Lab2: Binary Semantic Segmentation

Shyang-En Weng 翁祥恩 413551036

March 23, 2025

1 Introduction

In this lab, we will focus on binary semantic segmentation to classify the image with objects into the foreground (object of interest) and background. For the experiment, I implemented the UNet [7] and ResNet34+UNet trained and tested on the Oxford-IIIT Pet dataset [6], identifying the cats and dogs at the pixel level. The contributions are listed below:

- Both U-Net and ResNet34+U-Net implementations achieve a Dice score 93% and 91%, repectively.
- Various data augmentation techniques are applied during training, enhancing convergence and overall performance.
- Additional experiments on different upsampling methods are conducted and analyzed in this lab.

2 Implementation Details

2.1 Training

The main purpose of the training is to make the model learn the dataset. For the whole training process, it can be split into few steps, including data preparation, model settings, training, and validation. In this section, I will go through the process of training and validation in a single iteration, to multiple epochs.

The implementation of the training process is shown in Listing 1. In each iteration, first clear the gradients, then the model processes a mini-batch of input data, computes the forward pass to generate predictions, and evaluates the loss function by comparing the predictions with the ground truth labels. Next, the computed loss is then backpropagated through the network using an optimization algorithm to update the model parameters, also update the states of optimizer and scheduler. Last, the training loop continues for a predefined number of epochs to be met. In my implementation, Automatic Mixed Precision (AMP) was employed to accelerate the training process and maximize the batch size by optimizing memory usage and computational efficiency.

Listing 1: Implementation of training process.

```
best_dice_score = 0
    # Train the model
    print("Start training...")
    for epoch in range(args.epochs):
        train_loss = 0
        model.train()
        train_loop = tqdm(train_loader)
        for batch, sample in enumerate(train_loop):
            optimizer.zero_grad()

            inputs, masks = sample['image'], sample['mask']
            inputs, masks = inputs.to(device), masks.to(device)

            with autocast(enabled=enable_amp):
                outputs = model(inputs)
                loss = loss_fn(outputs, masks)

            # loss.backward()
            # optimizer.step()
            # scheduler.step()
            scaler.scale(loss).backward()
            scaler.step(optimizer)
            scaler.update()
            scheduler.step()

            train_loss += loss.item()
            train_loop.set_description(f"Epoch [{epoch}/{args.epochs}]")
            train_loop.set_postfix(loss = train_loss / (batch + 1))

            writer.add_scalar('Loss/train', loss.item(), epoch)

        # Validation
        dice_score_val, val_loss = evaluate(model, val_loader, device, loss_fn,

        # Save the model
        if dice_score_val > best_dice_score:
            best_dice_score = dice_score_val
            ckpt_path = os.path.join(ckpt_dir, f"model_{epoch}_{best_dice_score:
            torch.save(model.state_dict(), ckpt_path)
            print(f"Model saved to {ckpt_path}")
```

2.2 Evaluation

To objectively assess the performance of our method, we employ quantitative metrics that measure the accuracy and consistency of the predicted results. In this lab, I adopt the Dice Score, which is widely used in segmentation tasks due to its robustness in measuring spatial overlap between predicted and ground truth regions. The evaluation process similar to training, but without back-propagation, as shown in Listing 2.

Listing 2: Implementation of evaluation process.

```
def evaluate(net, data, device, loss_fn = None, writer = None, epoch = None):
    # implement the evaluation function here
    net.eval()
    with torch.no_grad():
        val_loss = 0
        dice_score_val = 0
        for sample in data:
            images, masks = sample['image'], sample['mask']
            images, masks = images.to(device), masks.to(device)
            outputs = net(images)
            dice_score_val += dice_score(outputs, masks)
            val_loss += loss_fn(outputs, masks).item() if loss_fn else 0
        val_loss /= len(data)
        dice_score_val /= len(data)
        print(f"Validation Dice Score: {dice_score_val}")
        print(f"Validation Loss: {val_loss}") if loss_fn else None

        if writer:
            writer.add_scalar('Loss/val', val_loss, epoch)
            writer.add_scalar('Dice/val', dice_score_val, epoch)

    return dice_score_val, val_loss
```

Dice Score, as shown as Dice similarity coefficient, is a metric to assess the similarity of two samples, as shown in Equation 1.

$$DiceScore = \frac{2|(x_{pred} \cap x_{gt}|}{|x_{pred} + |x_{gt}||} \tag{1}$$

where x_{pred} and x_{gt} are the predicted segmentation mask and ground truth segmentation mask, respectively. This metric quantifies the similarity between the two masks by measuring the proportion of overlapping pixels relative to the total number of pixels in the image. Specifically, it is defined as the ratio of the number of pixels that are common between the predicted and ground truth masks to the overall image size, providing an indication of segmentation accuracy. The implementation is shown in Listing 3.

Listing 3: Implementation of Dice Score.

```
def dice_score(pred_mask, gt_mask, smooth = 1e-8):
    # implement the Dice score here
    assert pred_mask.shape == gt_mask.shape

    pred_mask_flat = pred_mask.view(-1)
    gt_mask_flat = gt_mask.view(-1)

    intersection = torch.sum(pred_mask_flat * gt_mask_flat)
    union = torch.sum(pred_mask_flat + gt_mask_flat)
    dice = (2 * intersection + smooth) / (union + smooth)

    return dice
```

2.3 Inference

During the inference phase, the trained model is deployed to process new input data and generate predictions without further parameter updates. Unlike the training phase, where optimization and backpropagation are involved, inference focuses solely on the forward pass through the network without the gradients calculations, making it computationally efficient. The implementation is shown in Listing 4.

Listing 4: Implementation of inference process.

```
def inference(args, net, data, device):
    # implement the evaluation function here
    net.eval()
    with torch.no_grad():
        for i, sample in enumerate(data):
            image, mask = sample['image'], sample['mask']
            image, mask = image.to(device), mask.to(device)
            outputs = net(image)
            # Save the output masks
            save_images(args, i, image, outputs, mask)

def save_images(args, i, image, output, mask):
    # implement the save images function here
    save_path = os.path.join(args.save_path, args.model)
    if not os.path.exists(save_path):
        os.makedirs(save_path)
    save_img = torch.cat(
        [denormalize(image), output.repeat(1, 3, 1, 1),
        mask.repeat(1, 3, 1, 1)], 0)
    tv.utils.save_image(save_img, os.path.join(
        save_path, f"output_{i}.png"), nrow=3)
```

```
def denormalize(tensor, mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]):
    mean = torch.tensor(mean).view(-1, 1, 1).to(tensor.device)
    std = torch.tensor(std).view(-1, 1, 1).to(tensor.device)
    return tensor * std + mean
```

2.4 UNet and ResNet34+UNet

UNet [7] is a widely used architecture for image segmentation, featuring a symmetric encoder-decoder structure with skip connections that facilitate the direct transfer of spatial information from each of the encoding path to the decoding path. Furthermore, it decomposes and reconstructs the feature representations from a multi-scale perspective, enabling the model to capture both global context and fine-grained details.

The implementation of UNet and ResNet34 encoder-based UNet are shown in Listings 5 and 6, respectively. For UNet, the design follows the idea of the paper [7], and ResNet34 encoder-based UNet is modified from the UNet implementation. The model is split into five stages, where the downsampling and upsampling processes four times, the number of channels is multiplied two times in each stage, and the feature size is reduced two times through max-pooling. In the decoder, the upsampling process of each stage is concatenated with the skipped features from the encoder and the previous stage's output.

There are some slight modifications to my implementation. First, the processing convolution blocks in the corresponding stage have the exact feature sizes for each downsampling stage. Second, to match the stages of ResNet34 and UNet, the max-pooling layer in the first stage was removed, simplifying the design and making the integration of the ResNet encoder and UNet decoder a five-stage network. Third, the channel size after the concatenation in the decoder is set to twice the original channel size to match the size in each encoder stage. Moreover, I tried the bilinear upsampling method to evaluate the difference between the deconvolution approach, which is a part of my additional experiments. The model architecture of my implementation is shown in Figures 1 and 2.

Listing 5: Implementation of UNet.

```
class encoder_block(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(encoder_block, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, out_channels, 3, padding=1)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.conv2 = nn.Conv2d(out_channels, out_channels, 3, padding=1)
        self.bn2 = nn.BatchNorm2d(out_channels)
        self.pool = nn.MaxPool2d(2)

    def forward(self, x):
```

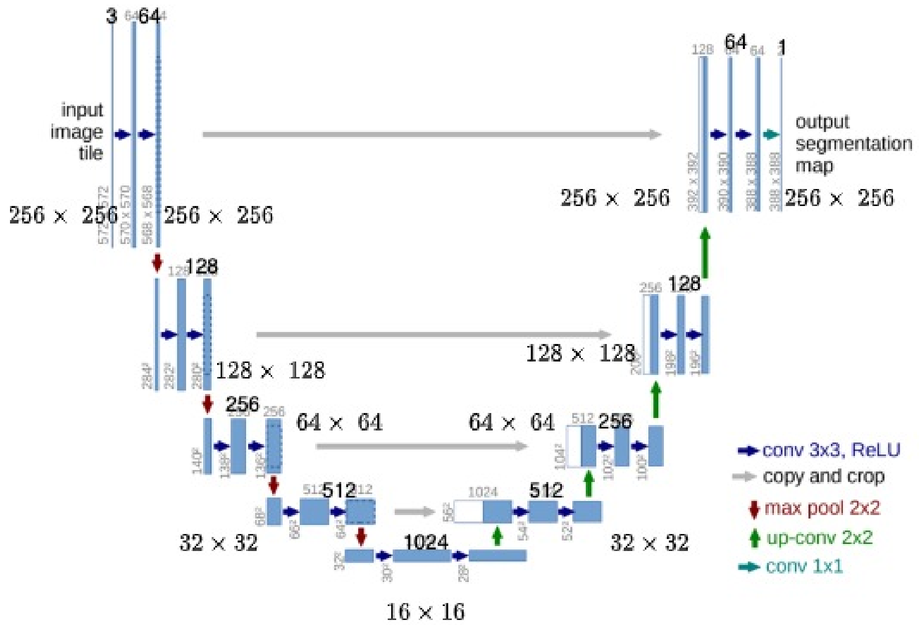


Figure 1: My implementation of UNet [7].

```
        x = F.relu(self.conv1(x))
        x = self.bn1(x)
        x = F.relu(self.conv2(x))
        x = self.bn2(x)
        x_down = self.pool(x)
        return x, x_down

class decoder_block(nn.Module):
    def __init__(self, in_channels, out_channels, upsample='Deconv'):
        super(decoder_block, self).__init__()
        if upsample == 'bilinear':
            self.up = nn.Upsample(
                scale_factor=2, mode='bilinear', align_corners=True)
        elif upsample == 'Deconv':
            self.up = nn.Sequential(
                nn.ConvTranspose2d(
                    in_channels//2, in_channels//2, 2, stride=2),
                nn.BatchNorm2d(in_channels//2),
                nn.ReLU(inplace=True)
            )
        else:
            raise ValueError(
```

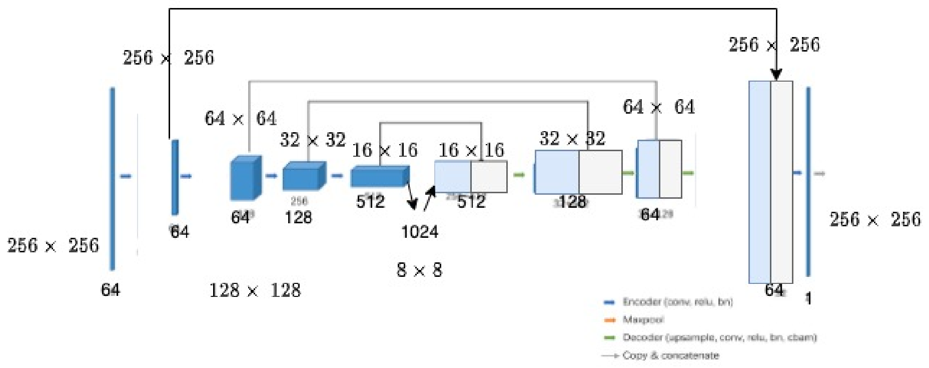


Figure 2: My implementation of ResNet34+UNet [3].

```
                "Invalid␣value␣for␣'upsample'.␣Use␣'bilinear'␣or␣'Deconv'")
        self.conv1 = nn.Conv2d(in_channels, out_channels, 3, padding=1)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.conv2 = nn.Conv2d(out_channels, out_channels, 3, padding=1)
        self.bn2 = nn.BatchNorm2d(out_channels)

class UNet(nn.Module):
    def __init__(self, in_channels=3, out_channels=1, n_channels=64):
        super(UNet, self).__init__()
        self.in_channels = in_channels
        self.out_channels = out_channels

        self.conv1 = nn.Conv2d(in_channels, n_channels, 3, stride=1, padding=1)
        self.down1 = encoder_block(n_channels, n_channels) # 64
        self.down2 = encoder_block(n_channels, n_channels*2) # 128
        self.down3 = encoder_block(n_channels*2, n_channels*4) # 256
        self.down4 = encoder_block(n_channels*4, n_channels*8) # 512

        self.mid = nn.Sequential(
            nn.Conv2d(n_channels*8, n_channels*16, 3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(n_channels*16, n_channels*8, 3, padding=1),
            nn.ReLU(inplace=True),
        )
        self.up1 = decoder_block(n_channels*16, n_channels*4) # 512
        self.up2 = decoder_block(n_channels*8, n_channels*2) # 256
        self.up3 = decoder_block(n_channels*4, n_channels)
        self.up4 = decoder_block(n_channels*2, n_channels)
        self.out = nn.Sequential(
            nn.Conv2d(n_channels, out_channels, 1),
            nn.Sigmoid()
        )

    def forward(self, x):
        assert x.shape[1] == self.in_channels

        x = F.relu(self.conv1(x))
        x1, x1_down = self.down1(x)
        x2, x2_down = self.down2(x1_down)
        x3, x3_down = self.down3(x2_down)
        x4, x4_down = self.down4(x3_down)

        x = self.mid(x4_down) # 512x32x32

        x = self.up1(x, x4) # 256x64x64
        x = self.up2(x, x3) # 128x128x128
        x = self.up3(x, x2) # 64x256x256
        x = self.up4(x, x1) # 64x512x512
        x = self.out(x)
        return x
```

Listing 6: Implementation of ResNet34 + UNet.

```
class Resblock(nn.Module):
    def __init__(self, in_channels, out_channels, downsample=False):
        super(Resblock, self).__init__()
        self.conv0 = nn.Conv2d(in_channels, out_channels, 1)
        self.conv1 = nn.Conv2d(out_channels, out_channels, 3, padding=1)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.conv2 = nn.Conv2d(out_channels, out_channels, 3, padding=1)
        self.bn2 = nn.BatchNorm2d(out_channels)
        self.identity = nn.Identity()
        self.downsample = downsample

    def forward(self, x):
        x = self.conv0(x)
        residual = self.identity(x)
        x = F.relu(self.conv1(x))
        x = self.bn1(x)
        x = F.relu(self.conv2(x))
        x = self.bn2(x)
        x += residual

        if self.downsample:
            x_down = F.max_pool2d(x, 2)
            return x, x_down
        return x

class ResNet34_encoder(nn.Module):
    def __init__(self, in_channels=3, n_channels=64):
        super(ResNet34_encoder, self).__init__()

        self.conv1 = nn.Sequential(
            nn.Conv2d(in_channels, n_channels, 7, stride=2, padding=3),
            nn.ReLU(inplace=True),
            nn.BatchNorm2d(n_channels),
            # nn.MaxPool2d(3, stride=2, padding=1)
        )
        self.conv2 = nn.Sequential(
            Resblock(n_channels, n_channels),
            Resblock(n_channels, n_channels),
            Resblock(n_channels, n_channels, downsample=True)
        )
        self.conv3 = nn.Sequential(
            Resblock(n_channels, n_channels*2),
            Resblock(n_channels*2, n_channels*2),
            Resblock(n_channels*2, n_channels*2),
            Resblock(n_channels*2, n_channels*2, downsample=True)
        )
        self.conv4 = nn.Sequential(
            Resblock(n_channels*2, n_channels*4),
            Resblock(n_channels*4, n_channels*4),
            Resblock(n_channels*4, n_channels*4),
            Resblock(n_channels*4, n_channels*4),
            Resblock(n_channels*4, n_channels*4),
            Resblock(n_channels*4, n_channels*4, downsample=True)
        )
        self.conv5 = nn.Sequential(
            Resblock(n_channels*4, n_channels*8),
            Resblock(n_channels*8, n_channels*8),
            Resblock(n_channels*8, n_channels*8, downsample=True)
        )
    def forward(self, x):
        x = self.conv1(x)
        x1, x1_down = self.conv2(x)
        x2, x2_down = self.conv3(x1_down)
        x3, x3_down = self.conv4(x2_down)
        x4, x4_down = self.conv5(x3_down)
        return x1, x2, x3, x4, x4_down

class decoder_block(nn.Module):
    def __init__(self, in_channels, out_channels, upsample='Deconv'):
        super(decoder_block, self).__init__()
        if upsample == 'bilinear':
            self.up = nn.Upsample(
                scale_factor=2, mode='bilinear', align_corners=True)
        elif upsample == 'Deconv':
            self.up = nn.Sequential(
                nn.ConvTranspose2d(
                    in_channels//2, in_channels//2, 2, stride=2),
                nn.BatchNorm2d(in_channels//2),
                nn.ReLU(inplace=True)
            )
        else:
            raise ValueError(
                "Invalid␣value␣for␣'upsample'.␣Use␣'bilinear'␣or␣'Deconv'")
        self.conv1 = nn.Conv2d(in_channels, out_channels, 3, padding=1)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.conv2 = nn.Conv2d(out_channels, out_channels, 3, padding=1)
        self.bn2 = nn.BatchNorm2d(out_channels)

    def forward(self, x1, x2):
        x1 = self.up(x1)
        x_cat = torch.cat([x1, x2], dim=1)
        x = F.relu(self.conv1(x_cat))
        x = self.bn1(x)
        x = F.relu(self.conv2(x))
        x = self.bn2(x)
        return x

class ResNet34_UNet(nn.Module):
    def __init__(self, in_channels=3, out_channels=1, n_channels=64):
        super(ResNet34_UNet, self).__init__()
        self.in_channels = in_channels
        self.out_channels = out_channels

        self.encoder = ResNet34_encoder(in_channels, n_channels)

        self.mid = nn.Sequential(
            nn.Conv2d(n_channels*8, n_channels*16, 3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(n_channels*16, n_channels*8, 3, padding=1),
            nn.ReLU(inplace=True),
        )
        self.up1 = decoder_block(n_channels*16, n_channels*4) # 512
        self.up2 = decoder_block(n_channels*8, n_channels*2) # 256
        self.up3 = decoder_block(n_channels*4, n_channels)
        self.up4 = decoder_block(n_channels*2, n_channels)
        self.up5 = nn.Sequential(
            nn.Upsample(scale_factor=2, mode='bilinear', align_corners=True),
            nn.Conv2d(n_channels, n_channels, 3, padding=1),
            nn.BatchNorm2d(n_channels),
            nn.ReLU(inplace=True),
        )
        self.out = nn.Sequential(
            nn.Conv2d(n_channels, out_channels, 1),
            nn.Sigmoid()
        )

    def forward(self, x):
        assert x.shape[1] == self.in_channels

        x1, x2, x3, x4, x4_down = self.encoder(x)

        x = self.mid(x4_down) # 512x16x16

        x = self.up1(x, x4) # 256x32x32
        x = self.up2(x, x3) # 128x64x64
        x = self.up3(x, x2) # 64x128x128
        x = self.up4(x, x1) # 64x256x256
        x = self.up5(x) # 64x512x512
        x = self.out(x) # 1x512x512
        return x
```

3 Data Preprocessing

In this section, I will introduce the preprocessing techniques used in the experiments. The Python library Albumentations [1], which is faster with a rich variety of transform operations, is used. Moreover, it supports the simultaneous process of the inputs and targets with shared transformations in a Python dictionary format, making the processing easier to utilize. The preprocessing techniques for training I have used include rotation, crop, blur, and elastic transformation, which do not overall corrupt the images' content but effectively extend the datasets with more variety. For the computations of training, validating, and testing, two preprocessing techniques: resizing and normalization are used in these tasks to reduce the computational cost and data redundancy.

The implementation is shown in Listing 7.

Listing 7: Implementation of preprocessing.

```
def load_dataset(args, mode):
    if mode == "train":
        transform = A.Compose([
            A.Resize(256, 256),
            A.Flip(p=0.5),
            A.RandomRotate90(p=0.5),
            A.RandomResizedCrop(256, 256, scale=(0.8, 1.0)),
            A.ShiftScaleRotate(p=0.5),

            A.GaussianBlur(blur_limit=(3, 7), p=0.5),

            A.ElasticTransform(p=0.5),
        ])
    elif mode == "valid":
        transform = A.Compose([
            A.Resize(256, 256),
        ])
    elif mode == "test":
        transform = A.Compose([
            A.Resize(256, 256),
        ])
    # implement the load dataset function here
    dataset = OxfordPetDataset(
        args.data_path, mode = mode,
        transform = transform)
    shuffle = True if mode == "train" else False
    loader = DataLoader(dataset,
        batch_size = args.batch_size,
        shuffle = True)

    return loader
```

4 Analyze the experiment results

In all experiments, the number of epochs, mini-batch size, and learning rate are set to 200, 32, and 10^{-3}, respectively. Training is optimized using the Adam optimizer with a cosine annealing scheduler to dynamically adjust the learning rate for improved convergence. Loss function is Binary Cross Entropy loss. The input image size is resized to 256×256 for all experiments. AMP is used for accelerating the training process.

The learning curves are presented in Figures 3 and 4, illustrating the training progress of both models. My implementation achieves a Dice score exceeding 90% for both models, demonstrating effective learning and segmentation capability. From the results, we observe that ResNet34+U-Net converges faster in the early stages of training, benefiting from the strong feature extraction capabilities of the ResNet34 backbone. However, U-Net achieves better performance in later epochs, suggesting that its symmetric encoder-decoder structure allows for more refined spatial feature learning over time. This trade-off between convergence speed and final accuracy highlights the differences in learning dynamics between the two architectures.

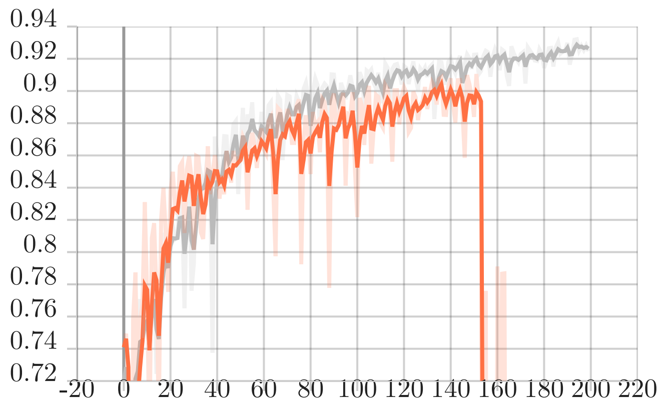


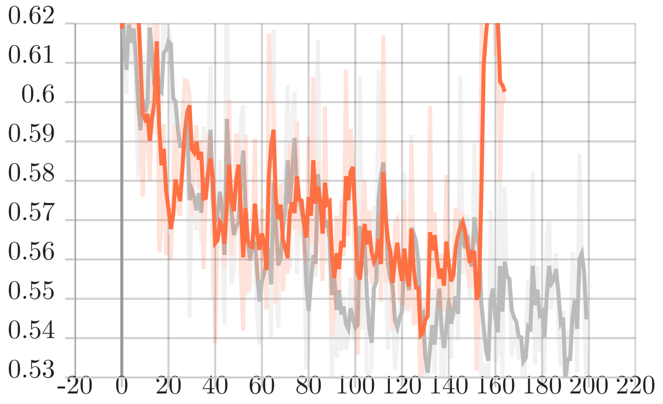
Figure 3: Comparison of UNet (gray) vs. ResNet34+UNet (orange) using dice score. It is important to note that the values of ResNet34+U-Net became NaN after approximately 150 epochs. This occurred because the loss value became too small or too large during AMP training, causing the gradient to approach zero or overflow. Despite this, the earlier training stages, where the model performed well, are still usable for comparison.

From a visual perspective, as shown in Figure 5, the segmentation results reveal certain limitations. The model struggles with transparent and fine structures (whiskers), where details are often missed. Additionally, regions in the background that share similar colors with the object (white building as part of the white dog) are sometimes misclassified as the foreground. These challenges are from coarse annotations and insufficient supervision for hard samples, which may lead to ambiguity in learning fine-grained details.

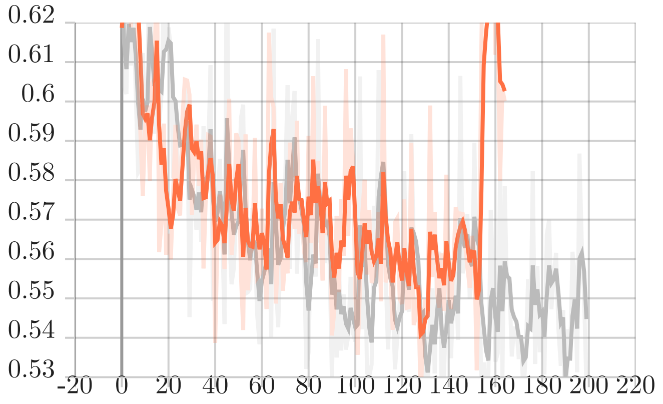
Furthermore, I compared two different upsampling approaches: bilinear interpolation and deconvolution, as shown in Figures 6 and 7. From the figures, we observe an unexpected drop in the Dice score and an increase in loss for U-Net with bilinear interpolation, suggesting instability of training which may caused by the AMP. In contrast, the model using deconvolution converges slightly slower initially but achieves marginally better performance in the later stages. These results highlight the importance of selecting an appropriate upsampling strategy, as it can impact model stability, convergence behavior, and overall segmentation performance.

5 Execution steps

To train, evaluate, and inference the model, please follow the instructions in Listing 8.



(a) training loss



(b) validation loss

Figure 4: Learning Curves comparison. UNet and ResNet34+UNet are shown as gray and orange, respectively. It is important to note that the values of ResNet34+U-Net became NaN after approximately 150 epochs. This occurred because the loss value became too small during AMP training, causing the gradient to approach zero. Despite this, the earlier training stages, where the model performed well, are still usable for comparison.

Listing 8: Implementation of execution.

```
# Check readme.md
# 1. Install
# create your own virtual environment
conda create -n lab2 python=3.8 -y
conda activate lab2
# please install Pytorch first then execute the following instructions
pip install torch==1.12.1+cu102 torchvision==0.13.1+cu102 torchaudio==0.12.1 \ -

pip install -r requirements.txt

# 2. train
python src/train.py --model unet --use_gpu --gpu 0
# Other instructions
# -e 200
# -lr 1e-3
# -b 32
# --data_path ./dataset # dataset's path
# If you want to accelerate the training process, use "--amp"

# The model will be stored in saved_model

# 3. inference
python src/inference.py --model unet --ckpt YOUR/MODEL/PATH --use_gpu --gpu 0
# --data_path ./dataset # dataset's path
# --save_path ./results/inferenc # saved images' dir
# --ckpt saved_models/train/unet/checkpoints/model_best.pth
# batch size only accept 1 -> -b =1
```

6 Discussion

The experiments show that ResNet34+U-Net converges faster, but the original U-Net achieves better final performance. Additionally, some limitations commonly occur in hard samples due to image content and annotation quality. Furthermore, the experiments on the upsampling module indicate that using conventional interpolation allows for faster initial convergence. Combining these insights, integrating interpolation with learnable deconvolution layers and designing a more efficient encoder with residual connections could be a promising approach. The learnable deconvolution layers can help refine feature reconstruction while maintaining computational efficiency, addressing the overfitting issues observed with pure interpolation. Meanwhile, an optimized residual encoder, potentially with channel/spatial attention mechanisms or vision transformer, could enhance feature extraction while preserving fast convergence. This hybrid design may lead to better generalization and improved segmentation accuracy. Last but not least, cleaning the data first or providing a efficient loss to select or filter the hard samples may be a more simple way to enhance the overall performance.

Through this lab, I have gained a deeper understanding of the U-Net architecture, which integrates an encoder-decoder structure with a multi-scale design for effective feature extraction and spatial information preservation, no matter using it or ResNet34+UNet. Its versatility has inspired variants like V-Net [5] for volumetric medical image segmentation using 3D convolutions and M-Net [4], which enhances multi-scale learning by incorporating downsampled images as additional inputs. Moreover, U-Net serves as a key backbone in modern deep learning applications, including diffusion models [2], where it plays a crucial role in the denoising process, refining noisy inputs to generate high-quality outputs. Its adaptability beyond segmentation highlights its significance in state-of-the-art computer vision research.

References

[1] Alexander Buslaev, Vladimir I. Iglovikov, Eugene Khvedchenya, Alex Parinov, Mikhail Druzhinin, and Alexandr A. Kalinin. Albumentations: Fast and flexible image augmentations. *Information*, 11(2), 2020.

[2] Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models. *Advances in neural information processing systems*, 33:6840–6851, 2020.

[3] Zeping Huang, Enze Qu, Yishuang Meng, Man Zhang, Qiuwen Wei, Xianghui Bai, and Xinling Zhang. Deep learning-based pelvic levator hiatus segmentation from ultrasound images. *European Journal of Radiology Open*, 9:100412, 2022.

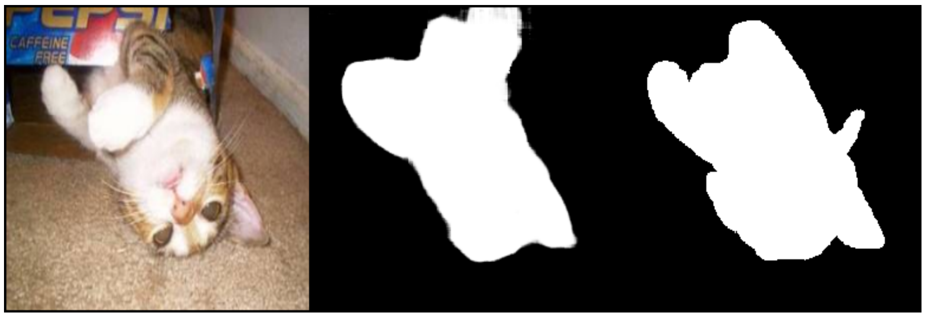
[4] Raghav Mehta and Jayanthi Sivaswamy. M-net: A convolutional neural network for deep brain structure segmentation. In *2017 IEEE 14th international symposium on biomedical imaging (ISBI 2017)*, pages 437–440. Ieee, 2017.

[5] Fausto Milletari, Nassir Navab, and Seyed-Ahmad Ahmadi. V-net: Fully convolutional neural networks for volumetric medical image segmentation. In *2016 fourth international conference on 3D vision (3DV)*, pages 565–571. Ieee, 2016.

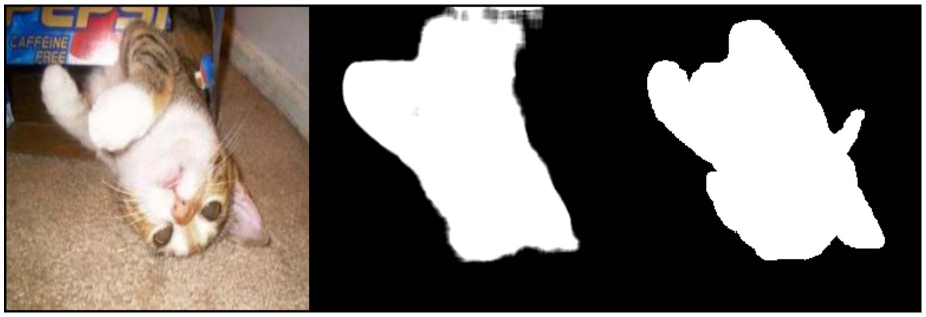
[6] Omkar M Parkhi, Andrea Vedaldi, Andrew Zisserman, and C. V. Jawahar. Cats and dogs. In *2012 IEEE Conference on Computer Vision and Pattern Recognition*, pages 3498–3505, 2012.

[7] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *Medical image computing*

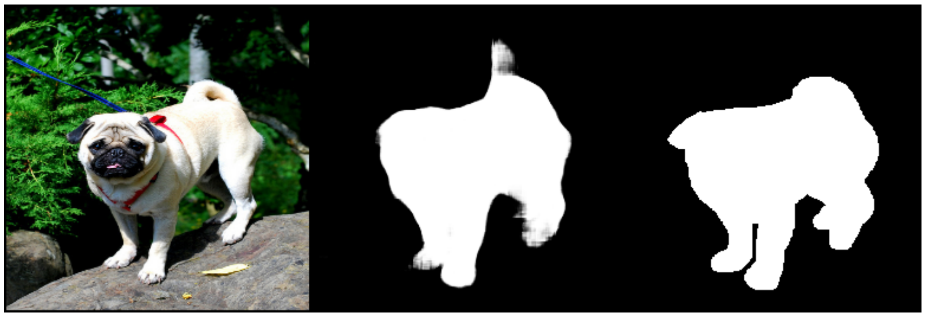
and computer-assisted intervention–MICCAI 2015: 18th international conference, Munich, Germany, October 5-9, 2015, proceedings, part III 18, pages 234–241. Springer, 2015.



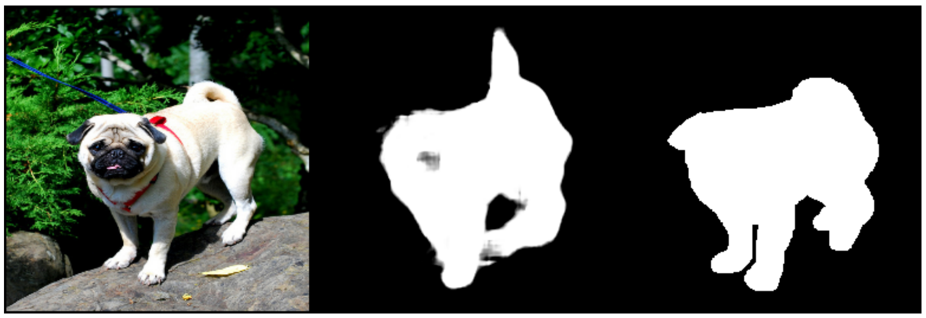
(a) Segmentation using UNet on cat image.



(b) Segmentation using ResNet34+UNet on cat image.



(c) Segmentation using UNet on dog image.



(d) Segmentation using ResNet34+UNet on dog image.

Figure 5: Visual comparison of UNet and ResNet34+UNet.

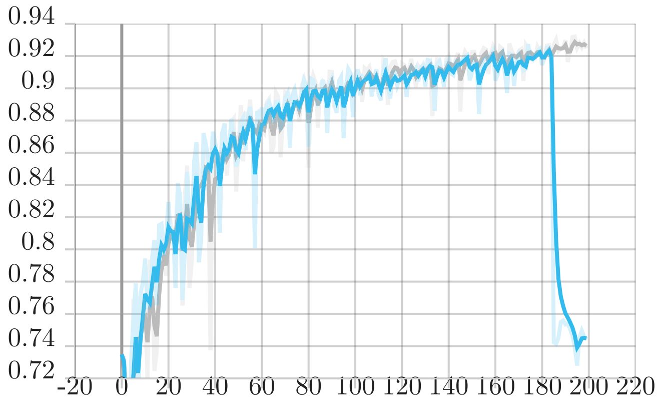
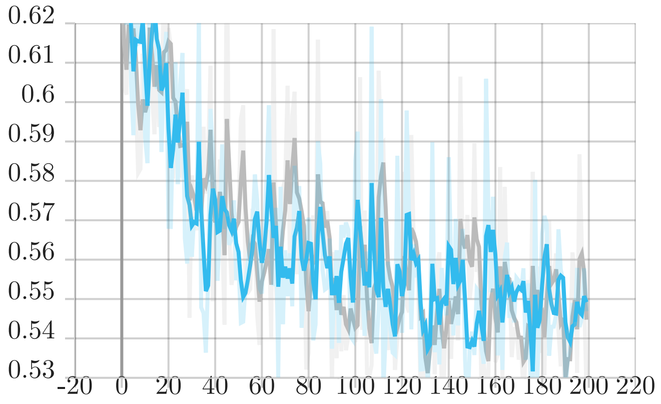
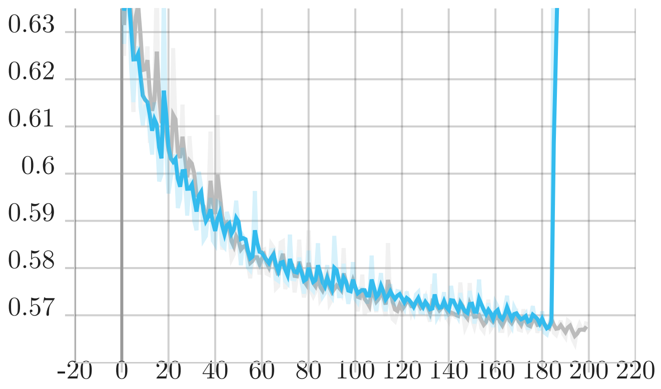


Figure 6: Comparison of UNet with deconvolution for upsampling (gray) vs. UNet with bilinear for upsampling (blue). The exploding loss in validation curve may caused by AMP training.



(a) training loss



(b) validation loss

Figure 7: Learning Curves comparison of UNet with deconvolution for upsampling (gray) vs. UNet with bilinear for upsampling (blue). The exploding loss in validation curve may caused by AMP training.