

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, pre-processing, 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.**

```
1 class DoubleConv(nn.Module):
2     """
3         Double Convolution block: (conv -> BN -> ReLU) *
4             2
5         """
6     def __init__(self, in_channels, out_channels,
7                  mid_channels=None):
8         super().__init__()
9         if not mid_channels:
10            mid_channels = out_channels
11
12         self.double_conv = nn.Sequential(
13             nn.Conv2d(in_channels, mid_channels,
14                     kernel_size=3, padding=1,
15                     bias=False),
16             nn.BatchNorm2d(mid_channels),
17             nn.ReLU(inplace=True),
18             nn.Conv2d(mid_channels, out_channels,
19                     kernel_size=3, padding=1,
20                     bias=False),
21             nn.BatchNorm2d(out_channels),
22             nn.ReLU(inplace=True)
23         )
24
25     def forward(self, x):
26         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.**

```
1 class Down(nn.Module):
2     """
3         Downsampling block: maxpool -> double conv
4         """
5     def __init__(self, in_channels, out_channels):
6         super().__init__()
7         self.maxpool_conv = nn.Sequential(
8             nn.MaxPool2d(2),
```

```

9         DoubleConv(in_channels, out_channels)
10    )
11
12    def forward(self, x):
13        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.

```

1 class Up(nn.Module):
2     """
3     Upsampling block: upconv -> double conv
4     """
5     def __init__(self, in_channels, out_channels):
6         super().__init__()
7
7         self.up = nn.ConvTranspose2d(
8             in_channels, in_channels // 2,
9                 kernel_size=2, stride=2)
10        self.conv = DoubleConv(in_channels,
11            out_channels)
12
13    def forward(self, x1, x2):
14        x1 = self.up(x1)
15
16        # Adjust dimensions if there's a mismatch
17        # (due to odd dimensions)
18        diffY = x2.size()[2] - x1.size()[2]
19        diffX = x2.size()[3] - x1.size()[3]
20
21        x1 = F.pad(x1, [diffX // 2, diffX - diffX
22            // 2,
23                diffY // 2, diffY - diffY
24                    // 2])
25
26        # Concatenate along the channel dimension
27        x = torch.cat([x2, x1], dim=1)
28        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.

```

1 class OutConv(nn.Module):
2     """
3     Output convolution block
4     """
5     def __init__(self, in_channels, out_channels):
6         super().__init__()
7         self.conv = nn.Conv2d(in_channels,
8             out_channels, kernel_size=1)
9
10    def forward(self, x):
11        return self.conv(x)

```

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

Code 5: Implementation of the complete UNet architecture.

```

1 class UNet(nn.Module):
2     """
3     Full UNet architecture
4     """
5     def __init__(self, n_channels=1, n_classes=2):
6         """
7             n_channels: Number of input channels
8                 → (e.g., 1 for grayscale, 3 for RGB)
9             n_classes: Number of output classes
10                → (e.g., 2 for binary segmentation)
11        """
12        super(UNet, self).__init__()
13        self.n_channels = n_channels
14        self.n_classes = n_classes
15
16        # Initial double convolution
17        self.inc = DoubleConv(n_channels, 64)
18
19        # Contracting path (encoder)
20        self.down1 = Down(64, 128)
21        self.down2 = Down(128, 256)
22        self.down3 = Down(256, 512)
23        self.down4 = Down(512, 1024)
24
25        # Expansive path (decoder)
26        self.up1 = Up(1024, 512)
27        self.up2 = Up(512, 256)
28        self.up3 = Up(256, 128)
29        self.up4 = Up(128, 64)
30
31        # Final convolution
32        self.outc = OutConv(64, n_classes)
33
34    def forward(self, x):
35        # Contracting path
36        x1 = self.inc(x)
37        x2 = self.down1(x1)
38        x3 = self.down2(x2)
39        x4 = self.down3(x3)
40        x5 = self.down4(x4)
41
42        # Expansive path with skip connections
43        x = self.up1(x5, x4)
44        x = self.up2(x, x3)
45        x = self.up3(x, x2)
46        x = self.up4(x, x1)
47
48        # Final convolution
49        pred = self.outc(x)
50
51        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 residual connections to improve gradient flow and prevent vanishing/exploding gradients.

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

For the implementation, I reference a public GitHub repository¹.

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.

```

1 class ResNetBasicBlock(nn.Module):
2     def __init__(self, in_channels, out_channels,
3                  stride=1):
4         super(ResNetBasicBlock, self).__init__()
5         self.conv1 = nn.Conv2d(
6             in_channels, out_channels,
7             kernel_size=3, stride=stride,
8             padding=1, bias=False)
9         self.bn1 = nn.BatchNorm2d(out_channels)
10        self.relu = nn.ReLU(inplace=True)
11        self.conv2 = nn.Conv2d(
12            out_channels, out_channels,
13            kernel_size=3, stride=1, padding=1,
14            bias=False)
15        self.bn2 = nn.BatchNorm2d(out_channels)
16        self.downsample = nn.Sequential()
17        if stride != 1 or in_channels !=
18            out_channels:
19            self.downsample = nn.Sequential(
20                nn.Conv2d(in_channels,
21                          out_channels,
22                          kernel_size=1,
23                          stride=stride,
24                          bias=False),
25                nn.BatchNorm2d(out_channels)
26            )
27
28    def forward(self, x):
29        identity = x
30
31        out = self.conv1(x)
32        out = self.bn1(out)
33        out = self.relu(out)
34
35        out = self.conv2(out)
36        out = self.bn2(out)
37
38        # Apply downsample to identity if needed
39        identity = self.downsample(identity)
40
41        # Add residual connection
42        out = out + identity
43        out = self.relu(out)
44
45        return out

```

The complete ResNet34 encoder is implemented as follows²:

Code 7: Implementation of the ResNet34 Encoder.

```

1 class ResNet34Encoder(nn.Module):
2     def __init__(self):
3         super(ResNet34Encoder, self).__init__()
4         # Define the ResNet34 encoder
5         self.conv1 = nn.Conv2d(3, 32,
6             kernel_size=7,
7             stride=2, padding=3,
8             bias=False)
9         self.bn1 = nn.BatchNorm2d(32)
10        self.maxpool = nn.MaxPool2d(kernel_size=3,
11            stride=2, padding=1)
12        self.conv2_x = self._make_layer(32, 64, 3)
13        self.conv3_x = self._make_layer(64, 128, 4,
14            stride=2)
15        self.conv4_x = self._make_layer(128, 256,
16            6, stride=2)
17        self.conv5_x = self._make_layer(256, 512,
18            3, stride=2)
19
20    def _make_layer(self, in_channels,
21                  out_channels, num_blocks, stride=1):
22        layers = []
23        for _ in range(num_blocks):
24            layers.append(
25                ResNetBasicBlock(in_channels,
26                                  out_channels, stride))
27        in_channels = out_channels
28        stride = 1
29        return nn.Sequential(*layers)
30
31    def forward(self, x):
32        x1 = self.conv1(x)
33        x1 = self.bn1(x1)
34        x1 = F.relu(x1)
35
36        # First block after maxpool
37        x2 = self.maxpool(x1)
38        x2 = self.conv2_x(x2)
39
40        # Remaining blocks
41        x3 = self.conv3_x(x2)
42        x4 = self.conv4_x(x3)
43        x5 = self.conv5_x(x4)
44
45        # Return feature maps for skip connections
46        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.

```

1 class ResNet34_UNet(nn.Module):
2     def __init__(self, n_channels=3, n_classes=1):
3         super(ResNet34_UNet, self).__init__()
4
5         self.encoder = ResNet34Encoder()
6
7         self.bridge = nn.Sequential(
8             nn.Conv2d(512, 1024, kernel_size=3,
9                     padding=1),
10            nn.BatchNorm2d(1024),
11            nn.ReLU(inplace=True),
12            nn.MaxPool2d(kernel_size=2, stride=2))
13
14         self.up1 = Up(1024, 512)
15         self.up2 = Up(512, 256)

```

¹<https://github.com/GohVh/resnet34-unet/blob/main/model.py>

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

```

16     self.up3 = Up(256, 128)
17     self.up4 = Up(128, 64)
18     self.up5 = Up(64, 32)
19     self.final_up = nn.ConvTranspose2d(32, 32,
20         kernel_size=2, stride=2)
21     # Final convolution
22     self.outc = OutConv(32, n_classes)
23
24     def forward(self, x):
25         # Encoder path
26         x1, x2, x3, x4, x5 = self.encoder(x)
27
28         # Bridge
29         x = self.bridge(x5)
30
31         # Decoder path with skip connections
32         x = self.up1(x, x5)
33         x = self.up2(x, x4)
34         x = self.up3(x, x3)
35         x = self.up4(x, x2)
36         x = self.up5(x, x1)
37
38         x = self.final_up(x)
39
40         # Final convolution
41         x = self.outc(x)
42
43         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 (BCEWithLogitLoss) 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.

```

1 def dice_score(pred_mask, gt_mask):
2     # Ensure binary masks with proper thresholding
3     # (detach from computation graph if needed)
4     with torch.no_grad():
5         # apply sigmoid to the prediction mask

```

```

5     pred_mask = (torch.sigmoid(pred_mask) >
6         0.5).float().flatten()
7     gt_mask = (gt_mask > 0.5).float().flatten()
8
9     intersection = torch.sum(pred_mask *
10        gt_mask)
11    union = torch.sum(pred_mask) +
12        torch.sum(gt_mask)
13
14    dice = (2.0 * intersection) / (union +
15        1e-8) # add a small epsilon to avoid
16        division by zero
17
18    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.

```

1 optimizer = optim.AdamW(model.parameters(),
2                         lr=args.learning_rate,
3                         weight_decay=args.j
4                         ↵ .weight_decay)

```

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

Code 11: Learning rate scheduler implementations.

```

1 if args.scheduler == 'plateau':
2     scheduler =
3         optim.lr_scheduler.ReduceLROnPlateau(
4             optimizer, mode='min', patience=3)
5 elif args.scheduler == 'cosine':
6     scheduler =
7         optim.lr_scheduler.CosineAnnealingLR(
8             optimizer, T_max=args.epochs)
9 elif args.scheduler == 'onecycle':
10    scheduler = optim.lr_scheduler.OneCycleLR(
11        optimizer,
12        max_lr=args.learning_rate,
13        epochs=args.epochs,
14        steps_per_epoch=len(train_loader),
15        pct_start=0.1,
16        anneal_strategy='cos')

```

```

16 else:
17     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.

```

1 # Training loop
2 for epoch in range(start_epoch, args.epochs):
3     print(f"\nEpoch {epoch+1}/{args.epochs}")
4
5     # Training phase
6     model.train()
7     train_loss = 0
8     train_dice = 0
9
10    train_pbar = tqdm(enumerate(train_loader),
11                      total=len(train_loader), desc="Training")
12    for i, batch in train_pbar:
13        images = batch['image'].to(device)
14        masks = batch['mask'].to(device)
15
16        # Forward pass
17        optimizer.zero_grad()
18        outputs = model(images)
19        loss = criterion(outputs, masks)
20
21        # Backward pass and optimize
22        loss.backward()
23        optimizer.step()
24
25        # Update metrics
26        batch_dice = dice_score(outputs, masks)
27        train_loss += loss.item()
28        train_dice += batch_dice

```

```

28
29
30
31     # Update the progress bar
32     train_pbar.set_postfix(loss=f"[loss] "
33                           f"\u2192 .item():.4f",
34                           dice=f"[batch_dice:] "
35                           f"\u2192 .4f",
36                           lr=f"[optimizer] "
37                           f"\u2192 .param_groups[0] "
38                           f"\u2192 ['lr']: .4E")
39
40
41     # Calculate average metrics
42     train_loss /= len(train_loader)
43     train_dice /= len(train_loader)
44
45     # Validation phase
46     model.eval()
47     val_loss = 0
48     val_dice = 0
49
50     with torch.no_grad():
51         for batch in tqdm(val_loader,
52                            total=len(val_loader),
53                            desc="Validation"):
54             images = batch['image'].to(device)
55             masks = batch['mask'].to(device)
56
57             outputs = model(images)
58             loss = criterion(outputs, masks)
59
60             val_loss += loss.item()
61             val_dice += dice_score(outputs, masks)
62
63     val_loss /= len(val_loader)
64     val_dice /= len(val_loader)
65
66     # Update ReduceLROnPlateau or CosineAnnealingLR
67     # scheduler
68     if args.scheduler == 'plateau':
69         scheduler.step(val_loss)
70     elif args.scheduler == 'cosine':
71         scheduler.step()
72
73     # Print progress
74     print(f"Train Loss: {train_loss:.4f}, Train"
75           f" Dice: {train_dice:.4f}")
76     print(f"Val Loss: {val_loss:.4f}, Val Dice: "
77           f"\u2192 {val_dice:.4f}")
78     print(f"Learning Rate: "
79           f"\u2192 {optimizer.param_groups[0]['lr']:.8f}")
80
81     # Save checkpoints and track best model
82     if val_dice > best_val_dice:
83         best_val_dice = val_dice
84         torch.save({
85             'model_state_dict': model.state_dict(),
86             'optimizer_state_dict': optimizer.state_dict(),
87             'val_dice': val_dice,
88             'epoch': epoch
89         }, os.path.join(args.save_dir,
90                         'best_model.pth'))

```

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

Code 13: Test set evaluation.

```

1 # Load best model for testing
2 best_model_path = os.path.join(args.save_dir,
3   ↪ 'best_model.pth')
4 if os.path.exists(best_model_path):
5   checkpoint = torch.load(best_model_path,
6     ↪ map_location=device)
7   model_]
8   ↪ .load_state_dict(checkpoint['model_state_dict'])
9   print(f"Loaded best model with validation Dice
10    ↪ score: {checkpoint['val_dice']:.4f}")
11
12 model.eval()
13 test_loss = 0
14 test_dice = 0
15
16 with torch.no_grad():
17   for batch in tqdm(test_loader,
18     ↪ total=len(test_loader), desc="Testing"):
19     images = batch['image'].to(device)
20     masks = batch['mask'].to(device)
21
22     outputs = model(images)
23     loss = criterion(outputs, masks)
24
25     test_loss += loss.item()
26     test_dice += dice_score(outputs, masks)
27
28 test_loss /= len(test_loader)
29 test_dice /= len(test_loader)
30
31 print(f"Test Loss: {test_loss:.4f}, Test Dice:
32    ↪ {test_dice:.4f}")
33
34 # Save final model with test results
35 torch.save({
36   'model_state_dict': model.state_dict(),
37   'test_loss': test_loss,
38   'test_dice': test_dice,
39 }, os.path.join(args.save_dir,
40   ↪ f'{args.model}_final.pth'))
41

```

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.

```

1 def preprocess_image(image_path, img_size=256):
2   """
3     Load and preprocess an image for inference
4
5   Args:
6     image_path: Path to the image file
7     img_size: Size to resize the image to
8
9   Returns:
10    Processed image tensor ready for model input
11  """
12  # Load image
13  image = Image.open(image_path).convert('RGB')
14
15  # Define preprocessing transforms
16  transform = T.Compose([
17    T.Resize((img_size, img_size)),
18    T.ToTensor(),
19    T.Normalize(mean=[0.485, 0.456, 0.406],
20      ↪ std=[0.229, 0.224, 0.225])
21

```

```

20  ])
21
22  # Apply transforms
23  image_tensor = transform(image)
24
25  return image_tensor, image
26
27 def predict(model, image_tensor, device,
28   ↪ threshold=0.5):
29   """
30     Generate a segmentation mask prediction
31
32   Args:
33     model: The trained segmentation model
34     image_tensor: Input image tensor
35     device: Device to run inference on
36     threshold: Threshold for binary segmentation
37
38   Returns:
39    Predicted mask as numpy array
40  """
41  with torch.no_grad():
42    # Add batch dimension and move to device
43    x = image_tensor.unsqueeze(0).to(device)
44
45    # Forward pass
46    output = model(x)
47
48    # Apply sigmoid and threshold
49    pred_mask = torch.sigmoid(output) >
50      ↪ threshold
51
52    # Convert to numpy
53    pred_mask = pred_mask.cpu().squeeze() [
54      ↪ .numpy().astype(np.uint8) *
55      ↪ 255
56
57  return pred_mask
58
59 def save_prediction(image, mask, output_path):
60   """
61   Visualize and save the prediction results
62
63   Args:
64     image: Original PIL image
65     mask: Predicted mask as numpy array
66     output_path: Path to save the visualization
67
68   # Create a figure with subplots
69   fig, axes = plt.subplots(1, 3, figsize=(15, 5))
70
71   # Plot original image
72   axes[0].imshow(image)
73   axes[0].set_title('Original Image')
74   axes[0].axis('off')
75
76   # Plot predicted mask
77   axes[1].imshow(mask, cmap='gray')
78   axes[1].set_title('Predicted Mask')
79   axes[1].axis('off')
80
81   # Plot overlay
82   image_np =
83     ↪ np.array(image.resize((mask.shape[1],
84     ↪ mask.shape[0])))
85   mask_rgb = np.stack([mask, np.zeros_like(mask),
86     ↪ np.zeros_like(mask)], axis=2)
87   overlay = image_np * 0.7 + mask_rgb * 0.3
88   overlay = np.clip(overlay, 0,
89     ↪ 255).astype(np.uint8)
90
91   axes[2].imshow(overlay)
92   axes[2].set_title('Overlay')
93
94

```

```

85 axes[2].axis('off')
86
87 plt.tight_layout()
88 plt.savefig(output_path, dpi=150,
89             bbox_inches='tight')
90 plt.close()
91
92 # Save raw mask as well
93 mask_path = output_path.replace('.png',
94                               '_mask.png')
95 mask_img = Image.fromarray(mask)
96 mask_img.save(mask_path)
97
98 def process_batch(model, image_paths, device,
99                     output_dir, img_size):
100     """
101     Process a batch of images
102
103     Args:
104         model: The trained segmentation model
105         image_paths: List of paths to image files
106         device: Device to run inference on
107         output_dir: Directory to save results
108         img_size: Input image size
109     """
110
111     for image_path in tqdm(image_paths,
112                           desc="Processing images"):
113         # Extract filename without extension
114         filename = os.path.splitext(os.path_
115                                     .basename(image_path))
116         [0]
117
118         # Preprocess image
119         image_tensor, original_image =
120             preprocess_image(image_path, img_size)
121
122         # Generate prediction
123         pred_mask = predict(model, image_tensor,
124                             device)
125
126         # Save visualization
127         output_path = os.path.join(output_dir,
128                                    f"{filename}_prediction.png")
129         save_prediction(original_image, pred_mask,
130                         output_path)

```

```

13     avg_dice: Average Dice score across the
14     → dataset
15     all_dices: List of individual Dice scores
16     → for each sample
17     """
18
19     net.eval()
20     all_dices = []
21
22     # Create output directory if it doesn't exist
23     if output_dir is not None and
24         → save_visualizations:
25         os.makedirs(output_dir, exist_ok=True)
26
27     with torch.no_grad():
28         for i, batch in enumerate(tqdm(data_loader,
29             → desc="Evaluating")):
30             images = batch['image'].to(device)
31             masks = batch['mask'].to(device)
32
33             # Forward pass
34             outputs = net(images)
35
36             # Calculate Dice score for each image in
37             → the batch
38             for j in range(images.size(0)):
39                 img_dice =
40                     → dice_score(outputs[j:j+1],
41                     → masks[j:j+1])
42                 all_dices.append(img_dice)
43
44             # Save visualization for some samples
45             if save_visualizations and output_dir
46                 → is not None and i % 5 == 0:
47                 for j in range(min(4,
48                     → images.size(0))): # Visualize
49                     up to 4 images per batch
50                     visualize_prediction(
51                         images[j].cpu(),
52                         masks[j].cpu(),
53                         outputs[j].cpu(),
54                         os.path.join(output_dir,
55                         → f'sample_{i}_{j}.png')
56                     )
57
58             # Calculate average Dice score
59             avg_dice = sum(all_dices) / len(all_dices)
60
61     return avg_dice, all_dices
62
63 def visualize_prediction(image, true_mask,
64     → pred_logits, save_path=None):
65     """
66     Visualize the model's prediction alongside the
67     → input image and ground truth mask
68
69     Args:
70         image: Input image tensor [C, H, W]
71         true_mask: Ground truth mask tensor [1, H,
72             → W]
73         pred_logits: Model's prediction logits
74             → tensor [1, H, W]
75         save_path: Path to save the visualization
76             → image
77     """
78
79     # Denormalize the image
80     mean = torch.tensor([0.485, 0.456,
81         → 0.406]).view(3, 1, 1)
82     std = torch.tensor([0.229, 0.224,
83         → 0.225]).view(3, 1, 1)
84     image = image * std + mean
85
86     # Convert tensors to numpy for visualization
87     image = image.permute(1, 2, 0).numpy()
88
89     # Save visualization
90     if save_path is not None:
91         plt.imsave(save_path, image)
92
93     return image
94
95     # Print summary statistics
96     print(f"Average Dice score: {avg_dice:.4f}")
97     print(f"Number of samples: {len(all_dices)}")
98     print(f"Output directory: {output_dir}")
99
100    # Print visualization
101    if save_path is not None:
102        print(f"Visualization saved to {save_path}")
103
104    return avg_dice, all_dices

```

```
1 def evaluate(net, data_loader, device,
2     ↪ output_dir=None, save_visualizations=False):
3     """
4         Evaluate a trained model on a dataset
5
6     Args:
7         net: The trained neural network model
8         data_loader: DataLoader for the evaluation
9         ↪ dataset
10        device: Device to run the evaluation on
11        ↪ (cuda or cpu)
12        output_dir: Directory to save visualization
13        ↪ results (if None, no visualizations are
14        ↪ saved)
15        save_visualizations: Whether to save
16        ↪ visualizations of segmentation results
17
18    Returns:
```

```

68     image = np.clip(image, 0, 1)
69
70     true_mask = true_mask.squeeze().numpy()
71
72     # Apply sigmoid to get probabilities and
73     # threshold to get binary mask
74     pred_probs = torch.sigmoid(pred_logits) |
75     # .squeeze().numpy()
76     pred_mask = (pred_probs >
77     # 0.5).astype(np.float32)
78
79     # Create the figure with three subplots
80     fig, axes = plt.subplots(1, 4, figsize=(16, 4))
81
82     # Plot the original image
83     axes[0].imshow(image)
84     axes[0].set_title('Original Image')
85     axes[0].axis('off')
86
87     # Plot the ground truth mask
88     axes[1].imshow(true_mask, cmap='gray')
89     axes[1].set_title('Ground Truth Mask')
90     axes[1].axis('off')
91
92     # Plot the predicted probability map
93     axes[2].imshow(pred_probs, cmap='viridis')
94     axes[2].set_title('Prediction Probabilities')
95     axes[2].axis('off')
96
97     # Plot the thresholded prediction mask
98     axes[3].imshow(pred_mask, cmap='gray')
99     axes[3].set_title('Predicted Mask')
100    axes[3].axis('off')
101
102    plt.tight_layout()
103
104    if save_path:
105        plt.savefig(save_path, dpi=150,
106                    bbox_inches='tight')
107        plt.close()
108    else:
109        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.

```

1  class OxfordPetDataset(torch.utils.data.Dataset):
2      def __init__(self, root, mode="train",
3                   transform=None):
4          assert mode in {"train", "valid", "test"}
5
6          self.root = root
7          self.mode = mode
8          self.transform = transform
9
10         self.images_directory =
11             os.path.join(self.root, "images")
12         self.masks_directory =
13             os.path.join(self.root, "annotations",
14                         "trimaps")
15
16         self.filenames = self._read_split() # read
17             # train/valid/test splits
18
19     def __len__(self):
20         return len(self.filenames)
21
22     def __getitem__(self, idx):
23         filename = self.filenames[idx]
24         image_path =
25             os.path.join(self.images_directory,
26                         filename + ".jpg")
27         mask_path =
28             os.path.join(self.masks_directory,
29                         filename + ".png")
30
31         image = np.array(Image.open(image_path) |
32                         .convert("RGB"))
33
34         trimap = np.array(Image.open(mask_path))
35         mask = self._preprocess_mask(trimap)
36
37         sample = dict(image=image, mask=mask,
38                         trimap=trimap)
39         return sample
40
41     @staticmethod
42     def _preprocess_mask(mask):
43         mask = mask.astype(np.float32)
44         mask[mask == 2.0] = 0.0
45         mask[(mask == 1.0) | (mask == 3.0)] = 1.0
46         return mask
47
48     def _read_split(self):
49         split_filename = "test.txt" if self.mode ==
50             "test" else "trainval.txt"
51         split_filepath = os.path.join(self.root,
52                                     "annotations", split_filename)
53         with open(split_filepath) as f:
54             split_data =
55                 f.read().strip("\n").split("\n")

```

```

42     filenames = [x.split(" ")[0] for x in
43         split_data]
44     if self.mode == "train": # 90% for train
45         filenames = [x for i, x in
46             enumerate(filenames) if i % 10 !=
47             0]
48     elif self.mode == "valid": # 10% for
49         validation
50         filenames = [x for i, x in
51             enumerate(filenames) if i % 10 ==
52             0]
53     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.

```

1  class SimpleOxfordPetDataset(OxfordPetDataset):
2      def __init__(self, root, mode="train"):
3          super().__init__(root, mode)
4          train_transform = T.Compose([
5              T.ToImage(),
6              T.RandomHorizontalFlip(p=0.5),
7              T.RandomVerticalFlip(p=0.5),
8              T.ColorJitter(brightness=0.2,
9                  contrast=0.2, saturation=0.2),
10             T.ToDtype(torch.float32, scale=True),
11         ])
12         val_transform = T.Compose([
13             T.ToImage(),
14             T.ToDtype(torch.float32, scale=True),
15         ])
16         self.transform = train_transform if mode ==
17             'train' else val_transform
18         self.normalize = T.Compose([
19             T.Normalize(mean=[0.485, 0.456, 0.406],
20                         std=[0.229, 0.224, 0.225])
21         ])
22
23     def __getitem__(self, *args, **kwargs):
24         sample = super().__getitem__(*args,
25             **kwargs)
26
27         # resize images
28         image = np.array(Image_
29             .fromarray(sample["image"]))
30             .resize((256, 256),
31             Image.BILINEAR)
32         mask = np.array(Image_
33             .fromarray(sample["mask"])).resize((256,
34             256), Image.NEAREST)
35         trimap = np.array(Image_
36             .fromarray(sample["trimap"]))
37             .resize((256, 256),
38             Image.NEAREST))
39
40         sample["trimap"] = np.expand_dims(trimap,
41             0)
42
43         sample["image"], sample["mask"] =
44             self.transform(image, mask)
45         sample["image"] =
46             self.normalize(sample["image"])
47
48         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 $\pm 20\%$.
- **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 256×256 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.

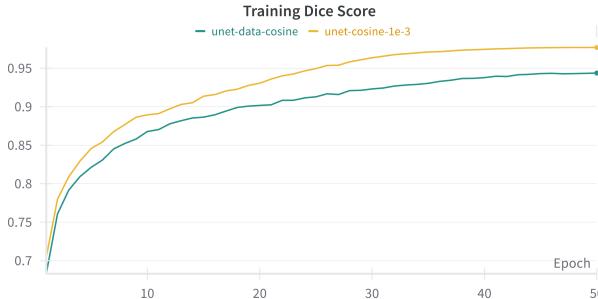


Figure 3: **The training Dice score over epochs.** `unet-data-cosine` is the model with data argumentation, while `unet-cosine-1e-3` is without data argumentation. The model without data argumentation achieves higher training Dice scores but performs worse on validation data.



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.

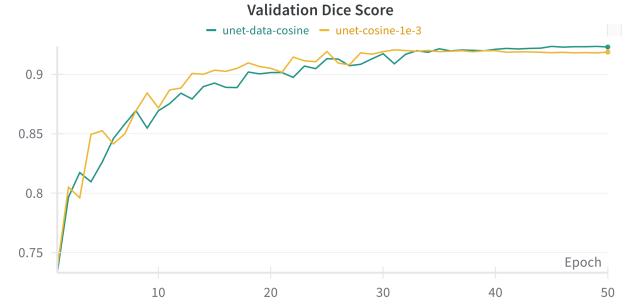


Figure 5: **The validation Dice score over epochs.** `unet-data-cosine` is the model with data argumentation, while `unet-cosine-1e-3` is without data argumentation. Data argumentation helps maintain higher validation Dice scores throughout training, demonstrating better generalization.

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 argumentation, while `resnet-cosine-1e-3` is without data augmentation.

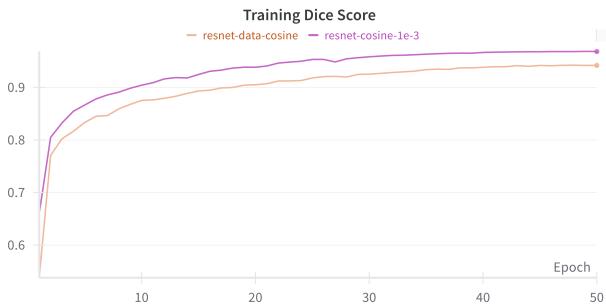


Figure 7: **The training Dice score over epochs for ResNet34_U-Net.** `resnet-data-cosine` is the model with data augmentation, while `resnet-cosine-1e-3` is without data augmentation. The non-augmented model shows higher training Dice scores but fails to generalize as well to validation data.

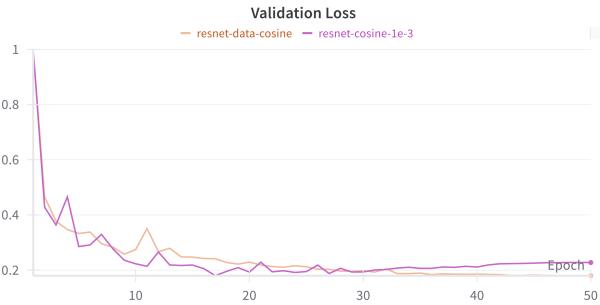


Figure 8: **The validation loss over epochs for ResNet34_U-Net.** `resnet-data-cosine` is the model with data augmentation, while `resnet-cosine-1e-3` is without data augmentation. The validation loss for the non-augmented model increases after epoch 20, indicating overfitting.

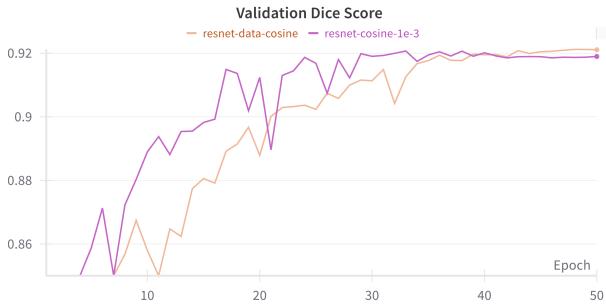


Figure 9: **The validation Dice score over epochs for ResNet34_U-Net.** `resnet-data-cosine` is the model with data augmentation, while `resnet-cosine-1e-3` is without data augmentation. The augmented model demonstrates more stable and higher validation performance throughout training.

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.

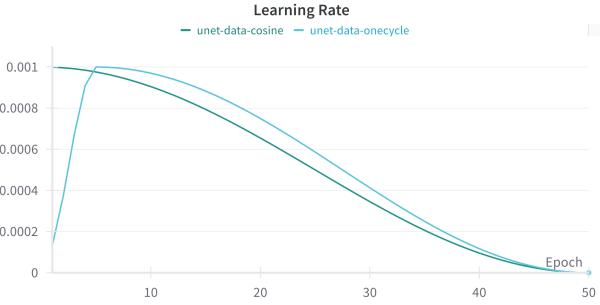


Figure 10: **Learning rate over epochs for U-Net models.** `unet-data-cosine` uses the CosineAnnealingLR scheduler, while `unet-data-onecycle` uses the OneCycleLR scheduler. Note the distinctive warm-up phase in the OneCycleLR approach.

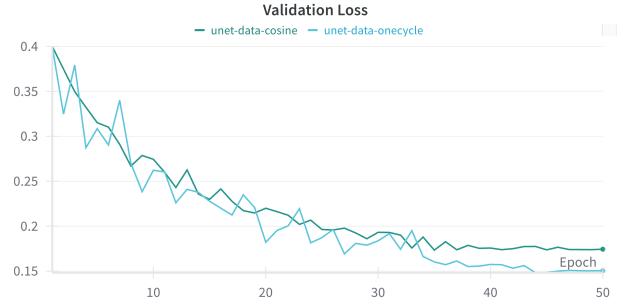


Figure 13: **Validation loss over epochs for U-Net models.** `unet-data-cosine` uses the CosineAnnealingLR scheduler, while `unet-data-onecycle` uses the OneCycleLR scheduler. The OneCycleLR approach maintains lower validation loss, indicating better generalization.

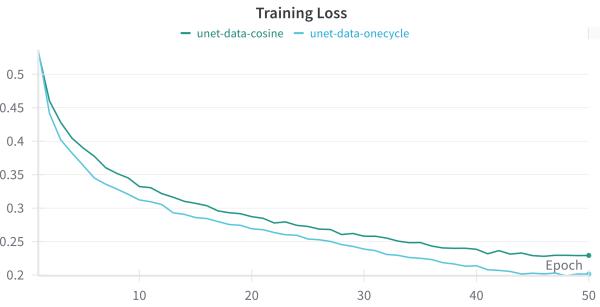


Figure 11: **Training loss over epochs for U-Net models.** `unet-data-cosine` uses the CosineAnnealingLR scheduler, while `unet-data-onecycle` uses the OneCycleLR scheduler. The OneCycleLR approach achieves lower training loss, particularly in the later stages of training.

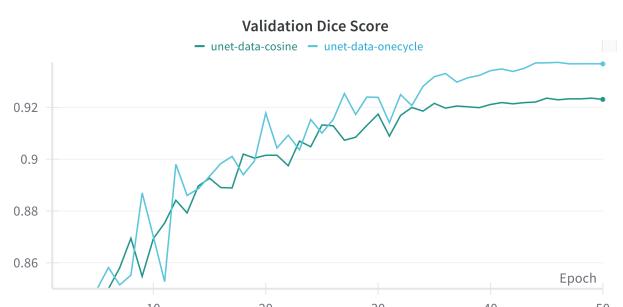


Figure 14: **Validation Dice score over epochs for U-Net models.** `unet-data-cosine` uses the CosineAnnealingLR scheduler, while `unet-data-onecycle` uses the OneCycleLR scheduler. The OneCycleLR approach achieves consistently higher validation Dice scores throughout training.

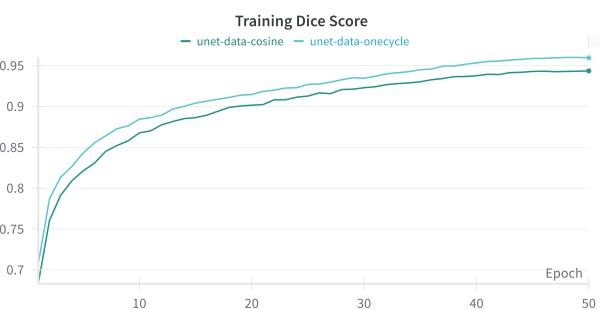


Figure 12: **Training Dice score over epochs for U-Net models.** `unet-data-cosine` uses the CosineAnnealingLR scheduler, while `unet-data-onecycle` uses the OneCycleLR scheduler. The OneCycleLR scheduler enables the model to achieve higher training Dice scores.

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.

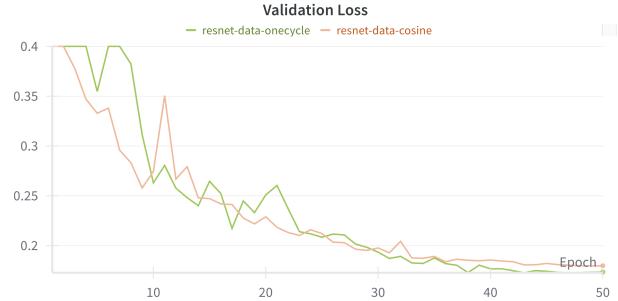


Figure 18: **Validation loss over epochs for ResNet34_U-Net models.** `resnet-data-cosine` uses the CosineAnnealingLR scheduler, while `resnet-data-onecycle` uses the OneCycleLR scheduler. The OneCycleLR approach maintains lower validation loss throughout most of the training process.



Figure 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.

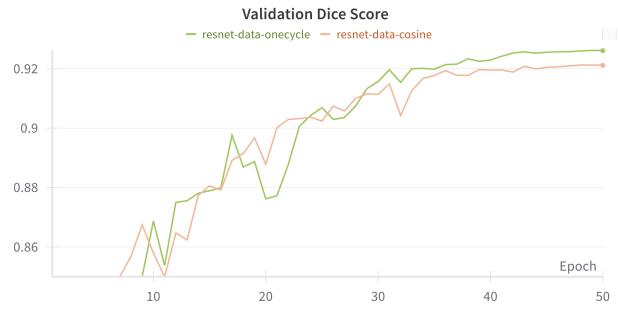


Figure 19: **Validation Dice score over epochs for ResNet34_U-Net models.** `resnet-data-cosine` uses the CosineAnnealingLR scheduler, while `resnet-data-onecycle` uses the OneCycleLR scheduler. The OneCycleLR approach demonstrates superior validation performance.



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.

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-

ceLR 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 ap-

proximately 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

I provide some segmentation results of both models in this section.

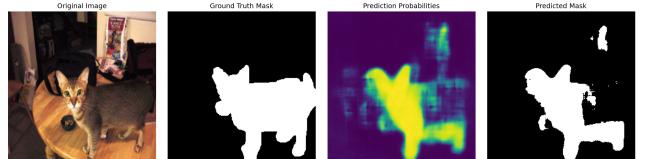


Figure 20: U-Net segmentation example 1. From left to right: original image, ground truth mask, prediction probability map, and binary prediction mask. The U-Net accurately captures the cat’s outline with high confidence.

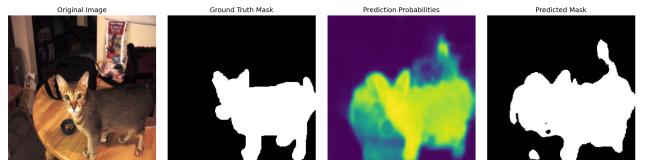


Figure 21: ResNet34_U-Net segmentation example 1. From left to right: original image, ground truth mask, prediction probability map, and binary prediction mask. The ResNet34_U-Net produces similar results to U-Net for this clear example.

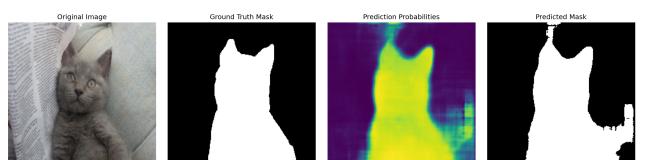


Figure 22: U-Net segmentation example 2. From left to right: original image, ground truth mask, prediction probability map, and binary prediction mask. The U-Net successfully segments the dog despite the challenging lighting conditions and complex pose.

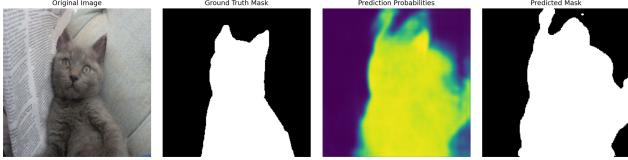


Figure 23: **ResNet34_U-Net segmentation example**
2. From left to right: original image, ground truth mask, prediction probability map, and binary prediction mask. The ResNet34_U-Net shows slightly less precise boundary delineation compared to U-Net for this example.

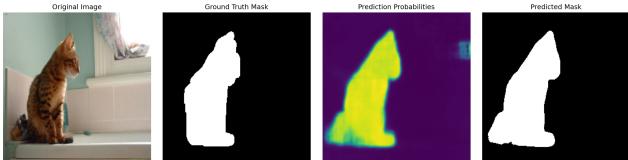


Figure 24: **U-Net segmentation example 3.** From left to right: original image, ground truth mask, prediction probability map, and binary prediction mask. The U-Net correctly identifies the cat and produces a clean segmentation mask.

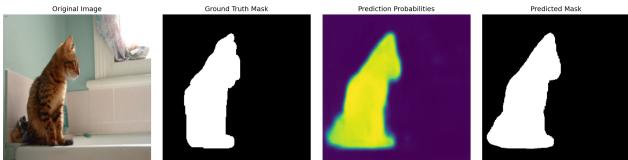


Figure 25: **ResNet34_U-Net segmentation example**
3. From left to right: original image, ground truth mask, prediction probability map, and binary prediction mask. The ResNet34_U-Net generates a more fragmented probability map with less confidence at the boundaries.

These visual examples illustrate the performance differences between the two architectures. While both models perform well on clear, well-defined images (as shown in Figures 24 and 25), the U-Net tends to produce more precise boundary segmentation with higher confidence at the object edges.

In the more challenging cases (Figures 22, 23, 20, and 21), the U-Net’s probability maps show stronger boundary definition and more consistent confidence across the pet’s body. The ResNet34_U-Net’s segmentations, while generally accurate, often display more uncertainty at object boundaries and occasionally produce more fragmented probability maps.

These visual results align with the quantitative findings, where U-Net achieved higher Dice scores and lower standard deviation. The examples suggest that U-Net’s architecture, with its direct skip connections and gradual spatial dimension reduction, better preserves the fine-grained spatial information necessary for precise boundary delineation in semantic segmentation tasks. The ResNet34_U-Net, while computationally more efficient, appears to sac-

rifice some precision in boundary detection, likely due to the more aggressive downsampling in its encoder path.

4 Execution steps

Please execute all scripts from the root directory.

4.1 Training

Code 18: The training commands.

```

1 python train.py --data\_\_path
    ↳ ./dataset/oxford-iiit-pet --epochs 50
    ↳ --scheduler onecycle --model unet --batch\_\_size
    ↳ 32 --learning-rate 0.001 # for training unet
2
3 python train.py --data\_\_path
    ↳ ./dataset/oxford-iiit-pet --epochs 50
    ↳ --scheduler onecycle --model resnet34\_unet
    ↳ --batch\_\_size 128 --learning-rate 0.001 # for
    ↳ training resnet34-unet

```

4.2 Evaluation

Code 19: The evaluation commands.

```

1 python src/evaluate.py --model
    ↳ saved_models/unet_final.pth --data\_path
    ↳ dataset/oxford-iiit-pet/ --model_type unet # for evalauting unet
2
3 python src/evaluate.py --model
    ↳ saved_models/resnet34_unet_final.pth
    ↳ --data\_path dataset/oxford-iiit-pet/
    ↳ --model_type resnet34_unet # for evalauting
    ↳ resnet34-unet

```

4.3 Inference

Code 20: The inference commands.

```

1 python src/inference.py --model
    ↳ saved_models/unet_final.pth --data\_path <image
    ↳ folder path> --model_type unet
2
3 python src/inference.py --model
    ↳ saved_models/resnet34_unet_final.pth
    ↳ --data\_path <image folder path> --model_type
    ↳ resnet34_unet

```

5 Discussion