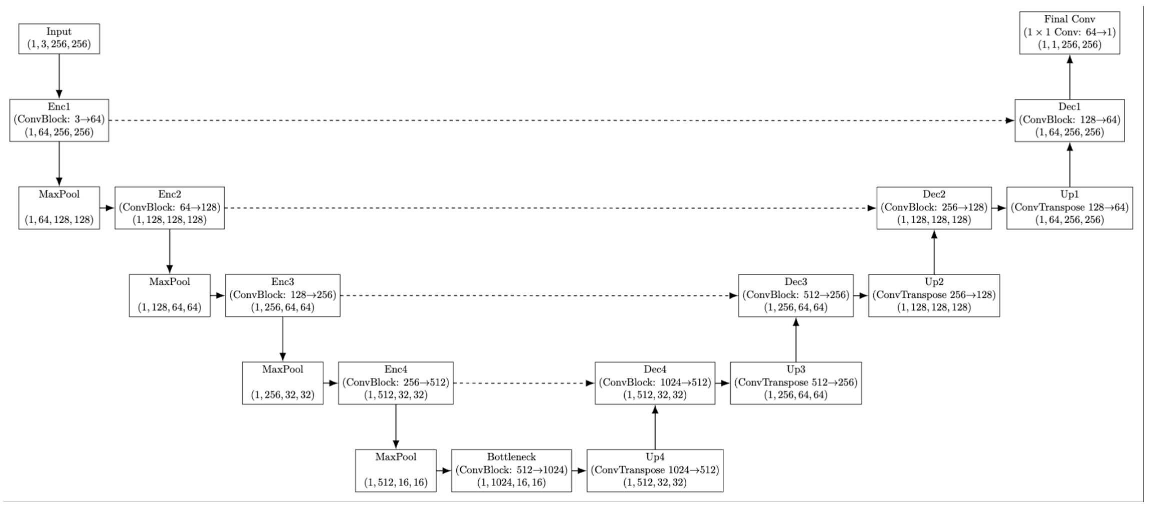
Lab2: Binary Semantic Segmentation

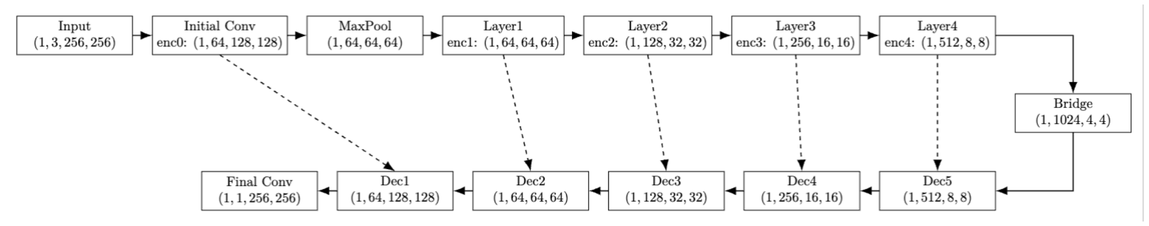
110550095 王皓平

1 Implementation Details

UNet Model


Input (1, 3, 256, 256)
Enc1 (ConvBlock: 3→64) (1, 64, 256, 256)
MaxPool (1, 64, 128, 128)
Enc2 (ConvBlock: 64→128) (1, 128, 128, 128)
MaxPool (1, 128, 64, 64)
Enc3 (ConvBlock: 128→256) (1, 256, 64, 64)
MaxPool (1, 256, 32, 32)
Enc4 (ConvBlock: 256→512) (1, 512, 32, 32)
MaxPool (1, 512, 16, 16)
Bottleneck (ConvBlock: 512→1024) (1, 1024, 16, 16)
Up4 (ConvTranspose 1024→512) (1, 512, 32, 32)
Dec4 (ConvBlock: 1024→512) (1, 512, 32, 32)
Up3 (ConvTranspose 512→256) (1, 256, 64, 64)
Dec3 (ConvBlock: 512→256) (1, 256, 64, 64)
Up2 (ConvTranspose 256→128) (1, 128, 128, 128)
Dec2 (ConvBlock: 256→128) (1, 128, 128, 128)
Up1 (ConvTranspose 128→64) (1, 64, 256, 256)
Dec1 (ConvBlock: 128→64) (1, 64, 256, 256)
Final Conv (1 × 1 Conv: 64→1) (1, 1, 256, 256)

ResNet34-UNet Model


Input (1, 3, 256, 256)
Initial Conv enc0: (1, 64, 128, 128)
MaxPool (1, 64, 64, 64)
Layer1 enc1: (1, 64, 64, 64)
Layer2 enc2: (1, 128, 32, 32)
Layer3 enc3: (1, 256, 16, 16)
Layer4 enc4: (1, 512, 8, 8)
Bridge (1, 1024, 4, 4)
Dec5 (1, 512, 8, 8)
Dec4 (1, 256, 16, 16)
Dec3 (1, 128, 32, 32)
Dec2 (1, 64, 64, 64)
Dec1 (1, 64, 128, 128)
Final Conv (1, 1, 256, 256)

1.1 Training

In this experiment, we train the model by repeatedly feeding data into the neural network over multiple iterations and continuously adjusting the model parameters based on the evaluation of a loss function, with the aim of optimizing segmentation performance.

1.1.1 Model and Loss Function

During training, we may choose different architectures like UNet ResNet34_UNet as the segmentation network. To balanced foreground-background pixel ratios, we combine BCE (*Binary CrossEntropy*) and Dice Loss. CrossEntropy measures the per-pixel probability discrepancy between the predictions and the ground truth, while Dice Loss reinforces the model' s ability to capture overlapping regions of the foreground. In practice, we simply sum these two losses to form the final loss function:

$$\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{BCE}} + \mathcal{L}_{\text{Dice}},$$

ensuring a balance between foreground-background classification and the completeness of the segmented regions.

1.1.2 Training Hyperparameters and Iteration

During the training phase, we set several key hyperparameters:

- **Learning Rate**: Controls the step size of the weight updates. If the learning rate is too high, the training process may become unstable; if it is too low, convergence will be slower.
- **Batch Size**: This determines the number of data samples fed into the network at each training step. A larger batch size might require more GPU memory, while a smaller one may lead to unstable model updates.
- **Epochs**: The number of complete passes through the entire training dataset, with parameter updates occurring after each pass. Increasing the number of epochs gives the model more opportunities to fully converge, but it also increases the risk of overfitting.
- **Optimizer**: In this project, we adopt the Adam (*Adaptive Moment Estimation*) optimizer for parameter updates. Adam automatically adjusting the learning rate for each parameter, leading to stable convergence and good performance on a variety of tasks.

1.2 Evaluation

To assess the performance of our segmentation model, we employ the Dice score. This metric measures the spatial overlap between the predicted segmentation mask and the ground truth mask. Formally, if P represents the set of predicted foreground pixels and T the ground truth foreground pixels, the Dice score is defined as:

$$\text{Dice} = \frac{2\,|P \cap T|}{|P| + |T|}.$$

The score ranges from 0 to 1, with 1 indicating perfect agreement between the prediction and the reference.

In our evaluation procedure, each input image is fed into the trained model to obtain a predicted mask, which is subsequently thresholded to produce a binary output. We then compute the Dice coefficient between this binarized prediction and the corresponding ground truth. By averaging the Dice scores over all samples in the validation or test set, we obtain a single quantitative measure of the model’ s segmentation accuracy.

1.3 Inference

During inference, we initialize the model (e.g., UNet or ResNet34_UNet) with its best-trained weights and feed batches of test data through a forward pass to obtain per-pixel foreground probabilities. We then apply a threshold 0.5 to generate binary masks. Finally, we use metrics such as the Dice score to quantitatively assess the model’ s segmentation quality.

2 Data Preprocessing

In this experiment, we employ a series of preprocessing and augmentation strategies on the Oxford-IIIT Pet Dataset to ensure that the model learns stable and generalizable representations under various image conditions. First, we split the data into train, valid, and test subsets based on a provided file list. Then, to maintain consistency across the dataset, we resize all images and masks to a fixed size 256. During the loading process, a custom dataset class (`OxfordPetDataset`) is used to handle file paths, read images, and convert them into NumPy arrays and `PIL` objects, with the `DataLoader` ultimately enabling batch processing and multi-threaded data handling.

2.1 Image Augmentation and Preprocessing

To enhance the model's robustness in foreground segmentation, we apply various augmentation techniques to the training images, including:

- **Horizontal Flip**: The image and its corresponding mask are flipped horizontally to simulate variability in object orientation.
- **Rotation**: The image is randomly rotated within a specified range of angles to reduce the model's dependence on a fixed orientation.
- **Brightness & Contrast Adjustment**: The lighting conditions of the image are altered within reasonable limits to enhance the model's adaptability to different exposure levels.

At the same time, we normalize the pixel values to the range $[0, 1]$, and optionally, we can further map the image distribution to a fixed mean and standard deviation using the Normalize command to accelerate model convergence.

2.2 Unique Data Mixing Strategies: MixUp and CutMix

Unlike conventional augmentation methods that only utilize basic transformations such as rotation and flipping, this experiment additionally incorporates two data mixing strategies, **MixUp** and **CutMix**, to boost data diversity:

- **MixUp**: The original image is blended with another randomly selected image by weighting their pixels using a λ value sampled from a Beta distribution. The corresponding masks are mixed in the same way and then binarized. This approach effectively blurs the boundaries between different objects, encouraging the model to learn more generalized features.
- **CutMix**: A random rectangular region is selected, and a patch from another image along with its corresponding mask is inserted into this region. This method preserves more of the original image structure while also introducing foreground samples of varying shapes and positions.

Since these mixing strategies are relatively uncommon in segmentation tasks, they represent one of the distinctive features of our preprocessing approach compared to standard methods, further enhancing the model’ s ability to discern the shapes and boundaries of the foreground.

3 Analyze the Experiment Result

3.1 Effect of Batch Size

To investigate how different batch sizes influence training dynamics and segmentation performance, we experimented with batch sizes of 2, 8, and 64. Figure 1 shows the training loss curves and the validation Dice score curves for these three settings.

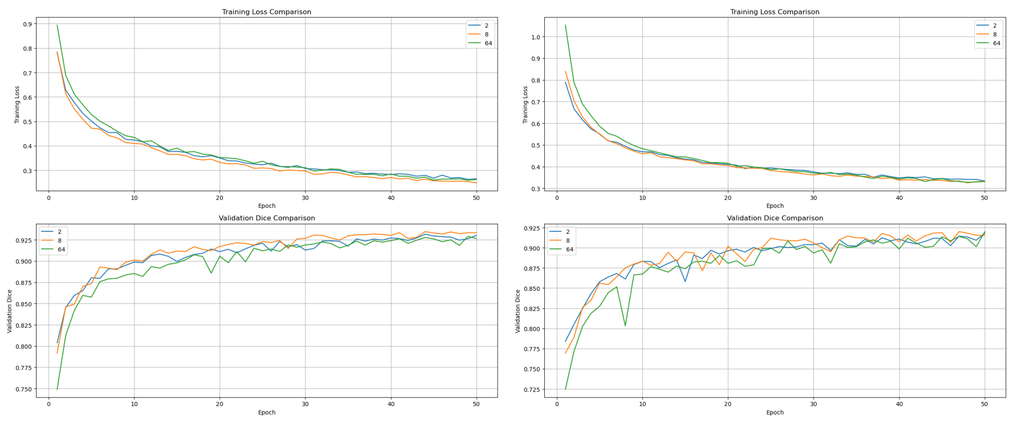


Figure 1: **Left:** Training loss comparison for batch sizes of 2, 8, and 64. **Right:** Validation Dice comparison for these batch sizes.

All three batch sizes yield relatively stable training loss curves. However, in terms of the Dice score comparison, a batch size of 8 performs better than 64. This may be because a smaller batch size introduces greater stochasticity during training, which can sometimes help the model escape poor local minima and improve generalization Overall, all three batch sizes achieve satisfactory segmentation performance.

3.1.1 Effect of Epoch Count

To investigate how different numbers of epochs influence the training and validation performance, we experimented with 20, 50, and 70 epochs. Figure 2 shows the training loss curves and validation Dice score curves for these three settings.

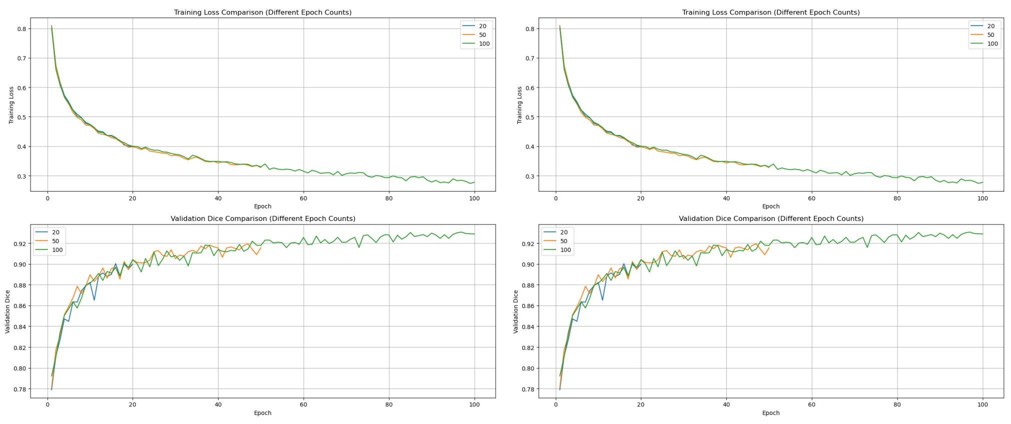


Figure 2: **Left:** Training loss comparison for 20, 50, and 70 epochs. **Right:** Corresponding validation Dice scores for these epoch counts.

Increasing the number of epochs generally reduces the training loss and improves the validation Dice score, indicating that the model has had enough time to converge. However, when the epoch count exceeds a certain range, the improvement in the validation Dice score tends to decline, showing that further increases in epochs yield limited benefits. In some cases, an excessively high number of epochs may lead to overfitting. Although the graph shows that 100 epochs achieve the best performance, in reality, 50 epochs are sufficient for this task.

3.2 Effect of Learning Rate

To investigate how different learning rates influence model convergence and segmentation performance, we conducted experiments with learning rates of `0.001`, `0.0001`, and `0.00001`. Figure 3 shows the training loss curves and validation Dice score curves for these three settings.

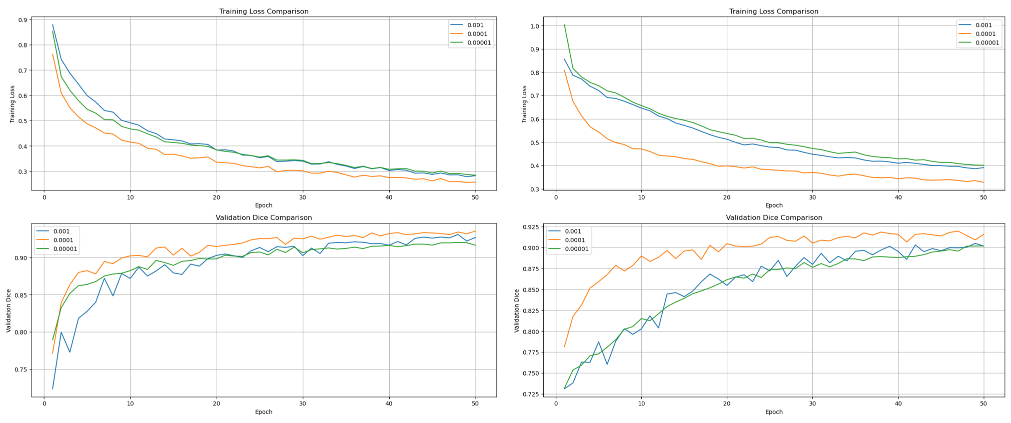


Figure 3: **Left:** Training loss comparison for learning rates 0.001, 0.0001, and 0.00001. **Right:** Corresponding validation Dice scores for these learning rates.

For both models, a moderate learning rate of 0.0001 yields the best performance in terms of training loss convergence and Dice score. The main reason is that it strikes a good balance between convergence speed and gradient update stability. When the learning rate is too high, the model may overfit in some cases or converge to a suboptimal solution. Conversely, if the learning rate is too low, the convergence process becomes too slow, requiring more training epochs to achieve comparable performance.

4 Execution Steps

To build the environment:

```
conda create -n dlp_hw2 python=3.10.16
conda activate dlp_hw2
pip install -r requirents.txt
```

To train the model:

```
python src/train.py --data_path ./dataset/oxford-iiit-pet -e 50 -b 4 -lr 1e-4
```

In this command:

- **–data_path** specifies the directory where the input data is stored.
- **–epochs** defines the number of complete passes over the training dataset.
- **–batch_size** sets the number of samples per batch during training.
- **–learning_rate** sets the initial learning rate for the optimizer.
- **–model_name** indicates which model architecture to use, default UNet.

To test the result:

```
python src/inference.py --model ./saved_models/best_unet_50_4_0.0001.pth \
--data_path ./dataset/oxford-iiit-pet -b 4
```

5 Discussion

5.1 Alternative Architectures

In this study, we primarily employed UNet and its ResNet34_UNet variant for binary semantic segmentation. However, based on our understanding of the models and the Oxford-IIIT Pet Dataset, the following alternative architectures may potentially yield better results:

1. **DeepLabv3+**: DeepLabv3+ uses dilated convolutions combined with an encoder-decoder structure to capture multi-scale contextual information. For pet images, which may include objects of varying sizes and poses, this architecture could better handle blurred boundaries and background noise, resulting in improved segmentation accuracy.

2. **PSPNet (Pyramid Scene Parsing Network)**: PSPNet leverages a pyramid pooling module to aggregate global contextual information, balancing local details with overall scene structure. This can be especially useful in differentiating complex foreground objects from the background, a common challenge in the dataset.

3. **HRNet (High-Resolution Network)**: HRNet maintains high-resolution representations throughout the network by fusing multi-scale features. This helps preserve spatial details crucial for precise boundary detection and may lead to enhanced segmentation of fine details in pet images.

4. **Transformer-based Segmentation Architectures (e.g., SegFormer, SETR)**: Transformer models excel in capturing long-range dependencies and modeling global context. Integrating transformers into segmentation networks may complement CNN-based approaches, resulting in improved feature fusion and overall segmentation performance.

5.1.1 Potential Research Directions

Based on the above alternative architectures and our current model' s performance, several research avenues are worth exploring:

- **Incorporating Attention Mechanisms and Adaptive Fusion**: Investigate the integration of attention modules (e.g., channel attention, spatial attention, or self-attention) into segmentation networks to enhance the discrimination of object boundaries and improve feature fusion.

- **Hybrid Loss Functions and Multi-task Learning**: Beyond the traditional CrossEntropy and Dice Loss, explore the combination of additional losses (e.g., edge loss, IoU loss) to provide a more balanced training signal. Additionally, adopting a multi-task learning framework that jointly tackles segmentation, edge detection, and depth estimation may result in more robust feature learning.

- **Self-supervised and Weakly-supervised Learning**: For scenarios where annotated data are limited or expensive, self-supervised pre-training or weakly-supervised learning strategies could be leveraged to enhance model generalization and reduce reliance on fully labeled datasets.

- **Neural Architecture Search (NAS)**: Utilize NAS to automatically discover optimal network architectures tailored to the segmentation task, potentially achieving a better balance between model complexity and computational efficiency.