endo_SwinUNETR_test_mk1

May 13, 2025

[1]:
```
import os
import numpy as np
from numpy.lib.stride_tricks import as_strided
import time
import matplotlib.pyplot as plt
from scipy.spatial.distance import directed_hausdorff

import torch
from torch.utils.data import DataLoader
from torch.utils.data import Dataset
from torch.utils.data import DataLoader, random_split
from torch.optim.lr_scheduler import StepLR
import torch.nn as nn
import torch.nn.functional as F
import torchvision.models as models

from pytorch_lightning import LightningDataModule
from pytorch_lightning import LightningModule
from pytorch_lightning import Trainer
from pytorch_lightning.callbacks import LearningRateMonitor, ModelCheckpoint
from pytorch_lightning.callbacks import EarlyStopping
from pytorch_lightning.loggers import TensorBoardLogger

from sklearn.model_selection import train_test_split

from monai.networks.nets import SwinUNETR
from monai.losses import DiceCELoss
from monai.metrics import DiceMetric, MeanIoU, HausdorffDistanceMetric,␣
 ↪ConfusionMatrixMetric
from monai.transforms import (
    AsDiscreted,
    Compose,
    Resized,
    EnsureChannelFirstd,
    LoadImaged,
    ScaleIntensityd,
    ToTensord,
```

```
    RandFlipd,
    RandZoomd,
    ToTensord,
    AsDiscreted,
    CenterSpatialCropd
)
```

```
C:\Users\dsumm\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.11_qbz5n
2kfra8p0\LocalCache\local-packages\Python311\site-
packages\ignite\handlers\checkpoint.py:17: DeprecationWarning: `TorchScript`
support for functional optimizers is deprecated and will be removed in a future
PyTorch release. Consider using the `torch.compile` optimizer instead.
  from torch.distributed.optim import ZeroRedundancyOptimizer
```

[2]:
```
# Custom dataset class for pytorch compatibility
# https://pytorch.org/tutorials/beginner/data_loading_tutorial.html
class EndoVis2017Dataset(Dataset):
    def __init__(self, label_subdir=None, test=False):
        self.data = []

        if label_subdir is None:
            raise ValueError("You must specify a `label_subdir` for ground␣
 ↪truth masks (e.g., 'instrument_seg_composite').")

        self.root_dir = "C:/Users/dsumm/OneDrive/Documents/UMD ENPM Robotics␣
 ↪Files/BIOE658B (Intro to Medical Image Analysis)/Project/dataset/test/"
        self.label_subdir = label_subdir

        # Recursively walk through directory to find left frame image paths and␣
 ↪GT image paths
        for subdir, dirs, files in os.walk(self.root_dir):
            if 'left_frames' in subdir:
                #print("Hit!")
                for file in sorted(files):
                    if file.endswith(('.png', '.jpg', '.jpeg')):
                        img_path = os.path.join(subdir, file)
                        #print(img_path)

                        gt_root = subdir.replace('left_frames', 'ground_truth')
                        mask_path = os.path.join(gt_root, self.label_subdir,␣
 ↪file)

                        if os.path.exists(mask_path):
                            #print("Hit!")
                            self.data.append({"image": img_path, "label":␣
 ↪mask_path})    # Dictionary for MONAI compatability
```

```
        if not test:
            transforms_list = [
                LoadImaged(keys=["image", "label"]),                                    # Loads image data and metadata from file path dictionaries
                EnsureChannelFirstd(keys=["image", "label"]),                           # Adjust or add the channel dimension of input data to ensure channel_first shape

                # Images are of nominal size 1280x1024 --> resizing for memory efficiency
                CenterSpatialCropd(keys=["image", "label"], roi_size=(1024, 1280)),     # Cropping background padding from images
                Resized(keys=["image", "label"], spatial_size=(256, 320)),              # Imported images are of various sizes: standardize to 320,256

                # Apply data augmentation techniques
                RandFlipd(keys=["image", "label"], prob=0.3, spatial_axis=1),           # Horizontal axis flip imposed w/ 30% prob
                #RandRotate90d(keys=["image", "label"], prob=0.3, max_k=3),             # Random 90° rotation imposed w/ 30% prob
                RandZoomd(keys=["image", "label"], prob=0.3, min_zoom=0.75, max_zoom=1.25), # Zoom range (+/-25%) imposed w/ 30% prob
                #RandAdjustContrastd(keys=["image"], prob=0.3, gamma=(0.75, 1.25)),     # Contrast variation (+/-25%) imposed w/ 30% prob

                ScaleIntensityd(keys=["image"]),                                        # Scale the intensity of input image to the value range 0-1
                ToTensord(keys=["image", "label"]),                                     # Ensure data is of tensor type for pytorch usage
            ]
        else:
            transforms_list = [
                LoadImaged(keys=["image", "label"]),                                    # Loads image data and metadata from file path dictionaries
                EnsureChannelFirstd(keys=["image", "label"]),                           # Adjust or add the channel dimension of input data to ensure channel_first shape

                # Images are of nominal size 1280x1024 --> resizing for memory efficiency
                CenterSpatialCropd(keys=["image", "label"], roi_size=(1024, 1280)),     # Cropping background padding from images
                Resized(keys=["image", "label"], spatial_size=(256, 320)),              # Imported images are of various sizes: standardize to 320,256
```

```
                ScaleIntensityd(keys=["image"]),                            #␣
↪Scale the intensity of input image to the value range 0-1
                ToTensord(keys=["image", "label"]),                         #␣
↪Ensure data is of tensor type for pytorch usage
            ]
        # Additional conditional transforms based on label_subdir
        if label_subdir == "binary_composite":
            transforms_list.append(AsDiscreted(keys=["label"], threshold=0.5)) ␣
↪         # Binary threshold for binary seg
        elif label_subdir == "part_seg_composite":
            transforms_list.append(AsDiscreted(keys=["label"], to_onehot=5))   ␣
↪         # 5 individual class labels for instrument independent part seg
        elif label_subdir == "TypeSegmentation":
            transforms_list.append(AsDiscreted(keys=["label"], to_onehot=8))   ␣
↪          # 8 individual class labels for part independent instrument seg
        elif label_subdir == "instrument_part_seg_composite":
            transforms_list.append(AsDiscreted(keys=["label"], to_onehot=21))  ␣
↪          # 26 individual class labels for instrument & part seg

        # Imposing MONAI transforms
        # https://docs.monai.io/en/stable/transforms.html
        self.transform = Compose(transforms_list)

    def __len__(self):
        # Returns number of imported samples
        length = len(self.data)
        return length

    def __getitem__(self, idx):
        # Return transformed sample from the dataset as dictated by the index
        sample = self.data[idx]
        return self.transform(sample)
```

[3]:
```
class MONAIDataLoader(LightningDataModule):
    def __init__(self, dataset=None, batch_size: int = None, img_size: int =␣
↪None, dimensions:int = None):
        super().__init__()
        if dataset is None:
            raise ValueError("No dataset given!")
        self.dataset = dataset
        self.test_dataset = dataset
        self.batch_size = batch_size
        self.pin_memory = True

        self.train, self.val = random_split(self.dataset, [
            int(len(self.dataset) * 0.8),
```

```
            len(self.dataset) - int(len(self.dataset) * 0.8)
        ])

        print(f"Train dataset size: {len(self.train)}")
        print(f"Validation dataset size: {len(self.val)}")
        print(f"Test dataset size: {len(self.test_dataset)}")

    def setup(self, stage=None):
        # required by PyTorch Lightning
        pass

    def train_dataloader(self):
        return DataLoader(self.train, batch_size=self.batch_size,␣
↪pin_memory=self.pin_memory)

    def val_dataloader(self):
        return DataLoader(self.val, batch_size=self.batch_size, pin_memory=self.
↪pin_memory)

    def test_dataloader(self):
        return DataLoader(self.test_dataset, batch_size=self.batch_size,␣
↪pin_memory=self.pin_memory)
```

[4]:

```
class SwinUNETR_Train(LightningModule):
    def __init__(self, img_size=(1, 3, 256, 320), batch_size=1, lr=0.001,␣
↪num_classes=1):
        super().__init__()

        self.save_hyperparameters()
        self.num_classes = num_classes
        print("num_classes", self.num_classes, num_classes, self.hparams.
↪num_classes)
        self.example_input_array = [torch.zeros(self.hparams.img_size)]

        self.test_step_outputs = []  # Initialize an empty list to store outputs

        self.dice_metric = DiceMetric(include_background=True,␣
↪reduction="mean", ignore_empty=True)
        self.iou_metric = MeanIoU(include_background=True, reduction="mean",␣
↪ignore_empty=True)
        self.hausdorff_metric = HausdorffDistanceMetric(
                                        include_background=True,
                                        distance_metric="euclidean",
                                        percentile=95,
                                        directed=False,
                                        reduction="mean"
                                    )
```

```
        self.confusion_metric = ConfusionMatrixMetric(
            metric_name=["precision", "recall", "f1 score"],
            include_background=False,
            compute_sample=False,
            reduction="mean"
        )

        # Metric tracking
        self.dice_scores = []
        self.iou_scores = []

        # Define SwinUNETR model from MONAI
        self.model = SwinUNETR(
            img_size=(256,320),
            in_channels=3,
            out_channels=self.num_classes,
            feature_size=48,                        # common starting point; can increase to 96/128
            drop_rate=0.1,                          # 10% dropout probability
            use_checkpoint=True,                     # Enable gradient checkpointing to save memory
            spatial_dims = 2,
        )

        # Using combined DICE and CE loss as loss function
        # Conditional loss function based on the number of classes
        if num_classes == 1:
            self.DICE_CE_Loss = DiceCELoss(
                include_background=False,  # Exclude background class
                sigmoid=True,  # Use softmax for multiclass segmentation
                softmax=False,  # Apply softmax for multiclass
                lambda_dice=1.0,  # Adjust the weight for Dice loss
                lambda_ce=1.0,  # Adjust the weight for Cross-Entropy loss
                reduction='mean'  # Use mean reduction
            )
        else:
            self.DICE_CE_Loss = DiceCELoss(
                include_background=False,  # Exclude background class
                sigmoid=False,  # Use softmax for multiclass segmentation
                softmax=True,  # Apply softmax for multiclass
                lambda_dice=1.0,  # Adjust the weight for Dice loss
                lambda_ce=1.0,  # Adjust the weight for Cross-Entropy loss
                reduction='mean'  # Use mean reduction
            )

        # Tracking losses for matplotlib
        self.train_losses = []
```

```python
        self.val_losses = []

        # For storing images for the last epoch
        self.last_image = []
        self.last_pred = []
        self.last_mask = []
        self.logged_epochs = []

    # Passes model inputs through U-net to get output predictions
    def forward(self, inputs):
        outputs = self.model(inputs)
        return outputs

    def test_step(self, batch, batch_idx):
        # Prepare input and ground truth
        inputs, gt_input = self._prepare_batch(batch)
        outputs = self.forward(inputs)

        if self.hparams.num_classes == 1:
            # Binary segmentation
            probs = torch.sigmoid(outputs)
            preds = (probs > 0.5).float()
            gt_input = (gt_input > 0.5).float()

        else:
            probs = torch.softmax(outputs, dim=1)
            preds = torch.nn.functional.one_hot(torch.argmax(probs, dim=1),␣
↪num_classes=self.num_classes)
            preds = preds.permute(0, 3, 1, 2).float()  # Shape: [B, C, H, W]

        # MONAI metrics
        self.dice_metric(y_pred=preds, y=gt_input)
        self.iou_metric(y_pred=preds, y=gt_input)

        # Hausdorff: safe only per image if non-empty
        for i in range(preds.shape[0]):
            pred_i = preds[i]
            gt_i = gt_input[i]
            if torch.any(pred_i) and torch.any(gt_i):  # Check both non-empty
                self.hausdorff_metric(y_pred=pred_i.unsqueeze(0), y=gt_i.
↪unsqueeze(0))
            else:
                print(f"[Info] Skipping HD metric for empty prediction or GT in␣
↪batch index {i}")
        #self.hausdorff_metric(y_pred=preds, y=gt_input)
        self.confusion_metric(y_pred=preds, y=gt_input)
```

```
        # Extract Dice, IoU, Hausdorff from MONAI
        # Aggregate & safely handle NaNs
        dice = torch.nan_to_num(self.dice_metric.aggregate(), nan=0.0).item()
        iou = torch.nan_to_num(self.iou_metric.aggregate(), nan=0.0).item()
        hausdorff = torch.nan_to_num(self.hausdorff_metric.aggregate(), nan=0.
↪0).item()
        #hausdorff = self.hausdorff_metric.aggregate().item()
        #hausdorff = float('nan') if torch.isnan(torch.tensor(hausdorff)) else␣
↪hausdorff
        #hausdorff = torch.nan_to_num(hausdorff, nan=0.0)

        self.dice_metric.reset()
        self.iou_metric.reset()
        self.hausdorff_metric.reset()

        # Extract precision, recall, f1 score
        confusion_metrics = self.confusion_metric.aggregate()
        precision, recall, f1 = [m.item() for m in confusion_metrics]
        self.confusion_metric.reset()

        # Log metrics
        self.log("test_dice", dice, prog_bar=True)
        self.log("test_iou", iou, prog_bar=True)
        self.log("test_hausdorff", hausdorff, prog_bar=True)
        self.log("test_precision", precision, prog_bar=True)
        self.log("test_recall", recall, prog_bar=True)
        self.log("test_f1", f1, prog_bar=True)

        # Return for aggregation
        out = {
            "test_dice": torch.tensor(dice),
            "test_iou": torch.tensor(iou),
            "test_precision": torch.tensor(precision),
            "test_recall": torch.tensor(recall),
            "test_f1": torch.tensor(f1),
            "test_hausdorff": torch.tensor(hausdorff)
        }

        self.test_step_outputs.append(out)
        return out

    def on_test_epoch_end(self):
        # Aggregate the results across all batches in the epoch
        avg_dice = torch.stack([x["test_dice"] for x in self.
↪test_step_outputs]).mean()
        avg_iou = torch.stack([x["test_iou"] for x in self.test_step_outputs]).
↪mean()
```

```
        avg_hausdorff = torch.stack([x["test_hausdorff"] for x in self.
↪test_step_outputs]).mean()
        avg_precision = torch.stack([x["test_precision"] for x in self.
↪test_step_outputs]).mean()
        avg_recall = torch.stack([x["test_recall"] for x in self.
↪test_step_outputs]).mean()
        avg_f1 = torch.stack([x["test_f1"] for x in self.test_step_outputs]).
↪mean()

        print(f"\n  Test Metrics:"
            f"\n    Dice        : {avg_dice.item():.4f}"
            f"\n    IoU         : {avg_iou.item():.4f}"
            f"\n    Hausdorff   : {avg_hausdorff.item():.4f}"
            f"\n    Precision   : {avg_precision.item():.4f}"
            f"\n    Recall      : {avg_recall.item():.4f}"
            f"\n    F1 Score    : {avg_f1.item():.4f}")

        # Clear for next epoch
        self.test_step_outputs.clear()

    def training_step(self, batch, batch_idx):
        # Gets labels for input and corresponding ground truth
        inputs, gt_input = self._prepare_batch(batch)

        # Call forward pass
        outputs = self.forward(inputs)

        # Compute DICE & CE loss based on current params
        loss = self.DICE_CE_Loss(outputs, gt_input)

        # Log DICE loss with PyTorch Lightning logger
        self.log(f"Train_Dice_CE_loss", loss, on_epoch=True, prog_bar=True)

        # Append train loss at the end of each epoch
        if batch_idx == len(batch) - 1:
            self.train_losses.append(loss.item())

        return loss

    def validation_step(self, batch, batch_idx):

        # Gets labels for input and corresponding ground truth
        inputs, gt_input = self._prepare_batch(batch)
        outputs = self.forward(inputs)
        loss = self.DICE_CE_Loss(outputs, gt_input)
        self.log("val_loss", loss, on_step=False, on_epoch=True, prog_bar=True)
```

```python
        if self.hparams.num_classes == 1:
            probs = torch.sigmoid(outputs)
            preds = (probs > 0.5).float()
            # Ensure ground truth is binary (i.e., 0 or 1)
            gt_input = (gt_input > 0.5).float()  # Threshold the ground truth if needed

            intersection = (preds * gt_input).sum()
            union = preds.sum() + gt_input.sum()
            bin_dice_score = 2.0 * intersection / (union + 1e-8)  # Avoid division by zero
            # IoU score calculation for binary segmentation
            bin_iou_score = intersection / (union - intersection + 1e-8)  # Avoid division by zero

            self.log("val_dice", bin_dice_score, on_step=False, on_epoch=True, prog_bar=True)
            self.log("val_iou", bin_iou_score, on_step=False, on_epoch=True, prog_bar=True)

        else:
            probs = torch.softmax(outputs, dim=1)
            preds = torch.nn.functional.one_hot(torch.argmax(probs, dim=1), num_classes=self.num_classes)
            preds = preds.permute(0, 3, 1, 2).float()  # Shape: [B, C, H, W]

            self.dice_metric(y_pred=preds, y=gt_input)
            self.iou_metric(y_pred=preds, y=gt_input)

        if self.trainer.sanity_checking:
            return  # skip logging during sanity check

        # Append validation loss at the end of each epoch
        if batch_idx == len(batch) - 1:
            self.val_losses.append(loss.item())

            # For binary segmentation: apply sigmoid and threshold
            if self.hparams.num_classes == 1:
                outputs = torch.sigmoid(outputs)
                outputs = (outputs > 0.5).float()  # Convert probabilities to binary mask
                self.dice_scores.append(bin_dice_score)
                self.iou_scores.append(bin_iou_score)

            # For multiclass segmentation: apply softmax
            else:
```

```
                outputs = torch.softmax(outputs, dim=1)  # Apply softmax for␣
↪multi-class outputs
                dice = self.dice_metric.aggregate()[0].item()
                #print("Dice", dice)
                iou = self.iou_metric.aggregate()[0].item()
                #print("IOU", iou)
                self.dice_metric.reset()
                self.iou_metric.reset()
                self.dice_scores.append(dice)
                self.iou_scores.append(iou)
                self.log("val_dice", dice, on_step=False, on_epoch=True,␣
↪prog_bar=True)
                self.log("val_iou", iou, on_step=False, on_epoch=True,␣
↪prog_bar=True)

            # Normalize and convert tensor to 3 channels (RGB) for visualization
            def process(last):
                # Detach from cpu to not interrupt training
                # https://stackoverflow.com/questions/63582590/
↪why-do-we-call-detach-before-calling-numpy-on-a-pytorch-tensor
                last = last[0].detach().cpu()

                # Min max normalization
                # https://www.codecademy.com/article/normalization
                last= (last - last.min()) / (last.max() - last.min() + 1e-8)

                # If grayscale, reshape last image to RGB for display by␣
↪replicating gray value twice
                # https://discuss.pytorch.org/t/convert-grayscale-images-to-rgb/
↪113422
                return last.repeat(3, 1, 1) if last.shape[0] == 1 else last

            current_epoch = self.current_epoch
            total_epochs = self.trainer.max_epochs
            #print("TE", total_epochs)

            if current_epoch == 0 or current_epoch == total_epochs - 1 or␣
↪current_epoch == total_epochs // 2:
                self.last_image.append(process(inputs))
                self.last_pred.append(process(outputs))
                self.last_mask.append(process(gt_input))
                self.logged_epochs.append(current_epoch)
                print(f"Logged image from epoch {current_epoch}")

        return loss
```

```
    #def predict_step(self, batch, batch_idx, dataloader_idx=0):
    #     return self(batch['image'])

    def configure_optimizers(self):
        #set optimizer
        optimizer = torch.optim.AdamW(self.parameters(), lr=self.hparams.lr,␣
↪weight_decay=1e-4)
        scheduler = StepLR(optimizer, step_size=5, gamma=0.5)  # halve LR every␣
↪5 epochs
        return {
            'optimizer': optimizer,
            'lr_scheduler': {
                'scheduler': scheduler,
                'interval': 'epoch',
                'frequency': 1
            }
        }

    def _prepare_batch(self, batch):
        return batch['image'], batch['label']

    # Plot training and val losses when needed
    def plot_losses(self):
        min_len = min(len(self.train_losses), len(self.val_losses))
        epochs = range(1, min_len + 1)

        # Plotting training vs validation loss
        plt.figure(figsize=(10, 6))
        plt.plot(epochs, self.train_losses[:len(epochs)], label="Training␣
↪Loss", color='blue')
        plt.plot(epochs, self.val_losses[:len(epochs)], label="Validation␣
↪Loss", color='orange')
        plt.title("Training vs Validation Loss")
        plt.xlabel("Epochs")
        plt.ylabel("Loss")
        plt.legend()
        plt.show()

    def plot_metrics(self):
        epochs = range(1, len(self.dice_scores) + 1)

        # Convert to CPU floats if necessary
        dice = [d.cpu().item() if torch.is_tensor(d) else d for d in self.
↪dice_scores]
        iou = [i.cpu().item() if torch.is_tensor(i) else i for i in self.
↪iou_scores]
```

```
        plt.figure(figsize=(10, 6))
        plt.plot(epochs, dice, label='Dice Coefficient')
        plt.plot(epochs, iou, label='IoU')
        plt.xlabel("Epochs")
        plt.ylabel("Score")
        plt.title("Validation Metrics Over Time")
        plt.legend()
        plt.show()

    def plot_result_by_epoch(self):
        total_epochs = len(self.last_image)
        print("Total Epochs:", total_epochs)

        if total_epochs < 5:
            print(f"Only {total_epochs} epochs recorded, plotting all.")
            selected_epochs = list(range(total_epochs))
        else:
            print(f"{total_epochs} epochs recorded, bug in code.")

        for epoch_idx in selected_epochs:
            epoch_num = self.logged_epochs[epoch_idx] if hasattr(self,␣
↪"logged_epochs") else epoch_idx
            img = self.last_image[epoch_idx]
            pred = self.last_pred[epoch_idx]
            mask = self.last_mask[epoch_idx]

            fig, ax = plt.subplots(1, 3, figsize=(12, 4))

            ax[0].imshow(np.transpose(img.numpy(), (1, 2, 0)))
            ax[0].set_title(f"Epoch {epoch_num} - Image")
            ax[0].axis("off")

            if self.hparams.num_classes == 1:
                ax[1].imshow(np.transpose(pred.numpy(), (1, 2, 0)))
                ax[1].set_title(f"Epoch {epoch_num} - Prediction")
                ax[1].axis("off")

                ax[2].imshow(np.transpose(mask.numpy(), (1, 2, 0)))
                ax[2].set_title(f"Epoch {epoch_num} - Ground Truth")
                ax[2].axis("off")
            else:
                # Define the colormap and normalization
                num_classes = self.hparams.num_classes
                cmap = plt.get_cmap('viridis', num_classes)
                bounds = np.arange(num_classes + 1) - 0.5
                norm = plt.matplotlib.colors.BoundaryNorm(bounds, cmap.N)
```

```
                # Convert one-hot encoded predictions and masks to␣
↪single-channel class labels
                pred_mask = torch.argmax(pred, dim=0).cpu().numpy()
                true_mask = torch.argmax(mask, dim=0).cpu().numpy()

                # Apply consistent colormap and normalization
                im1 = ax[1].imshow(pred_mask, cmap=cmap, norm=norm)
                ax[1].set_title(f"Epoch {epoch_num} - Prediction")
                ax[1].axis("off")

                im2 = ax[2].imshow(true_mask, cmap=cmap, norm=norm)
                ax[2].set_title(f"Epoch {epoch_num} - Ground Truth")
                ax[2].axis("off")

                im_for_cbar = im1  # just need one mappable

                # Adjust layout to leave space at the bottom
                fig.subplots_adjust(bottom=0.25) # tweak this if labels get cut␣
↪off

                # Add a new axis below the plots for the colorbar
                cbar_ax = fig.add_axes([0.1, 0.1, 0.8, 0.10])  # [left, bottom,␣
↪width, height]
                cbar = fig.colorbar(im_for_cbar, cax=cbar_ax,␣
↪orientation='horizontal', ticks=np.arange(num_classes))

                # Set class labels
                if num_classes == 5:
                    cbar.ax.set_xticklabels(['Background', 'Shaft', 'Wrist',␣
↪'Claspers', 'Probe'])
                elif num_classes == 8:
                    cbar.ax.set_xticklabels(['Background', 'Bipolar Forceps',␣
↪'Prograsp Forceps', 'Large Needle Driver',
                                             'Vessel Sealer', 'Grasping␣
↪Retractor', 'Monopolar Curved Scissors', 'Other'])

                    plt.setp(cbar.ax.get_xticklabels(), rotation=30,␣
↪ha="right", rotation_mode="anchor")
                elif num_classes == 21:
                    cbar.ax.set_xticklabels([
                        "Background",
                        "Bipolar Forceps Shaft", "Bipolar Forceps Wrist",␣
↪"Bipolar Forceps Claspers",
                        "Prograsp Forceps Shaft", "Prograsp Forceps Wrist",␣
↪"Prograsp Forceps Claspers",
```

```
                        "Large Needle Driver Shaft", "Large Needle Driver␣
↪Wrist", "Large Needle Driver Claspers",
                        "Vessel Sealer Shaft", "Vessel Sealer Wrist", "Vessel␣
↪Sealer Claspers",
                        "Grasping Retractor Shaft", "Grasping Retractor Wrist",␣
↪"Grasping Retractor Claspers",
                        "Monopolar Curved Scissors Shaft", "Monopolar Curved␣
↪Scissors Wrist", "Monopolar Curved Scissors Claspers",
                        "Other Probe","Other Probe"
                    ])
                    plt.setp(cbar.ax.get_xticklabels(), rotation=45,␣
↪ha="right", rotation_mode="anchor")

                cbar.set_label('Class ID')

        plt.show()
```

[5]:

```
binary_SwinUNETR_model = SwinUNETR_Train(num_classes=1)
binary_SwinUNETR_model.load_state_dict(torch.load('C:/Users/dsumm/OneDrive/
 ↪Documents/UMD ENPM Robotics Files/BIOE658B (Intro to Medical Image Analysis)/
 ↪Project/results/SwinUNETR/SwinUNETRmodels/binary_SwinUNETR_model.pth'))

binary_endo_images = EndoVis2017Dataset(label_subdir='binarySegmentation',␣
 ↪test=True)
binary_endo_data = MONAIDataLoader(dataset=binary_endo_images, batch_size=20)

trainer = Trainer(accelerator="gpu", devices=1)
trainer.test(model=binary_SwinUNETR_model, datamodule=binary_endo_data)
```

```
num_classes 1 1 1
```

```
monai.networks.nets.swin_unetr SwinUNETR.__init__:img_size: Argument `img_size`
has been deprecated since version 1.3. It will be removed in version 1.5. The
img_size argument is not required anymore and checks on the input size are run
during forward().
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.
```

```
Using default `ModelCheckpoint`. Consider installing `litmodels` package to
enable `LitModelCheckpoint` for automatic upload to the Lightning model
registry.
GPU available: True (cuda), used: True
TPU available: False, using: 0 TPU cores
HPU available: False, using: 0 HPUs
You are using a CUDA device ('NVIDIA GeForce RTX 4070 Laptop GPU') that has
Tensor Cores. To properly utilize them, you should set
`torch.set_float32_matmul_precision('medium' | 'high')` which will trade-off
precision for performance. For more details, read https://pytorch.org/docs/stabl
e/generated/torch.set_float32_matmul_precision.html#torch.set_float32_matmul_pre
cision
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]

Train dataset size: 720
Validation dataset size: 180
Test dataset size: 900

The 'test_dataloader' does not have many workers which may be a bottleneck.
Consider increasing the value of the `num_workers` argument` to `num_workers=31`
in the `DataLoader` to improve performance.

Testing: |          | 0/? [00:00<?, ?it/s]


 Test Metrics:
  Dice       : 0.9156
  IoU        : 0.8632
  Hausdorff  : 24.0247
  Precision  : 0.9038
  Recall     : 0.9528
  F1 Score   : 0.9187
```

Test metric	DataLoader 0
test_dice	0.9156012535095215
test_f1	0.9186905026435852
test_hausdorff	24.024734497070312
test_iou	0.8631661534309387
test_precision	0.9038184285163879
test_recall	0.9528173804283142

[5]:
```
[{'test_dice': 0.9156012535095215,
  'test_iou': 0.8631661534309387,
  'test_hausdorff': 24.024734497070312,
  'test_precision': 0.9038184285163879,
  'test_recall': 0.9528173804283142,
```

```
'test_f1': 0.9186905026435852}]
```

[6]:
```
binary_SwinUNETR_model.eval().cuda()  # <<< This is important
N_BATCHES = 10  # Set number of batches to evaluate
times = []

with torch.no_grad():
    for i, batch in enumerate(binary_endo_data.test_dataloader()):
        if i >= N_BATCHES:
            break
        inputs = batch["image"].cuda()
        start_time = time.time()
        outputs = binary_SwinUNETR_model(inputs)
        torch.cuda.synchronize()  # Ensures accurate timing on GPU
        end_time = time.time()
        times.append(end_time - start_time)

avg_infer_time = np.mean(times) / inputs.shape[0]  # Per image
print(f"Average inference time per image over {N_BATCHES * inputs.shape[0]}␣
↪images: {avg_infer_time:.6f} seconds")
```

```
Average inference time per image over 200 images: 0.015349 seconds
```

[7]:
```
part_seg_SwinUNETR_model = SwinUNETR_Train(num_classes=5)
part_seg_SwinUNETR_model.load_state_dict(torch.load('C:/Users/dsumm/OneDrive/
↪Documents/UMD ENPM Robotics Files/BIOE658B (Intro to Medical Image Analysis)/
↪Project/results/SwinUNETR/SwinUNETRmodels/part_seg_SwinUNETR_model.pth'))

part_seg_endo_images = EndoVis2017Dataset(label_subdir='part_seg_composite',␣
↪test=True)
part_seg_endo_data = MONAIDataLoader(dataset=part_seg_endo_images,␣
↪batch_size=10)

trainer = Trainer(accelerator="gpu", devices=1)
trainer.test(model=part_seg_SwinUNETR_model, datamodule=part_seg_endo_data)
```

```
num_classes 5 5 5
Train dataset size: 720
Validation dataset size: 180
Test dataset size: 900
```

```
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.
Using default `ModelCheckpoint`. Consider installing `litmodels` package to
enable `LitModelCheckpoint` for automatic upload to the Lightning model
registry.
GPU available: True (cuda), used: True
TPU available: False, using: 0 TPU cores
HPU available: False, using: 0 HPUs
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]

Testing: |          | 0/? [00:00<?, ?it/s]

the ground truth of class 4 is all 0, this may result in nan/inf distance.
the prediction of class 4 is all 0, this may result in nan/inf distance.
the ground truth of class 3 is all 0, this may result in nan/inf distance.
the prediction of class 2 is all 0, this may result in nan/inf distance.
the ground truth of class 2 is all 0, this may result in nan/inf distance.


 Test Metrics:
  Dice       : 0.7915
  IoU        : 0.7053
  Hausdorff  : 41.3787
  Precision  : 0.7683
  Recall     : 0.8137
  F1 Score   : 0.7826
```

Test metric	DataLoader 0
test_dice	0.7915090322494507
test_f1	0.7826336026191711
test_hausdorff	41.378692626953125
test_iou	0.7053414583206177
test_precision	0.7682565450668335
test_recall	0.8136885166168213

```
[7]: [{'test_dice': 0.7915090322494507,
       'test_iou': 0.7053414583206177,
       'test_hausdorff': 41.378692626953125,
       'test_precision': 0.7682565450668335,
       'test_recall': 0.8136885166168213,
       'test_f1': 0.7826336026191711}]
```

[8]:
```
part_seg_SwinUNETR_model.eval().cuda()  # <<< This is important
N_BATCHES = 20  # Set number of batches to evaluate
times = []

with torch.no_grad():
    for i, batch in enumerate(part_seg_endo_data.test_dataloader()):
        if i >= N_BATCHES:
            break
        inputs = batch["image"].cuda()
        start_time = time.time()
        outputs = part_seg_SwinUNETR_model(inputs)
        torch.cuda.synchronize()  # Ensures accurate timing on GPU
        end_time = time.time()
        times.append(end_time - start_time)

avg_infer_time = np.mean(times) / inputs.shape[0]  # Per image
print(f"Average inference time per image over {N_BATCHES * inputs.shape[0]}␣
↪images: {avg_infer_time:.6f} seconds")
```

```
Average inference time per image over 200 images: 0.014753 seconds
```

[9]:
```
instr_seg_SwinUNETR_model = SwinUNETR_Train(num_classes=8)
instr_seg_SwinUNETR_model.load_state_dict(torch.load('C:/Users/dsumm/OneDrive/
↪Documents/UMD ENPM Robotics Files/BIOE658B (Intro to Medical Image Analysis)/
↪Project/results/SwinUNETR/SwinUNETRmodels/instr_seg_SwinUNETR_model.pth'))

instr_seg_endo_images = EndoVis2017Dataset(label_subdir='TypeSegmentation',␣
↪test=True)
instr_seg_endo_data = MONAIDataLoader(dataset=instr_seg_endo_images,␣
↪batch_size=5)

trainer = Trainer(accelerator="gpu", devices=1)
trainer.test(model=instr_seg_SwinUNETR_model, datamodule=instr_seg_endo_data)
```

```
num_classes 8 8 8
```

```
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.
```

```
Using default `ModelCheckpoint`. Consider installing `litmodels` package to
enable `LitModelCheckpoint` for automatic upload to the Lightning model
registry.
GPU available: True (cuda), used: True
TPU available: False, using: 0 TPU cores
HPU available: False, using: 0 HPUs
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]

Train dataset size: 720
Validation dataset size: 180
Test dataset size: 900

Testing: |          | 0/? [00:00<?, ?it/s]

the ground truth of class 5 is all 0, this may result in nan/inf distance.
the ground truth of class 6 is all 0, this may result in nan/inf distance.
the ground truth of class 7 is all 0, this may result in nan/inf distance.
the prediction of class 5 is all 0, this may result in nan/inf distance.
the prediction of class 6 is all 0, this may result in nan/inf distance.
the prediction of class 7 is all 0, this may result in nan/inf distance.
the prediction of class 3 is all 0, this may result in nan/inf distance.
the prediction of class 1 is all 0, this may result in nan/inf distance.


 Test Metrics:
  Dice       : 0.5318
  IoU        : 0.4814
  Hausdorff  : 38.7343
  Precision  : 0.6533
  Recall     : 0.6854
  F1 Score   : 0.6613
```

Test metric	DataLoader 0
test_dice	0.5317918062210083
test_f1	0.6613436341285706
test_hausdorff	38.73433303833008
test_iou	0.481431782245636
test_precision	0.6533302068710327
test_recall	0.6854087710380554

[9]:
```
[{'test_dice': 0.5317918062210083,
  'test_iou': 0.481431782245636,
  'test_hausdorff': 38.73433303833008,
  'test_precision': 0.6533302068710327,
  'test_recall': 0.6854087710380554,
  'test_f1': 0.6613436341285706}]
```

[10]:

```
instr_seg_SwinUNETR_model.eval().cuda()  # <<< This is important
N_BATCHES = 40  # Set number of batches to evaluate
times = []

with torch.no_grad():
    for i, batch in enumerate(instr_seg_endo_data.test_dataloader()):
        if i >= N_BATCHES:
            break
        inputs = batch["image"].cuda()
        start_time = time.time()
        outputs = instr_seg_SwinUNETR_model(inputs)
        torch.cuda.synchronize()  # Ensures accurate timing on GPU
        end_time = time.time()
        times.append(end_time - start_time)

avg_infer_time = np.mean(times) / inputs.shape[0]  # Per image
print(f"Average inference time per image over {N_BATCHES * inputs.shape[0]}␣
↪images: {avg_infer_time:.6f} seconds")
```

```
Average inference time per image over 200 images: 0.645149 seconds
```