Optimizing Fetal Brain MRI Segmentation: Exploring UNet Variants and Label-Specific Architectures

Team Members
Aniketh Vijesh (AM.EN.U4AIE22009)
Shrisharanyan Vasu (AM.EN.U4AIE22150)

Abstract

This study explores the application of a UNet-based architecture for segmenting fetal brain regions in MRI scans using the FETA 2024 dataset. The primary objective is to accurately segment different parts of fetal brain development. The study so far has focused on extensive experimentation on different transformations and augmentations to help the model generalize well. The experiments were designed to assess the impact of these modifications on segmentation performance, with the Dice similarity coefficient as the primary evaluation metric. Results indicate that incorporating advanced preprocessing techniques significantly improves segmentation accuracy in the validation set, achieving a Dice score of above 7.2 in specific configurations. Following this further research is being conducted on a new architecture that leverages a single UNet for each label. The intuition behind this architecture is that training for each label might respond more effectively to distinct sets of hyperparameters and transforms.

Dataset Details

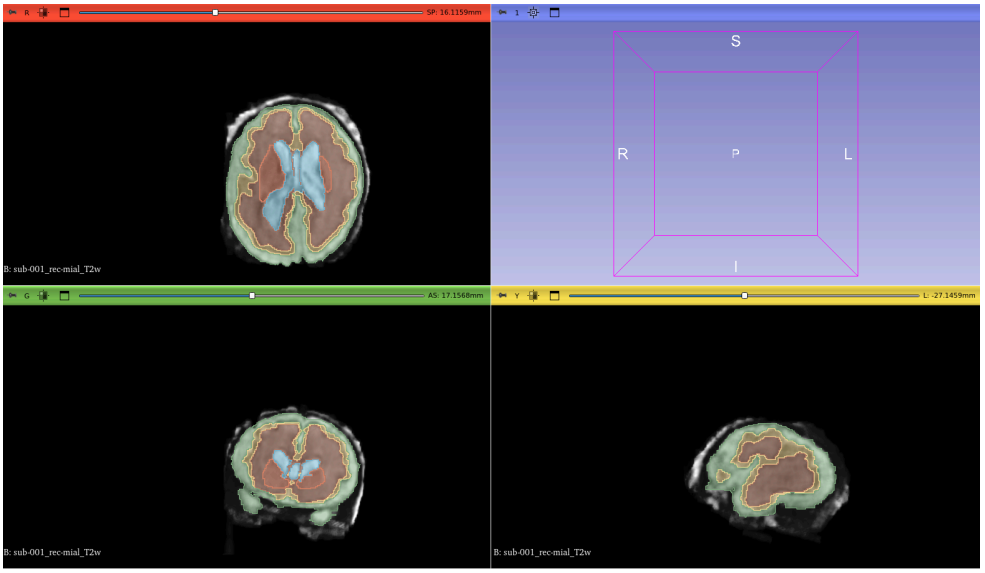
The dataset that we are using is the Fetal Tissue Annotation(FeTA) 2024 dataset. FeTA Dataset: Fetal Tissue Annotations dataset for machine-learning and deep-learning-based image segmentation algorithm development. The dataset was released as part of the FeTA 2024 challenge at the Medical Image Computing and Computer Assisted Intervention (MICCAI) 2024 conference.

The dataset consists of 120 T2-weighted fetal brain reconstructions from two different institutions with a corresponding label map that was manually segmented into 7 different tissues/labels, which are as follows:

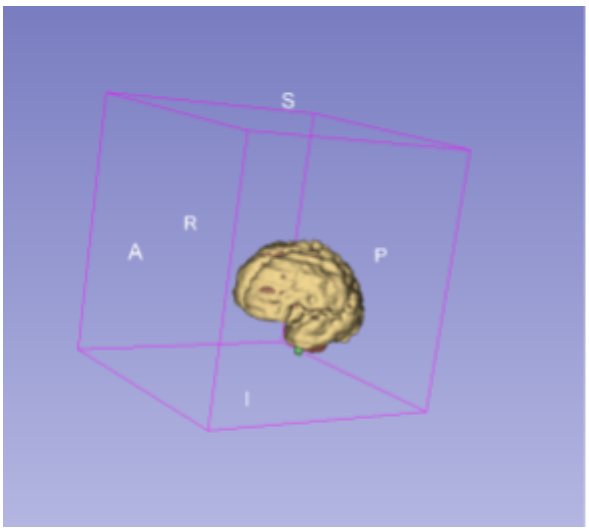
1. External Cerebrospinal Fluid
2. Grey Matter
3. White Matter
4. Ventricles
5. Cerebellum
6. Deep Grey Matter
7. Brainstem

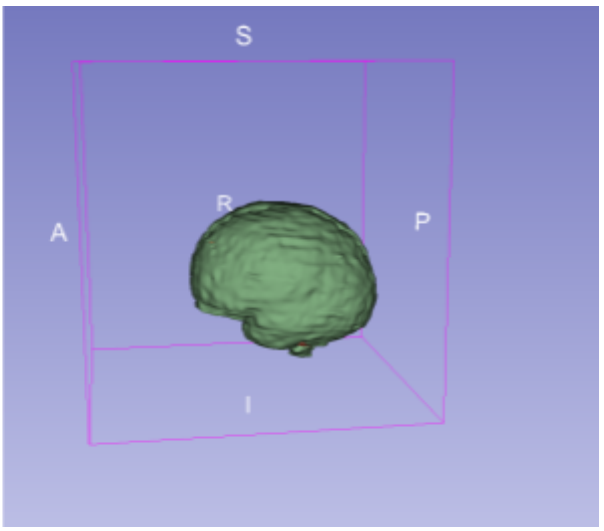
The data also contains a class 0, which is considered to be background and other unwanted features.
Each sample will be 256x256x256 voxels, but there is a variance in the resolution due to the data being collected from multiple different sources.

Some visualisations of the dataset



3D visualisation





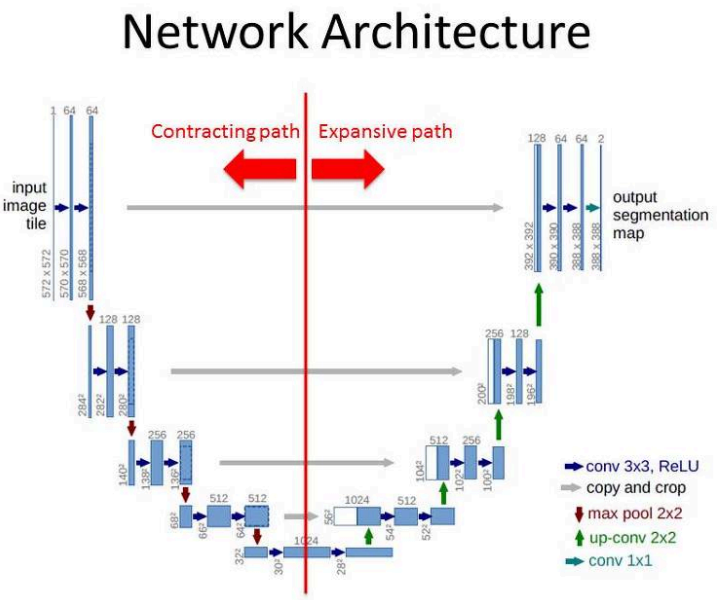
Model Architecture

MONAI's built-in UNet from monai.networks.net.Unet was used for running these experimentations.

UNet

The UNet architecture is structured into two primary sections: the encoder (contracting path) and the decoder (expansive path), connected by skip connections. The encoder comprises a series of convolutional blocks, each consisting of two 3x3 convolutions (stride=1, padding=1) followed by ReLU activations and a 2x2 max pooling layer (stride=2). The pooling layers progressively reduce the spatial dimensions while doubling the number of channels at each level, typically starting from 64 channels and increasing to 128, 256, 512, and 1024 channels at the bottleneck. This allows the encoder to capture increasingly abstract and complex features.

At the bottleneck, the feature maps undergo two 3x3 convolutions, followed by upsampling in the decoder. The decoder mirrors the encoder, with transposed convolutions (stride=2) used for upsampling, which halves the number of channels at each level. Each upsampling step is followed by concatenation with the corresponding encoder feature maps via skip connections and two 3x3 convolutional layers with ReLU activations. The output layer typically consists of a 1x1 convolution to map the final feature maps to the desired number of output channels (e.g., for segmentation labels), followed by a softmax or sigmoid activation depending on whether the task is multi-class or binary segmentation.



Below is a rough implementation [1]

Encoder

```
class Encoder(nn.Module):
    def __init__(self, in_channels):
        super(Encoder, self).__init__()
        self.enc1 = self.conv_block(in_channels, 64)
        self.enc2 = self.conv_block(64, 128)
        self.enc3 = self.conv_block(128, 256)
        self.enc4 = self.conv_block(256, 512)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
        self.feature_maps = []
```

Bottleneck

```
class Bottleneck(nn.Module):
    def __init__(self, in_channels):
        super(Bottleneck, self).__init__()
        self.bottleneck = nn.Sequential(
            nn.Conv2d(in_channels, 1024, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(1024, 1024, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
        )
        self.feature_maps = []

    def forward(self, x):
        self.feature_maps.append(x)
        bottleneck = self.bottleneck(nn.MaxPool2d(2)(x))
        self.feature_maps.append(bottleneck)
        return bottleneck
```

```
class Decoder(nn.Module):
    def __init__(self, out_channels):
        super(Decoder, self).__init__()
        self.upconv4 = self.upconv_block(1024, 512)
        self.upconv3 = self.upconv_block(512, 256)
        self.upconv2 = self.upconv_block(256, 128)
        self.upconv1 = self.upconv_block(128, 64)
        self.final_conv = nn.Conv2d(64, out_channels, kernel_size=1)
        self.feature_maps = []
```

Decoder

The key feature of the UNets are the skip connections that concatenate the feature maps extracted by the encoder to the decoder in order to ensure smoother gradient flow, following the concepts of ResNets.

```
    def forward(self, x, enc1, enc2, enc3, enc4):
        self.feature_maps.append(x)

        up4 = self.upconv4(x)
        up4 = torch.concat([up4, enc4], dim=1)
        dec4 = self.conv_block(1024, 512)(up4)
        self.feature_maps.append(dec4)
```

UNet-per-label

The U-Net-per-label architecture will extend the classic U-Net design by employing a distinct U-Net model for each segmentation label. Each model will retain the standard U-Net structure but will be tailored to optimize for a specific label. The encoder and decoder will maintain the same configuration: 3x3 convolutions with stride=1 and padding=1, ReLU activations, and channel scaling starting from 64 channels in the first layer and doubling at each downsampling step. However, the hyperparameters, such as the number of channels, learning rate, and data augmentation strategies, will be fine-tuned separately for each U-Net, allowing for label-specific customization.

The key architectural difference will lie in the separation of training processes for each label, which will ensure that each network independently learns features pertinent to its respective label. This modularity will allow for experimentation with different loss functions (e.g., Dice loss for one label and focal loss for another) and transformations (e.g., rotation or elastic deformations), offering a flexible and effective approach to addressing challenges in multi-label segmentation tasks.

Code Till Date

Base Definition and initialising the WandB run

```
import torch
import torch.nn.functional as F
import wandb
from monai.networks.nets import UNet
from monai.networks.layers import Norm
from monai.losses import DiceLoss
from monai.metrics import DiceMetric
from monai.transforms import AsDiscrete
from monai.inferers import sliding_window_inference
import numpy as np


gpu_device = "cuda:0" if torch.cuda.is_available else "cpu"
print(gpu_device)
device = torch.device("cuda:0")  # or "cpu"
model = UNet(
    spatial_dims=3,
    in_channels=1,
    out_channels=8,  # 8 classes including background
    channels=(16, 32, 64, 256),
    strides=(2, 2, 2, 2),
    norm=Norm.BATCH,
).to(device)

# Initialize Wandb
wandb.init(project="FETA-model-training")

# Loss and metric configurations
loss_function = DiceLoss(
    smooth_nr=0,
    squared_pred=True,
    to_onehot_y=False,
    softmax=False
)

optimizer = torch.optim.Adam(model.parameters(), 1e-4)

# Post-processing transforms
post_label = AsDiscrete()  #
post_pred = AsDiscrete()
```

Transform definition and WandB logging

```
train_transforms = monai.transforms.Compose(
    [
        monai.transforms.LoadImaged(keys=["image", "label"]),
        monai.transforms.EnsureChannelFirstd(keys=["image", "label"]),
        monai.transforms.ResizeWithPadOrCropd(keys=["image", "label"], spatial_size=(128, 128, 128)),
        monai.transforms.Orientationd(keys=["image", "label"], axcodes="RAS"),
        monai.transforms.AdjustContrastd(keys=["image"], gamma=1.5),
        monai.transforms.NormalizeIntensityd(keys=["image"], nonzero=True, channel_wise=True)
    ]
)

val_transforms = monai.transforms.Compose(
    [
        monai.transforms.LoadImaged(keys=["image", "label"]),
        monai.transforms.EnsureChannelFirstd(keys=["image", "label"]),
        monai.transforms.ResizeWithPadOrCropd(keys=["image", "label"], spatial_size=(128, 128, 128)),
        monai.transforms.Orientationd(keys=["image", "label"], axcodes="RAS"),
        monai.transforms.NormalizeIntensityd(keys=["image"], nonzero=True, channel_wise=True)
    ]
)

def log_transforms_to_wandb(train_transforms, val_transforms):
    train_transform_details = [str(transform) for transform in train_transforms.transforms]
    val_transform_details = [str(transform) for transform in val_transforms.transforms]

    wandb.log({
        "train_transforms": train_transform_details,
        "val_transforms": val_transform_details
    })
log_transforms_to_wandb(train_transforms, val_transforms)
```

```
    train_ds = CacheDataset(data=train_files, transform=train_transforms, cache_rate=1.0, num_workers=2)
    train_loader = DataLoader(train_ds, batch_size=2, shuffle=True, num_workers=2,collate_fn=pad_list_data_collate)

    val_ds = CacheDataset(data=val_files, transform=val_transforms, cache_rate=1.0, num_workers=2)
    val_loader = DataLoader(val_ds, batch_size=1, num_workers=2,collate_fn=pad_list_data_collate)
```

Validation

```python
# Validation Function
def validation(model, val_loader, loss_function, device, epoch):
    model.eval()
    val_loss = 0

    # Initialize classwise dice metrics
    class_dice_metrics = {
        f"class_{i}": DiceMetric(include_background=False, reduction="mean")
        for i in range(8)  # 8 classes
    }

    with torch.no_grad():
        progress_bar = tqdm(val_loader, desc=f"Validation Epoch {epoch + 1}", unit="batch")
        for val_data in progress_bar:
            val_images, val_labels = val_data["image"].to(device), val_data["label"].to(device)

            val_labels = val_labels.squeeze(1)
            val_labels_one_hot = F.one_hot(val_labels.long(), num_classes=8)
            val_labels_one_hot = val_labels_one_hot.permute(0, 4, 1, 2, 3).float()

            outputs = model(val_images)
            loss = loss_function(outputs, val_labels_one_hot)
            val_loss += loss.item()

            # Calculate classwise Dice scores
            for class_idx in range(8):
                class_key = f"class_{class_idx}"
                class_dice_metrics[class_key](
                    y_pred=[outputs[:, class_idx:class_idx+1]],
                    y=[val_labels_one_hot[:, class_idx:class_idx+1]]
                )

            progress_bar.set_postfix(loss=loss.item())

    # Compute average loss and classwise Dice scores
    val_loss = val_loss / len(val_loader)

    # Calculate and log classwise dice scores
    val_dice_scores = {}
    mean_dice = 0
    for class_idx in range(8):
        class_key = f"class_{class_idx}"
        dice_val = class_dice_metrics[class_key].aggregate().item()

        # Ensure dice values are between 0 and 1
        if dice_val > 1 or dice_val < 0:
            dice_val = abs(dice_val % 1)

        val_dice_scores[f"val_dice_{class_key}"] = dice_val
        mean_dice += dice_val

    # Calculate mean dice across all classes
    mean_dice = mean_dice / 8

    # Log all metrics to wandb
    wandb.log({
        "validation_loss": val_loss,
        "validation_dice": mean_dice,
        "epoch": epoch + 1,
        **val_dice_scores  # Unpack all classwise dice scores
    })

    print(f"[Validation] Epoch {epoch+1}: Loss: {val_loss:.4f}, Mean Dice: {mean_dice:.4f}")
    return val_loss, mean_dice
```

Training

```python
# Training Function
def train(model, global_step, train_loader, val_loader, loss_function, optimizer, device, dice_val_best, global_step_best, epoch, early_stopping):
    model.train()
    epoch_loss = 0

    # Initialize classwise dice metrics
    class_dice_metrics = {
        f"class_{i}": DiceMetric(include_background=False, reduction="mean")
        for i in range(8)  # 8 classes
    }

    progress_bar = tqdm(train_loader, desc=f"Training Epoch {epoch + 1}", unit="batch")
    for step, train_data in enumerate(progress_bar):
        images, labels = train_data["image"].to(device), train_data["label"].to(device)

        labels = labels.squeeze(1)
        labels_one_hot = F.one_hot(labels.long(), num_classes=8).permute(0, 4, 1, 2, 3).float()

        optimizer.zero_grad()
        outputs = model(images)
        loss = loss_function(outputs, labels_one_hot)
        epoch_loss += loss.item()

        loss.backward()
        optimizer.step()

        # Update classwise metrics
        for class_idx in range(8):
            class_key = f"class_{class_idx}"
            class_dice_metrics[class_key](
                y_pred=[outputs[:, class_idx:class_idx+1]],
                y=[labels_one_hot[:, class_idx:class_idx+1]]
            )

        if step % 10 == 0:
            progress_bar.set_postfix(loss=loss.item())

    # Calculate and log training metrics
    epoch_loss = epoch_loss / len(train_loader)

    # Calculate classwise dice scores
    train_dice_scores = {}
    mean_train_dice = 0
    for class_idx in range(8):
        class_key = f"class_{class_idx}"
        dice_val = class_dice_metrics[class_key].aggregate().item()

        if dice_val > 1 or dice_val < 0:
            dice_val = abs(dice_val % 1)

        train_dice_scores[f"train_dice_{class_key}"] = dice_val
        mean_train_dice += dice_val

    mean_train_dice = mean_train_dice / 8

    # Log all metrics to wandb
    wandb.log({
        "train_loss": epoch_loss,
        "train_dice":mean_train_dice,
        "epoch": epoch + 1,
        **train_dice_scores  # Unpack all classwise dice scores
    })

    print(f"[Training] Epoch {epoch + 1}: Loss: {epoch_loss:.4f}, Mean Dice: {mean_train_dice:.4f}")
```

```python
# Log all metrics to wandb
wandb.log({
    "train_loss": epoch_loss,
    "train_dice":mean_train_dice,
    "epoch": epoch + 1,
    **train_dice_scores  # Unpack all classwise dice scores
})

print(f"[Training] Epoch {epoch + 1}: Loss: {epoch_loss:.4f}, Mean Dice: {mean_train_dice:.4f}")

# Validation
val_loss, val_dice = validation(model, val_loader, loss_function, device, epoch)

# Early stopping check
stop_training = early_stopping.should_stop(val_loss)
if stop_training:
    print("Early stopping triggered")
    return global_step, dice_val_best, global_step_best, stop_training

if val_dice > dice_val_best:
    dice_val_best = val_dice
    global_step_best = global_step

global_step += 1
return global_step, dice_val_best, global_step_best, stop_training
```

Main Training loop

```python
max_epochs = 250
early_stopping = EarlyStopping(patience=15, min_delta=0.0001)
global_step = 0
dice_val_best = 0.0
global_step_best = 0
# Training Loop
for epoch in range(max_epochs):
    global_step, dice_val_best, global_step_best, stop_training = train(
        model,
        global_step,
        train_loader,
        val_loader,
        loss_function,
        optimizer,
        device,
        dice_val_best,
        global_step_best,
        epoch,
        early_stopping
    )

    if stop_training:
        break
# Finish Wandb logging
wandb.finish()
```

Results and Analysis

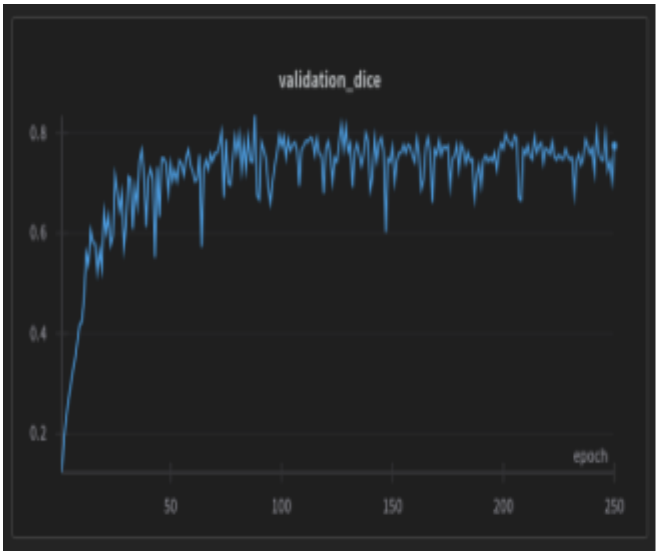
We conducted close to 70 experiments with different sets of transforms and different hyperparameters.

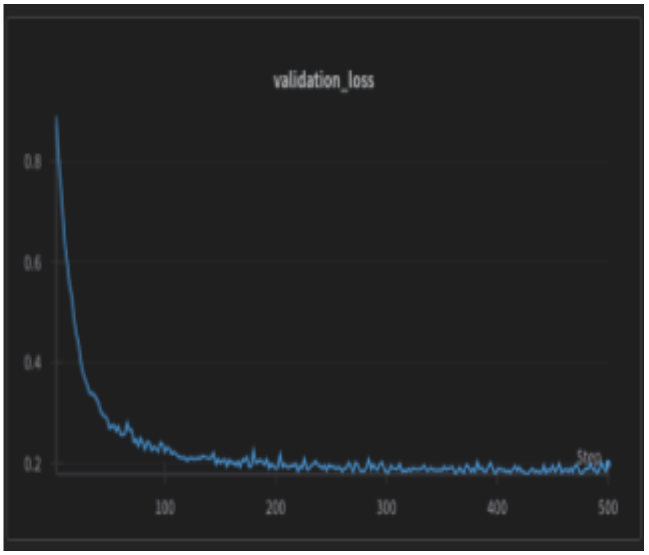
epoch	train_dice	train_loss	validation_dic	validation_loss
250	0.71988	0.1383	0.7745	0.1978
200	0.72588	0.15162	0.74287	0.19499
200	0.59964	0.34026	0.70475	0.22543
200	0.64301	0.33943	0.65907	0.22151
200	0.57234	0.33804	0.60644	0.24073
61	0.5053	0.44011	0.52273	0.33209
64	0.41153	0.44021	0.43602	0.40665
100	0.51673	0.38857	0.35061	0.50774

The Best Experiment

A dice score of 0.77 on validation set was obtained in a run of 250 epochs.

The below graphs are WandB loggings which show the dice curve and the loss curve.





Below are the class-wise validation dice scores:

```
val_dice_class_0:  0.9500665664672852
val_dice_class_1:  0.6834901571273804
val_dice_class_2:  0.6502509713172913
val_dice_class_3:  0.8552199602127075
val_dice_class_4:  0.7948681116104126
val_dice_class_5:  0.9145325422286988
val_dice_class_6:  0.6984032988548279
val_dice_class_7:  0.6491819024085999
```

The classes as previously mentioned are:

1. External Cerebrospinal Fluid
2. Grey Matter
3. White Matter
4. Ventricles
5. Cerebellum
6. Deep Grey Matter
7. Brainstem

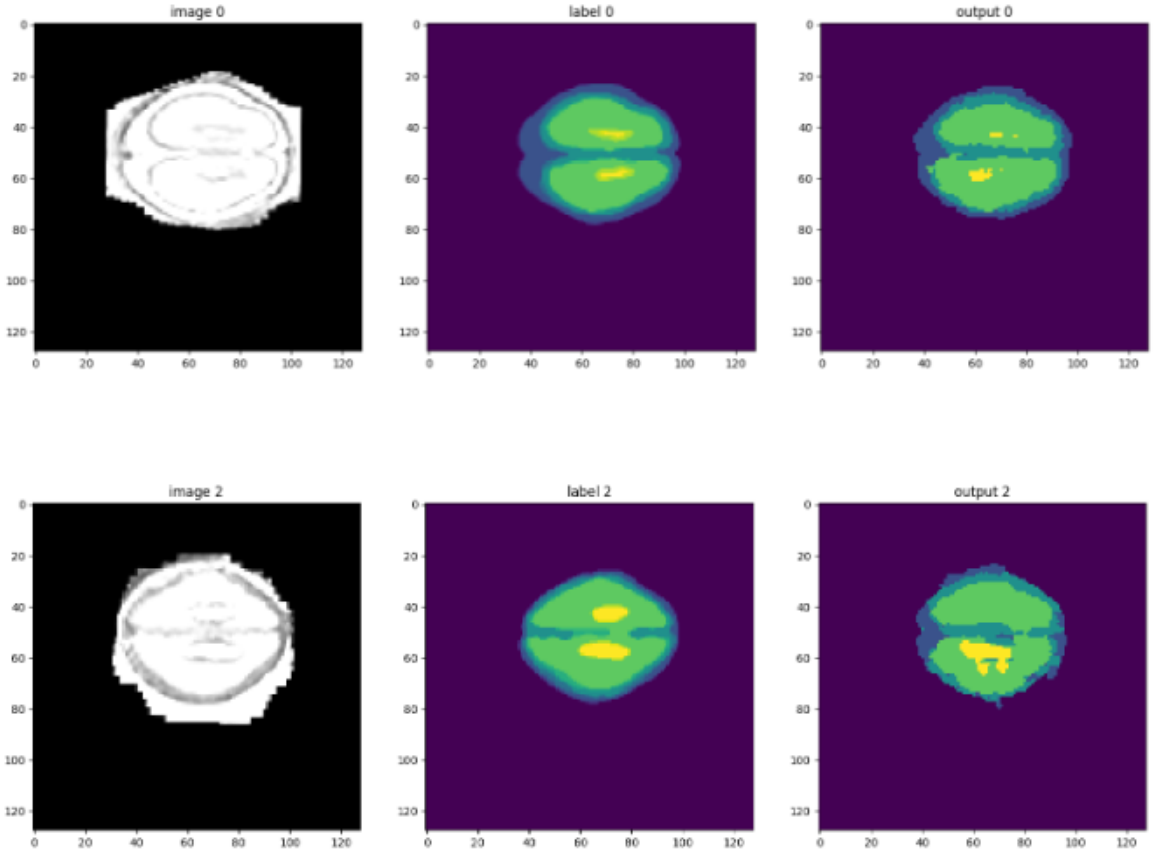
excluding background (class 0).
From the above results, we are able to infer that Cerebellum has the highest dice score followed by White Matter (we will ignore the background for now).

This experiment was run with more extensive preprocessing transforms than augmentation transforms.

```
▼ train_transforms: [] 6 items
  0: "<monai.transforms.io.dictionary.LoadImaged object at 0x7f4b08968fd0>"
  1: "<monai.transforms.utility.dictionary.EnsureChannelFirstd object at 0x7f4b08968700>"
  2: "<monai.transforms.croppad.dictionary.ResizeWithPadOrCropd object at 0x7f4b08968970>"
  3: "<monai.transforms.spatial.dictionary.Orientationd object at 0x7f4b0896b0d0>"
  4: "<monai.transforms.intensity.dictionary.AdjustContrastd object at 0x7f4b0896a230>"
  5: "<monai.transforms.intensity.dictionary.NormalizeIntensityd object at 0x7f4b08968b50>"
```

The high results despite the use of less augmentations could be the result of the simplicity of the dataset. The biggest problem faced was the varying intensities of the voxels since the data was collected from various distinct places. The transforms NormalizeIntensityd helps with improving the clarity of the dataset. The AdjustContrastd, despite being a transform primarily for augmentation, also helps with the same.

The below figures help visualise the performance of the model on the validation.



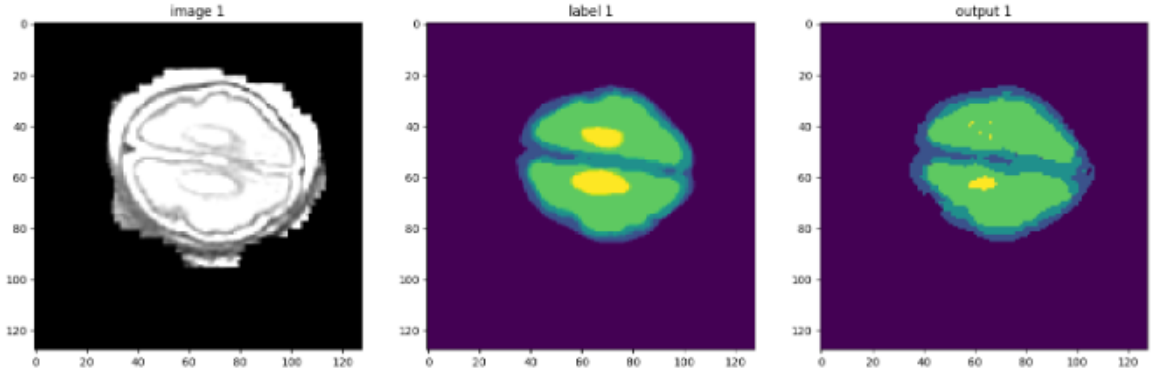


image 1
label 1
output 1
0
20
40
60
80
100
120

Pending Modules

Data Preprocessing and Augmentation for UNet-per-label:

- Implement label-specific preprocessing pipelines, including customized data normalization and augmentation strategies tailored to each segmentation label.
- Experiment with advanced augmentation techniques such as elastic deformations, rotations, and intensity scaling to optimize model generalization.

Pipeline Development:

- Create individual training pipelines for each U-Net, ensuring independent backpropagation and gradient updates for label-specific learning.
- Compare the performance of the UNet-per-label approach against a unified UNet model to validate the effectiveness of the modular architecture.

References and Attachments

[1] U-Net: What's that all about?
https://www.kaggle.com/code/spellsharp/u-net-what-s-that-all-about
[2] Kaggle Notebook
https://www.kaggle.com/code/spellsharp/feta-challenge-2024
[3] GitHub Link
https://github.com/TheHuntsman4/FeTA-2024
[4] WandB logs
https://wandb.ai/anikethvij-personal/FETA-model-training