Computer Vision PA #3 Report

20195159 Wooseok Jeon

Abstract

This Programming Assignment #3 focuses on the segmentation, particularly binary segmentation. Segmentation refers to dividing an image into various segments, either to simplify its representation or to enhance its meaningfulness for analytical purposes. Binary segmentation, a specific type of segmentation, focuses on splitting the image into two distinct parts: the foreground and the background. First, the U-Net model is employed to obtain initial coarse segmentation. Additionally, a Spatial Propagation Network (SPN) is utilized to ascertain pixel or segment affinity, which is essentially a measure of similarity. The U-Net based model is also applied for this purpose. To further refine the initial coarse segmentation, the Convolutional Spatial Propagation Network (CSPN) and Dynamic Spatial Propagation Network (DySPN) models are implemented. This comprehensive approach aims to achieve improved segmentation results. As a result, more refined segmentation was achieved using CSPN and DySPN techniques compared to the initial coarse segmentation obtained with the Unet model.

1. Method

All the code execution was carried out in a Colab environment. However, due to insufficient RAM, the process of reconstructing the data was executed in the Visual Studio environment.

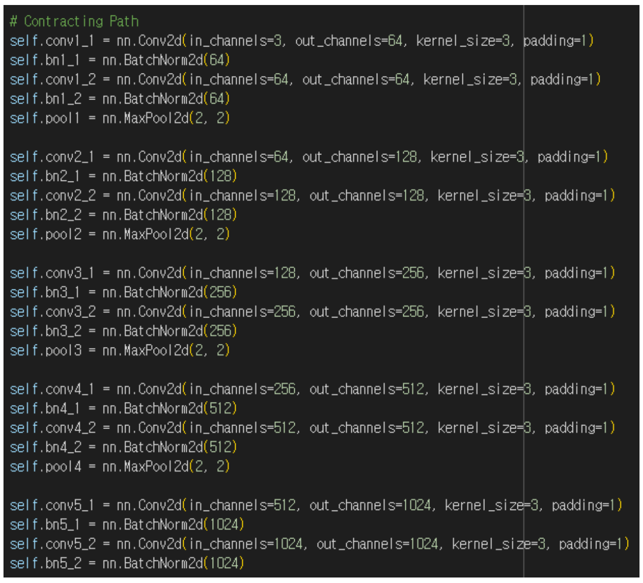
1.1. Step 0: Preprocessing

The provided dataset is known as SimpleOxfordPetDataset, which contains images of dogs and cats, along with binary segmentation information stored in a dictionary format. The process involves dividing this dataset into a training dataset and a validation dataset. Afterwards, the data is prepared for training the model through the data loader.

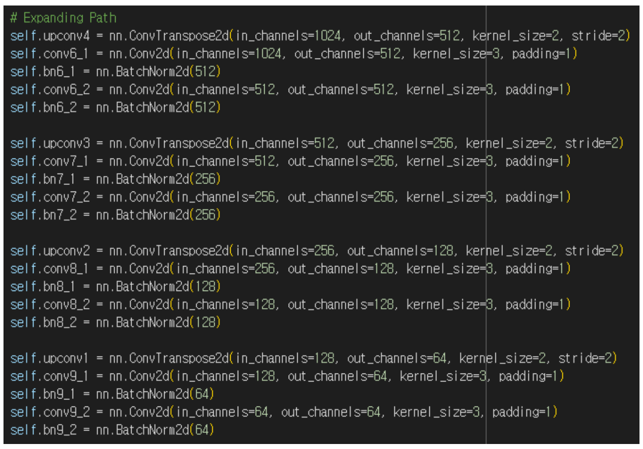
1.2. Step 1: Get Coarse Segmentation

The U-Net model is used for training the dataset. U-Net belongs to the encoder-decoder based models. The fundamental idea of encoder-decoder models is to increase the number of channels while reducing the dimensions during the encoding phase to capture features of the input image. This approach allows the model to extract image features using both low and high-dimensional information, enabling precise location identification. The structure of U-Net is broadly divided into two paths: the Contracting Path and the Expanding Path. The Contracting Path, also known as the encoder, involves repeated operations of 3 X 3 convolution, batch normalization, ReLU operations, and max pooling. The Expanding Path, known as the decoder, replaces max pooling with Up convolution operations and also employs the aforementioned concatenation. The final layer includes a 1 X 1 convolution operation for segmentation prediction.

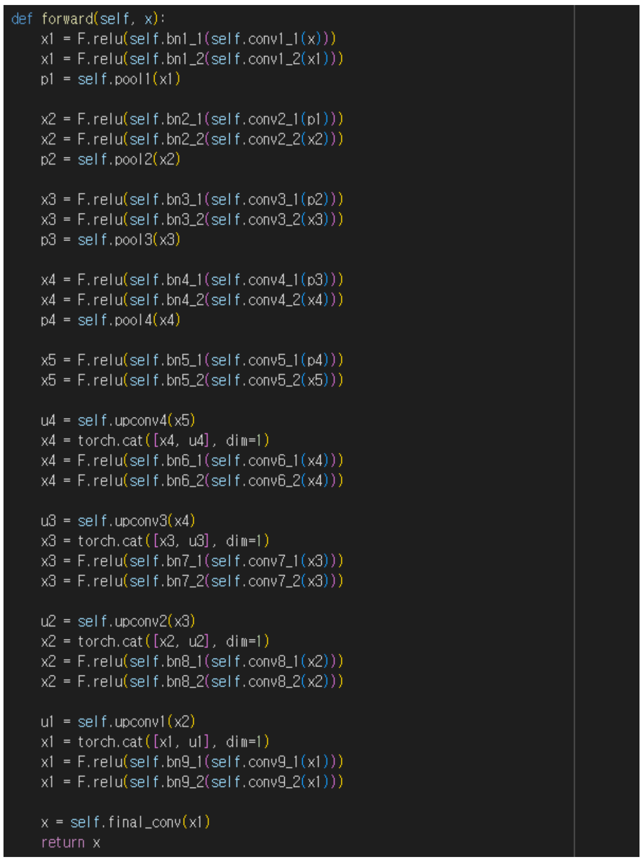
Below is the U-Net model code that I use. Each part consists of the encoder section known as the contracting path, the decoder section known as the expanding path, and the forward part.



```
# Contracting Path
self.conv1_1 = nn.Conv2d(in_channels=3, out_channels=64, kernel_size=3, padding=1)
self.bn1_1 = nn.BatchNorm2d(64)
self.conv1_2 = nn.Conv2d(in_channels=64, out_channels=64, kernel_size=3, padding=1)
self.bn1_2 = nn.BatchNorm2d(64)
self.pool1 = nn.MaxPool2d(2, 2)

self.conv2_1 = nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, padding=1)
self.bn2_1 = nn.BatchNorm2d(128)
self.conv2_2 = nn.Conv2d(in_channels=128, out_channels=128, kernel_size=3, padding=1)
self.bn2_2 = nn.BatchNorm2d(128)
self.pool2 = nn.MaxPool2d(2, 2)

self.conv3_1 = nn.Conv2d(in_channels=128, out_channels=256, kernel_size=3, padding=1)
self.bn3_1 = nn.BatchNorm2d(256)
self.conv3_2 = nn.Conv2d(in_channels=256, out_channels=256, kernel_size=3, padding=1)
self.bn3_2 = nn.BatchNorm2d(256)
self.pool3 = nn.MaxPool2d(2, 2)

self.conv4_1 = nn.Conv2d(in_channels=256, out_channels=512, kernel_size=3, padding=1)
self.bn4_1 = nn.BatchNorm2d(512)
self.conv4_2 = nn.Conv2d(in_channels=512, out_channels=512, kernel_size=3, padding=1)
self.bn4_2 = nn.BatchNorm2d(512)
self.pool4 = nn.MaxPool2d(2, 2)

self.conv5_1 = nn.Conv2d(in_channels=512, out_channels=1024, kernel_size=3, padding=1)
self.bn5_1 = nn.BatchNorm2d(1024)
self.conv5_2 = nn.Conv2d(in_channels=1024, out_channels=1024, kernel_size=3, padding=1)
self.bn5_2 = nn.BatchNorm2d(1024)
```



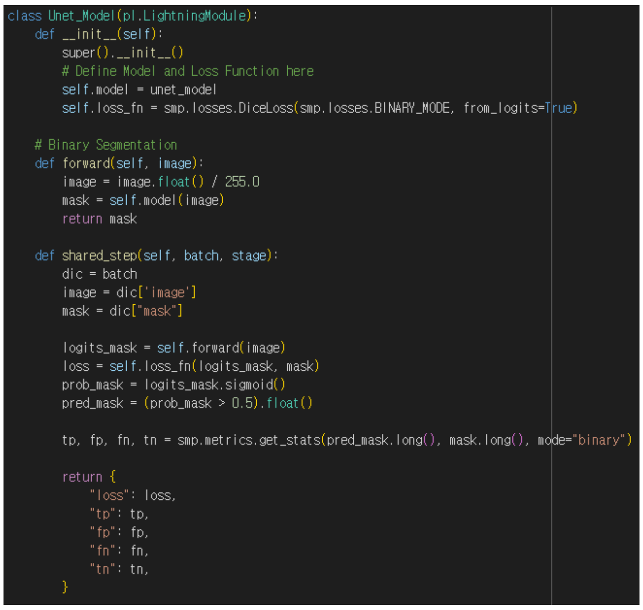
```
# Expanding Path
self.upconv4 = nn.ConvTranspose2d(in_channels=1024, out_channels=512, kernel_size=2, stride=2)
self.conv6_1 = nn.Conv2d(in_channels=1024, out_channels=512, kernel_size=3, padding=1)
self.bn6_1 = nn.BatchNorm2d(512)
self.conv6_2 = nn.Conv2d(in_channels=512, out_channels=512, kernel_size=3, padding=1)
self.bn6_2 = nn.BatchNorm2d(512)

self.upconv3 = nn.ConvTranspose2d(in_channels=512, out_channels=256, kernel_size=2, stride=2)
self.conv7_1 = nn.Conv2d(in_channels=512, out_channels=256, kernel_size=3, padding=1)
self.bn7_1 = nn.BatchNorm2d(256)
self.conv7_2 = nn.Conv2d(in_channels=256, out_channels=256, kernel_size=3, padding=1)
self.bn7_2 = nn.BatchNorm2d(256)

self.upconv2 = nn.ConvTranspose2d(in_channels=256, out_channels=128, kernel_size=2, stride=2)
self.conv8_1 = nn.Conv2d(in_channels=256, out_channels=128, kernel_size=3, padding=1)
self.bn8_1 = nn.BatchNorm2d(128)
self.conv8_2 = nn.Conv2d(in_channels=128, out_channels=128, kernel_size=3, padding=1)
self.bn8_2 = nn.BatchNorm2d(128)

self.upconv1 = nn.ConvTranspose2d(in_channels=128, out_channels=64, kernel_size=2, stride=2)
self.conv9_1 = nn.Conv2d(in_channels=128, out_channels=64, kernel_size=3, padding=1)
self.bn9_1 = nn.BatchNorm2d(64)
self.conv9_2 = nn.Conv2d(in_channels=64, out_channels=64, kernel_size=3, padding=1)
self.bn9_2 = nn.BatchNorm2d(64)
```



```
def forward(self, x):
    x1 = F.relu(self.bn1_1(self.conv1_1(x)))
    x1 = F.relu(self.bn1_2(self.conv1_2(x1)))
    p1 = self.pool1(x1)

    x2 = F.relu(self.bn2_1(self.conv2_1(p1)))
    x2 = F.relu(self.bn2_2(self.conv2_2(x2)))
    p2 = self.pool2(x2)

    x3 = F.relu(self.bn3_1(self.conv3_1(p2)))
    x3 = F.relu(self.bn3_2(self.conv3_2(x3)))
    p3 = self.pool3(x3)

    x4 = F.relu(self.bn4_1(self.conv4_1(p3)))
    x4 = F.relu(self.bn4_2(self.conv4_2(x4)))
    p4 = self.pool4(x4)

    x5 = F.relu(self.bn5_1(self.conv5_1(p4)))
    x5 = F.relu(self.bn5_2(self.conv5_2(x5)))

    u4 = self.upconv4(x5)
    x4 = torch.cat([x4, u4], dim=1)
    x4 = F.relu(self.bn6_1(self.conv6_1(x4)))
    x4 = F.relu(self.bn6_2(self.conv6_2(x4)))

    u3 = self.upconv3(x4)
    x3 = torch.cat([x3, u3], dim=1)
    x3 = F.relu(self.bn7_1(self.conv7_1(x3)))
    x3 = F.relu(self.bn7_2(self.conv7_2(x3)))

    u2 = self.upconv2(x3)
    x2 = torch.cat([x2, u2], dim=1)
    x2 = F.relu(self.bn8_1(self.conv8_1(x2)))
    x2 = F.relu(self.bn8_2(self.conv8_2(x2)))

    u1 = self.upconv1(x2)
    x1 = torch.cat([x1, u1], dim=1)
    x1 = F.relu(self.bn9_1(self.conv9_1(x1)))
    x1 = F.relu(self.bn9_2(self.conv9_2(x1)))

    x = self.final_conv(x1)
    return x
```

Now, to train the dataset using the U-net model, a pre-provided PyTorch Lightning module is utilized. The DiceLoss function is employed as the loss function. During training, the mask output from the forward function is passed through a sigmoid function, and then the Intersection over Union (IoU) is calculated with the original mask for training.

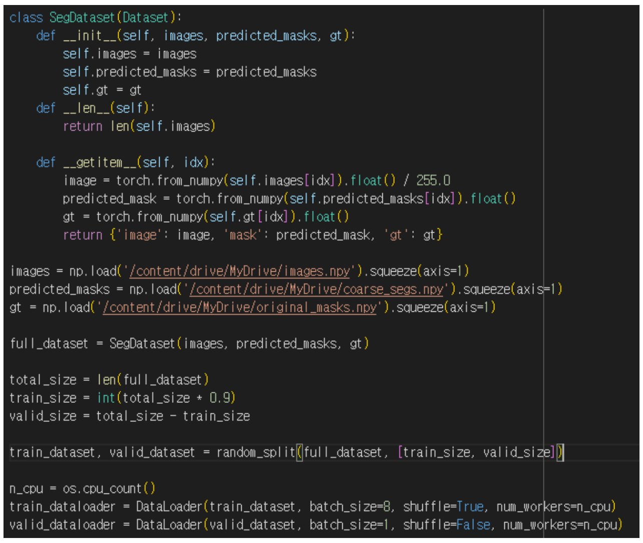
Below is the code that illustrates this process.



```
class Unet_Model(pl.LightningModule):
    def __init__(self):
        super().__init__()
        # Define Model and Loss Function here
        self.model = unet_model
        self.loss_fn = smp.losses.DiceLoss(smp.losses.BINARY_MODE, from_logits=True)

    # Binary Segmentation
    def forward(self, image):
        image = image.float() / 255.0
        mask = self.model(image)
        return mask

    def shared_step(self, batch, stage):
        dic = batch
        image = dic['image']
        mask = dic["mask"]

        logits_mask = self.forward(image)
        loss = self.loss_fn(logits_mask, mask)
        prob_mask = logits_mask.sigmoid()
        pred_mask = (prob_mask > 0.5).float()

        tp, fp, fn, tn = smp.metrics.get_stats(pred_mask.long(), mask.long(), mode="binary")

        return {
            "loss": loss,
            "tp": tp,
            "fp": fp,
            "fn": fn,
            "tn": tn,
        }
```

Now, as I need to refine the coarse segmentation I've obtained, it's necessary to save these data as npy files and recreate the dataset. The reconstructed dataset is stored in a dictionary format containing the image, the original mask, and the coarse mask. During this process, the PyTorch dataset module was utilized to rebuild the dataloader.

Below is the code that encapsulates this process.



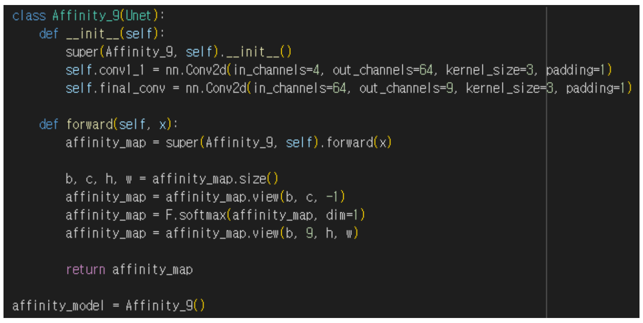
```
class SegDataset(Dataset):
    def __init__(self, images, predicted_masks, gt):
        self.images = images
        self.predicted_masks = predicted_masks
        self.gt = gt
    def __len__(self):
        return len(self.images)

    def __getitem__(self, idx):
        image = torch.from_numpy(self.images[idx]).float() / 255.0
        predicted_mask = torch.from_numpy(self.predicted_masks[idx]).float()
        gt = torch.from_numpy(self.gt[idx]).float()
        return {'image': image, 'mask': predicted_mask, 'gt': gt}

images = np.load('/content/drive/MyDrive/images.npy').squeeze(axis=1)
predicted_masks = np.load('/content/drive/MyDrive/coarse_segs.npy').squeeze(axis=1)
gt = np.load('/content/drive/MyDrive/original_masks.npy').squeeze(axis=1)

full_dataset = SegDataset(images, predicted_masks, gt)

total_size = len(full_dataset)
train_size = int(total_size * 0.9)
valid_size = total_size - train_size

train_dataset, valid_dataset = random_split(full_dataset, [train_size, valid_size])

n_cpu = os.cpu_count()
train_dataloader = DataLoader(train_dataset, batch_size=8, shuffle=True, num_workers=n_cpu)
valid_dataloader = DataLoader(valid_dataset, batch_size=1, shuffle=False, num_workers=n_cpu)
```

1.3. Step 2: Convolutional Spatial Propagation Network

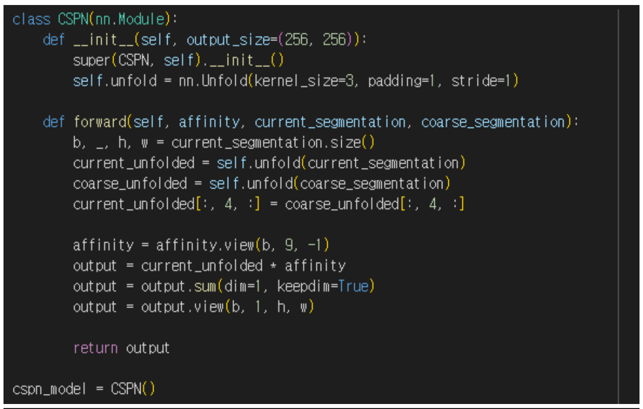
The Convolutional Spatial Propagation Network (CSPN) is a method in computer vision that utilizes a linear propagation model, incorporating recurrent convolutional operations to effectively disseminate information. CSPN's distinctive feature is its proficiency in learning the affinity matrix, which represents the relationships among neighboring pixels, through a deep convolutional neural network (CNN). This technique refines initial coarse segmentation by leveraging affinity information.

To acquire a 3 X 3 size affinity matrix, modifications were made to the final layer of the U-net. These modifications involved adjusting the layer to have a kernel size of 3, padding size of 1, and output layer size of 9. Following this, a softmax function was applied to normalize the nine affinity values, ensuring their sum equals one. Subsequently, the process involved propagating information from neighborhood pixels to the center pixel using these affinity values. CSPN underwent a total of six iterations in this procedure.

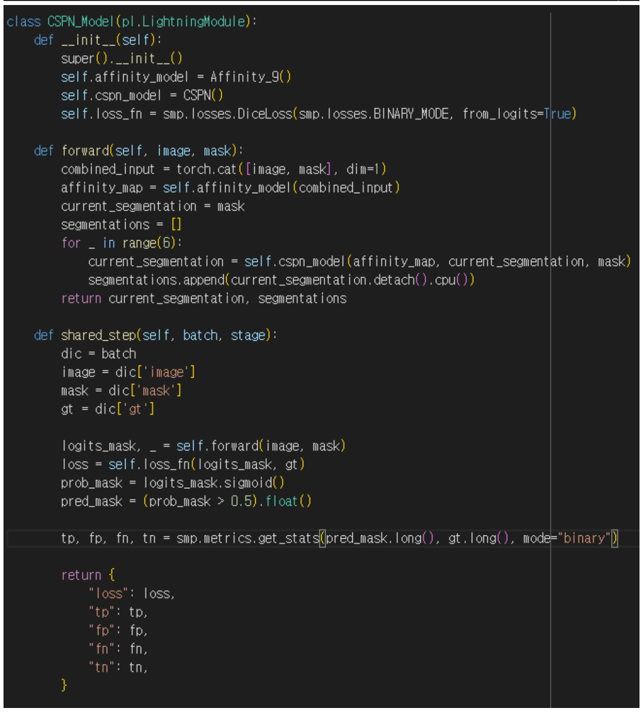
Below is the code for the newly defined Affinity Map class and training module.



```
class Affinity_9(Unet):
    def __init__(self):
        super(Affinity_9, self).__init__()
        self.conv1_1 = nn.Conv2d(in_channels=4, out_channels=64, kernel_size=3, padding=1)
        self.final_conv = nn.Conv2d(in_channels=64, out_channels=9, kernel_size=3, padding=1)

    def forward(self, x):
        affinity_map = super(Affinity_9, self).forward(x)

        b, c, h, w = affinity_map.size()
        affinity_map = affinity_map.view(b, c, -1)
        affinity_map = F.softmax(affinity_map, dim=1)
        affinity_map = affinity_map.view(b, 9, h, w)

        return affinity_map

affinity_model = Affinity_9()
```



```python
class CSPN(nn.Module):
    def __init__(self, output_size=(256, 256)):
        super(CSPN, self).__init__()
        self.unfold = nn.Unfold(kernel_size=3, padding=1, stride=1)

    def forward(self, affinity, current_segmentation, coarse_segmentation):
        b, _, h, w = current_segmentation.size()
        current_unfolded = self.unfold(current_segmentation)
        coarse_unfolded = self.unfold(coarse_segmentation)
        current_unfolded[:, 4, :] = coarse_unfolded[:, 4, :]

        affinity = affinity.view(b, 9, -1)
        output = current_unfolded * affinity
        output = output.sum(dim=1, keepdim=True)
        output = output.view(b, 1, h, w)

        return output

cspn_model = CSPN()
```



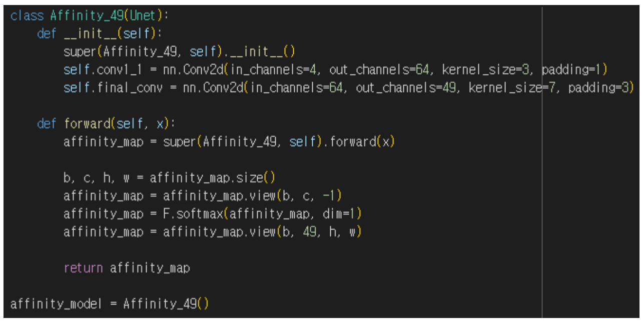
```python
class CSPN_Model(pl.LightningModule):
    def __init__(self):
        super().__init__()
        self.affinity_model = Affinity_9()
        self.cspn_model = CSPN()
        self.loss_fn = smp.losses.DiceLoss(smp.losses.BINARY_MODE, from_logits=True)

    def forward(self, image, mask):
        combined_input = torch.cat([image, mask], dim=1)
        affinity_map = self.affinity_model(combined_input)
        current_segmentation = mask
        segmentations = []
        for _ in range(6):
            current_segmentation = self.cspn_model(affinity_map, current_segmentation, mask)
            segmentations.append(current_segmentation.detach().cpu())
        return current_segmentation, segmentations

    def shared_step(self, batch, stage):
        dic = batch
        image = dic['image']
        mask = dic['mask']
        gt = dic['gt']

        logits_mask, _ = self.forward(image, mask)
        loss = self.loss_fn(logits_mask, gt)
        prob_mask = logits_mask.sigmoid()
        pred_mask = (prob_mask > 0.5).float()

        tp, fp, fn, tn = smp.metrics.get_stats(pred_mask.long(), gt.long(), mode="binary")

        return {
            "loss": loss,
            "tp": tp,
            "fp": fp,
            "fn": fn,
            "tn": tn,
        }
```

1.4. *Step 3: Dynamic Spatial Propagation Network*

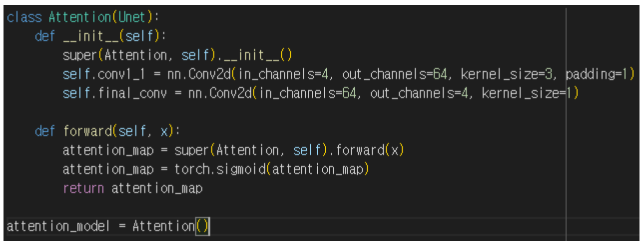
The Dynamic Spatial Propagation Network (DySPN) is an affinity-based technique designed to improve initial coarse segmentation. It stands out from its predecessor CSPN, by employing a more dynamic approach : attention.

The first step involves modifying the final layer of U-net to obtain a 7 X 7 affinity matrix, achieved by adjusting the kernel size to 7, the padding size to 3, and the output layer size to 49. Next, a softmax function is applied to ensure that the sum of the 49 affinity values equals one. Following this, these affinity values are used to propagate information from neighboring pixels to the center pixel. Similar to CSPN, which typically requires six iterations, DySPN also necessitates six iterations. This model is more efficient in effectively segmenting different neighborhood parts according to distance and independently generating attention maps, refining these into adaptive affinity matrices.

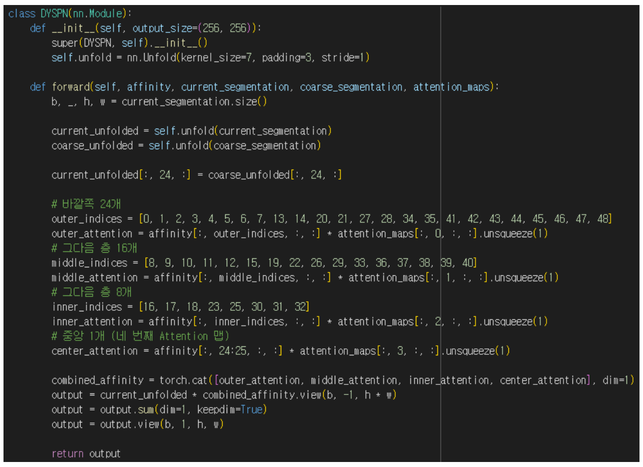
Below is the code for the newly defined Attention Map class and the DySPN class, as well as the code for the redefined Affinity Map class with training module.



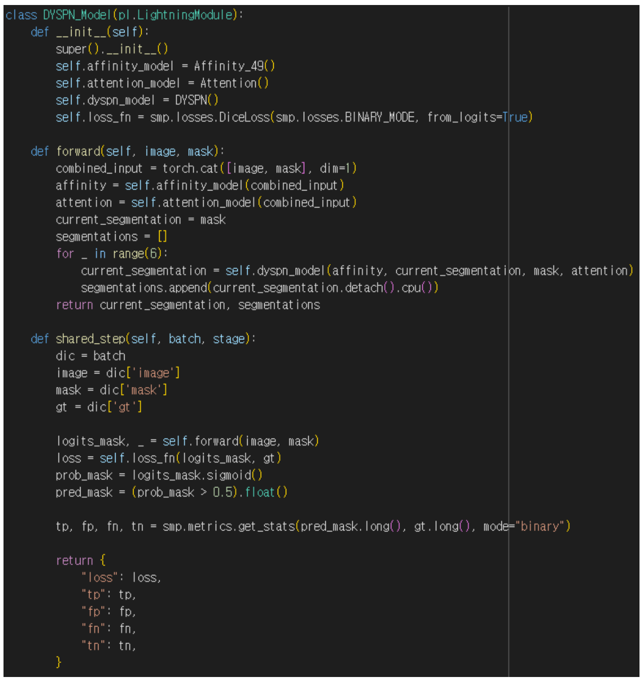
```python
class Affinity_49(Unet):
    def __init__(self):
        super(Affinity_49, self).__init__()
        self.conv1_1 = nn.Conv2d(in_channels=4, out_channels=64, kernel_size=3, padding=1)
        self.final_conv = nn.Conv2d(in_channels=64, out_channels=49, kernel_size=7, padding=3)

    def forward(self, x):
        affinity_map = super(Affinity_49, self).forward(x)

        b, c, h, w = affinity_map.size()
        affinity_map = affinity_map.view(b, c, -1)
        affinity_map = F.softmax(affinity_map, dim=1)
        affinity_map = affinity_map.view(b, 49, h, w)

        return affinity_map

affinity_model = Affinity_49()
```



```python
class Attention(Unet):
    def __init__(self):
        super(Attention, self).__init__()
        self.conv1_1 = nn.Conv2d(in_channels=4, out_channels=64, kernel_size=3, padding=1)
        self.final_conv = nn.Conv2d(in_channels=64, out_channels=4, kernel_size=1)

    def forward(self, x):
        attention_map = super(Attention, self).forward(x)
        attention_map = torch.sigmoid(attention_map)
        return attention_map

attention_model = Attention()
```



```python
class DYSPN(nn.Module):
    def __init__(self, output_size=(256, 256)):
        super(DYSPN, self).__init__()
        self.unfold = nn.Unfold(kernel_size=7, padding=3, stride=1)

    def forward(self, affinity, current_segmentation, coarse_segmentation, attention_maps):
        b, _, h, w = current_segmentation.size()

        current_unfolded = self.unfold(current_segmentation)
        coarse_unfolded = self.unfold(coarse_segmentation)

        current_unfolded[:, 24, :] = coarse_unfolded[:, 24, :]

        # 바깥쪽 24개
        outer_indices = [0, 1, 2, 3, 4, 5, 6, 7, 13, 14, 20, 21, 27, 28, 34, 35, 41, 42, 43, 44, 45, 46, 47, 48]
        outer_attention = affinity[:, outer_indices, :, :] * attention_maps[:, 0, :, :].unsqueeze(1)
        # 그다음 층 16개
        middle_indices = [8, 9, 10, 11, 12, 15, 19, 22, 26, 29, 33, 36, 37, 38, 39, 40]
        middle_attention = affinity[:, middle_indices, :, :] * attention_maps[:, 1, :, :].unsqueeze(1)
        # 그다음 층 8개
        inner_indices = [16, 17, 18, 23, 25, 30, 31, 32]
        inner_attention = affinity[:, inner_indices, :, :] * attention_maps[:, 2, :, :].unsqueeze(1)
        # 중앙 1개 (네 번째 Attention 맵)
        center_attention = affinity[:, 24:25, :, :] * attention_maps[:, 3, :, :].unsqueeze(1)

        combined_affinity = torch.cat([outer_attention, middle_attention, inner_attention, center_attention], dim=1)
        output = current_unfolded * combined_affinity.view(b, -1, h * w)
        output = output.sum(dim=1, keepdim=True)
        output = output.view(b, 1, h, w)

        return output
```



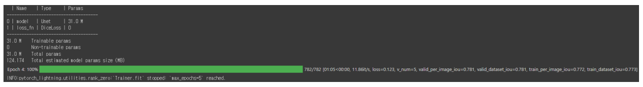
```python
class DYSPN_Model(pl.LightningModule):
    def __init__(self):
        super().__init__()
        self.affinity_model = Affinity_49()
        self.attention_model = Attention()
        self.dyspn_model = DYSPN()
        self.loss_fn = smp.losses.DiceLoss(smp.losses.BINARY_MODE, from_logits=True)

    def forward(self, image, mask):
        combined_input = torch.cat([image, mask], dim=1)
        affinity = self.affinity_model(combined_input)
        attention = self.attention_model(combined_input)
        current_segmentation = mask
        segmentations = []
        for _ in range(6):
            current_segmentation = self.dyspn_model(affinity, current_segmentation, mask, attention)
            segmentations.append(current_segmentation.detach().cpu())
        return current_segmentation, segmentations

    def shared_step(self, batch, stage):
        dic = batch
        image = dic['image']
        mask = dic['mask']
        gt = dic['gt']

        logits_mask, _ = self.forward(image, mask)
        loss = self.loss_fn(logits_mask, gt)
        prob_mask = logits_mask.sigmoid()
        pred_mask = (prob_mask > 0.5).float()

        tp, fp, fn, tn = smp.metrics.get_stats(pred_mask.long(), gt.long(), mode="binary")

        return {
            "loss": loss,
            "tp": tp,
            "fp": fp,
            "fn": fn,
            "tn": tn,
        }
```

2. Experiment & Result

The number of epochs is 5, and the learning rate is 0.001 in every process.

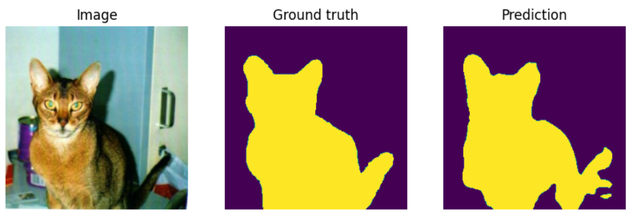
2.1. Coarse Segmentation

The final loss after training the Unet model was 0.123, with a valid per image IOU of 0.781, valid dataset IOU of 0.781, train per image IOU of 0.772, and train dataset IOU of 0.773.



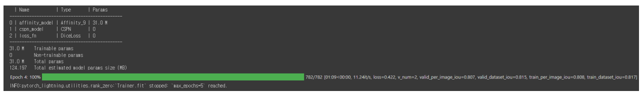
```
  | Name    | Type     | Params
--------------------------------------
0 | model   | Unet     | 31.0 M
1 | loss_fn | DiceLoss | 0
--------------------------------------
31.0 M    Trainable params
0         Non-trainable params
31.0 M    Total params
124.174   Total estimated model params size (MB)
Epoch 4: 100% 782/782 [01:05<00:00, 11.86it/s, loss=0.123, v_num=5, valid_per_image_iou=0.781, valid_dataset_iou=0.781, train_per_image_iou=0.772, train_dataset_iou=0.773]
INFO:pytorch_lightning.utilities.rank_zero:`Trainer.fit` stopped: `max_epochs=5` reached.
```

Observing the initial coarse segmentation results obtained from the trained model, it is evident that, while generally accurate, there is still room for improvement.



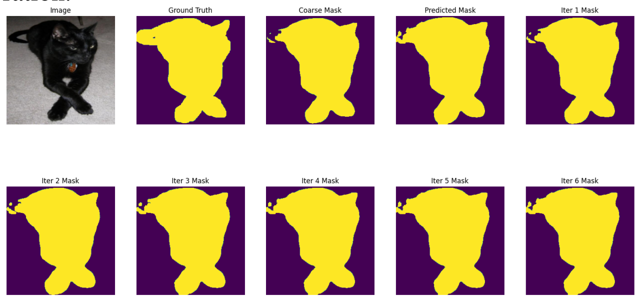
2.2. Refined Segmentation: CSPN

The final loss after training the CSPN model was 0.422, with a valid per image IOU of 0.807, valid dataset IOU of 0.815, train per image IOU of 0.808, and train dataset IOU of 0.817. Therefore, better segmentation has been achieved compared to the initial one.



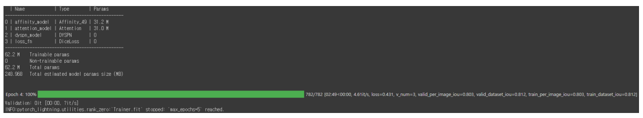
```
  | Name           | Type       | Params
--------------------------------------------------
0 | affinity_model | Affinity_9 | 31.0 M
1 | cspn_model     | CSPN       | 0
2 | loss_fn        | DiceLoss   | 0
--------------------------------------------------
31.0 M    Trainable params
0         Non-trainable params
31.0 M    Total params
124.197   Total estimated model params size (MB)
Epoch 4: 100% 782/782 [01:09<00:00, 11.24it/s, loss=0.422, v_num=2, valid_per_image_iou=0.807, valid_dataset_iou=0.815, train_per_image_iou=0.808, train_dataset_iou=0.817]
INFO:pytorch_lightning.utilities.rank_zero:`Trainer.fit` stopped: `max_epochs=5` reached.
```

The image below visualizes the results of the CSPN. The results show that performance progressively improves with each iteration.



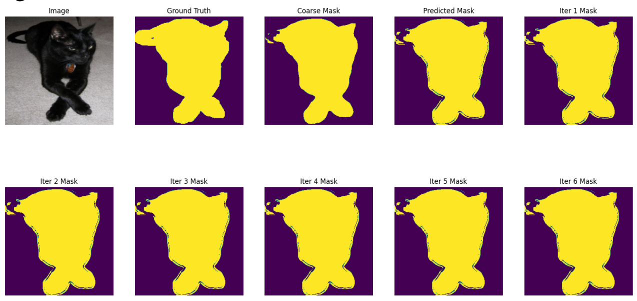
2.3. Refined Segmentation: DySPN

The final loss after training the DySPN model was 0.431, with a valid per image IOU of 0.803, valid dataset IOU of 0.812, train per image IOU of 0.803, and train dataset IOU of 0.812. Contrary to the theory, it did not yield better results than the CSPN.



```
  | Name            | Type        | Params
---------------------------------------------------
0 | affinity_model  | Affinity_49 | 31.2 M
1 | attention_model | Attention   | 31.0 M
2 | dyspn_model     | DYSPN       | 0
3 | loss_fn         | DiceLoss    | 0
---------------------------------------------------
62.2 M    Trainable params
0         Non-trainable params
62.2 M    Total params
248.968   Total estimated model params size (MB)

Epoch 4: 100% 782/782 [02:49<00:00, 4.61it/s, loss=0.431, v_num=3, valid_per_image_iou=0.803, valid_dataset_iou=0.812, train_per_image_iou=0.803, train_dataset_iou=0.812]
Validation: 0it [00:00, ?it/s]
INFO:pytorch_lightning.utilities.rank_zero:`Trainer.fit` stopped: `max_epochs=5` reached.
```

The image below visualizes the results of the CSPN. The results show that the segmentation around the periphery of the image becomes more refined with each iteration.



As a result, more refined segmentation was achieved using CSPN and DySPN techniques compared to the initial coarse segmentation obtained with the Unet model.

3. References

1. Cheng et al. “Learning Depth with Convolutional Spatial Propagation Network” (2019)
2. Lin et al. “Dynamic Spatial Propagation Network for Depth Completion” (2022)