1. Implementation Details

The core task of this project is to implement binary semantic segmentation using deep learning. The main modules include data loading, model training, evaluation, as well as inference and result visualization. Below are the detailed implementation specifics and how each part collaborates.

1. **Data Loading and Preprocessing**
    - **Data Loading**
      The data loading module is located in oxford_pet.py and provides the load_dataset function. This function loads the corresponding dataset based on the passed mode parameter (e.g., "train", "valid", or "test") and preprocesses both the images and their corresponding binary annotations. The preprocessing steps include:
        - Resizing and normalizing the images (for example, scaling pixel values to the range [0, 1]).
        - The output format is a dictionary containing the keys "image" and "mask" (for the test set, if the Ground Truth does not exist, "mask" is set to None).
    - **DataLoader**
      In both training and inference scripts, the torch.utils.data.DataLoader is used to package the dataset into batches, which facilitates subsequent model iteration reading. During training, shuffle=True is set, while for validation and testing, shuffle=False is used.
2. **Model Construction**
    - **Model Architecture**
      The project primarily uses UNet as the segmentation model, defined in models/unet.py. The model takes a 3-channel color image as input and outputs a single-channel binary prediction map. Additionally, an alternative implementation, ResNet34_UNet (presented as comments in the code), is provided to compare the performance of different architectures.
    - **Device and Weight Loading**
      Both training and inference automatically select GPU (if available) or CPU based on the current environment. After training, the model weights with the best performance are saved to a specified directory; during inference, the saved weights are loaded to restore the model state.

3. **Model Training**
    - **Training Process**

      The training code is located in train.py. In each epoch:
        - The training set is iterated over, extracting images and masks from each batch. Images are converted in type (if the pixel range is 0–255, they are normalized to [0, 1]) to ensure they match the model's input requirements.
        - **Forward Propagation:** The images are passed through the model to obtain output logits.
        - **Loss Calculation:** BCEWithLogitsLoss is used as the loss function. Since the model outputs logits, there is no need to apply a sigmoid before computing the loss.
        - **Backpropagation and Parameter Update:** The Adam optimizer is used to update the model parameters.
        - The average training loss for each epoch is recorded.
    - **Evaluation and Metric Recording**

      At the end of each epoch, the program performs evaluation on the validation set in two ways:
        - It calculates the average loss on the validation set.
        - It computes the average Dice Score through the evaluate() function. This function applies sigmoid activation and binarizes the model predictions, then compares them with the Ground Truth to compute the Dice Score. The training script records the training loss, validation loss, training Dice Score, and validation Dice Score for each epoch. Finally, the plot_training_metrics function is used to plot these metrics as line charts and save them as PNG files for intuitive analysis of the training process.
    - **Best Model Saving**

      When the validation Dice Score of a particular epoch exceeds the current best value, the current model weights are saved to a specified directory, ensuring that the best model can be loaded during inference.
4. **Model Inference and Result Visualization**
    - **Inference Process**

      The inference code is located in inference.py, and the process is as follows:

- The test set is loaded using load_dataset (with mode set to "test"), and data is read in batches through DataLoader.
- The best model weights saved during training are loaded, and the model is set to evaluation mode.
- For each batch of images, a forward pass is performed to obtain the predicted mask. The predicted mask is processed with sigmoid activation and binarized using a threshold of >0.5.

- **Result Visualization**

  To visually compare the model's predictions with the Ground Truth (if available), the project implements the visualize_comparison function (defined in utils.py).

  This function displays the results in a single image with three subplots:

  1. The original input image.
  2. The image with the Ground Truth Mask overlaid (the masked area is filled with black).
  3. The image with the Predicted Mask overlaid (the masked area is filled with black). To achieve a fully opaque effect (i.e., the masked area is completely black), when alpha=1, the function directly returns the result with the masked area painted black, avoiding semi-transparent blending. The final result is saved as a PNG file for inspection.

5. **Module Collaboration Summary**
   - The **Data Loading Module** is responsible for reading and preprocessing images and annotations from the raw dataset, outputting them in a unified format (in the form of a dictionary).
   - The **Model Module** (either UNet or ResNet34_UNet) implements the construction and forward propagation of the segmentation network.
   - The **Training Module** uses the data loader and model to perform forward and backward propagation, loss calculation, metric evaluation, and model saving.
   - The **Evaluation Module** measures the segmentation performance of the model using the Dice Score.
   - The **Visualization Module** utilizes line charts to display the changes in training and validation loss, as well as Dice Scores, and shows a comparison of the original image, the Ground Truth, and the predicted mask during inference.

Each module collaborates through a unified data format and interfaces (e.g., each batch contains the keys "image" and "mask" in a dictionary), ensuring that the entire process of training, evaluation, and inference is coherent and reproducible.

Model Design and Architecture Details

1. UNet Model

Design Concept:
The UNet model was originally developed for biomedical image segmentation. Its main idea is to use an encoder-decoder structure. The encoder gradually downsamples the input to capture high-level abstract features, while the corresponding upsampling with skip connections gradually recovers the spatial details of the image. This design effectively preserves local spatial information while maintaining global semantic context, which is essential for precise segmentation tasks.

Architecture Details:

1. **Encoder (Downsampling):**
    - Multiple consecutive convolutional blocks (DoubleConv) are used, where each DoubleConv block includes two convolution layers, batch normalization, and ReLU activation.
    - After each DoubleConv block, a max pooling layer is applied, which halves the feature map size while progressively increasing the number of channels.
2. **Bottleneck:**
    - At the deepest part of the encoder, the number of channels is further doubled and processed by a DoubleConv block to capture the richest semantic features.
3. **Decoder (Upsampling):**
    - Upsampling is performed using transposed convolutions (here, nn.ConvTranspose2d) to restore the feature map dimensions.
    - After each upsampling step, the corresponding encoder features are concatenated with the upsampled features via skip connections, then merged using another DoubleConv block.

- Finally, a 1x1 convolution is used to convert the number of channels to match the target segmentation classes (in this case, 1 for binary segmentation).

4. **Output and Upsampling:**
    - To ensure the output size matches the original input, bilinear interpolation is applied after the decoder to upsample the final segmentation map to the same size as the input image.

Training Settings:

- Input images are 3-channel color images, and the output is a single-channel binary segmentation map.
- The loss function used is BCEWithLogitsLoss; since the model outputs logits, there is no need to apply a sigmoid before loss calculation.
- The Adam optimizer is used for parameter updates, with the learning rate typically adjusted based on the validation results.
- During training, the training and validation losses as well as segmentation performance metrics (e.g., Dice Score) are recorded, and the model weights are saved when the validation performance is best.

Modifications Compared to the Original U-Net:
While our implementation is fundamentally aligned with the original U-Net paper by Ronneberger et al., which employs an encoder–decoder structure with skip connections for feature fusion at different levels, we have made several modifications:

1. **Inclusion of Batch Normalization:**
    - **Original Paper:** The original U-Net paper did not use Batch Normalization.
    - **Our Implementation:** BatchNorm is added after each convolutional layer.
    - **Reason:** Batch Normalization stabilizes training, accelerates convergence, and reduces sensitivity to initialization, thereby enhancing the model's generalization ability.
2. **ReLU Activation and In-place Operation:**
    - **Original Paper:** The original paper uses ReLU as the activation function without specifying the use of in-place operations.
    - **Our Implementation:** ReLU is applied with inplace=True to save memory.

- **Reason:** In-place operations reduce memory usage, which is particularly important when processing high-resolution images, without affecting model performance.

3. **Upsampling Strategy:**
   - **Original Paper:** The original U-Net uses transposed convolution for upsampling and requires cropping the skip connection features to match dimensions.
   - **Our Implementation:** Two upsampling options are provided:
     - Using bilinear upsampling via nn.Upsample and applying padding when necessary to match dimensions.
     - Alternatively, using nn.ConvTranspose2d for transposed convolution.
   - **Reason:** Bilinear upsampling has lower computational cost and avoids the checkerboard artifacts sometimes introduced by transposed convolutions. Padding helps to maintain consistent dimensions for feature fusion, making the implementation more flexible and easier to debug.
4. **Output Layer and Dimension Recovery:**
   - **Original Paper:** The final 1x1 convolution converts the feature map into the segmentation result, but cropping may be required to match the original image size in some cases.
   - **Our Implementation:** Bilinear interpolation (F.interpolate) is additionally applied after the decoder to upsample the output to the original image size.
   - **Reason:** This simplifies the post-processing step, ensuring that the final segmentation map matches the original image dimensions and eliminating the need for additional cropping.

---

2. ResNet34+UNet Model

Design Concept:

The ResNet34+UNet model introduces the ResNet34 encoder into the UNet architecture. ResNet34 utilizes residual blocks to effectively mitigate the vanishing gradient problem in deep neural networks, allowing for a deeper and more stable network. By combining the ResNet34 encoder with the UNet decoder, this model

leverages the pre-designed ResNet34 as a powerful feature extractor and then uses upsampling with skip connections to reconstruct the segmentation map.

Architecture Details:

1. **ResNet34 Encoder:**
    - The initial convolution layer (7x7 convolution followed by BatchNorm and ReLU) performs initial feature extraction from the input image, followed by a max pooling layer to further reduce the dimensions.
    - Following the ResNet34 architecture, the network proceeds through several layers composed of BasicBlocks, outputting feature maps with 64, 128, 256, and 512 channels respectively.
    - In each BasicBlock, a 3x3 convolution, BatchNorm, and ReLU are applied, followed by another 3x3 convolution and BatchNorm. A shortcut path (possibly with dimensionality reduction) is added, and finally, ReLU activation is applied.
2. **UNet Decoder:**
    - The decoder integrates skip connection information from the ResNet34 encoder (e.g., outputs from conv1, layer1, layer2, and layer3) and uses Up modules to progressively restore the feature map dimensions.
    - Each Up module performs upsampling, concatenates the upsampled features with the corresponding encoder features, and then merges them using a DoubleConv block.
    - Finally, a 1x1 convolution maps the channels to the number of classes for the segmentation task, and interpolation is used to adjust the final output size to match the original image dimensions.

Training Settings:

- Thanks to the rich feature extraction capabilities of the ResNet34 encoder, the ResNet34+UNet model generally achieves better segmentation performance in complex scenarios.
- The training process is similar to that of the UNet, using BCEWithLogitsLoss with the Adam optimizer, and learning rate and other hyperparameters are adjusted based on validation performance.
- During training, the losses and Dice Scores for each epoch are recorded to facilitate subsequent performance analysis and model selection.

Modifications Specific to Our Implementation:

The ResNet34+UNet model in our implementation largely retains the powerful feature extraction ability of the original ResNet34. However, to adapt it for the segmentation task, we made the following adjustments (which differ from the design described in literature for ultrasound pelvic muscle arch segmentation):

1. **Removal of the Classification Layer:**
    - **Original Design:** In image classification, ResNet34 ends with global average pooling, a fully connected layer (FC), and a softmax output to generate a 1000-dimensional classification result.
    - **Our Adjustment:** To preserve spatial information, we removed the average pooling and fully connected layers, retaining only the final convolutional block (e.g., the 512-channel feature map), which produces a high-resolution feature map with spatial dimensions. After processing through the UNet decoder, the final output is upsampled to match the original image size (and the number of channels is adjusted as needed for the segmentation task, e.g., 1 or 256).
2. **Connection to the UNet Decoder:**
    - **Original Design:** The original ResNet34 is designed solely for classification and does not include upsampling or skip connections.
    - **Our Adjustment:** We use ResNet34 as the encoder and pass intermediate features (from layers such as conv1, layer1, layer2, and layer3) as skip connections into the UNet decoder. This combines the powerful feature extraction of ResNet34 with the fine detail recovery of the UNet, thereby improving the precision and accuracy of the segmentation results.
3. **Channel Adjustment and Fusion:**
    - **Original Design:** For classification, ResNet34 ultimately outputs a 512-channel feature map, which is then processed by average pooling and fully connected layers, losing spatial resolution.
    - **Our Adjustment:** To adapt to the UNet decoder, we modified the ResNet34 structure to preserve the complete spatial dimensions at the end of the encoder (e.g., maintaining the feature map size at H/32 × W/32). Upsampling and skip connections then fuse features from the encoder's other layers. In this process, additional convolution or upsampling layers (such as ConvTranspose2d or bilinear upsampling) may be used to reduce the number of channels so that the final output feature map meets the segmentation task requirements.

In our implementation, the ResNet34+UNet model fully leverages the feature extraction capabilities of ResNet34. However, to adapt it for segmentation, we had to remove the average pooling, fully connected layers, and softmax output used for classification, retaining the convolutional feature maps with spatial information and fusing them with the UNet decoder. This approach preserves more spatial details, thereby producing high-quality pixel-level segmentation results and improving the model's generalization ability in complex scenarios.

2. Data Preprocessing

In this project, we use the Oxford-IIIT Pet dataset for binary semantic segmentation. The data preprocessing and augmentation strategies mainly include the following steps:

1. **Data Download and File Organization**
    - The system first checks whether the "images" and "annotations" subfolders exist in the directory. If not, the required image and annotation archives are automatically downloaded from the internet using the OxfordPetDataset.download() method and then extracted to the specified directory with extract_archive().
    - This process ensures that the data is automatically downloaded and organized during the initial run, making subsequent data loading more convenient.
2. **Dataset Class Design**
    - The basic OxfordPetDataset class is responsible for reading the images and their corresponding trimaps (i.e., the original annotations). Within this class, the _preprocess_mask method converts the trimap into a binary mask (treating areas labeled as 1 and 3 as foreground, and areas labeled as 2 as background).
    - To increase the diversity of training data, we implemented the SimpleOxfordPetDataset class by inheriting from OxfordPetDataset. In training mode, each original sample generates two versions:
        - Even indices: Use the original image directly.
        - Odd indices: Apply random data augmentation.
3. **Data Augmentation Strategies**
    - In the SimpleOxfordPetDataset.__getitem__ method, we perform random augmentation on training samples, mainly including:

        - **Random Cropping and Zoom Augmentation:**
          Simulate a zoom effect by randomly cropping a region of the image and then resizing it back to the original dimensions. This method helps the model learn features at different scales, thereby enhancing its robustness to variations in object scale.
        - **Grayscale Conversion:**
          Convert the image to grayscale and then back to RGB. This operation forces the model not to rely solely on color information but to pay more attention to shape and texture features, which is helpful in scenarios with significant color variations.
        - **Brightness Adjustment (Color Jitter):**
          Randomly adjust the brightness of the image (multiplying by a random factor) so that the image maintains good segmentation performance under different lighting conditions. This helps the model generalize better in various illumination environments.
    - Finally, regardless of whether the sample is original or augmented, all images are resized to 256×256. They are then converted into numpy arrays and reformatted from HWC to CHW to match the model's input requirements.
4. **Comparison with Standard Preprocessing Methods and Advantages**
    - **Standard Methods:**
      Standard preprocessing typically includes only image resizing, normalization, and basic format conversion, lacking handling of geometric or color variations.
    - **Advantages of Our Approach:**
        - **Enhanced Diversity:**
          Through random cropping, zooming, grayscale conversion, and brightness adjustment, a more diverse set of training samples is generated, reducing the risk of overfitting and improving the model's generalization across different scenarios.
        - **Scale Robustness:**
          Random cropping and zooming enable the model to learn features at different scales, which is beneficial for handling objects with significant size variations.
        - **Illumination Invariance:**
          Brightness adjustments help the model become less sensitive

to changes in lighting conditions, ensuring consistent segmentation performance.

- **Reduced Dependence on Color:**
  Grayscale conversion encourages the model to focus on shape and texture information, further enhancing its stability in cases of abnormal color conditions.

In summary, by incorporating multiple data augmentation strategies into our preprocessing pipeline, we not only increase the quantity of training samples but also enable the model to generalize better in diverse environments and across objects of various scales. This approach provides a significant advantage over standard methods that only involve simple resizing and normalization.

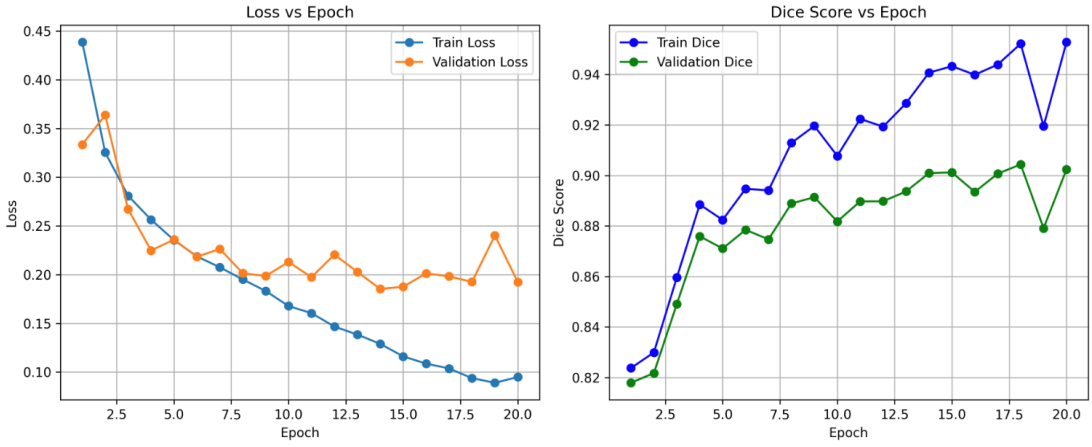
3. Analyze the experiment results

1. Comparison of Different Models and Augmentation

Strategies

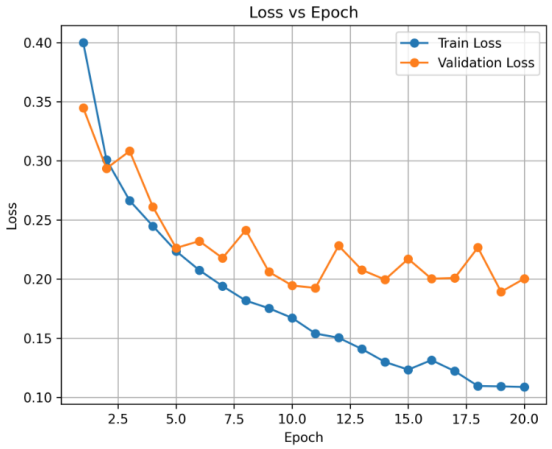
This experiment mainly compared the following four configurations：

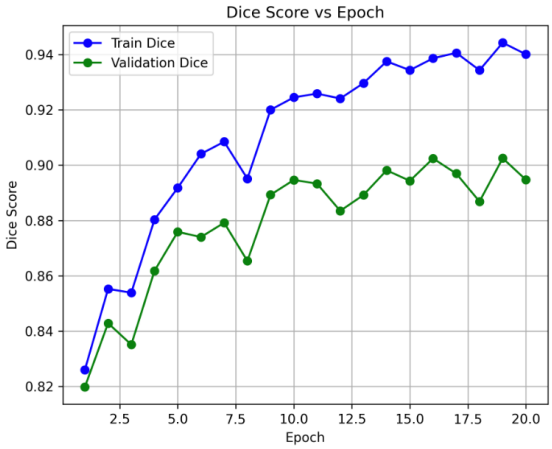
1. UNet (Without Data Augmentation)
   - **Test Dice Score:** 0.9054
   - The graphs show that as the number of epochs increases, both the Train Loss and Validation Loss continuously decrease, while the Dice Score rises accordingly. The performance stabilizes at around the 15th to 20th epoch.



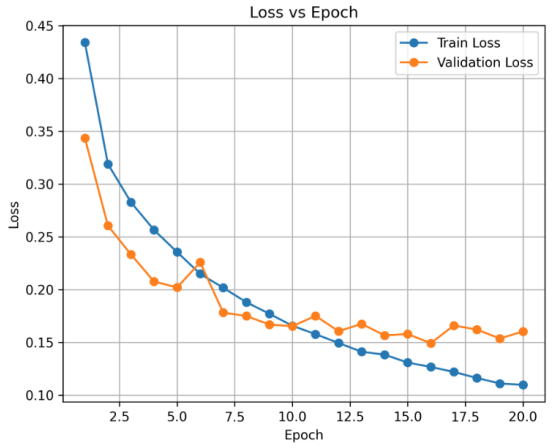
2. ResNet34 + UNet (Without Data Augmentation)
   - **Test Dice Score:** 0.9038
   - Compared to the pure UNet, ResNet34+UNet shows a similar

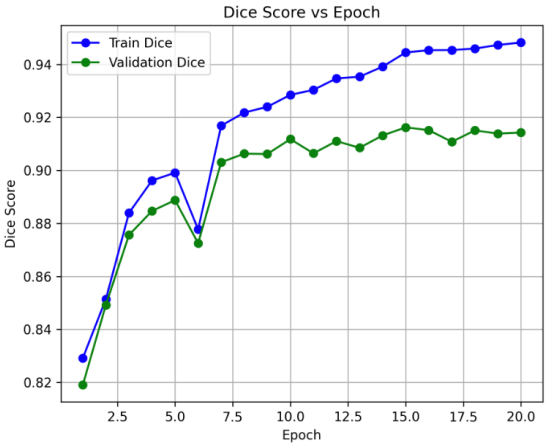
convergence speed in the early stages. However, the final Dice Score is slightly lower, which might be due to insufficient data or augmentation, and thus the full potential of ResNet34 as an encoder may not have been completely exploited.



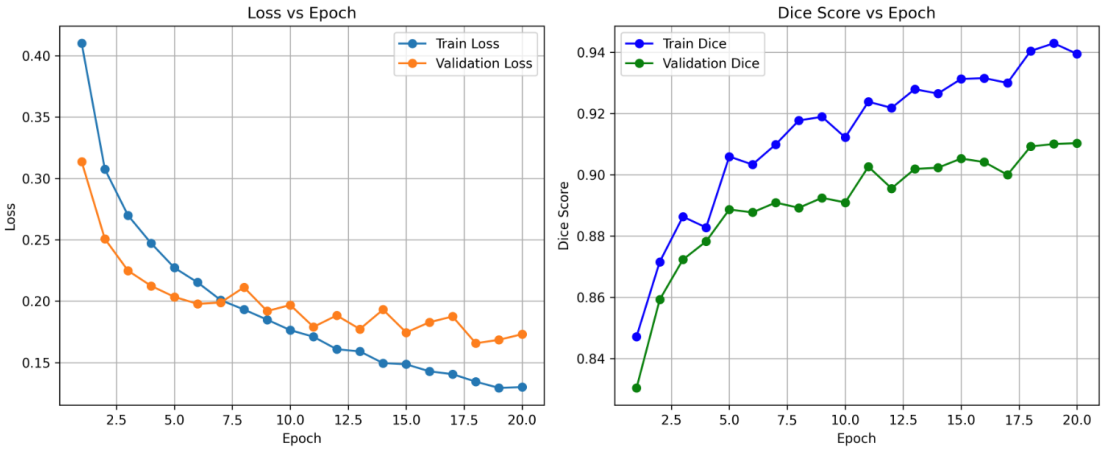


3. **UNet + Data Augmentation**
   - **Test Dice Score:** 0.9221
   - After introducing augmentation strategies (such as random cropping, brightness adjustment, and grayscale conversion), both the Train Loss and Validation Loss decrease faster. The final Test Dice Score is significantly improved to 0.9221, demonstrating that data augmentation is very effective in enhancing generalization.





4. **ResNet34 + UNet + Data Augmentation**
   - **Test Dice Score:** 0.9195
   - With the help of data augmentation, the final Dice Score is also higher than the non-augmented version. However, it is slightly lower than the UNet + Augmentation configuration (0.9221). This might be due to hyperparameter settings or the decoder section still needing further fine-tuning, or differences in how the augmentation is adapted.



2. Different Hyperparameters, Model Configurations, and Training Strategies

1. **Batch Size (Batch Size = 8)**
    - Choosing a batch size of 8 is a balance between achieving stable convergence and maintaining training speed, given the GPU memory limitations. If the batch size is too small, the loss curve will show larger fluctuations; if it is too large, more GPU memory might be required.
2. **Learning Rate (Learning Rate = 1e-4)**
    - For example, when using the Adam optimizer, a learning rate of 1e-4 is commonly used for semantic segmentation tasks, achieving a good balance of convergence speed and stability. A learning rate that is too high might cause loss oscillations in the first few epochs, while a rate that is too low may require more epochs to converge.
3. **Model Configuration: UNet vs. ResNet34+UNet**
    - **UNet:** A fully custom encoder-decoder structure that is suitable for small to medium-sized datasets and allows for easy channel adjustments.
    - **ResNet34+UNet:** Uses the preset ResNet34 as the encoder, which is better at extracting high-level features; however, if the dataset is limited or the augmentation is insufficient, it might not fully exploit the advantages of ResNet34.
4. **Training Strategy: Data Augmentation**
    - Experimental results indicate that augmentation strategies are critical for improving the final Dice Score. Techniques such as random

cropping, grayscale conversion, and brightness adjustment increase the model's adaptability to changes in lighting, scale, and color.
    - The graphs also show that with augmentation, both Validation Loss and Validation Dice converge more smoothly and reach higher scores in fewer epochs.

3. Analysis of Dice Score Trends

1. Relationship Between Loss and Dice Score
    - In the early epochs, as the loss rapidly decreases, the Dice Score rises quickly, indicating that the model is learning key features to distinguish between the foreground (pets) and the background.
    - After approximately 10 to 15 epochs, the curves tend to flatten or show slight fluctuations, suggesting that the model is nearing saturation. Further improvements in accuracy may require additional augmentation or hyperparameter fine-tuning.
2. Intermediate Fluctuations
    - In some graphs, the Dice Score shows temporary declines in certain epochs. This may be caused by learning rate adjustments or the effects of data augmentation, resulting in short-term instability. However, the overall trend continues to rise.
3. Final Stable Region
    - After around 15 to 20 epochs, both the Train Loss and Validation Loss stabilize, and the Dice Score remains above 0.90. To further improve performance, more complex augmentation strategies, fine-tuning of the decoder architecture, or the introduction of higher capacity models should be considered.

4. Impact of Dataset Characteristics on Model Performance

- In the Oxford-IIIT Pet dataset, pets have rich contours and the backgrounds are varied. Without data augmentation, the model might perform poorly when encountering pets in different poses or lighting conditions.
- Augmentation strategies can significantly increase data diversity, making the model more robust to variations in size, angle, and brightness.

- If the dataset is limited (e.g., only using the default train/valid split), larger models like ResNet34+UNet may not fully train or be as sensitive to augmentation, resulting in a slightly lower final test score than expected.

5. conclusion

- **Data Augmentation is Key:** Both UNet and ResNet34+UNet can achieve higher Dice Scores under augmentation strategies.
- **UNet vs. ResNet34+UNet:** Their performance is similar; if the dataset or augmentation is insufficient, ResNet34+UNet may not significantly outperform a custom UNet. With more data or stronger augmentation, ResNet34+UNet could potentially learn richer features.
- **Hyperparameters and Training Strategies:** Batch size, learning rate, and augmentation methods all have a significant impact on convergence speed and final performance, and should be considered comprehensively based on GPU resources and dataset characteristics.

In summary, the experimental results indicate that incorporating data augmentation effectively improves the model's ability to recognize pet contours. This results in an approximate 1.5 to 2 percentage point increase in the Dice Score on the test set. In terms of model selection, with well-tuned augmentation and hyperparameters, ResNet34+UNet still has room for further improvement; however, the current results show that a UNet with fewer parameters can achieve very good segmentation performance.

4. Execution steps

1. **Environment Setup**
    - In the project root directory, run the following command to install the required packages:

      pip install -r requirements.txt

    - Ensure that the requirements.txt file includes all dependencies needed for the project (e.g., PyTorch, tqdm, numpy, matplotlib, Pillow, etc.).
2. **Training Phase**
    - By default, the program uses the following parameters:

- data_path = "../dataset/oxford-iiit-pet"
- epochs = 20
- batch_size = 8
- learning_rate = 1e-4

- Execute train.py (assumed to be in the src directory), for example:

```
cd src
python train.py
```

- If you wish to customize the parameters, you can specify them in the command line, for example:

```
python train.py --data_path "../dataset/oxford-iiit-pet" \
                --epochs 20 \
                --batch_size 8 \
                --learning_rate 1e-4
```

- After training, the best model weights will be saved in the ../saved_models/ directory (e.g., unet_model_best.pth or res34_unet_aug_9103_model_best.pth).

3. Inference Phase

- By default, the program uses the following parameters:
  - model = "../saved_models/res34_unet_aug_9103_model_best.pth"
  - data_path = "../dataset/oxford-iiit-pet"
  - batch_size = 8
- Execute inference.py (also assumed to be in the src directory), for example:

```
python inference.py
```

- If you wish to customize the model weight path or other parameters, you can specify them manually, for example:

```
python inference.py --model "../saved_models/unet_model_best.pth" \
                    --data_path "../dataset/oxford-iiit-pet" \
                    --batch_size 8
```

- Upon completion of inference, the program will perform segmentation on the test set and calculate the average Dice Score. You can also view the prediction results via the visualization function (such as visualize_comparison).

4. **Reproducing Results**
   - In the same environment (with the same package versions installed), you can reproduce the experimental results and test Dice Score by training with the default parameters and saving the model, then running the inference script with the same parameters.
   - If parameters (e.g., epochs, batch size, learning rate, etc.) are modified, the final segmentation results and loss/metric curves might vary slightly, but the overall workflow remains unchanged.

These are the specific commands and parameter details for installing the packages, training the model, and performing inference and evaluation in the project, ensuring the reproducibility of the results.

5. Discussion

1. **Alternative Network Architectures**
   - **DeepLab Series:**
     Models such as DeepLabv3+ combine atrous convolution with Atrous Spatial Pyramid Pooling (ASPP), enabling them to capture multi-scale contextual information simultaneously. This approach is promising for scenarios with high-detail requirements, such as pet images.
   - **SegFormer:**
     SegFormer employs a Transformer-based encoder, breaking away from traditional CNN architectures. It is expected to have a broader receptive field in images and to be more sensitive to long-range feature dependencies, potentially providing more refined segmentation results.
2. **Improvement Proposals**
   - **Loss Function Adjustment:**
     The currently used BCEWithLogitsLoss may be less sensitive in cases where the foreground and background classes are highly imbalanced.

If the dataset contains relatively few foreground regions, consider incorporating Dice Loss, Focal Loss, or IoU Loss to improve the learning effectiveness for the minority class.

- **Stronger Data Augmentation Strategies:**
  Although we have already applied random cropping, brightness adjustment, and similar techniques, further diversity in the data could be achieved by considering rotations, mirror flipping, random occlusion (Cutout), or mixed augmentation techniques (Mixup/CutMix).
- **Learning Rate Scheduling:**
  Employing dynamic scheduling strategies such as Cosine Annealing or Step LR could help the model converge more smoothly in later stages of training, preventing occasional fluctuations in the Dice Score.

3. Future Research Directions
   - **Multi-task Learning:**
     Training the model to perform both classification (e.g., pet breed classification for cats and dogs) and segmentation simultaneously might enable the encoder to learn richer features, potentially improving segmentation performance.
   - **Lightweight Models and Edge Deployment:**
     For applications on mobile devices or edge computing, consider using lightweight backbone networks such as MobileNet or ShuffleNet combined with a smaller decoder, to achieve real-time segmentation with reduced resource consumption.
   - **Semi-supervised or Weakly Supervised Segmentation:**
     If additional unlabeled pet images are available, exploring semi-supervised or weakly supervised methods may further enhance the model's adaptability to different breeds and background scenes.

Overall, although this project already achieves satisfactory segmentation accuracy for pet images, there is still room for further improvement through in-depth research and adjustments in network architectures, loss functions, data augmentation, and training strategies. These improvements could also extend the application to more complex scenarios or domains.