Ex8

December 2, 2024

1 Exercise 8: Supervised segmentation with a 2D U-Net

1.1 Introduction

Neural networks (NNs) are powerful computational models inspired by the structure and function of the human brain, designed to learn patterns from data through layers of interconnected nodes. These networks, particularly Convolutional Neural Networks (CNNs), are exceptionally well-suited for image-related tasks, as they excel at recognizing spatial hierarchies in data, such as edges, textures, and shapes. In supervised segmentation, where the goal is to assign a label to each pixel in an image (e.g., classifying tissues in medical scans), NNs leverage labeled training data to learn complex features that distinguish between different regions, even in challenging scenarios with subtle intensity differences or irregular shapes.

For this task, the dataset consists of 2D coronal slices of MR scans and their corresponding tissue segmentations, where each pixel is annotated with its respective tissue class. Neural networks like U-Net are widely used in this domain due to their encoder-decoder architecture, which combines feature extraction with precise localization. The encoder learns high-level features, while the decoder reconstructs the spatial information for pixel-wise predictions. This architecture is particularly effective for medical imaging, as it can handle the variability in tissue shapes and intensities while maintaining fine-grained segmentation accuracy. Improving the MONAI BasicUNet for this dataset involves optimizing the architecture and loss functions to enhance segmentation performance and better address the nuances of the MR images. Further detail of the changes is provided below.

Additionally, the model has been trained on the HPC, and edited in VS Code.

1.2 Initial steps

To run the code you need to do two things:

1) Install monai
2) Upload the training and validation data so you can train the model

The following two code blocks have the commands to do that.

[ ]:
```
# Install monai, you only need to do this the first time the note book is run
!pip install "monai[all]"
```

[ ]:
```
# Upload the training data zip file. Click on Browse and upload the file.
# The zip file will show up under the folder icon on the left.
```

```
from google.colab import files

uploaded = files.upload()

for fn in uploaded.keys():
  print('User uploaded file "{name}" with length {length} bytes'.format(
      name=fn, length=len(uploaded[fn])))
```

[ ]:
```
# Unzip the training data
!unzip training_data_exercise.zip
```

1.3 U-Net training code

The following code block has the U-Net training code. The code bits you should modify have a TODO-statement above them

[2]:
```
import logging
import os
import sys
from glob import glob
import argparse
import numpy as np
import matplotlib.pyplot as plt
import torch
from torch.utils.tensorboard import SummaryWriter
import torch.nn.functional as F

import monai
from monai.data import create_test_image_2d, list_data_collate,␣
 ↪decollate_batch, DataLoader
from monai.inferers import sliding_window_inference
from monai.metrics import DiceMetric
from monai.transforms import (
    Activations,
    EnsureChannelFirstd,
    AsDiscrete,
    Compose,
    LoadImaged,
    RandSpatialCropd,
    RandFlipd,
    RandAffined,
    RandGaussianNoised,
    RandShiftIntensityd,
    EnsureTyped
)
from monai.visualize import plot_2d_or_3d_image
```

```
[3]: def _get_data(train_dir):
         images = sorted(glob(os.path.join(train_dir, "*T1w.npy")))
         segs = sorted(glob(os.path.join(train_dir, "*T1w_tissue_segmentation.npy")))
         files = [{"img": img, "seg": seg} for img, seg in zip(images, segs)]
         return files


     def _get_training_transforms():
         roi_size = [200, 200]

         train_transforms = Compose(
             [
                 # Load image and label
                 LoadImaged(keys=["img", "seg"]),
                 EnsureChannelFirstd(keys=["img", "seg"]),

                 # Random spatial cropping with the specified ROI size
                 RandSpatialCropd(keys=["img", "seg"], roi_size=roi_size),

                 # Data Augmentation
                 RandFlipd(keys=["img", "seg"], spatial_axis=[0], prob=0.5),  # Flip␣
       ↪along axis 0
                 RandFlipd(keys=["img", "seg"], spatial_axis=[1], prob=0.5),  # Flip␣
       ↪along axis 1

                 RandAffined(
                     keys=["img", "seg"],
                     prob=0.7,  # Probability of applying affine transformations
                     rotate_range=(0.1, 0.1, 0.0),  # Random rotations (radians)
                     translate_range=(10, 10, 0),  # Random translations (pixels)
                     scale_range=(0.1, 0.1, 0.0),  # Random scaling
                     mode=("bilinear", "nearest"),  # Interpolation modes
                 ),

                 # Non-Spatial just applied to image
                 RandGaussianNoised(keys=["img"], prob=0.2, mean=0.0, std=0.1),  #␣
       ↪Add random Gaussian noise
                 RandShiftIntensityd(keys=["img"], offsets=0.1, prob=0.5),  #␣
       ↪Intensity shifts

                 # Ensure final data format
                 EnsureTyped(keys=["img", "seg"]),
             ]
         )


         return train_transforms, roi_size
```

```
def _get_validation_transforms():
    val_transforms = Compose(
        [
            LoadImaged(keys=["img", "seg"]),
            EnsureChannelFirstd(keys=["img", "seg"]),
        ]
    )

    return val_transforms

def _get_data_loaders(train_files, train_transforms, val_files, val_transforms):

    train_ds = monai.data.Dataset(data=train_files, transform=train_transforms)
    train_loader = DataLoader(
        train_ds,
        batch_size=2,
        shuffle=True,
        num_workers=2,
        collate_fn=list_data_collate,
        pin_memory=torch.cuda.is_available(),
    )
    # create a validation data loader
    val_ds = monai.data.Dataset(data=val_files, transform=val_transforms)
    val_loader = DataLoader(val_ds, batch_size=1, num_workers=2,␣
↪collate_fn=list_data_collate)

    return train_loader, val_loader

def _loss_function(outputs, labels):
    # Dice Loss from MONAI
    dice_loss_fn = monai.losses.DiceLoss(to_onehot_y=True, softmax=True)
    dice_loss = dice_loss_fn(outputs, labels)

    # Cross-Entropy loss from PyTorch
    ce_loss_fn = torch.nn.CrossEntropyLoss()

    # CrossEntropyLoss expects labels without one-hot encoding.
    # Remove one-hot encoding from labels if it's applied earlier.
    labels = torch.argmax(labels, dim=1)  # Convert from one-hot to class␣
↪indices
    ce_loss = ce_loss_fn(outputs, labels)

    # Weighted combination of losses
    dice_weight = 0.8
    ce_weight = 0.2
    full_loss = dice_weight * dice_loss + ce_weight * ce_loss

    return full_loss

def _get_model(device):
    model = monai.networks.nets.BasicUNet(
        spatial_dims=2,
        in_channels=1,
        out_channels=4,
        features=(32, 64, 128, 256, 512, 32),
        act="LeakyReLU",
        norm="instance",  # Instance normalization for medical imaging
        dropout=0.4,  # Regularization
    ).to(device)

    return model

def _get_optimizer(model):
    optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)

    return optimizer

# Training function start here
def train(train_dir, validation_dir):
    monai.config.print_config()
    logging.basicConfig(stream=sys.stdout, level=logging.INFO)

    # Get the training data files (images and segmentation)
    train_files = _get_data(train_dir)

    # Set up training transforms.
    train_transforms, roi_size = _get_training_transforms()

    # Get the validation data files (images and segmentations)
    val_files = _get_data(validation_dir)

    # Get the validation transforms.
    val_transforms = _get_validation_transforms()


    # Get data loaders for running the training and validation
    train_loader, val_loader = _get_data_loaders(train_files, train_transforms,␣
 ↪val_files, val_transforms)

    # The device variable is used to automatically run the training on
    # the GPU if its available, otherwise the CPU is used
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    model = _get_model(device)

    optimizer = _get_optimizer(model)


    # Final thing before training. Setup the validation metric so we can measure
    # the validation performance during training.
    # I'm not including the background into the validation dice as we are more␣
↪interest
    # in the foreground structures.
    dice_metric = DiceMetric(include_background=False, reduction="mean",␣
↪get_not_nans=False)

    # Define post-processing steps for the output so we can compare it to the␣
↪validation segmentations
    # The loss does this automatically because we have defined
    # to_onehot_y=True, softmax=True
    # so this is not necessary for training, just for validation
    post_trans = Compose([Activations(softmax=True), AsDiscrete(argmax=True)])

    # Okay start the training loop
    # Feel free to change the number of
    # epochs and the other parameters

    # Validation interval
    val_interval = 10

    # Keep track of the best metric & epoch
    best_metric = -1
    best_metric_epoch = -1

    epochs = 700

    # Adding early stopping
    patience = 20
    epochs_no_improve = 0
    stop_training = False

    # Losses and metrics
    epoch_loss_values = list()
    metric_values = list()

    # Save a log into a directory called log
    writer = SummaryWriter(log_dir='./log')

    for epoch in range(epochs):
        if stop_training:
            break

        print("-" * 100)
        print(f"epoch {epoch + 1}/{epochs}")
        model.train()
        epoch_loss = 0
        step = 0
        for batch_data in train_loader:
            step += 1

            #Get training data for this batch
            inputs, labels = batch_data["img"].to(device), batch_data["seg"].
↪to(device)
            optimizer.zero_grad()

            # Push the input through the model
            outputs = model(inputs)

            # Get the loss
            loss = _loss_function(outputs, labels)

            # Backpropagate and call the optimizer
            loss.backward()
            optimizer.step()

            # Store the losses
            epoch_loss += loss.item()
            print(f"{step}/{len(train_loader)}, train_loss: {loss.item():.4f}")
            writer.add_scalar("train_loss", loss.item(), len(train_loader) *␣
↪epoch + step)

        # These statements plot examples into the tensorboard log dile
        plot_2d_or_3d_image(inputs[0], epoch + 1, writer, index=0,␣
↪tag="input_image")
        plot_2d_or_3d_image(labels[0], epoch + 1, writer, index=0,␣
↪tag="input_labeling")
        output_tmp = [post_trans(i) for i in decollate_batch(outputs)]
        plot_2d_or_3d_image(output_tmp[0], epoch + 1, writer, index=0,␣
↪tag="input_prediction")

        # Save the average loss per epoch
        epoch_loss /= step
        epoch_loss_values.append(epoch_loss)
        print(f"epoch {epoch + 1} average loss: {epoch_loss:.4f}")

        # Run the validation, every val_interval steps
        if (epoch + 1) % val_interval == 0:
            model.eval()

            # The no_grad is just to tell torch that it doesn't need to␣
↪backpropagate here
            with torch.no_grad():
                val_images = None
                val_labels = None
                val_outputs = None
                im_num = 0
                for val_data in val_loader:
                    val_images, val_labels = val_data["img"].to(device),␣
↪val_data["seg"].to(device)
                    sw_batch_size = 4
                    val_outputs = sliding_window_inference(val_images,␣
↪roi_size, sw_batch_size, model)
                    val_outputs = [post_trans(i) for i in␣
↪decollate_batch(val_outputs)]
                    # compute metric for current iteration
                    dice_metric(y_pred=val_outputs, y=val_labels)

                    # Plot the validation images, labels and predictions into␣
↪the log file
                    plot_2d_or_3d_image(val_images, epoch + 1, writer, index=0,␣
↪tag="validation_image"+str(im_num))
                    plot_2d_or_3d_image(val_labels, epoch + 1, writer, index=0,␣
↪tag="validation_labeling"+str(im_num))
                    plot_2d_or_3d_image(val_outputs, epoch + 1, writer,␣
↪index=0, tag="validation_prediction"+str(im_num))
                    im_num += 1

                # aggregate the final mean dice result
                metric = dice_metric.aggregate().item()
                # reset the status for next validation round
                dice_metric.reset()
                metric_values.append(metric)

                # Keep track of the best metric and save the best model
                if metric > best_metric:
                    best_metric = metric
                    best_metric_epoch = epoch + 1
                    epochs_no_improve = 0
                    torch.save(model.state_dict(),␣
↪"best_metric_model_segmentation2d_dict.pth")
                    print("saved new best metric model")
                else:
```

```
                epochs_no_improve += 1
            print(
                "current epoch: {} current mean dice: {:.4f} best mean dice:
↪ {:.4f} at epoch {}".format(
                    epoch + 1, metric, best_metric, best_metric_epoch
                )
            )
            writer.add_scalar("val_mean_dice", metric, epoch + 1)

            # Check early stopping condition
            if epochs_no_improve >= patience:
                print(f"Early stopping triggered after {epoch + 1} epochs!")
                stop_training = True

    print(f"train completed, best_metric: {best_metric:.4f} at epoch:␣
↪{best_metric_epoch}")
    writer.close()

# This function is used for the testing the performance once the test data
# is released
def test(test_dir):
    monai.config.print_config()
    logging.basicConfig(stream=sys.stdout, level=logging.INFO)

    test_files = _get_data(test_dir)
    test_transforms = _get_validation_transforms()
    test_ds = monai.data.Dataset(data=test_files, transform=test_transforms)
    # sliding window inference need to input 1 image in every iteration
    test_loader = DataLoader(test_ds, batch_size=1, num_workers=2,␣
↪collate_fn=list_data_collate)
    dice_metric = DiceMetric(include_background=False, reduction="mean",␣
↪get_not_nans=False)
    post_trans = Compose([Activations(softmax=True), AsDiscrete(argmax=True)])
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

    if not os.path.exists('output'):
        os.mkdir('output')

    model = _get_model(device)
    model.load_state_dict(torch.load("best_metric_model_segmentation2d_dict.
↪pth", map_location=torch.device('cpu')))

    model.eval()

    sub = 0
    with torch.no_grad():
        for test_data in test_loader:
            test_images, test_labels = test_data["img"].to(device),␣
↪test_data["seg"].to(device)
            # define sliding window size and batch size for windows inference
            roi_size = (200, 200)
            sw_batch_size = 4
            val_outputs = sliding_window_inference(test_images, roi_size,␣
↪sw_batch_size, model)
            val_outputs = [post_trans(i) for i in decollate_batch(val_outputs)]
            val_labels = decollate_batch(test_labels)
            # compute metric for current iteration
            dice_metric(y_pred=val_outputs, y=val_labels)

            for val_output, val_label in zip(val_outputs, val_labels):
                labels = val_output.unique().tolist()
                out_shape = list(val_label.shape)
                out_shape[0] = len(labels)
                tmp = np.zeros(tuple(out_shape))
                for i in labels:
                    tmp[int(i), :, :] = val_output.cpu() == int(i)

                plt.imsave(os.path.join('output', str(sub)+'_unet_segmentation.
↪png'), np.moveaxis(tmp[1:,:,:], 0, -1))

                tmp = np.zeros(tuple(out_shape))
                for i in labels:
                    tmp[int(i), :, :] = val_label.cpu() == int(i)

                plt.imsave(os.path.join('output',  str(sub)+'_segmentation.
↪png'), np.moveaxis(tmp[1:,:,:], 0, -1))
                sub+=1

        # aggregate the final mean dice result
        print("evaluation metric:", dice_metric.aggregate().item())

        # reset the status
        dice_metric.reset()
```

1.4 Code explanation

Next we are going to detail the changes done throughout the code to improve the performance of the model.

Enhanced Data Augmentation

First, additional data augmentation techniques are introduced to make the model more robust to variations in the input data. For instance, random flipping along spatial axes and affine transformations (including random rotations, translations, and scaling) help the model generalize better to unseen data. Non-spatial augmentations such as adding random Gaussian noise and intensity shifts further enhance diversity in the input data. The first set of changes are applied to the images and the segmentation, while the non-spatial are applied only on the images. These augmentations are especially beneficial in medical imaging, where variations in orientation, intensity, and positioning can significantly affect model performance.

Updated Loss Function

The original loss function relied solely on Dice Loss, which is well-suited for segmentation tasks but can struggle when handling class imbalances. In the improved version, a hybrid loss function is implemented by combining Dice Loss with Cross-Entropy Loss. This combination leverages Dice Loss's ability to measure overlap and Cross-Entropy Loss's focus on classification accuracy. Weighted contributions from each loss term (e.g., 80% Dice Loss and 20% Cross-Entropy Loss) allow fine-tuning the optimization to balance segmentation performance and class predictions effectively.

Refined Model Architecture

The improved model architecture enhances its capacity and regularization. The number of features in each layer is increased (e.g., from 16–128 in the original to 32–512 in the improved), which allows the network to learn more complex features. Instance normalization is used instead of batch normalization, which is better suited for medical images due to typically smaller batch sizes. Additionally, dropout is introduced for regularization to reduce overfitting, which after trying different values is kept at 0.4.

Improved Optimizer

The optimizer is updated from Stochastic Gradient Descent (SGD) to Adam, which generally converges faster and requires less manual tuning of learning rates. This change improves training efficiency and stability, especially for segmentation tasks that often involve high-dimensional optimization.

Training Pipeline Enhancements

The training loop code retains all essential components while including early stopping. We have added a patience parameter of 20, so if the loss does not improve in 20 epochs the training stops. This allows reducing training time if the condition holds. Finally after trying different values, the number of epochs is set to 700.

In summary, the enhanced code builds upon the original by introducing robust data augmentation, improving model architecture, and refining loss functions and optimizers. These updates collectively aim to improve segmentation accuracy, generalization, and overall robustness of the model, allowing us to improve the evaluation metric from 0.8302255868911743 to 0.935719907283783 on the validation set.

1.5 Running the training code

The following code block will run the training code and keep track of it using tensorboard

```
[ ]: # Fire up training
train('./training_data_exercise/training_data/', './training_data_exercise/
 ↪validation_data/')
```

```
[ ]: # Visualize using tensorboard
%load_ext tensorboard
# Fire up tensorboard
%tensorboard --logdir log
```

1.6 Testing the model

Once you want to test the model (and the test data is released) you can run the code below. Just note that, you need to have the saved model configuration (best_metric_model_segmentation2d_dict.pth) uploaded or saved in to google colab. If you need to upload it, you can use the upload function in the beginning of this notebook. Also you need to execute the code block with the U-Net code because this imports the necessary files, along with the testing function. The test code will also plot the segmentation in an output folder (./output). Remember to download these if you want to save them. Below I'm passing the validation data folder to the test function just to demonstrate how it works. Once the test data is released you should upload it to google colab and then change the path that is passed to the function.

```
[7]: test('training_data_exercise/validation_data')
```

```
MONAI version: 1.4.0
Numpy version: 1.26.4
Pytorch version: 2.5.1+cu124
MONAI flags: HAS_EXT = False, USE_COMPILED = False, USE_META_DICT = False
MONAI rev id: 46a5272196a6c2590ca2589029eed8e4d56ff008
MONAI __file__: /zhome/2b/8/212341/medical-image-analysis-
dtu/w10/.venv/lib64/python3.9/site-packages/monai/__init__.py

Optional dependencies:
Pytorch Ignite version: 0.4.11
ITK version: 5.4.0
Nibabel version: 5.3.2
scikit-image version: 0.22.0
scipy version: 1.13.1
Pillow version: 11.0.0
Tensorboard version: 2.18.0
gdown version: 5.2.0
TorchVision version: 0.20.1+cu124
tqdm version: 4.67.1
lmdb version: 1.5.1
psutil version: 6.1.0
pandas version: 2.2.3
einops version: 0.8.0
transformers version: 4.40.2
mlflow version: 2.18.0
pynrrd version: 1.1.1
clearml version: 1.16.5

For details about installing the optional dependencies, please visit:
    https://docs.monai.io/en/latest/installation.html#installing-the-
recommended-dependencies

BasicUNet features: (16, 32, 32, 64, 128, 16).
```

```
You are using `torch.load` with `weights_only=False` (the current default
value), which uses the default pickle module implicitly. It is possible to
construct malicious pickle data which will execute arbitrary code during
unpickling (See
https://github.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-models for
more details). In a future release, the default value for `weights_only` will be
flipped to `True`. This limits the functions that could be executed during
unpickling. Arbitrary objects will no longer be allowed to be loaded via this
mode unless they are explicitly allowlisted by the user via
`torch.serialization.add_safe_globals`. We recommend you start setting
`weights_only=True` for any use case where you don't have full control of the
loaded file. Please open an issue on GitHub for any issues related to this
experimental feature.
```

```
evaluation metric: 0.8302255868911743
```

[ ]:
```python
# Final
test('training_data_exercise/validation_data')
```

```
MONAI version: 1.4.0
Numpy version: 1.26.4
Pytorch version: 2.5.1+cu124
MONAI flags: HAS_EXT = False, USE_COMPILED = False, USE_META_DICT = False
MONAI rev id: 46a5272196a6c2590ca2589029eed8e4d56ff008
MONAI __file__: /zhome/2b/8/212341/medical-image-analysis-
dtu/w10/.venv/lib64/python3.9/site-packages/monai/__init__.py

Optional dependencies:
Pytorch Ignite version: 0.4.11
ITK version: 5.4.0
Nibabel version: 5.3.2
scikit-image version: 0.22.0
scipy version: 1.13.1
Pillow version: 11.0.0
Tensorboard version: 2.18.0
gdown version: 5.2.0
TorchVision version: 0.20.1+cu124
tqdm version: 4.67.1
lmdb version: 1.5.1
psutil version: 6.1.0
pandas version: 2.2.3
einops version: 0.8.0
transformers version: 4.40.2
mlflow version: 2.18.0
```

```
pynrrd version: 1.1.1
clearml version: 1.16.5

For details about installing the optional dependencies, please visit:
    https://docs.monai.io/en/latest/installation.html#installing-the-
recommended-dependencies

BasicUNet features: (32, 64, 128, 256, 512, 32).
You are using `torch.load` with `weights_only=False` (the current default
value), which uses the default pickle module implicitly. It is possible to
construct malicious pickle data which will execute arbitrary code during
unpickling (See
https://github.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-models for
more details). In a future release, the default value for `weights_only` will be
flipped to `True`. This limits the functions that could be executed during
unpickling. Arbitrary objects will no longer be allowed to be loaded via this
mode unless they are explicitly allowlisted by the user via
`torch.serialization.add_safe_globals`. We recommend you start setting
`weights_only=True` for any use case where you don't have full control of the
loaded file. Please open an issue on GitHub for any issues related to this
experimental feature.
evaluation metric: 0.935719907283783
```

[4]:
```python
# Final with test set
test('test_data')
```

```
MONAI version: 1.4.0
Numpy version: 1.26.4
Pytorch version: 2.5.1
MONAI flags: HAS_EXT = False, USE_COMPILED = False, USE_META_DICT = False
MONAI rev id: 46a5272196a6c2590ca2589029eed8e4d56ff008
MONAI __file__:
/Users/<username>/Documents/Master/2A/MIA/lab/w10/.env/lib/python3.12/site-
packages/monai/__init__.py

Optional dependencies:
Pytorch Ignite version: NOT INSTALLED or UNKNOWN VERSION.
ITK version: NOT INSTALLED or UNKNOWN VERSION.
Nibabel version: NOT INSTALLED or UNKNOWN VERSION.
scikit-image version: NOT INSTALLED or UNKNOWN VERSION.
scipy version: NOT INSTALLED or UNKNOWN VERSION.
Pillow version: 11.0.0
Tensorboard version: 2.18.0
gdown version: NOT INSTALLED or UNKNOWN VERSION.
TorchVision version: NOT INSTALLED or UNKNOWN VERSION.
tqdm version: NOT INSTALLED or UNKNOWN VERSION.
lmdb version: NOT INSTALLED or UNKNOWN VERSION.
```

```
psutil version: 6.1.0
pandas version: NOT INSTALLED or UNKNOWN VERSION.
einops version: NOT INSTALLED or UNKNOWN VERSION.
transformers version: NOT INSTALLED or UNKNOWN VERSION.
mlflow version: NOT INSTALLED or UNKNOWN VERSION.
pynrrd version: NOT INSTALLED or UNKNOWN VERSION.
clearml version: NOT INSTALLED or UNKNOWN VERSION.

For details about installing the optional dependencies, please visit:
    https://docs.monai.io/en/latest/installation.html#installing-the-
recommended-dependencies

BasicUNet features: (32, 64, 128, 256, 512, 32).
```

```
/var/folders/v6/q9xwr1gn2dvdngsx947dpj280000gn/T/ipykernel_4570/2209174152.py:29
3: FutureWarning: You are using `torch.load` with `weights_only=False` (the
current default value), which uses the default pickle module implicitly. It is
possible to construct malicious pickle data which will execute arbitrary code
during unpickling (See
https://github.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-models for
more details). In a future release, the default value for `weights_only` will be
flipped to `True`. This limits the functions that could be executed during
unpickling. Arbitrary objects will no longer be allowed to be loaded via this
mode unless they are explicitly allowlisted by the user via
`torch.serialization.add_safe_globals`. We recommend you start setting
`weights_only=True` for any use case where you don't have full control of the
loaded file. Please open an issue on GitHub for any issues related to this
experimental feature.
  model.load_state_dict(torch.load("best_metric_model_segmentation2d_dict.pth",
map_location=torch.device('cpu')))
```

```
evaluation metric: 0.8888344168663025
```

1.6.1 Execution details

Above we have the first execution of the code without any changes. Then we can see the best model on the validation set, as mentioned before improving the performance from around 0.830 to 0.936. Then finally, we run the model on the test data achieving an performance of 0.88. Given this value we can state there is no overfitting, since the performance of the model only drops by 0.05 from validation to test set.

1.7 Conclusions