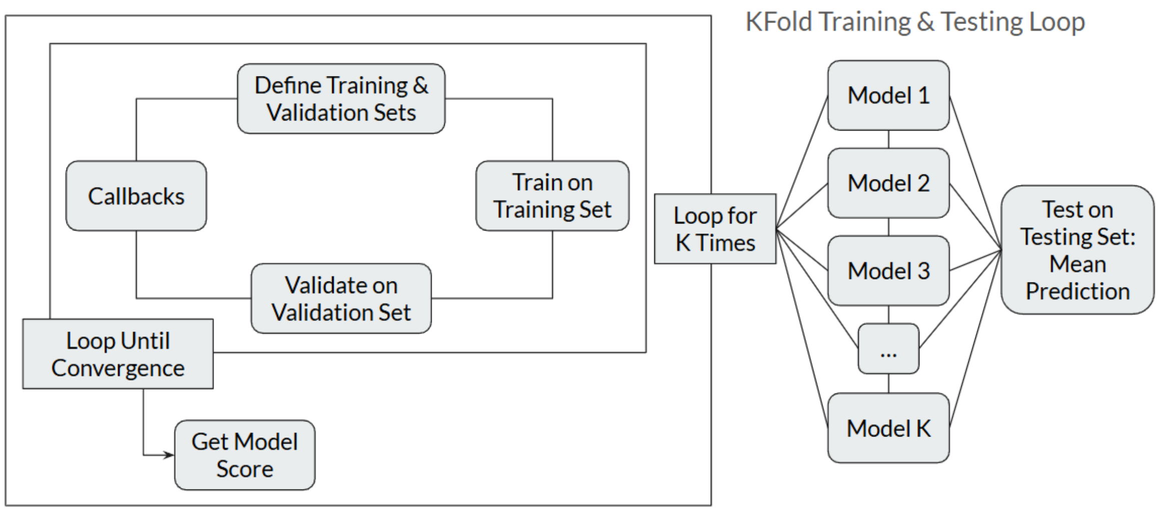
For cross-validation, we use the KFold cross-validation technique with 3 folds. The training pipeline does not change much: instead of randomly creating training and validation patient IDs, we use KFold with 3 splits and then iterate through each split. Each split holds training and validation patient IDs. For testing, we load the best models trained on each fold. During ensemble prediction, since the models output raw logits, we first apply the sigmoid function to convert these logits into probabilities. Then, we sum the probabilities across all models, average them, and use the resulting mean probabilities for final predictions.

```python
def testEnsemble(models, dataloader, metric_fn, metric_fn_batch, writer):
    step = 0
    # reset metrics at the beginning of validation
    metric_fn.reset()
    metric_fn_batch.reset()
    # post-processing transforms for predictions
    post_activation = Activations(sigmoid=True)
    post_pred = AsDiscrete(threshold=0.5)
    # start testing
    with torch.no_grad():
        for batch_data in dataloader:
            step += 1
            # retrieve data from the dataloader
            inputs = batch_data["modalities"].to(DEVICE)
            labels = batch_data["mask"].to(DEVICE).long()
            # ensemble forward pass (average prediction)
            summed_outputs = torch.zeros((inputs.shape[0], 3, *inputs.shape
                [2:]), device=DEVICE)
            for model in models:
                outputs = model(inputs)
                probabilities = post_activation(outputs)  # convert raw
                    outputs to probabilities
                summed_outputs += probabilities
            ensemble_outputs = summed_outputs / len(models)
            # apply post-processing to outputs (sigmoid, discrete with
                threshold)
            outputs_post_pred = post_pred(ensemble_outputs)
            # metric score
            metric_fn(y_pred=outputs_post_pred, y=labels)
            metric_fn_batch(y_pred=outputs_post_pred, y=labels)
            # log predictions for visualization
            if step % 5 == 0:
                middle_depth = inputs.shape[-1] // 2
                writer.add_images("Test/KFold/Images/Inputs", inputs[0:1, 0:1,
                     :, :, middle_depth], step)
                writer.add_images("Test/KFold/Images/Labels", labels[0:1, :,
                    :, :, middle_depth], step)
                writer.add_images("Test/KFold/Images/Predictions",
                    outputs_post_pred[0:1, :, :, :, middle_depth], step)
    # aggregate the metrics over all batches for the entire epoch
    # ...
    # extract per-class metrics from metric_batch
    # ...
    # write metric results to TensorBoard and return it
    # ...
    return epoch_metric
```

Figure 17: scripts/model/test_kfold.py


Default Training & Testing Loop
Define Training & Validation Sets
Callbacks
Train on Training Set
Validate on Validation Set
Loop Until Convergence
Get Model Score
Test on Testing Set

(a) Default machine learning training and testing pipeline.


KFold Training & Testing Loop
Define Training & Validation Sets
Callbacks
Train on Training Set
Validate on Validation Set
Loop Until Convergence
Get Model Score
Loop for K Times
Model 1
Model 2
Model 3
...
Model K
Test on Testing Set: Mean Prediction

(b) KFold machine learning training and testing pipeline.

Figure 18: Visualization for default and KFold machine learning training and testing pipelines. (top: default, bottom: KFold).

6 Results

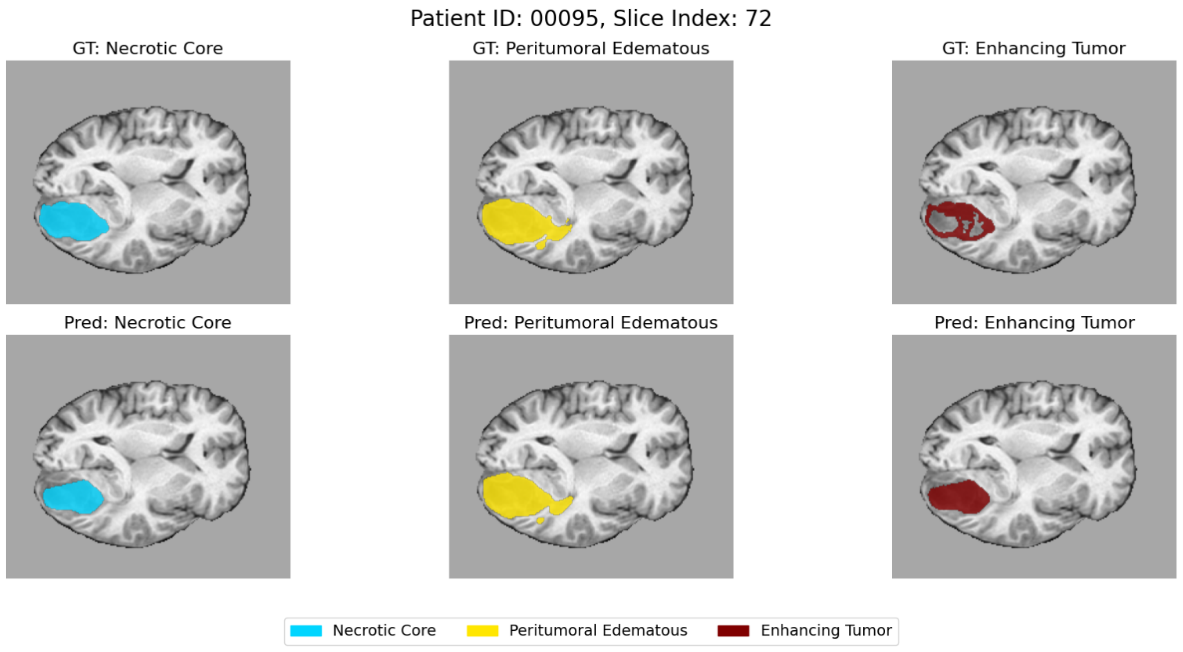


Figure 19: Ground truth and predicted classes for the patient 00095 overlayed on T1 modality image. Each column corresponds to one of the three tumor classes (Necrotic Core, Peritumoral Edematous, and Enhancing Tumor). The top row shows ground truth masks overlaid on the T1 modality, and the bottom row shows the predicted masks. Different colors indicate each class, and the T1 scan is shown in grayscale.

Method	Default Training Procedure, T1 Modality	Default Training Procedure, Multiple Modalities	**KFold Training Procedure, Multiple Modalities**
Dice Score	0.6935	0.7576	**0.7616**

Table 1: Dice Score results for different training procedures and modalities. The best technique and score is highlighted in bold.

From the results, we see that the introduction of additional modalities (T1, T2, and FLAIR) significantly improved the model performance, as we had anticipated. Additionally, employing the KFold cross-validation technique increased the accuracy of the predictions, although not substantially. In KFold, we use an ensemble prediction technique, where the ensemble is trained on the whole dataset, rather than just a subset of it, which generally leads to better generalization and accuracy. One of the reasons why there is no great improvement with KFold is the fact that the BRaT dataset is already well balanced and comprehensive, providing sufficient diversity in the training and validation splits without the need to use the cross-validation technique. Another reason is the fact that we used data augmentation techniques during training, which effectively increased the diversity of the training data without the need for additional folds to achieve good generalization.