Automatic Glioma Segmentation Pipeline using BraTS Dataset and 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
from scripts.model.test import testingPipeline
from scripts.utils.utils import setGlobalSeed

def run():
    # set python, random and pytorch seeds for reproducibility
    setGlobalSeed()
    # define modalities to load
    modalities = ["t1"] # this will load "t1"modality and segmentation mask
        later in createDataloader() function.
    # start training pipeline
    trainingPipeline(modalities=modalities)
    # start testing pipeline
    testingPipeline(modalities=modalities, kfold=False)

if __name__ == "__main__":
    run()
```

Figure 1: main.py

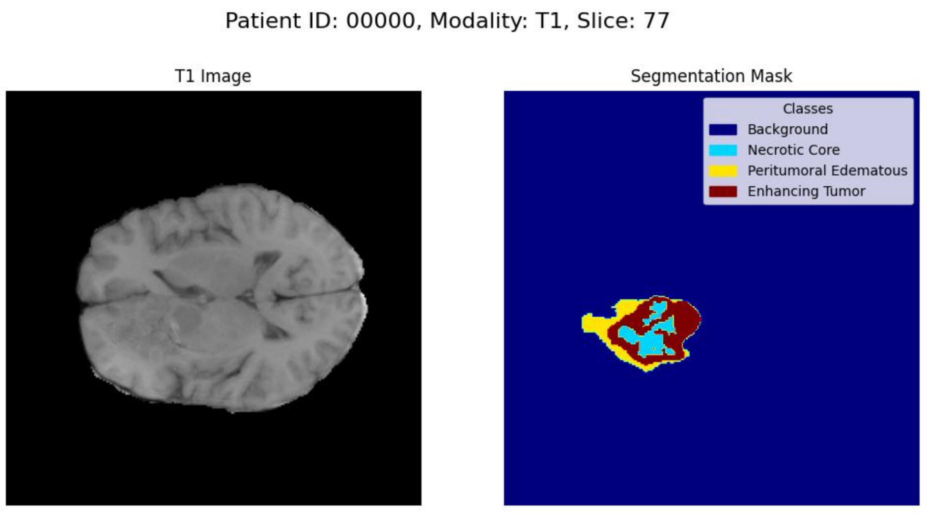


Figure 2: Visualization for T1 modality and segmentation mask for patient with `ID=00000`.

1.b DataLoader

```python
def createDataloader(patient_ids: list, modalities: list, batch_size: int =
    BATCH_SIZE, train: bool = True):
    # prepare data in MONAI format
    data_dicts = [
        {
            f"modality_{i}": f"{DATA_PATH}/{PATIENT_FOLDER_NAME}{pid}/{
                PATIENT_FOLDER_NAME}{pid}_{modality}.nii"
            for i, modality in enumerate(modalities)
        } | {
            "mask": f"{DATA_PATH}/{PATIENT_FOLDER_NAME}{pid}/{
                PATIENT_FOLDER_NAME}{pid}_seg.nii"
        }
        for pid in patient_ids
    ]
    # define data augmentation and transforms
    transforms = defineTransforms(modalities=modalities, train=train)
    # 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 3: `scripts/data/dataloader.py`

1.c Data Augmentations

```python
def defineTransforms(modalities: list, train: bool = True):
    if train:
        transforms = Compose([
            LoadImaged(keys=[f"modality_{i}" for i in range(len(modalities))]
                + ["mask"]),
            EnsureChannelFirstd(keys=[f"modality_{i}" for i in range(len(
                modalities))]),
            ConvertToMultiChannelBasedOnBratsClassesd(keys="mask"),
            ConcatItemsd(keys=[f"modality_{i}" for i in range(len(modalities))
                ], name="modalities", dim=0),
            Orientationd(keys=["modalities", "mask"], axcodes="RAS"),
            Spacingd(
                keys=["modalities", "mask"],
                pixdim=(1.0, 1.0, 1.0),
                mode=("bilinear", "nearest"),
            ),
            RandRotate90d(keys=["modalities", "mask"], prob=0.5),
            RandFlipd(keys=["modalities", "mask"], prob=0.5, spatial_axis=1),
            RandFlipd(keys=["modalities", "mask"], prob=0.5, spatial_axis=2),
            RandScaleIntensityd(keys="modalities", factors=0.1, prob=0.25),
            RandShiftIntensityd(keys="modalities", offsets=0.1, prob=0.25),
            RandGaussianNoised(keys="modalities", prob=0.25),
            RandBiasFieldd(keys="modalities", prob=0.5, degree=2),
            NormalizeIntensityd(keys="modalities", nonzero=True, channel_wise=
                True),
            CustomSpatialCrop(keys=["modalities", "mask"], start=5, end=-6),
            ToTensord(keys=["modalities", "mask"])
        ])
    else:
        transforms = Compose([
            LoadImaged(keys=[f"modality_{i}" for i in range(len(modalities))]
                + ["mask"]),
            EnsureChannelFirstd(keys=[f"modality_{i}" for i in range(len(
                modalities))]),
            ConvertToMultiChannelBasedOnBratsClassesd(keys="mask"),
            ConcatItemsd(keys=[f"modality_{i}" for i in range(len(modalities))
                ], name="modalities", dim=0),
            Orientationd(keys=["modalities", "mask"], axcodes="RAS"),
            Spacingd(
                keys=["modalities", "mask"],
                pixdim=(1.0, 1.0, 1.0),
                mode=("bilinear", "nearest"),
            ),
            NormalizeIntensityd(keys="modalities", nonzero=True, channel_wise=
                True),
            CustomSpatialCrop(keys=["modalities", "mask"], start=5, end=-6),
            ToTensord(keys=["modalities", "mask"])
        ])
    return transforms
```

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

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.

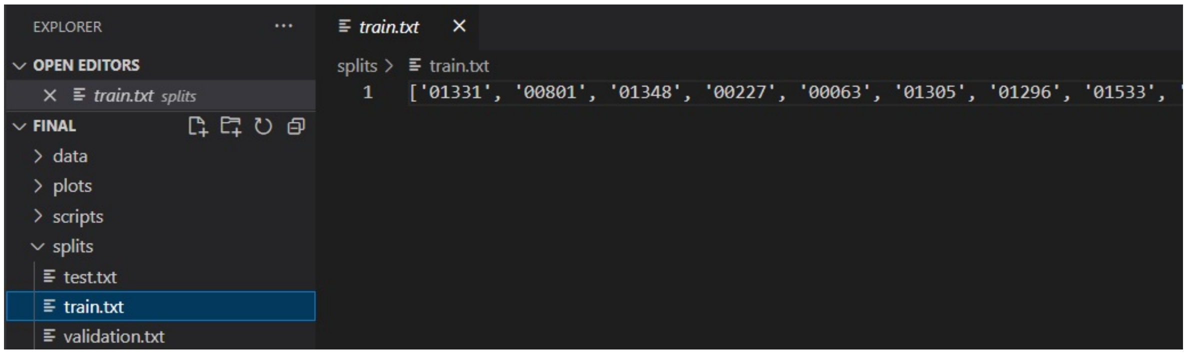
```python
def createSplits():
    # retrieve patient IDs
    patient_ids = getPatientIDs()
    # shuffle IDs
    random.shuffle(patient_ids)
    # get length for train, validation and test sets
    len_train = int(len(patient_ids)*TRAIN_PERCENTAGE)
    len_val = int(len(patient_ids)*VAL_PERCENTAGE)
    # 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 5: `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 6: `scripts/data/split.py`



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

Figure 7: Content of `train.txt` file holding training patient IDs. The result of running `createSplits()` function.

3 Model Selection and Training

3.a Training Pipeline

```python
def trainingPipeline(modalities: list) -> None:
    train_ids, val_ids, _ = getSplits()
    trainloader = createDataloader(patient_ids=train_ids, modalities=modalities, train=True)
    valloader = createDataloader(patient_ids=val_ids, modalities=modalities, train=False)
    model = UNet(spatial_dims=3, in_channels=len(modalities), out_channels=3,
        channels=(16, 32, 64, 128, 256), strides=(2, 2, 2, 2),
        dropout=0.2, num_res_units=2, norm="batch",).to(DEVICE)
    loss_fn = DiceLoss(
        to_onehot_y=False, sigmoid=True, include_background=True,
        reduction="mean", smooth_nr=1e-5, smooth_dr=1e-5, squared_pred=False)
    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)
    post_pred = Compose([Activations(sigmoid=True), AsDiscrete(threshold=0.5)])
    optimizer = optim.AdamW(model.parameters(), lr=LR)
    scheduler = optim.lr_scheduler.CosineAnnealingWarmRestarts(
        optimizer, T_0=10, T_mult=1, eta_min=1e-6)
    best_metric = 2 # dice loss is between 0 and 1
    best_metric_epoch = -1
    epochs_no_improve = 0
    writer = SummaryWriter()
    best_model_path_modalities = BEST_MODEL_PATH.replace(".pth", "_".join(modalities) + ".pth")
    # training
    for epoch in range(EPOCHS):
        # train
        train_loss  = train(model, trainloader, loss_fn, optimizer, epoch, writer)
        # validate
        val_loss, val_dice = validate(model, valloader, post_pred, loss_fn, metric_fn, metric_fn_batch, epoch, writer)
        # learning rate scheduler step
        scheduler.step(val_loss)
        for param_group in optimizer.param_groups:
            lr = param_group["lr"]
            writer.add_scalar("Train/LearningRate", lr, epoch)
            break
        # early stopping
        if val_loss < best_metric:
            best_metric, best_metric_epoch, epochs_no_improve = val_loss, epoch + 1, 0
            # save the best model
            torch.save(model.state_dict(), best_model_path_modalities)
        else:
            epochs_no_improve += 1
        if epochs_no_improve >= PATIENCE:
            break
    writer.close()
```

Figure 8: scripts/model/train.py

```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 9: `scripts/model/train.py`

3.b 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 10: scripts/model/train.py

3.c Early Stopping

```
...
        # early stopping
        if val_loss < best_metric:
            best_metric = val_loss
            best_metric_epoch = epoch + 1
            epochs_no_improve = 0
            # save the best model
            torch.save(model.state_dict(), BEST_MODEL_PATH)
        else:
            epochs_no_improve += 1
        if epochs_no_improve >= PATIENCE:
            break
...
```

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

3.d 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", # reduction over the batch
        smooth_nr=1e-5, # numerator smoothing to avoid division by zero
        smooth_dr=1e-5, # denominator smoothing to avoid division by zero
        squared_pred=False # do not square predictions
    )
...
```

Figure 12: `scripts/model/train.py`, in `trainingPipeline()` function.

3.e Saving Model Weights

We save only the best-performing model checkpoint, i.e. the model that achieves the lowest validation loss at the end of an epoch.

```
...
            # save the best model
            torch.save(model.state_dict(), BEST_MODEL_PATH)
            print(f"    Saved new best model at epoch {epoch + 1}")
...
```

Figure 13: 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:

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

   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 14: `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, kfold: bool = False):
    best_model_path_modalities = BEST_MODEL_PATH.replace(".pth", "_".join(
        modalities) + ".pth")
    if kfold:
        kfold_results = readJSON(path=KFOLD_RESULTS_PATH)
        best_fold_idx = kfold_results["best_fold_idx"]
        best_model_file = best_model_path_modalities.replace(".pth", f"_fold-{
            best_fold_idx}.pth")
    else:
        best_model_file = best_model_path_modalities
    # define model architecture
    model = UNet(
        spatial_dims=3, in_channels=len(modalities), out_channels=3,
        channels=(16, 32, 64, 128, 256), strides=(2, 2, 2, 2), dropout=0.2,
        num_res_units=2, norm="batch").to(DEVICE)
    # load the state_dict from the file
    state_dict = torch.load(best_model_file, map_location=torch.device(DEVICE)
        )
    # load the state_dict into the model
    model.load_state_dict(state_dict)
    print(f"Model weights loaded successfully from {best_model_file}")
    return model
```

Figure 15: scripts/model/test.py

4.c Test Pipeline

```python
def testingPipeline(modalities: list, kfold: bool = False):
    # retrieve testing patient IDs
    _, _, test_ids = getSplits()
    # define testing dataloader
    testloader = createDataloader(patient_ids=test_ids, modalities=modalities,
         train=False)
    # load trained model
    model = loadTrainedUNet(modalities=modalities, kfold=kfold)
    # 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
    metric = test(model=model, 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 16: 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 17: 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
from scripts.model.test import testingPipeline
from scripts.utils.utils import setGlobalSeed

def run():
    # set python, random and pytorch seeds for reproducibility
    setGlobalSeed()
    # define modalities to load
    modalities = ["t1", "t2", "flair"] # this will load "t1", "t2", and "flair
        " modalities and segmentation mask later in createDataloader()
        function.
    # start training pipeline
    trainingPipeline(modalities=modalities)
    # start testing pipeline
    testingPipeline(modalities=modalities, kfold=False)

if __name__ == "__main__":
    run()
```

Figure 18: `main.py`

5.c Multiple Modalities Results

See in **Results** section.

5.d Cross-validation

For cross-validation, we use the KFold cross-validation technique with 5 folds. The training pipeline does not change much: instead of randomly creating training and validation patient IDs, we use KFold with 5 splits and then iterate through each split. Each split holds training and validation patient IDs. For testing, we load the model that performed the best among all the folds using the JSON file stored under `KFOLD_RESULTS_PATH`.

```python
def trainingPipelineKFold(modalities: list, n_splits: int = 5) -> None:
    train_ids, val_ids, _ = getSplits()
    all_ids = train_ids + val_ids
    kf = KFold(n_splits=n_splits, shuffle=True, random_state=SEED)
    writer = SummaryWriter()
    best_fold_metric = 2.0
    best_fold_epoch = -1
    best_fold_idx = -1
    folds_results = {}
    for fold_idx, (train_index, val_index) in enumerate(kf.split(all_ids)):
        fold_train_ids = [all_ids[i] for i in train_index]
        fold_val_ids = [all_ids[i] for i in val_index]
        ... # same as in trainingPipeline()
        for epoch in range(EPOCHS):
            ... # same as in trainingPipeline()
        folds_results[fold_idx+1] = {
            "best_metric": best_metric,
            "best_epoch": best_metric_epoch
        }
        if best_metric < best_fold_metric:
            best_fold_metric, best_fold_epoch, best_fold_idx = best_metric,
                best_metric_epoch, fold_idx + 1
    kfold_results = {
        "best_fold_idx": best_fold_idx,
        "best_fold_metric": best_fold_metric,
        "folds_results": folds_results
    }
    writeJSON(kfold_results, KFOLD_RESULTS_PATH)
    writer.close()
```

Figure 19: `scripts/model/train_kfold.py`

6 Results