Deep Learning Lab 2 - Binary Semantic Segmentation

110550088 李杰穎

March 25, 2025

1 Implementation Details

1.1 Code Structure

The implementation is organized into modular components, with each functionality encapsulated in separate files within the `src/` directory, following the required project structure:

- `models/`: Contains the neural network architectures.
  - `unet.py`: Implements the standard UNet architecture.
  - `resnet.py`: Implements the ResNet34 encoder with UNet decoder.
- `oxford_pet.py`: Handles dataset loading, preprocessing, and augmentation.
- `train.py`: Contains the training loop and related utilities.
- `evaluate.py`: Implements validation and evaluation metrics.
- `inference.py`: Handles model inference on unseen images.
- `utils.py`: Contains utility functions like Dice score calculation.

1.2 Network Architectures

For this lab, I implemented two encoder-decoder architectures for binary semantic segmentation: a standard UNet and a ResNet34-UNet hybrid.

1.2.1 UNet Architecture

The UNet architecture follows the original design from Ronneberger et al., consisting of a contracting path (encoder) and an expansive path (decoder) with skip connections between corresponding layers.

Contracting Path (Encoder): The encoder consists of repeated blocks of two 3×3 convolutions followed by a ReLU activation and a 2×2 max pooling operation with stride 2. Each downsampling step doubles the number of feature channels.

Figure 1: **UNet architecture.** The network consists of a contracting path (left) and an expansive path (right). Skip connections connect corresponding layers between the encoder and decoder paths.

Double Convolution Block: The basic building block of the UNet is the double convolution, implemented as:

Code 1: **Implementation of the Double Convolution block.**

```python
class DoubleConv(nn.Module):
    """
    Double Convolution block: (conv -> BN -> ReLU) * 2
    """
    def __init__(self, in_channels, out_channels, mid_channels=None):
        super().__init__()
        if not mid_channels:
            mid_channels = out_channels

        self.double_conv = nn.Sequential(
            nn.Conv2d(in_channels, mid_channels, kernel_size=3, padding=1, bias=False),
            nn.BatchNorm2d(mid_channels),
            nn.ReLU(inplace=True),
            nn.Conv2d(mid_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)
```

Down-sampling Block: Each down-sampling step in the encoder consists of a max pooling operation followed by a double convolution:

Code 2: **Implementation of the Down-sampling block.**

```python
class Down(nn.Module):
    """
    Downsampling block: maxpool -> double conv
    """
    def __init__(self, in_channels, out_channels):
        super().__init__()
        self.maxpool_conv = nn.Sequential(
            nn.MaxPool2d(2),
```

```
            DoubleConv(in_channels, out_channels)
        )

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

Expansive Path (Decoder): The decoder consists of up-sampling operations followed by double convolutions. Each up-sampling step halves the number of feature channels. Skip connections from the encoder are concatenated with the corresponding decoder features to provide localization information.

Code 3: **Implementation of the Up-sampling block.**

```
class Up(nn.Module):
    """
    Upsampling block: upconv -> double conv
    """
    def __init__(self, in_channels, out_channels):
        super().__init__()

        self.up = nn.ConvTranspose2d(
            in_channels, in_channels // 2,
            ↪ kernel_size=2, stride=2)
        self.conv = DoubleConv(in_channels,
        ↪ out_channels)

    def forward(self, x1, x2):
        x1 = self.up(x1)

        # Adjust dimensions if there's a mismatch
        ↪ (due to odd dimensions)
        diffY = x2.size()[2] - x1.size()[2]
        diffX = x2.size()[3] - x1.size()[3]

        x1 = F.pad(x1, [diffX // 2, diffX - diffX
        ↪ // 2,
                        diffY // 2, diffY - diffY
                        ↪ // 2])

        # Concatenate along the channel dimension
        x = torch.cat([x2, x1], dim=1)
        return self.conv(x)
```

Final Output Layer: The final layer uses a 1×1 convolution to map the feature vector to the desired number of classes (1 for binary segmentation):

Code 4: **Implementation of the Output Convolution block.**

```
class OutConv(nn.Module):
    """
    Output convolution block
    """
    def __init__(self, in_channels, out_channels):
        super().__init__()
        self.conv = nn.Conv2d(in_channels,
        ↪ out_channels, kernel_size=1)

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

Complete UNet Architecture: The complete UNet architecture combines these components:

Code 5: **Implementation of the complete UNet architecture.**

```
class UNet(nn.Module):
    """
    Full UNet architecture
    """
    def __init__(self, n_channels=1, n_classes=2):
        """
        Args:
            n_channels: Number of input channels
            ↪ (e.g., 1 for grayscale, 3 for RGB)
            n_classes: Number of output classes
            ↪ (e.g., 2 for binary segmentation)
        """
        super(UNet, self).__init__()
        self.n_channels = n_channels
        self.n_classes = n_classes

        # Initial double convolution
        self.inc = DoubleConv(n_channels, 64)

        # Contracting path (encoder)
        self.down1 = Down(64, 128)
        self.down2 = Down(128, 256)
        self.down3 = Down(256, 512)
        self.down4 = Down(512, 1024)

        # Expansive path (decoder)
        self.up1 = Up(1024, 512)
        self.up2 = Up(512, 256)
        self.up3 = Up(256, 128)
        self.up4 = Up(128, 64)

        # Final convolution
        self.outc = OutConv(64, n_classes)

    def forward(self, x):
        # Contracting path
        x1 = self.inc(x)
        x2 = self.down1(x1)
        x3 = self.down2(x2)
        x4 = self.down3(x3)
        x5 = self.down4(x4)

        # Expansive path with skip connections
        x = self.up1(x5, x4)
        x = self.up2(x, x3)
        x = self.up3(x, x2)
        x = self.up4(x, x1)

        # Final convolution
        pred = self.outc(x)

        return pred
```

The architecture has 5 levels with an initial channel count of 64, which doubles at each down-sampling step until 1024 channels at the bottleneck. The total parameter count for the UNet model (with 3 input channels and 1 output channel) is approximately 31.04 million.

1.2.2 ResNet34-UNet Architecture

The ResNet34-UNet hybrid architecture combines a ResNet34 backbone as the encoder with a UNet-style decoder path. This architecture leverages the power of resid-

ual learning for feature extraction while maintaining the precise localization capabilities of UNet.

For the implementation, I reference a public GitHub repository[1].

ResNet34 Encoder: The encoder is based on ResNet34, which consists of residual blocks with identity mappings. Each residual block contains two 3×3 convolutional layers with batch normalization and ReLU activations:

Code 6: **Implementation of the ResNet34 Basic Block.**

```python
class ResNetBasicBlock(nn.Module):
    def __init__(self, in_channels, out_channels, stride=1):
        super(ResNetBasicBlock, 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 = nn.Sequential()
        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)

        # Apply downsample to identity if needed
        identity = self.downsample(identity)

        # Add residual connection
        out = out + identity
        out = self.relu(out)

        return out
```

The complete ResNet34 encoder is implemented as follows[2]:

Code 7: **Implementation of the ResNet34 Encoder.**

```python
class ResNet34Encoder(nn.Module):
    def __init__(self):
        super(ResNet34Encoder, self).__init__()
        # Define the ResNet34 encoder
        self.conv1 = nn.Conv2d(3, 32, kernel_size=7,
                               stride=2, padding=3,
                               bias=False)
        self.bn1 = nn.BatchNorm2d(32)
        self.maxpool = nn.MaxPool2d(kernel_size=3,
        stride=2, padding=1)
        self.conv2_x = self._make_layer(32, 64, 3)
        self.conv3_x = self._make_layer(64, 128, 4,
        stride=2)
        self.conv4_x = self._make_layer(128, 256,
        6, stride=2)
        self.conv5_x = self._make_layer(256, 512,
        3, stride=2)

    def _make_layer(self, in_channels,
    out_channels, num_blocks, stride=1):
        layers = []
        for _ in range(num_blocks):
            layers
            .append(ResNetBasicBlock(in_channels,
            out_channels, stride))
            in_channels = out_channels
            stride = 1
        return nn.Sequential(*layers)

    def forward(self, x):
        x1 = self.conv1(x)
        x1 = self.bn1(x1)
        x1 = F.relu(x1)

        # First block after maxpool
        x2 = self.maxpool(x1)
        x2 = self.conv2_x(x2)

        # Remaining blocks
        x3 = self.conv3_x(x2)
        x4 = self.conv4_x(x3)
        x5 = self.conv5_x(x4)

        # Return feature maps for skip connections
        return x1, x2, x3, x4, x5
```

Complete ResNet34-UNet Architecture: The complete ResNet34-UNet combines the ResNet34 encoder with a UNet-style decoder:

Code 8: **Implementation of the ResNet34-UNet architecture.**

```python
class ResNet34_UNet(nn.Module):
    def __init__(self, n_channels=3, n_classes=1):
        super(ResNet34_UNet, self).__init__()

        self.encoder = ResNet34Encoder()

        self.bridge = nn.Sequential(
            nn.Conv2d(512, 1024, kernel_size=3,
            padding=1),
            nn.BatchNorm2d(1024),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2)
        )

        self.up1 = Up(1024, 512)
        self.up2 = Up(512, 256)
```

[1] https://github.com/GohVh/resnet34-unet/blob/main/model.py

[2] For ResNet-34 implementation, I also referenced https://ithelp.ithome.com.tw/articles/10333931

```
        self.up3 = Up(256, 128)
        self.up4 = Up(128, 64)
        self.up5 = Up(64, 32)
        self.final_up = nn.ConvTranspose2d(32, 32, kernel_size=2, stride=2)
        # Final convolution
        self.outc = OutConv(32, n_classes)

    def forward(self, x):
        # Encoder path
        x1, x2, x3, x4, x5 = self.encoder(x)

        # Bridge
        x = self.bridge(x5)

        # Decoder path with skip connections
        x = self.up1(x, x5)
        x = self.up2(x, x4)
        x = self.up3(x, x3)
        x = self.up4(x, x2)
        x = self.up5(x, x1)

        x = self.final_up(x)

        # Final convolution
        x = self.outc(x)

        return x
```

The ResNet34-UNet architecture benefits from the residual connections in the encoder, which help with gradient flow during backpropagation and enable deeper network training. The total parameter count for the ResNet34-UNet model is approximately 38.22M.

1.3 Loss Function

For the binary semantic segmentation task, I chose the Binary Cross-Entropy with Logits Loss (BCEWithLogitsLoss) as the primary loss function:

BCEWithLogitsLoss combines a Sigmoid activation and Binary Cross-Entropy loss in a single function, providing better numerical stability. The loss function calculates the pixel-wise binary cross-entropy between the predicted logits and target masks.

The binary cross-entropy loss for pixel i is defined as:

$$-[y_i \cdot \log(\sigma(x_i)) + (1 - y_i) \cdot \log(1 - \sigma(x_i))] \quad (1)$$

where $\sigma(x_i)$ is the sigmoid function applied to the predicted logit x_i, and y_i is the ground truth label (0 or 1).

1.4 Evaluation Metric

For evaluating the segmentation performance, I implemented the Dice score, which is a widely used metric for semantic segmentation tasks:

Code 9: **Implementation of the Dice Score metric.**

```
def dice_score(pred_mask, gt_mask):
    # Ensure binary masks with proper thresholding (detach from computation graph if needed)
    with torch.no_grad():
        # apply sigmoid the the prediction mask
        pred_mask = (torch.sigmoid(pred_mask) > 0.5).float().flatten()
        gt_mask = (gt_mask > 0.5).float().flatten()

        intersection = torch.sum(pred_mask * gt_mask)
        union = torch.sum(pred_mask) + torch.sum(gt_mask)

        dice = (2.0 * intersection) / (union + 1e-8)  # add a small epsilon to avoid division by zero

        return dice.item()
```

The Dice score measures the overlap between the predicted segmentation mask and the ground truth mask. It is calculated as twice the area of intersection divided by the sum of the areas of both masks:

$$\text{Dice} = \frac{2 \times |X \cap Y|}{|X| + |Y|} \quad (2)$$

where X is the predicted mask and Y is the ground truth mask. The Dice score ranges from 0 (no overlap) to 1 (perfect overlap).

1.5 Training Procedure

The training procedure is implemented in the `train.py` file and consists of the following key components:

Optimizer: I used the AdamW optimizer, which combines the benefits of Adam with decoupled weight decay regularization:

Code 10: **Optimizer implementation.**

```
optimizer = optim.AdamW(model.parameters(),
                        lr=args.learning_rate,
                        weight_decay=args.weight_decay)
```

Learning Rate Scheduler: To improve convergence, I implemented several learning rate scheduling options:

Code 11: **Learning rate scheduler implementations.**

```
if args.scheduler == 'plateau':
    scheduler = optim.lr_scheduler.ReduceLROnPlateau(
        optimizer, mode='min', patience=3)
elif args.scheduler == 'cosine':
    scheduler = optim.lr_scheduler.CosineAnnealingLR(
        optimizer, T_max=args.epochs)
elif args.scheduler == 'onecycle':
    scheduler = optim.lr_scheduler.OneCycleLR(
        optimizer,
        max_lr=args.learning_rate,
        epochs=args.epochs,
        steps_per_epoch=len(train_loader),
        pct_start=0.1,
        anneal_strategy='cos'
    )
```

```
else:
    scheduler = None
```

These schedulers help in dynamically adjusting the learning rate during training:

- **ReduceLROnPlateau**: Reduces the learning rate when the validation loss plateaus.
- **CosineAnnealingLR**: Gradually reduces the learning rate following a cosine curve.
- **OneCycleLR**: Implements the one-cycle policy that increases the learning rate to a maximum value and then decreases it.

I will discuss the performance of cosine annealing and one cycle learning rate scheduler in Sec. 3.

Training Loop: The main training loop consists of:

- Forward pass through the model to generate predictions.
- Loss calculation using BCEWithLogitsLoss.
- Backward pass to compute gradients.
- Parameter updates using the AdamW optimizer.
- Validation phase after each epoch to evaluate model performance.
- Learning rate adjustment using the chosen scheduler.
- Checkpointing to save the best model based on validation Dice score.

Code 12: **Training loop implementation.**

```
# Training loop
for epoch in range(start_epoch, args.epochs):
    print(f"\nEpoch {epoch+1}/{args.epochs}")

    # Training phase
    model.train()
    train_loss = 0
    train_dice = 0

    train_pbar = tqdm(enumerate(train_loader), total=len(train_loader), desc="Training")
    for i, batch in train_pbar:
        images = batch['image'].to(device)
        masks = batch['mask'].to(device)

        # Forward pass
        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, masks)

        # Backward pass and optimize
        loss.backward()
        optimizer.step()

        # Update metrics
        batch_dice = dice_score(outputs, masks)
        train_loss += loss.item()
        train_dice += batch_dice


        # Update the progress bar
        train_pbar.set_postfix(loss=f"{loss.item():.4f}",
                               dice=f"{batch_dice:.4f}",
                               lr=f"{optimizer.param_groups[0]['lr']:.4E}")

        # Update OneCycleLR scheduler if used
        if args.scheduler == 'onecycle':
            scheduler.step()

    # Calculate average metrics
    train_loss /= len(train_loader)
    train_dice /= len(train_loader)

    # Validation phase
    model.eval()
    val_loss = 0
    val_dice = 0

    with torch.no_grad():
        for batch in tqdm(val_loader, total=len(val_loader), desc="Validation"):
            images = batch['image'].to(device)
            masks = batch['mask'].to(device)

            outputs = model(images)
            loss = criterion(outputs, masks)

            val_loss += loss.item()
            val_dice += dice_score(outputs, masks)

    val_loss /= len(val_loader)
    val_dice /= len(val_loader)

    # Update ReduceLROnPlateau or CosineAnnealingLR scheduler
    if args.scheduler == 'plateau':
        scheduler.step(val_loss)
    elif args.scheduler == 'cosine':
        scheduler.step()

    # Print progress
    print(f"Train Loss: {train_loss:.4f}, Train Dice: {train_dice:.4f}")
    print(f"Val Loss: {val_loss:.4f}, Val Dice: {val_dice:.4f}")
    print(f"Learning Rate: {optimizer.param_groups[0]['lr']:.8f}")

    # Save checkpoints and track best model
    if val_dice > best_val_dice:
        best_val_dice = val_dice
        torch.save({
            'model_state_dict': model.state_dict(),
            'optimizer_state_dict': optimizer.state_dict(),
            'val_dice': val_dice,
            'epoch': epoch
        }, os.path.join(args.save_dir, 'best_model.pth'))
```

Evaluation on Test Set: After training, the model is evaluated on the test set:

Code 13: **Test set evaluation.**

```python
# Load best model for testing
best_model_path = os.path.join(args.save_dir, 'best_model.pth')
if os.path.exists(best_model_path):
    checkpoint = torch.load(best_model_path, map_location=device)
    model.load_state_dict(checkpoint['model_state_dict'])
    print(f"Loaded best model with validation Dice score: {checkpoint['val_dice']:.4f}")

model.eval()
test_loss = 0
test_dice = 0

with torch.no_grad():
    for batch in tqdm(test_loader, total=len(test_loader), desc="Testing"):
        images = batch['image'].to(device)
        masks = batch['mask'].to(device)

        outputs = model(images)
        loss = criterion(outputs, masks)

        test_loss += loss.item()
        test_dice += dice_score(outputs, masks)

test_loss /= len(test_loader)
test_dice /= len(test_loader)

print(f"Test Loss: {test_loss:.4f}, Test Dice: {test_dice:.4f}")

# Save final model with test results
torch.save({
    'model_state_dict': model.state_dict(),
    'test_loss': test_loss,
    'test_dice': test_dice,
}, os.path.join(args.save_dir, f'{args.model}_final.pth'))

```

1.6 Inference and Visualization

For the inference phase, I implemented functionality to apply the trained models to unseen images and visualize the segmentation results:

Code 14: **Inference implementation.**

```python
def preprocess_image(image_path, img_size=256):
    """
    Load and preprocess an image for inference

    Args:
        image_path: Path to the image file
        img_size: Size to resize the image to

    Returns:
        Processed image tensor ready for model input
    """
    # Load image
    image = Image.open(image_path).convert('RGB')

    # Define preprocessing transforms
    transform = T.Compose([
        T.Resize((img_size, img_size)),
        T.ToTensor(),
        T.Normalize(mean=[0.485, 0.456, 0.406],
                    std=[0.229, 0.224, 0.225])
    ])

    # Apply transforms
    image_tensor = transform(image)

    return image_tensor, image

def predict(model, image_tensor, device, threshold=0.5):
    """
    Generate a segmentation mask prediction

    Args:
        model: The trained segmentation model
        image_tensor: Input image tensor
        device: Device to run inference on
        threshold: Threshold for binary segmentation

    Returns:
        Predicted mask as numpy array
    """
    with torch.no_grad():
        # Add batch dimension and move to device
        x = image_tensor.unsqueeze(0).to(device)

        # Forward pass
        output = model(x)

        # Apply sigmoid and threshold
        pred_mask = torch.sigmoid(output) > threshold

        # Convert to numpy
        pred_mask = pred_mask.cpu().squeeze().numpy().astype(np.uint8) * 255

    return pred_mask

def save_prediction(image, mask, output_path):
    """
    Visualize and save the prediction results

    Args:
        image: Original PIL image
        mask: Predicted mask as numpy array
        output_path: Path to save the visualization
    """
    # Create a figure with subplots
    fig, axes = plt.subplots(1, 3, figsize=(15, 5))

    # Plot original image
    axes[0].imshow(image)
    axes[0].set_title('Original Image')
    axes[0].axis('off')

    # Plot predicted mask
    axes[1].imshow(mask, cmap='gray')
    axes[1].set_title('Predicted Mask')
    axes[1].axis('off')

    # Plot overlay
    image_np = np.array(image.resize((mask.shape[1], mask.shape[0])))
    mask_rgb = np.stack([mask, np.zeros_like(mask), np.zeros_like(mask)], axis=2)
    overlay = image_np * 0.7 + mask_rgb * 0.3
    overlay = np.clip(overlay, 0, 255).astype(np.uint8)

    axes[2].imshow(overlay)
    axes[2].set_title('Overlay')
```

```
    axes[2].axis('off')

    plt.tight_layout()
    plt.savefig(output_path, dpi=150, bbox_inches='tight')
    plt.close()

    # Save raw mask as well
    mask_path = output_path.replace('.png', '_mask.png')
    mask_img = Image.fromarray(mask)
    mask_img.save(mask_path)

def process_batch(model, image_paths, device, output_dir, img_size):
    """
    Process a batch of images

    Args:
        model: The trained segmentation model
        image_paths: List of paths to image files
        device: Device to run inference on
        output_dir: Directory to save results
        img_size: Input image size
    """
    for image_path in tqdm(image_paths, desc="Processing images"):
        # Extract filename without extension
        filename = os.path.splitext(os.path.basename(image_path))[0]

        # Preprocess image
        image_tensor, original_image = preprocess_image(image_path, img_size)

        # Generate prediction
        pred_mask = predict(model, image_tensor, device)

        # Save visualization
        output_path = os.path.join(output_dir, f"{filename}_prediction.png")
        save_prediction(original_image, pred_mask, output_path)
```

To evaluate the model's performance across the entire test set, I calculate the average and median Dice score and generate visualizations of the segmentation results:

Code 15: **Test set evaluation with visualization.**

```
def evaluate(net, data_loader, device, output_dir=None, save_visualizations=False):
    """
    Evaluate a trained model on a dataset

    Args:
        net: The trained neural network model
        data_loader: DataLoader for the evaluation dataset
        device: Device to run the evaluation on (cuda or cpu)
        output_dir: Directory to save visualization results (if None, no visualizations are saved)
        save_visualizations: Whether to save visualizations of segmentation results

    Returns:
        avg_dice: Average Dice score across the dataset
        all_dices: List of individual Dice scores for each sample
    """
    net.eval()
    all_dices = []

    # Create output directory if it doesn't exist
    if output_dir is not None and save_visualizations:
        os.makedirs(output_dir, exist_ok=True)

    with torch.no_grad():
        for i, batch in enumerate(tqdm(data_loader, desc="Evaluating")):
            images = batch['image'].to(device)
            masks = batch['mask'].to(device)

            # Forward pass
            outputs = net(images)

            # Calculate Dice score for each image in the batch
            for j in range(images.size(0)):
                img_dice = dice_score(outputs[j:j+1], masks[j:j+1])
                all_dices.append(img_dice)

            # Save visualization for some samples
            if save_visualizations and output_dir is not None and i % 5 == 0:
                for j in range(min(4, images.size(0))):  # Visualize up to 4 images per batch
                    visualize_prediction(
                        images[j].cpu(),
                        masks[j].cpu(),
                        outputs[j].cpu(),
                        os.path.join(output_dir, f'sample_{i}_{j}.png')
                    )

    # Calculate average Dice score
    avg_dice = sum(all_dices) / len(all_dices)

    return avg_dice, all_dices

def visualize_prediction(image, true_mask, pred_logits, save_path=None):
    """
    Visualize the model's prediction alongside the input image and ground truth mask

    Args:
        image: Input image tensor [C, H, W]
        true_mask: Ground truth mask tensor [1, H, W]
        pred_logits: Model's prediction logits tensor [1, H, W]
        save_path: Path to save the visualization image
    """
    # Denormalize the image
    mean = torch.tensor([0.485, 0.456, 0.406]).view(3, 1, 1)
    std = torch.tensor([0.229, 0.224, 0.225]).view(3, 1, 1)
    image = image * std + mean

    # Convert tensors to numpy for visualization
    image = image.permute(1, 2, 0).numpy()
```

```
    image = np.clip(image, 0, 1)

    true_mask = true_mask.squeeze().numpy()

    # Apply sigmoid to get probabilities and threshold to get binary mask
    pred_probs = torch.sigmoid(pred_logits).squeeze().numpy()
    pred_mask = (pred_probs > 0.5).astype(np.float32)

    # Create the figure with three subplots
    fig, axes = plt.subplots(1, 4, figsize=(16, 4))

    # Plot the original image
    axes[0].imshow(image)
    axes[0].set_title('Original Image')
    axes[0].axis('off')

    # Plot the ground truth mask
    axes[1].imshow(true_mask, cmap='gray')
    axes[1].set_title('Ground Truth Mask')
    axes[1].axis('off')

    # Plot the predicted probability map
    axes[2].imshow(pred_probs, cmap='viridis')
    axes[2].set_title('Prediction Probabilities')
    axes[2].axis('off')

    # Plot the thresholded prediction mask
    axes[3].imshow(pred_mask, cmap='gray')
    axes[3].set_title('Predicted Mask')
    axes[3].axis('off')

    plt.tight_layout()

    if save_path:
        plt.savefig(save_path, dpi=150, bbox_inches='tight')
        plt.close()
    else:
        plt.show()
```

This evaluation framework allows for both quantitative assessment of the model's performance through the Dice score and qualitative assessment through visualization of the segmentation masks.

1.7 Hyperparameters

For training both the UNet and ResNet34-UNet models, I used the following hyperparameters:

- **Image size**: 256x256 pixels
- **Batch size**: 32 (UNet), 128 (ResNet34-UNet)
- **Optimizer**: AdamW
- **Learning rate**: 1e-3
- **Weight decay**: 1e-2
- **Scheduler**: OneCycleLR with 10% warm-up epochs
- **Number of epochs**: 50
- **Loss function**: BCEWithLogitsLoss

These hyperparameters were selected based on experimental validation and best practices in semantic segmentation tasks.

2 Data Preprocessing

Data preprocessing is a crucial step for achieving good performance in semantic segmentation tasks. The Oxford-IIIT Pet Dataset contains images of cats and dogs with pixel-level annotations for pet regions.

Dataset Handling: I implemented a custom dataset class that inherits from PyTorch's `Dataset` class to handle the Oxford-IIIT Pet Dataset:

Code 16: **Implementation of the Oxford Pet Dataset class.**

```
class OxfordPetDataset(torch.utils.data.Dataset):
    def __init__(self, root, mode="train", transform=None):
        assert mode in {"train", "valid", "test"}

        self.root = root
        self.mode = mode
        self.transform = transform

        self.images_directory = os.path.join(self.root, "images")
        self.masks_directory = os.path.join(self.root, "annotations", "trimaps")

        self.filenames = self._read_split()  # read train/valid/test splits

    def __len__(self):
        return len(self.filenames)

    def __getitem__(self, idx):
        filename = self.filenames[idx]
        image_path = os.path.join(self.images_directory, filename + ".jpg")
        mask_path = os.path.join(self.masks_directory, filename + ".png")

        image = np.array(Image.open(image_path).convert("RGB"))

        trimap = np.array(Image.open(mask_path))
        mask = self._preprocess_mask(trimap)

        sample = dict(image=image, mask=mask, trimap=trimap)
        return sample

    @staticmethod
    def _preprocess_mask(mask):
        mask = mask.astype(np.float32)
        mask[mask == 2.0] = 0.0
        mask[(mask == 1.0) | (mask == 3.0)] = 1.0
        return mask

    def _read_split(self):
        split_filename = "test.txt" if self.mode == "test" else "trainval.txt"
        split_filepath = os.path.join(self.root, "annotations", split_filename)
        with open(split_filepath) as f:
            split_data = f.read().strip("\n").split("\n")
```

```
        filenames = [x.split(" ")[0] for x in split_data]
        if self.mode == "train":  # 90% for train
            filenames = [x for i, x in enumerate(filenames) if i % 10 != 0]
        elif self.mode == "valid":  # 10% for validation
            filenames = [x for i, x in enumerate(filenames) if i % 10 == 0]
        return filenames
```

Preprocessing and Augmentation: To enhance model generalization and performance, I applied several data preprocessing and augmentation techniques:

Code 17: **Implementation of data preprocessing and augmentation.**

```
class SimpleOxfordPetDataset(OxfordPetDataset):
    def __init__(self, root, mode="train"):
        super().__init__(root, mode)
        train_transform = T.Compose([
            T.ToImage(),
            T.RandomHorizontalFlip(p=0.5),
            T.RandomVerticalFlip(p=0.5),
            T.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2),
            T.ToDtype(torch.float32, scale=True),
        ])
        val_transorm = T.Compose([
            T.ToImage(),
            T.ToDtype(torch.float32, scale=True),
        ])
        self.transform = train_transform if mode == 'train' else val_transorm
        self.normalize = T.Compose([
            T.Normalize(mean=[0.485, 0.456, 0.406],
                        std=[0.229, 0.224, 0.225])
        ])

    def __getitem__(self, *args, **kwargs):
        sample = super().__getitem__(*args, **kwargs)

        # resize images
        image = np.array(Image.fromarray(sample["image"]).resize((256, 256), Image.BILINEAR))
        mask = np.array(Image.fromarray(sample["mask"]).resize((256, 256), Image.NEAREST))
        trimap = np.array(Image.fromarray(sample["trimap"]).resize((256, 256), Image.NEAREST))

        sample["trimap"] = np.expand_dims(trimap, 0)


        sample["image"], sample["mask"] = self.transform(image, mask)
        sample["image"] = self.normalize(sample["image"])

        return sample
```

The preprocessing and augmentation pipeline includes:

- **Resizing**: All images and masks are resized to 256×256 pixels for consistent input dimensions. This is done by provided code.
- **Data type conversion**: Images and masks are converted to PyTorch tensors with `float32` data type and normalized to [0,1] range.
- **Data augmentation (training only)**:
  - **Random horizontal flip** with 50% probability.
  - **Random vertical flip** with 50% probability.
  - **Color jittering** with brightness, contrast, and saturation adjustments of ±20%.
- **Normalization**: Images are normalized using the ImageNet mean and standard deviation. Models learn faster with normalization.
- **Mask preprocessing**: The trimap annotations (which have values 1, 2, and 3) are converted to binary masks where 1 and 3 represent the foreground (pet) and 2 represents the background.

These augmentation techniques help prevent overfitting and improve the model's ability to generalize to unseen images with varying lighting conditions, orientations, and appearances. We will also discuss the performance difference with and without data argumentation.

3 Experiment Results Analysis

3.1 Data argumentation

In this experiment, I evaluated model performance with and without data augmentation. The baseline approach involved only resizing images to 256x256 pixels and normalizing them using ImageNet's mean and standard deviation. For comprehensive assessment, I tested both U-Net and ResNet34_U-Net architectures under identical training conditions: a learning rate of 0.001 with cosine annealing scheduler, trained over 50 epochs. This controlled comparison allowed me to isolate the impact of data augmentation on segmentation quality.

U-Net. As illustrated in Figure 2 and Figure 3, the model trained without data augmentation exhibits lower training loss and higher training Dice scores. However, examining the validation loss in Figure 4 reveals a concerning pattern: after epoch 25, the validation loss for the model without data argumentation increases. This trend clearly indicates an overfitting problem stemming from the lack of data augmentation in the training pipeline. In contrast, the model trained with data augmentation maintains stable validation performance, effectively mitigating the overfitting issue and achieving higher validation Dice scores by the end of training, as demonstrated in Figure 5. The benefits of data augmentation extend to test set performance as well, with the U-Net model with data argumentation

achieving a Dice score of 0.9269, compared to just 0.9209 for the baseline model under identical training conditions.

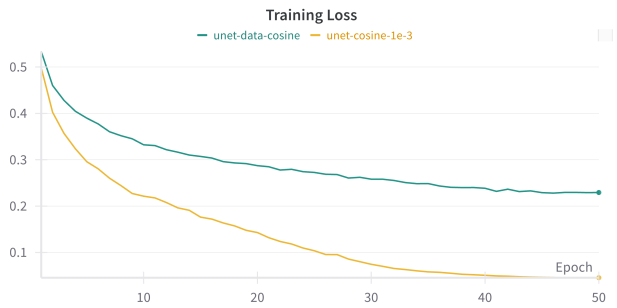

Figure 2: **The training loss over epochs.** `unet-data-cosine` is the model with data argumentation, while `unet-cosine-1e-3` is without data argumentation.

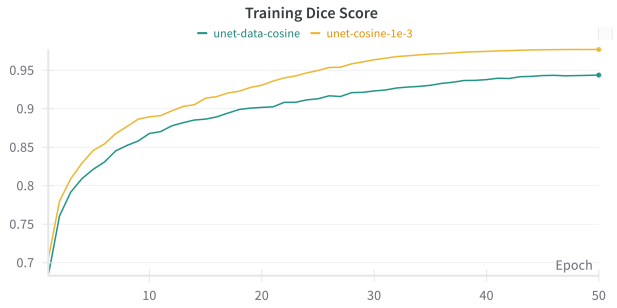

Figure 3: **The training Dice score over epochs.** `unet-data-cosine` is the model with data argumentation, while `unet-cosine-1e-3` is without data argumentation. The model without data argumentation achieves higher training Dice scores but performs worse on validation data.

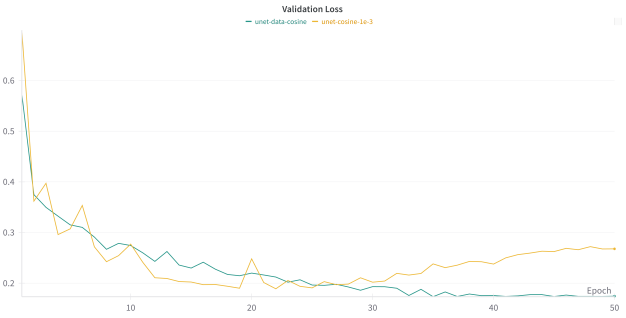

Figure 4: **The validation loss over epochs.** `unet-data-cosine` is the model with data argumentation, while `unet-cosine-1e-3` is without data argumentation. The validation loss for the model without data argumentation increases after epoch 25, indicating overfitting.

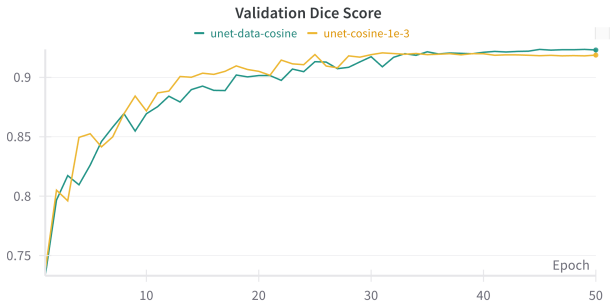

Figure 5: **The validation Dice score over epochs.** `unet-data-cosine` is the model with data argumentation, while `unet-cosine-1e-3` is without data argumentation. Data argumentation helps maintain higher validation Dice scores throughout training, demonstrating better generalization.

ResNet34_U-Net. We observe similar validation loss behavior on the ResNet34_U-Net architecture. As shown in Figure 6 and Figure 7, the model without data augmentation achieves lower training loss and higher training Dice scores initially. However, examining the validation metrics reveals the same pattern of overfitting. In Figure 8, we can see that the validation loss for the non-augmented model begins to increase around epoch 20, while the augmented model maintains a steadier, lower validation loss throughout training. This translates to superior validation Dice scores for the augmented model, as displayed in Figure 9.

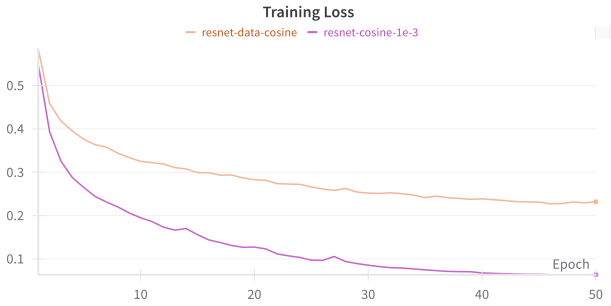

Figure 6: **The training loss over epochs for ResNet34_U-Net.** `resnet-data-cosine` is the model with data augmentation, while `resnet-cosine-1e-3` is without data augmentation.

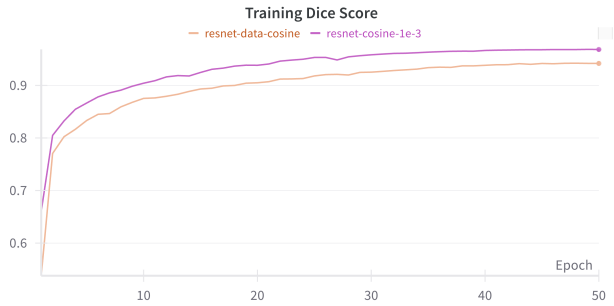

Figure 7: **The training Dice score over epochs for ResNet34_U-Net.** `resnet-data-cosine` is the model with data augmentation, while `resnet-cosine-1e-3` is without data augmentation. The non-augmented model shows higher training Dice scores but fails to generalize as well to validation data.

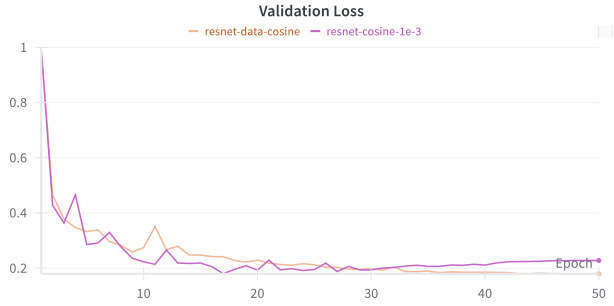

Figure 8: **The validation loss over epochs for ResNet34_U-Net.** `resnet-data-cosine` is the model with data augmentation, while `resnet-cosine-1e-3` is without data augmentation. The validation loss for the non-augmented model increases after epoch 20, indicating overfitting.

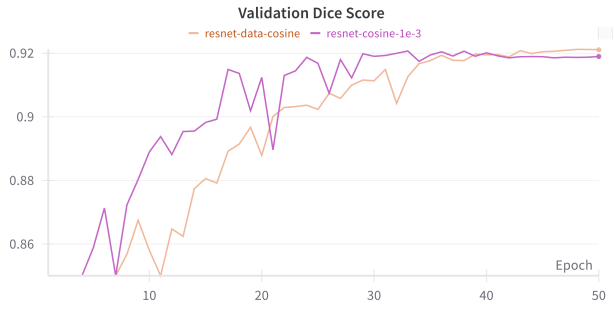

Figure 9: **The validation Dice score over epochs for ResNet34_U-Net.** `resnet-data-cosine` is the model with data augmentation, while `resnet-cosine-1e-3` is without data augmentation. The augmented model demonstrates more stable and higher validation performance throughout training.

Based on these consistent findings across both architectures, we conclude that data augmentation is essential for mitigating overfitting and improving generalization in semantic segmentation tasks with the Oxford-IIIT Pet Dataset. The augmentation techniques, including random horizontal and vertical flips and color jittering, effectively increase the diversity of the training data without requiring additional labeled samples. This approach leads to more robust models that perform better on unseen data. Therefore, we will apply data augmentation in all successive experiments to ensure optimal model performance and generalization capability.

3.2 Learning rate scheduling

In this experiment, I evaluated both models using two different learning rate schedulers: CosineAnnealingLR and OneCycleLR. The CosineAnnealingLR scheduler follows a cosine wave function to gradually decrease the learning rate from its initial value to near-zero over the training period. In contrast, the OneCycleLR scheduler implements a three-phase approach: it begins with a low initial learning rate, gradually increases it during the first 10% of training epochs until reaching the maximum specified value, and then decreases it following a cosine curve back to a minimal value by the end of training. This comparison allows us to assess how different learning rate trajectories affect model convergence and final performance.

U-Net. For the U-Net architecture, we observe distinct differences in training dynamics between the two schedulers. As shown in Figure 10, the learning rate trajectories differ significantly, with OneCycleLR featuring an initial warm-up phase followed by a steeper decline. This learning rate pattern appears to benefit the U-Net model, as evidenced by the training loss in Figure 11 and training Dice score in Figure 12. The OneCycleLR scheduler achieves lower training loss and higher training Dice scores, particularly in the later stages of training.

More importantly, the validation metrics demonstrate the superiority of the OneCycleLR approach. In Figure 13, we can see that the validation loss for the OneCycleLR-trained model is consistently lower after the initial epochs, suggesting better generalization. This translates to higher validation Dice scores as illustrated in Figure 14, with the OneCycleLR model maintaining a clear advantage throughout most of the training process.

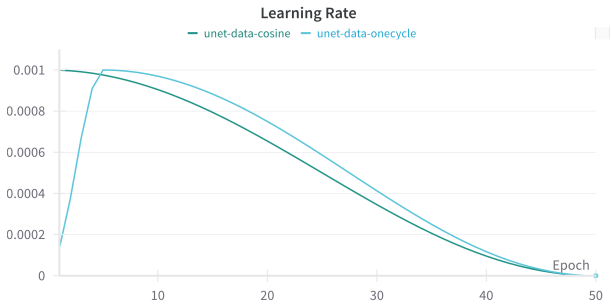

Figure 10: **Learning rate over epochs for U-Net models.** `unet-data-cosine` uses the CosineAnnealingLR scheduler, while `unet-data-onecycle` uses the OneCycleLR scheduler. Note the distinctive warm-up phase in the OneCycleLR approach.

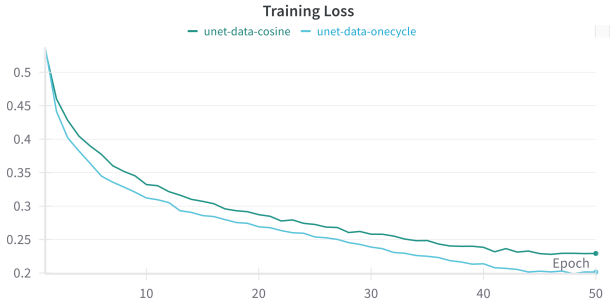

Figure 11: **Training loss over epochs for U-Net models.** `unet-data-cosine` uses the CosineAnnealingLR scheduler, while `unet-data-onecycle` uses the OneCycleLR scheduler. The OneCycleLR approach achieves lower training loss, particularly in the later stages of training.

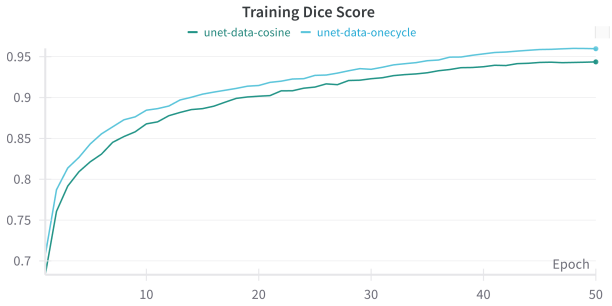

Figure 12: **Training Dice score over epochs for U-Net models.** `unet-data-cosine` uses the CosineAnnealingLR scheduler, while `unet-data-onecycle` uses the OneCycleLR scheduler. The OneCycleLR scheduler enables the model to achieve higher training Dice scores.

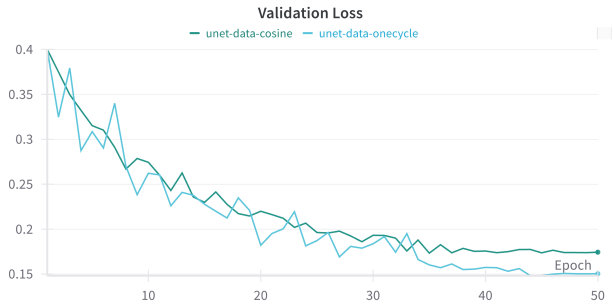

Figure 13: **Validation loss over epochs for U-Net models.** `unet-data-cosine` uses the CosineAnnealingLR scheduler, while `unet-data-onecycle` uses the OneCycleLR scheduler. The OneCycleLR approach maintains lower validation loss, indicating better generalization.

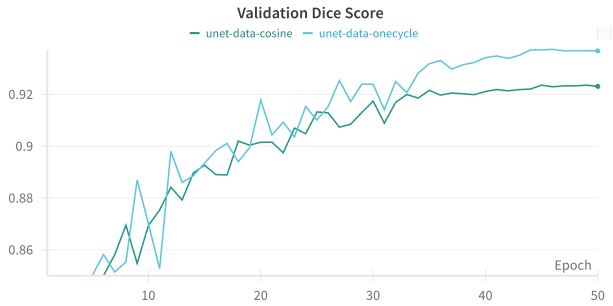

Figure 14: **Validation Dice score over epochs for U-Net models.** `unet-data-cosine` uses the CosineAnnealingLR scheduler, while `unet-data-onecycle` uses the OneCycleLR scheduler. The OneCycleLR approach achieves consistently higher validation Dice scores throughout training.

ResNet34_U-Net. Similar trends emerge when applying these learning rate schedulers to the ResNet34_U-Net architecture. As shown in Figure 15, the learning rate patterns follow the same principles as with the U-Net model. The training loss in Figure 16 shows that the OneCycleLR scheduler initially leads to higher losses during the warm-up phase, but quickly achieves lower training loss as the learning rate decreases. This is mirrored in the training Dice scores depicted in Figure 17, where the OneCycleLR model rapidly improves after the initial epochs.

The validation performance, as illustrated in Figure 18 and Figure 19, demonstrates that the OneCycleLR approach produces a model with better generalization capabilities. The validation loss is consistently lower, and the validation Dice score is higher for the model trained with OneCycleLR.

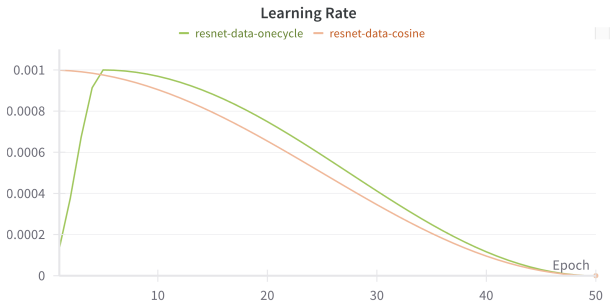

Figure 15: **Learning rate over epochs for ResNet34_U-Net models.** `resnet-data-cosine` uses the CosineAnnealingLR scheduler, while `resnet-data-onecycle` uses the OneCycleLR scheduler, showing the characteristic warm-up and cool-down phases.

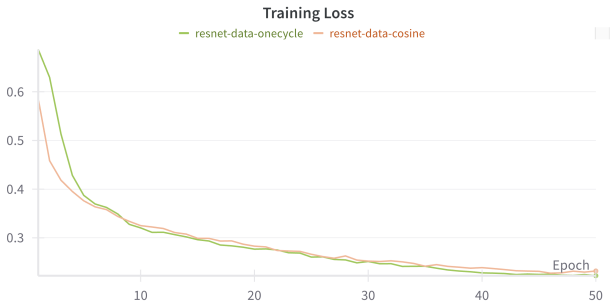

Figure 16: **Training loss over epochs for ResNet34_U-Net models.** `resnet-data-cosine` uses the CosineAnnealingLR scheduler, while `resnet-data-onecycle` uses the OneCycleLR scheduler. After the initial warm-up phase, the OneCycleLR approach achieves lower training loss.

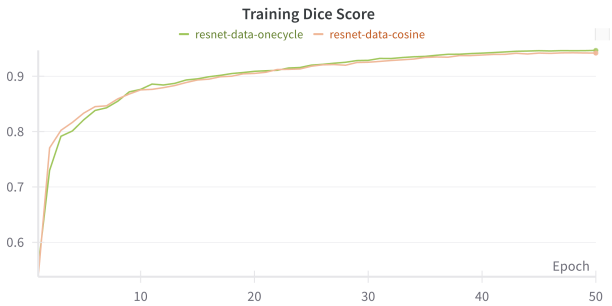

Figure 17: **Training Dice score over epochs for ResNet34_U-Net models.** `resnet-data-cosine` uses the CosineAnnealingLR scheduler, while `resnet-data-onecycle` uses the OneCycleLR scheduler. The OneCycleLR scheduler enables faster improvement in training Dice scores after the initial warm-up phase.

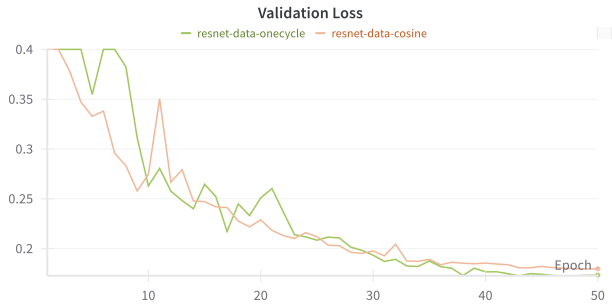

Figure 18: **Validation loss over epochs for ResNet34_U-Net models.** `resnet-data-cosine` uses the CosineAnnealingLR scheduler, while `resnet-data-onecycle` uses the OneCycleLR scheduler. The OneCycleLR approach maintains lower validation loss throughout most of the training process.

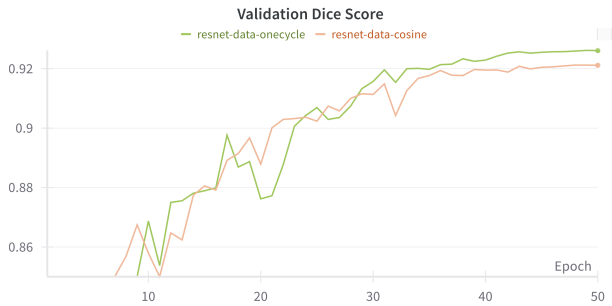

Figure 19: **Validation Dice score over epochs for ResNet34_U-Net models.** `resnet-data-cosine` uses the CosineAnnealingLR scheduler, while `resnet-data-onecycle` uses the OneCycleLR scheduler. The OneCycleLR approach demonstrates superior validation performance.

These results consistently demonstrate that the OneCycleLR scheduler outperforms the CosineAnnealingLR scheduler for both architectures. The initial warm-up phase in OneCycleLR allows the models to explore the parameter space more effectively before gradually reducing the learning rate. This approach appears to help the models escape poor local minima early in training while still allowing for fine-tuning of weights in later epochs. The improvement is substantial, with OneCycleLR yielding approximately a 1% increase in Dice score for U-Net. Based on these findings, we will use the OneCycleLR scheduler in all subsequent experiments to maximize model performance.

3.3 Model architectures

After establishing the optimal data preprocessing and training strategies, I conducted a comprehensive comparison of the U-Net and ResNet34_U-Net architectures to determine their relative strengths in binary semantic segmentation. Both models were trained with identical hyperparameters: using data augmentation, the OneCy-

cleLR scheduler, a base learning rate of 0.001, and training for 50 epochs.

Table 1 presents the detailed performance metrics of both architectures on the test set.

Table 1: **Performance comparison of U-Net and ResNet34_U-Net architectures.** Both models were trained with data augmentation, OneCycleLR scheduler, and identical hyperparameters.

Metric	U-Net	ResNet34_U-Net
Average Dice Score	0.9291	0.9179
Median Dice Score	0.9528	0.9455
Standard Deviation	0.0842	0.0937
Minimum Dice Score	0.0000	0.0000
Maximum Dice Score	0.9982	0.9958
Total Training Time (RTX 4090)	30m	16m

Interestingly, the U-Net architecture outperformed the more complex ResNet34_U-Net across all accuracy metrics. The U-Net achieved a higher average Dice score (0.9291 vs. 0.9179) and median Dice score (0.9528 vs. 0.9455), while also demonstrating lower variability with a standard deviation of 0.0842 compared to 0.0937 for ResNet34_U-Net. This suggests that the simpler U-Net architecture offers more consistent performance across the test dataset.

Both models struggled with certain challenging images, as evidenced by the minimum Dice score of 0.0000, indicating complete segmentation failure on at least one test sample. However, they both achieved excellent results on well-defined samples, with maximum Dice scores exceeding 0.99.

The superior performance of U-Net is somewhat surprising given that ResNet34_U-Net incorporates residual connections, which typically aid in training deeper networks by mitigating gradient vanishing problems. However, for this particular dataset and binary segmentation task, the simpler U-Net architecture appears to be more effective. This could be attributed to several factors:

1. The Oxford-IIIT Pet Dataset may not be complex enough to benefit from the additional capacity of ResNet34_U-Net.

2. The ResNet34 encoder's aggressive use of maxpooling and strided convolutions to rapidly reduce spatial dimensions likely causes loss of fine-grained spatial information that's crucial for precise boundary segmentation. While this design choice makes ResNet efficient for classification tasks, it can be detrimental for pixel-level segmentation where spatial precision is paramount.

3. The U-Net's more direct skip connections between corresponding encoder and decoder layers may preserve spatial information more effectively for this specific segmentation task.

Despite the performance advantage of U-Net, it's worth noting that the ResNet34_U-Net offers a significant computational efficiency benefit, with total training time approximately 47% faster than the standard U-Net (16 minutes vs. 30 minutes for the complete 50-epoch training run on an RTX 4090). This efficiency stems from ResNet's design choices that reduce feature map sizes earlier in the network, resulting in fewer operations in the deeper layers.

These findings highlight that architectural complexity does not always translate to better performance, and that the standard U-Net remains a strong baseline for semantic segmentation tasks, particularly when properly trained with data augmentation and appropriate learning rate scheduling. However, in scenarios where computational resources or training time are limited, the ResNet34_U-Net could be a reasonable alternative with only a modest decrease in segmentation quality.

3.4 Visual results

4 Execution steps

5 Discussion