Task 4

Task 4

Data Handling and Preprocessing (10 Points)

a. You can focus for now on loading the T1-weighted images and the matching labels.
b. Create a dataloader for the data using PyTorch's Dataloader (or Monai's Dataloader class)
c. Create suitable augmentations for the task to solve. Please note: If you apply transformations to the input data, you should think about if you need to apply any transformation to the label of the image as well.

We didnt used Monai's Dataloader class, because we wanted to use the data in the HPC. The dataset implementation is availabel in the `task4/brats_segmentations/dataloader.py`

`file name: ../task4/brats_segmentation/dataloader.py`

```
class ConvertToMultiChannelBasedOnBratsClassesd(MapTransform):
    def __call__(self, data):
        d = dict(data)
        for key in self.keys:
            print(f"Original label shape: {d[key].shape}, unique values: {torch.unique(d[key]
            result = []

            # Tumor Core (TC): Combine label 1 and label 4
            tc = torch.logical_or(d[key] == 1, d[key] == 4)
            result.append(tc)
            print(f"Tumor Core (TC) unique values: {torch.unique(tc)}")

            # Whole Tumor (WT): Combine label 1, label 2, and label 4
            wt = torch.logical_or(torch.logical_or(d[key] == 2, d[key] == 4), d[key] == 1)
            result.append(wt)
```

```
            print(f"Whole Tumor (WT) unique values: {torch.unique(wt)}")

            # Enhancing Tumor (ET): Only label 4
            et = d[key] == 4
            result.append(et)
            print(f"Enhancing Tumor (ET) unique values: {torch.unique(et)}")

            # Stack binary masks into multi-channel format
            d[key] = torch.stack(result, dim=0).float()
            print(f"Transformed label shape: {d[key].shape}, unique values: {torch.unique(d[
        return d



# Preprocessing and augmentation pipeline
def get_transforms(roi_size, augment=True):
    """
    Generate transforms for data preprocessing and augmentation.

    Args:
        roi_size (tuple): Size of the region of interest for cropping.
        augment (bool): Whether to apply augmentations.

    Returns:
        Compose: Transformation pipeline.
    """
    transforms = [
        LoadImaged(keys=["t1", "t1ce", "t2", "flair", "label"]),
        EnsureChannelFirstd(keys=["t1", "t1ce", "t2", "flair", "label"]),
        ConcatItemsd(keys=["t1", "t1ce", "t2", "flair"], name="image"),
        Orientationd(keys=["image", "label"], axcodes="RAS"),
        Spacingd(keys=["image", "label"], pixdim=(1.0, 1.0, 1.0), mode=("bilinear", "nearest"
        NormalizeIntensityd(keys="image", nonzero=True, channel_wise=True),
        ConvertToMultiChannelBasedOnBratsClassesd(keys=["label"]),  # One-hot encode labels
    ]

    if augment:
        transforms.extend([
            RandSpatialCropd(keys=["image", "label"], roi_size=roi_size, random_size=False),
            RandFlipd(keys=["image", "label"], prob=0.5, spatial_axis=0),
            RandFlipd(keys=["image", "label"], prob=0.5, spatial_axis=1),
```

```
            RandFlipd(keys=["image", "label"], prob=0.5, spatial_axis=2),
        ])

    transforms.append(ToTensord(keys=["image", "label"]))
    return Compose(transforms)


# DataLoader setup with CacheDataset
def get_dataloaders(split_dir, roi_size, batch_size, num_workers=4):
    """
    Create data loaders for training, validation, and testing.

    Args:
        split_dir (str): Path to the directory containing split JSON files.
        roi_size (tuple): Size of the region of interest for cropping.
        batch_size (int): Batch size for data loaders.
        num_workers (int): Number of workers for data loading.

    Returns:
        tuple: Training, validation, and test DataLoaders.
    """
    # Generate transforms for train and validation/test
    train_transform = get_transforms(roi_size, augment=True)
    val_transform = get_transforms(roi_size, augment=False)

    # Load dataset splits
    with open(f"{split_dir}/train.txt", "r") as f:
        train_files = json.load(f)
    with open(f"{split_dir}/validation.txt", "r") as f:
        val_files = json.load(f)
    with open(f"{split_dir}/test.txt", "r") as f:
        test_files = json.load(f)

    # Use CacheDataset for faster loading
    train_ds = CacheDataset(data=train_files, transform=train_transform, cache_rate=0.8, num
    val_ds = CacheDataset(data=val_files, transform=val_transform, cache_rate=1.0, num_worke
    test_ds = CacheDataset(data=test_files, transform=val_transform, cache_rate=1.0, num_wor

    # Create data loaders
    train_loader = DataLoader(train_ds, batch_size=batch_size, shuffle=True, num_workers=num
    val_loader = DataLoader(val_ds, batch_size=batch_size, shuffle=False, num_workers=num_wo
    test_loader = DataLoader(test_ds, batch_size=batch_size, shuffle=False, num_workers=num_
```

```
    return train_loader, val_loader, test_loader


# Visualization utility using Weights & Biases
def visualize_samples(loader, num_samples=3, project_name="brats_segmentation", slice_axis=2
    """
    Visualize 2D slices of images and labels using Weights & Biases.
    Logs a table of images with segmentation masks overlaid.

    Args:
        loader: DataLoader to fetch samples from.
        num_samples: Number of samples to visualize.
        project_name: W&B project name.
        slice_axis: Axis along which to slice (0=coronal, 1=sagittal, 2=axial).
    """
    wandb.init(project=project_name)
    class_labels = {
        0: "Background",
        1: "Tumor Core",
        2: "Whole Tumor",
        3: "Enhancing Tumor",
    }

    table = wandb.Table(columns=["Slice Index", "Image", "Ground Truth"])

    for i, batch in enumerate(loader):
        if i >= num_samples:
            break

        # Extract image and label
        image = batch["image"][0].cpu().numpy()  # Shape: [C, H, W, D]
        label = batch["label"][0].cpu().numpy()  # Shape: [C, H, W, D]

        # Choose slices along the given axis
        slice_index = image.shape[slice_axis + 1] // 2  # Middle slice along the given axis
        image_slice = np.take(image[0], slice_index, axis=slice_axis)
        label_slices = [np.take(label[c], slice_index, axis=slice_axis) for c in range(label

        # Combine labels into a single mask for visualization
        combined_label = np.zeros_like(label_slices[0])
        for c, mask in enumerate(label_slices, 1):  # Start class IDs from 1
            combined_label[mask > 0] = c
```

```
        # Log the image and labels
        table.add_data(
            slice_index,
            wandb.Image(image_slice, caption=f"Slice {slice_index}"),
            wandb.Image(
                image_slice,
                masks={
                    "ground_truth": {
                        "mask_data": combined_label,
                        "class_labels": class_labels,
                    }
                },
                caption=f"Slice {slice_index}",
            ),
        )

    wandb.log({"Visualization Samples": table})
    wandb.finish()



if __name__ == "__main__":
    # Example usage
    split_dir = "./splits/split3"
    roi_size = (128, 128, 128)
    batch_size = 1
    num_workers = 4

    train_loader, val_loader, test_loader = get_dataloaders(split_dir, roi_size, batch_size,

    # Inspect one batch from the training DataLoader
    print("Testing the training DataLoader...")
    for batch in train_loader:
        print(f"Image shape: {batch['image'].shape}")
        print(f"Label shape: {batch['label'].shape}")
        print(f"Label unique values: {torch.unique(batch['label'])}")
        break

    # Visualize samples using Weights & Biases
    print("Visualizing samples with W&B...")
    visualize_samples(train_loader)
```