Binary Semantic Segmentation
Lab2 Report

Student: PO-JUI, CHU
ID: 11028141

Deep Learning
Spring 2025
Date Submitted: March 25, 2025

Contents

List of Figures

List of Tables

Lab 2 Introduction

Binary Semantic Segmentation models typically classify each pixel in an image into two categories: 'foreground' and 'background'. They are commonly used in applications where a clear distinction between target objects and non-target areas is required, such as lesion segmentation in medical images or road area recognition.

In this assignment, the task is to identify the locations of animals in images. The model's prediction is a matrix containing only 0s and 1s, where 1 indicates that the model judges the location as an object and 0 represents the background.

In this experiment, two Binary Semantic Segmentation models (UNet and ResNet34+UNet) are trained using the Oxford-IIIT Pet Dataset. The Dice Score is used as the accuracy metric — the higher the score, the more precise the prediction.

1 Implementation Details

1.1 Training, Evaluation, and Inference Process

1.1.1 Training Process

During the training phase, the training and validation datasets are created using the `load_dataset` function in `oxford_pet.py` and processed in batches using `DataLoader`. The program first declares the model architecture (UNet or ResNet34_UNet) based on the selection, then employs the Adam optimizer, loss functions (Binary Cross Entropy combined with Dice Loss or Tversky Loss), and tools such as `pandas` and `tqdm` to record the training progress.

For each epoch, the loss and Dice score are computed and accumulated. During the validation phase, the model weights corresponding to the highest Dice Score are saved, and the training and validation results are recorded in an Excel file (`model_performence.xlsx`) for later graph analysis.

Train.py code is shown in Figure 1.

```
def train(args):
    # Load datasets
    train_dataset = load_dataset(args.data_path, mode="train")
    valid_dataset = load_dataset(args.data_path, mode="valid")
    train_loader = DataLoader(train_dataset, batch_size=args.batch_size, shuffle=True)
    valid_loader = DataLoader(valid_dataset, batch_size=1, shuffle=False)

    # Build model
    if args.model == "unet":
        model = UNetModel(in_channels=3, out_channels=1).to(args.device)
    else:
        model = ResNet34_UNet(in_channels=3, out_channels=1).to(args.device)

    optimizer = optim.Adam(model.parameters(), lr=args.learning_rate)

    best_dice = 0.0  # Track the best validation Dice score

    # Create a list to record performance per epoch
    performance_records = []
```

```python
for epoch in range(1, args.epochs + 1):
    model.train()
    epoch_losses, epoch_dices = [], []
    epoch_start_time = time.time()
    progress_bar = tqdm(enumerate(train_loader), total=len(train_loader), bar_format='{l_bar}{bar}')

    for i, batch in progress_bar:
        images = batch["image"].to(args.device)
        masks = batch["mask"].to(args.device)
        optimizer.zero_grad()

        outputs = model(images)
        loss = compute_loss(outputs, masks, loss_type=args.loss_type)
        loss.backward()
        optimizer.step()

        epoch_losses.append(loss.item())

        with torch.no_grad():
            epoch_dices.append(compute_dice_score(outputs, masks))

        progress_bar.set_description(
            f"Epoch {epoch}/{args.epochs} | Loss: {loss.item():.4f} | Dice: {epoch_dices[-1]:.4f} | Iter: {i+1}/{len(train_loader)}"
        )
```

```python
# Evaluate on validation set
val_losses, val_dices = evaluate(model, valid_loader, args.device, loss_type=args.loss_type)
avg_train_loss = np.mean(epoch_losses)
avg_val_loss = np.mean(val_losses)
avg_train_dice = np.mean(epoch_dices)
avg_val_dice = np.mean(val_dices)

print(f"Epoch {epoch} => Train Loss: {avg_train_loss:.4f}, Train Dice: {avg_train_dice:.4f} | Valid Loss: {avg_val_loss:.4f},
Valid Dice: {avg_val_dice:.4f}")

# Record performance for current epoch
performance_records.append({
    "Epoch": epoch,
    "Train Loss": avg_train_loss,
    "Train Dice": avg_train_dice,
    "Valid Loss": avg_val_loss,
    "Valid Dice": avg_val_dice
})

# Save model if validation dice score > 0.9 and improved
if avg_val_dice > 0.9 and avg_val_dice > best_dice:
    best_dice = avg_val_dice
    save_dir = os.path.join("..", "saved_models")
    if not os.path.exists(save_dir):
        os.makedirs(save_dir)
    model_path = os.path.join(save_dir, f"{args.model}.pth")
    torch.save(model.state_dict(), model_path)
```

```python
# Ensure outputs folder exists
outputs_dir = "outputs"
if not os.path.exists(outputs_dir):
    os.makedirs(outputs_dir)

# Convert performance records to DataFrame and rename columns based on model type
df_new = pd.DataFrame(performance_records)
if args.model == "unet":
    new_columns = {
        "Train Loss": "Unet Train Loss",
        "Train Dice": "Unet Train Dice",
        "Valid Loss": "Unet Val Loss",
        "Valid Dice": "Unet Val Dice"
    }
    desired_columns = ["Epoch", "Unet Train Loss", "Unet Train Dice", "Unet Val Loss", "Unet Val Dice"]
else:
    new_columns = {
        "Train Loss": "ResNet34_Unet Train Loss",
        "Train Dice": "ResNet34_Unet Train Dice",
        "Valid Loss": "ResNet34_Unet Val Loss",
        "Valid Dice": "ResNet34_Unet Val Dice"
    }
    desired_columns = ["Epoch", "ResNet34_Unet Train Loss", "ResNet34_Unet Train Dice", "ResNet34_Unet Val Loss", "ResNet34_Unet Val Dice"]

df_new = df_new.rename(columns=new_columns)
df_new.set_index("Epoch", inplace=True)

output_excel = os.path.join(outputs_dir, "model_performence.xlsx")

if os.path.exists(output_excel):
    df_existing = pd.read_excel(output_excel)
    df_existing.set_index("Epoch", inplace=True)
    # Overwrite columns corresponding to the current model
    for col in new_columns.values():
        df_existing[col] = df_new[col]
    # Also add new epochs if any exist in df_new but not in df_existing
    df_merged = df_existing.combine_first(df_new)
else:
    df_merged = df_new

df_merged.reset_index(inplace=True)
df_merged = df_merged[desired_columns]
```

Figure 1: Train.py code.

1.1.2 Evaluation Process

In evaluation, the model is set to evaluation mode (which turns off BatchNorm updates and dropout), and `torch.no_grad()` is used to save memory.

During evaluation, loss (using BCE combined with Dice or Tversky loss) and Dice scores are computed for each batch, and the final performance indicator is the average of these scores.

```python
import numpy as np
import torch
import torch.nn as nn
from utils import compute_dice_score, compute_loss

def evaluate(model: nn.Module, dataloader, device: torch.device, loss_type: str = 'bce_dice'):
    model.eval()
    losses, dice_scores = [], []
    with torch.no_grad():
        for batch in dataloader:
            images = batch["image"].to(device)
            masks = batch["mask"].to(device)
            pred_masks = model(images)
            loss = compute_loss(pred_masks, masks, loss_type=loss_type)
            losses.append(loss.item())
            dice_scores.append(compute_dice_score(pred_masks, masks))

    avg_loss = np.mean(losses)
    avg_dice = np.mean(dice_scores)

    return losses, dice_scores
```

Figure 2: Evaluation process.

1.1.3 Inference Process

During inference, based on the called model name ('unet.pth' or 'resnet34_unet.pth'), the corresponding pretrained model is loaded. Then, inference is performed on the test dataset by calculating the Dice score for each sample, and finally, the average Dice score for the test set is output.

```python
def main():
    args = get_args()
    device = torch.device(args.device if torch.cuda.is_available() else 'cpu')
    if args.model == "unet.pth":
        model = UNetModel(in_channels=3, out_channels=1).to(device)
    elif args.model == "resnet34_unet.pth":
        model = ResNet34_UNet(in_channels=3, out_channels=1).to(device)
    else:
        raise ValueError("Unsupported model file")

    model_path = os.path.join("..", "saved_models", args.model)
    if not os.path.exists(model_path):
        raise FileNotFoundError(f"Model file not found: {model_path}")

    state_dict = torch.load(model_path, map_location=device)
    model.load_state_dict(state_dict)
    print(f"Loaded state dict from {model_path}\n(inference on unet model.)")

    model.eval()

    test_dataset = load_dataset(args.data_path, mode="test")
    test_loader = DataLoader(test_dataset, batch_size=args.batch_size, shuffle=False)

    total_dice = 0.0
    count = 0
```

```python
    with torch.no_grad():
        for batch in test_loader:
            images = batch["image"].to(device)
            masks = batch["mask"].to(device)
            preds = model(images)

            for i in range(images.size(0)):
                dice = compute_dice_score(preds[i], masks[i])
                total_dice += dice
                count += 1

    mean_dice = total_dice / count if count > 0 else 0.0
    print(f"Mean Dice Score on test set: {mean_dice:.4f}")

def get_args():
    parser = argparse.ArgumentParser(description='Test model and compute mean Dice score')
    parser.add_argument('--model', default='unet.pth', choices=["unet.pth", "resnet34_unet.pth"],
                        help='Name of the model file in ../saved_models directory')
    parser.add_argument('--data_path', type=str, default="../dataset/oxford-iiit-pet/",
                        help='Path to the input data')
    parser.add_argument('--batch_size', '-b', type=int, default=1,
                        help='Batch size for testing')
    parser.add_argument('--device', type=str, default='cuda',
                        help='Device to use for testing')
    return parser.parse_args()
```

Figure 3: Inference process.

1.2 Model Architecture and Design Details

1.2.1 UNet Model

The UNet architecture adopts a classic encoder–decoder structure. The encoder consists of multiple layers of double convolution blocks, as shown in Figure 4.

```python
class ConvBlock_Double(nn.Module):

    def __init__(self, in_channels, out_channels):
        super(ConvBlock_Double, self).__init__()
        self.double_conv = nn.Sequential(
            nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=1, padding=1, bias=False),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True),
            nn.Conv2d(out_channels, out_channels, kernel_size=3, stride=1, padding=1, bias=False),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True)
        )

    def forward(self, x):
        return self.double_conv(x)
```

Figure 4: Double convolution blocks.

There are a total of four blocks, with downsampling performed between layers via max pooling. The bottleneck layer in the middle further abstracts high-level features.

The decoder uses transposed convolutions for upsampling and fuses these with the corresponding skip connections. Finally, a 1×1 convolution outputs the segmentation mask, and a Sigmoid function maps the values to the $[0, 1]$ range. UNet model process is shown in Figure 5.

```python
class UNetModel(nn.Module):

    def __init__(self, in_channels, out_channels, features=[64, 128, 256, 512]):
        super(UNetModel, self).__init__()
        self.encoders = nn.ModuleList()
        self.decoders = nn.ModuleList()
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)

        # Build the encoder (downsampling) path.
        for feature in features:
            self.encoders.append(ConvBlock_Double(in_channels, feature))
            in_channels = feature  # update in_channels for the next layer

        # Bottleneck between encoder and decoder.
        self.bottleneck = ConvBlock_Double(features[-1], features[-1] * 2)

        # Build the decoder (upsampling) path.
        reversed_features = features[::-1]
        for feature in reversed_features:
            # Transposed convolution for upsampling.
            self.decoders.append(
                nn.ConvTranspose2d(feature * 2, feature, kernel_size=2, stride=2)
            )
            # Double convolution after concatenation.
            self.decoders.append(ConvBlock_Double(feature * 2, feature))
```

```python
        # Final convolution to map to the desired output channels.
        self.final = nn.Sequential(
            nn.Conv2d(features[0], out_channels, kernel_size=1),
            nn.Sigmoid()
        )

    def forward(self, x):
        skip_connections = []

        # Encoder: apply each block and store outputs for skip connections.
        for encoder in self.encoders:
            x = encoder(x)
            skip_connections.append(x)
            x = self.pool(x)

        # Bottleneck.
        x = self.bottleneck(x)

        for decoder in self.decoders:
            if isinstance(decoder, nn.ConvTranspose2d):
                x = decoder(x)
                x = torch.cat((x, skip_connections.pop()), dim=1)
            else:
                x = decoder(x)
        x = self.final(x)
        return x
```

Figure 5: UNet.

1.2.2 ResNet34_UNet Model

In my ResNet34_UNet model, I first use the basic module from ResNet34, called `BasicBlock`, as the core building block to form the encoder. Each `BasicBlock` consists of two 3×3 convolutional layers, with each convolution followed by batch normalization and a ReLU activation. Since ResNet34 requires downsampling in some layers, whenever the input and output sizes or the number of channels do not match, the `BasicBlock` applies a 1×1 convolution (with batch normalization) as a shortcut to reduce the dimensions. This shortcut connection helps prevent the vanishing gradient problem and captures multi-scale features more effectively.

```
class BasicBlock(nn.Module):
    def __init__(self, in_channels, out_channels, stride=1):
        super(BasicBlock, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=stride,
                               padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3, stride=1,
                               padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(out_channels)

        self.downsample = None
        if stride != 1 or in_channels != out_channels:
            self.downsample = nn.Sequential(
                nn.Conv2d(in_channels, out_channels, kernel_size=1, stride=stride, bias=False),
                nn.BatchNorm2d(out_channels)
            )

    def forward(self, x):
        identity = x
        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)
        out = self.conv2(out)
        out = self.bn2(out)
        if self.downsample is not None:
            identity = self.downsample(x)
        out += identity
        out = self.relu(out)
        return out
```

Figure 6: BasicBlock in ResNet34_UNet model.

Next, I define a function called `make_layer` to stack multiple `BasicBlocks`. In this function, the first `BasicBlock` uses a larger stride (for example, stride = 2) to perform downsampling, while the remaining blocks use a stride of 1, focusing only on convolution operations. This approach follows the original ResNet34 configuration (with 3, 4, 6, and 3 `BasicBlocks` in each respective layer) to build the encoder.

```
def make_layer(block, in_channels, out_channels, blocks, stride):
    layers = []
    layers.append(block(in_channels, out_channels, stride))
    for _ in range(1, blocks):
        layers.append(block(out_channels, out_channels, stride=1))
    return nn.Sequential(*layers)

class ConvBlock_Double(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(ConvBlock_Double, self).__init__()
        self.double_conv = nn.Sequential(
            nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1, bias=False),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True),
            nn.Conv2d(out_channels, out_channels, kernel_size=3, padding=1, bias=False),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True)
        )

    def forward(self, x):
        return self.double_conv(x)
```

Figure 7: The subfunction.

In the initial part of ResNet34_UNet, a 7×7 convolution (stride $=$ 2, padding $=$ 3) is used first, followed by batch normalization, ReLU, and a 3×3 max pooling. This section serves as the pre-module of ResNet34 for initial low-level feature extraction and spatial downsampling.

Then, the model moves into the encoder section, which mainly uses the `BasicBlock`. As mentioned before, when the sizes or channels do not match between convolution layers, a 1×1 convolution is applied to ensure that the features can be properly added together. The encoder is divided into four levels according to the ResNet34 configuration. The first level keeps the same size and channel count, while the subsequent levels progressively reduce the spatial dimensions and increase the number of channels, ultimately reducing the image to an 8×8 size with 512 channels.

```
class UpBlock(nn.Module):
    def __init__(self, in_channels, skip_channels, out_channels):
        super(UpBlock, self).__init__()
        # Transposed convolution to upsample the feature map
        self.up = nn.ConvTranspose2d(in_channels, out_channels, kernel_size=2, stride=2)
        # Double convolution after concatenating the skip connection
        self.conv = ConvBlock_Double(out_channels + skip_channels, out_channels)

    def forward(self, x, skip):
        x = self.up(x)
        x = torch.cat([skip, x], dim=1)
        x = self.conv(x)
        return x
```

Figure 8: Up block.

```
class ResNet34_UNet(nn.Module):
    def __init__(self, in_channels=3, out_channels=1):
        super(ResNet34_UNet, self).__init__()
        # Initial convolutional layer: 7x7 conv with stride 2 then maxpool
        # Input: 256x256 -> conv1 output: 128x128 with 64 channels
        self.conv1 = nn.Conv2d(in_channels, 64, kernel_size=7, stride=2, padding=3, bias=False)
        self.bn1 = nn.BatchNorm2d(64)
        self.relu = nn.ReLU(inplace=True)
        # MaxPool reduces the resolution to 64x64
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)

        # Encoder: ResNet layers
        self.layer1 = make_layer(BasicBlock, 64, 64, blocks=3, stride=1)    # Output: 64, 64x64
        self.layer2 = make_layer(BasicBlock, 64, 128, blocks=4, stride=2)   # Output: 128, 32x32
        self.layer3 = make_layer(BasicBlock, 128, 256, blocks=6, stride=2)  # Output: 256, 16x16
        self.layer4 = make_layer(BasicBlock, 256, 512, blocks=3, stride=2)  # Output: 512, 8x8
```

Figure 9: Encoder structure.

After the encoder, a `ConvBlock_Double` is used to further process the feature map from layer4, keeping the channel number at 512. This helps to lower the computational load while integrating high-level features.

In the decoder, `UpBlock` modules are used (as shown in Figure 8). Each `UpBlock` first upsamples the feature map using a transposed convolution and then directly concatenates it with the corresponding features from the encoder. After concatenation, a double convolution block is applied to fuse the features, gradually restoring the resolution. This process is repeated until the original input size is reached. Finally, a 1×1 convolution is used to generate the segmentation mask, and a Sigmoid function maps the output to the range $[0, 1]$ for the final segmentation result.

```python
# Bottleneck: using double conv block to keep channels at 512 (no expansion to 1024)
self.bottleneck = ConvBlock_Double(512, 512)  # Output: 512, 8x8

# Decoder: upsample and merge skip connections directly (no 1x1 projection)
self.up1 = UpBlock(512, skip_channels=256, out_channels=256)  # Upsample from 8x8 to 16x16, merge with layer3
self.up2 = UpBlock(256, skip_channels=128, out_channels=128)  # Upsample from 16x16 to 32x32, merge with layer2
self.up3 = UpBlock(128, skip_channels=64, out_channels=64)    # Upsample from 32x32 to 64x64, merge with layer1
self.up4 = UpBlock(64, skip_channels=64, out_channels=64)     # Upsample from 64x64 to 128x128, merge with conv1

# Final block: two transposed convolutions to upsample from 128x128 to 256x256,
# then 1x1 convolution to generate the segmentation map with a Sigmoid activation.
self.last = nn.Sequential(
    nn.ConvTranspose2d(64, 64, kernel_size=2, stride=2),  # 128x128 -> 256x256
    nn.BatchNorm2d(64),
    nn.ReLU(inplace=True),
    nn.Conv2d(64, out_channels, kernel_size=1),
    nn.Sigmoid()
)
```

```python
    def forward(self, x):
        # Encoder
        x0 = self.relu(self.bn1(self.conv1(x)))  # x0: 64, 128x128
        x1 = self.maxpool(x0)                     # x1: 64, 64x64
        x1 = self.layer1(x1)                      # x1: 64, 64x64
        x2 = self.layer2(x1)                      # x2: 128, 32x32
        x3 = self.layer3(x2)                      # x3: 256, 16x16
        x4 = self.layer4(x3)                      # x4: 512, 8x8

        # Bottleneck
        b = self.bottleneck(x4)                   # b: 512, 8x8

        # Decoder with skip connections
        d1 = self.up1(b, x3)                      # d1: 256, 16x16
        d2 = self.up2(d1, x2)                     # d2: 128, 32x32
        d3 = self.up3(d2, x1)                     # d3: 64, 64x64
        d4 = self.up4(d3, x0)                     # d4: 64, 128x128

        out = self.last(d4)                       # out: out_channels, 256x256
        return out

# Example usage:
if __name__ == "__main__":
    model = ResNet34_UNet(in_channels=3, out_channels=1)
    x = torch.randn(1, 3, 256, 256)
    y = model(x)
    print("Output shape:", y.shape)  # Expected: torch.Size([1, 1, 256, 256])
```

Figure 10: Architecture in ResNet34_UNet model.

1.2.3 Loss Function

In the experiments, BCE combined with Dice Loss or Tversky Loss is used as the loss function to balance the imbalance between foreground and background. Experiments showed that for this dataset, BCE with Dice Loss works better, which may be related to the larger proportion of objects present.

This loss design considers both pixel-level accuracy and region-level overlap, effectively addressing the severe imbalance between foreground and background.

```python
def compute_dice_loss(pred_mask: torch.Tensor, gt_mask: torch.Tensor, eps: float = 1e-8) -> torch.Tensor:
    intersection = torch.sum(pred_mask * gt_mask) + eps
    union = torch.sum(pred_mask) + torch.sum(gt_mask) + eps
    loss = 1 - (2 * intersection / union)
    return loss

def compute_tversky_loss(pred_mask: torch.Tensor, gt_mask: torch.Tensor, alpha: float = 0.5,
                         beta: float = 0.5, eps: float = 1e-6) -> torch.Tensor:
    pred_flat = pred_mask.view(-1)
    gt_flat = gt_mask.view(-1)
    TP = (pred_flat * gt_flat).sum()
    FP = ((1 - gt_flat) * pred_flat).sum()
    FN = (gt_flat * (1 - pred_flat)).sum()
    tversky_index = (TP + eps) / (TP + alpha * FP + beta * FN + eps)
    return 1 - tversky_index
```

Figure 11: Loss function: BCE with Dice Loss or Tversky Loss.

```python
def compute_tversky_loss(pred_mask: torch.Tensor, gt_mask: torch.Tensor, alpha: float = 0.5, beta: float = 0.5, eps: float = 1e-6) -> torch.Tensor:
    pred_flat = pred_mask.view(-1)
    gt_flat = gt_mask.view(-1)
    TP = (pred_flat * gt_flat).sum()
    FP = ((1 - gt_flat) * pred_flat).sum()
    FN = (gt_flat * (1 - pred_flat)).sum()
    tversky_index = (TP + eps) / (TP + alpha * FP + beta * FN + eps)
    return 1 - tversky_index

def compute_loss(pred_mask: torch.Tensor, gt_mask: torch.Tensor, loss_type: str = 'bce_dice') -> torch.Tensor:
    bce = nn.BCELoss()(pred_mask, gt_mask)
    if loss_type == 'bce_dice':
        extra_loss = compute_dice_loss(pred_mask, gt_mask)
    elif loss_type == 'bce_tversky':
        extra_loss = compute_tversky_loss(pred_mask, gt_mask)
    else:
        raise ValueError("Unsupported loss_type")
    return bce + extra_loss
```

Figure 12: Choose: BCE with Dice Loss or Tversky Loss.

1.2.4 Model Performance Chart (graph.py)

During training, the code records the performance of each epoch and finally outputs the results to an Excel file (`model_performence.xlsx`). This file is then used by `graph.py` to generate trend charts for loss and accuracy (Dice score) and to calculate the minimum, maximum, initial, and final percentage changes of the metrics.

2 Data Preprocessing

2.1 Data Augmentation and Transformation

I used the transforms provided by the `torchvision` package for data augmentation, which integrates well with the PyTorch ecosystem, is highly efficient, and supports parallel processing. However, since `torchvision` lacks direct support for synchronous transformation of images and masks, I customized two transformation functions (`train_transform` and `valid_transform`) using `torchvision.transforms.functional` to manually perform geometric transformations, ensuring that the image and mask are transformed in sync. Note that color adjustments (e.g., `ColorJitter`) are applied only to the image.

Training Phase (`train_transform`):

- Convert the numpy array to a PIL image for further processing.
- Apply random horizontal flip, and random resizing with cropping (`RandomResizedCrop`) to enhance the model's robustness to scale and position changes.
- Apply random affine transformations (simulating shift, scale, and rotation) to further increase data diversity.
- Use Color Jitter to adjust the image's brightness, contrast, saturation, and hue.
- Finally, convert the image to a tensor and perform normalization (based on ImageNet's mean and standard deviation), and convert the mask to a tensor while adding a channel dimension.

Validation and Testing Phase (`valid_transform`):

- Only adjust the image size and perform normalization.

2.2 What Makes Your Method Unique?

Among the unique approaches, I customized a synchronous augmentation function that ensures the image and its corresponding mask remain aligned during geometric transformations. Additionally, I added `ColorJitter` to increase data diversity by adjusting the image's HSI without changing the positions of objects, improving the model's generalization ability.

3 Analyze the Experiment Results

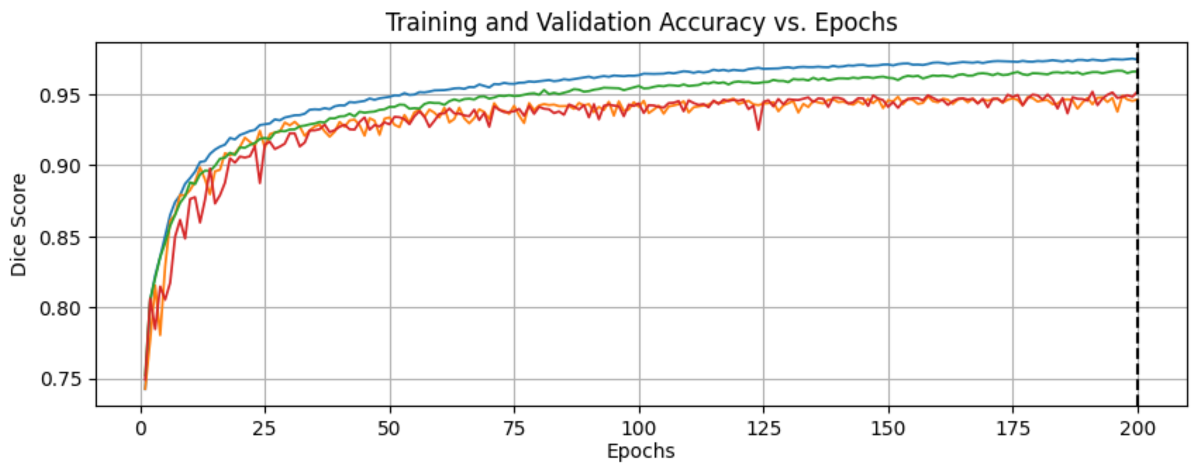
3.1 Experiment Setup and Hyperparameter Tuning

Experiment model hyperparameters:

- Epochs: Preset to 200.
- Batch size: Set based on GPU memory (e.g., 32).
- Learning rate: Set to 1×10^{-3}.
- Loss function: Choose 'bce_dice' or 'bce_tversky' depending on the experiment.

3.2 Observations and Analysis

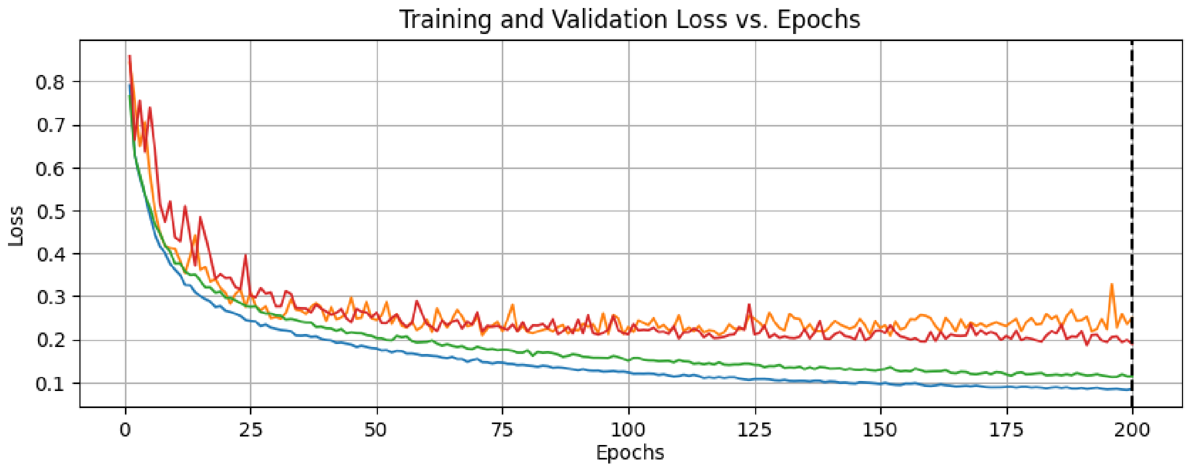
From the experimental results, it can be seen that ResNet34_UNet improves more rapidly in the early stages of training, indicating that its structure helps quickly capture multi-scale features. However, in the later stages, UNet shows a slightly higher Mean Dice Score on the training dataset, with the difference between the two being about 1%, while on the validation set the difference is minimal. This suggests that ResNet34_UNet may have better generalization ability.



Model	Min	Max	Start	End	Change (%)
— Unet Train Dice	0.7424	0.9752	0.7424	0.9746	↑ 31.28%
— Unet Val Dice	0.7426	0.9491	0.7426	0.9463	↑ 27.43%
— ResNet34_Unet Train Dice	0.7521	0.9667	0.7521	0.9662	↑ 28.47%
— ResNet34_Unet Val Dice	0.7492	0.9519	0.7492	0.9518	↑ 27.04%

Table 1: Observations from dice chart.

Furthermore, as shown in Table 2, UNet generally outperforms ResNet34_UNet in terms of training loss, but during the later validation stages an upward trend is observed — overfitting appears around epoch 120. This aligns with our inference that the generalization ability of the ResNet34_UNet architecture might be better.



Model	Min	Max	Start	End	Change (%)
— Unet Train Loss	0.0819	0.7902	0.7902	0.0842	↓ 89.34%
— Unet Val Loss	0.2084	0.843	0.843	0.2495	↓ 70.40%
— ResNet34_Unet Train Loss	0.1119	0.7653	0.7653	0.1133	↓ 85.20%
— ResNet34_Unet Val Loss	0.1862	0.8582	0.8582	0.1892	↓ 77.95%

Table 2: Observations from loss chart.

3.3 Inference Results

The experimental results are presented in Table 3. We can observe that the difference in performance on the testing dataset between the two models is minimal, with both achieving a Mean Dice Score of about 0.94, although ResNet34_UNet performs slightly better during testing.

UNet Mean Dice Score	ResNet34_UNet Mean Dice Score
/content/lab3/src Loaded state dict from ../saved_models/unet.pth (inference on unet model.) Mean Dice Score on test set: 0.9399	/content/lab3/src Loaded state dict from ../saved_models/resnet34 (inference on resnet34_unet model.) Mean Dice Score on test set: 0.9456

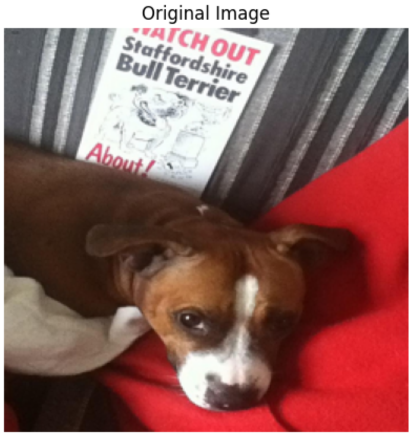
Table 3: Inference results on testing dataset.

3.4 Further Observations and Analysis

The experimental results in the table above show that each model has its own advantages and disadvantages. When the target object and background have similar tones, the UNet model tends to mistakenly classify that area as background; in contrast, ResNet34_UNet is better at fully recognizing and covering target objects that have tones similar to the background. Additionally, ResNet34_UNet does not perform as well as UNet for irregular objects, but it shows relatively better performance for smoother objects.

Original Image

Original Image



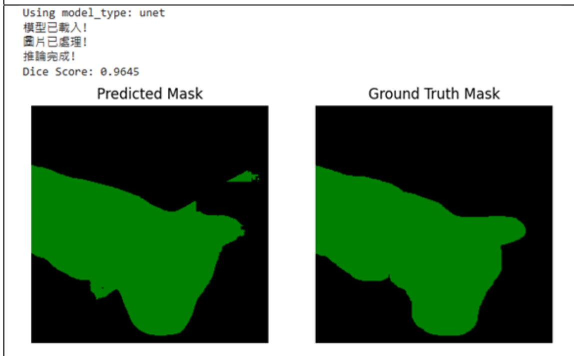
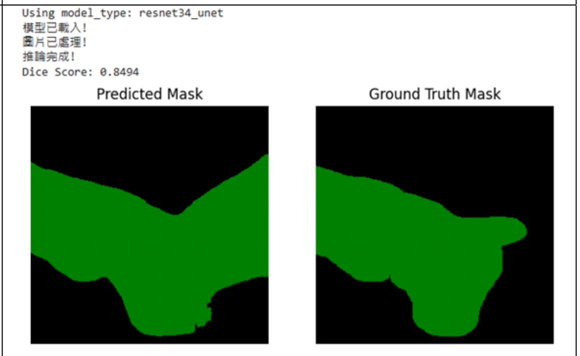
Result on UNet	Result on ResNet34_UNet
	
	

Table 4: Inference results comparison.

3.3 Controversies Regarding Deep Residual Networks (ResNet)

In the field of deep learning, ResNet (Deep Residual Learning for Image Recognition) has rapidly become a mainstream model in many image tasks since its proposal in 2015. However, despite its significant practical achievements, there remain many controversies in both academia and industry regarding its internal mechanisms.

- Some researchers believe that ResNet's ability to train very deep neural networks is mainly due to its 'residual connections', which improve gradient propagation. According to this view, in backpropagation, traditional deep networks may struggle to update parameters due to gradients diminishing (or exploding) layer by layer; residual connections provide a shortcut for gradients to pass directly to shallower layers, thereby improving training stability.
- However, the original ResNet paper indicates that thanks to the use of Batch Normalization (BN), the signals in both forward and backward propagation remain within a healthy range. Thus, the degradation problem is not solely due to vanishing gradients. Simply stating that 'residual connections improve gradient flow' does not fully explain why deeper networks achieve better performance.

My view is that while residual connections do help alleviate gradient issues to a certain extent, this is only part of the explanation. In fact, they also change the network's loss landscape, making the optimization process less likely to get stuck in poor local minima. In other words, the residual structure not only helps the gradient 'flow back' but also makes it easier for the network to learn subtle adjustments — learning the 'residual' rather than directly approximating the target mapping.

Although there is still some debate about the detailed internal mechanisms, both engineering practice and extensive experimental results have fully demonstrated the value of residual connections. This structure allows very deep networks to be successfully trained and to outperform traditional models in various applications. In the future, as theoretical research advances, we may be able to understand these phenomena more comprehensively from a mathematical and physical perspective, but for now, the empirical results have clearly proven the value of residual connections.

4 Execution Steps

4.1 Execution Commands and Parameter Settings (Training)

- model: unet or resnet34_unet
- device: cpu or cuda
- data_path: '../dataset/oxford-iiit-pet/' (default)
- epochs: 200 (default)
- batch_size: 32 (default)
- learning_rate: 1×10^{-3} (default)
- loss_type: bce_dice or bce_tversky

4.2 Execution Commands and Parameter Settings (Testing)

- model: unet.pth or resnet34_unet.pth
- device: cpu or cuda
- data_path: '../dataset/oxford-iiit-pet/' (default)
- batch_size: 1 (default)
- learning_rate: 1×10^{-3} (default)

5 Discussion

5.1 Exploration of Alternative Architectures

In the future, besides the current UNet and ResNet34_UNet architectures, alternative architectures such as DeepLabv3, PSPNet, or Attention UNet can be explored. DeepLabv3/DeepLabv3+ employ atrous convolution to capture a broader receptive field, enabling more effective multi-scale feature learning; PSPNet utilizes a Pyramid Pooling Module to extract global features, which helps mitigate the lack of contextual information in images; and Attention UNet leverages an attention mechanism to automatically emphasize important target regions, thereby further enhancing segmentation accuracy and performance.

5.2 Potential Research Directions

Regarding data augmentation, more advanced techniques such as CutMix, MixUp, or GAN-based data generation methods could be introduced to further improve the model's generalization ability. In addition, exploring model fusion and ensemble learning strategies—by integrating predictions from different architectures—may further enhance the stability and accuracy of segmentation results. Moreover, incorporating attention mechanisms or designing more refined multi-scale feature fusion strategies could enable the model to capture more detailed information, ultimately improving segmentation performance, particularly along boundary regions.

References

[1] Ronneberger, O., Fischer, P., & Brox, T. (2015). 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* (pp. 234–241). Springer.

[2] He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 770–778).

[3] Huang, Z., Qu, E., Meng, Y., Zhang, M., Wei, Q., Bai, X., & Zhang, X. (2022). Deep learning-based pelvic levator hiatus segmentation from ultrasound images. *European Journal of Radiology Open, 9*, 100412.

[4] Zhao, R., Qian, B., Zhang, X., Li, Y., Wei, R., Liu, Y., & Pan, Y. (2020, November). Rethinking dice loss for medical image segmentation. In *2020 IEEE international conference on data mining (ICDM)* (pp. 851–860). IEEE.

[5] Salehi, S. S. M., Erdogmus, D., & Gholipour, A. (2017, September). Tversky loss function for image segmentation using 3D fully convolutional deep networks. In *International workshop on machine learning in medical imaging* (pp. 379–387). Cham: Springer.

[6] Jadon, S. (2020, October). A survey of loss functions for semantic segmentation. In *2020 IEEE conference on computational intelligence in bioinformatics and computational biology (CIBCB)* (pp. 1–7). IEEE.

[7] Li, H., Xu, Z., Taylor, G., Studer, C., & Goldstein, T. (2018). Visualizing the loss landscape of neural nets. *Advances in neural information processing systems, 31.*

[8] Ding, X., Zhang, X., Ma, N., Han, J., Ding, G., & Sun, J. (2021). Repvgg: Making vgg-style convnets great again. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition* (pp. 13733–13742).

[9] The Oxford-IIIT Pet Dataset.