# Lab 3: Binary Semantic Segmentation

Yu-Hsiang, Chen - 313553004
July 25, 2024

## 1 Overview

In this lab, we focused on the challenging task of binary semantic segmentation using deep learning techniques. Specifically, we implemented, trained, and evaluated two models: UNet and ResNet34_UNet. The goal of this task was to classify each pixel in an image as either foreground (pet) or background.
The dataset used for this task was the Oxford-IIIT Pet Dataset, which consists of images of cats and dogs with pixel-level annotations. This dataset provides a valuable resource for training and evaluating segmentation models.

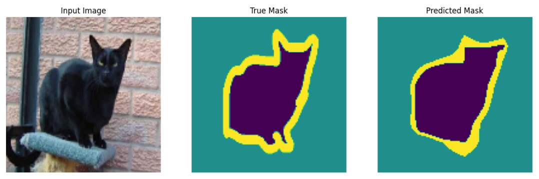


Figure 1: Oxford-IIIT Pet task

### 1.1 Data Preparation

The data preparation phase involved several key steps:

- **Loading and Preprocessing:** We loaded the Oxford-IIIT Pet Dataset and resized each image and its corresponding mask to $256 \times 256$ pixels to ensure uniform input dimensions.
- **Data Splitting:** The dataset was split into training, validation, and test sets to evaluate model performance effectively.
- **Normalization:** Images were normalized to have zero mean and unit variance to improve the convergence of our models during training.

### 1.2 Model Architecture

We designed and implemented two encoder-decoder network architectures:

- **UNet:** A widely used architecture for image segmentation tasks, characterized by its encoder-decoder structure with skip connections. The encoder downsamples the input image to capture context, while the decoder upsamples it to generate the segmentation mask.
- **ResNet34_UNet:** This model combines a ResNet34 backbone as the encoder with a UNet-style decoder. The ResNet34 encoder leverages pre-trained residual blocks to capture deep features, which are then passed to the decoder for upsampling and segmentation mask generation.

### 1.3 Training and Evaluation

Both models were trained using the Adam optimizer, with a learning rate tuned using Optuna for hyperparameter optimization. The loss function used was the Dice Loss, defined as:

$$\text{Dice Loss} = 1 - \frac{2|P \cap G|}{|P| + |G|} \tag{1}$$

where $P$ is the predicted segmentation mask, and $G$ is the ground truth mask. This loss function is particularly suited for segmentation tasks as it directly optimizes for the Dice score.
We monitored the training progress using validation datasets and evaluated the models' performance using the Dice Score, which is calculated as:

$$\text{Dice Score} = \frac{2|P \cap G|}{|P| + |G|} \tag{2}$$

Both models achieved a Dice score higher than 0.93, indicating their effectiveness in segmenting pet regions in the images.

### 1.4 Inference and Visualization

The trained models were applied to unseen images from the test set to assess their generalization capabilities. We visualized the segmentation masks overlaid on the original images to qualitatively evaluate the performance. The inference pipeline involved loading the trained model weights, processing the input images, and generating segmentation masks.

## 2 Implementation Details

### 2.1 Details of Training, Evaluating, and Inferencing Code

#### 2.1.1 Training Code

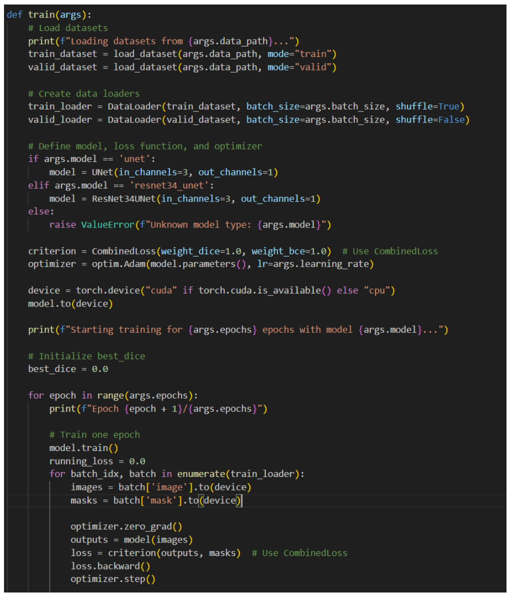
The training code for our models is designed to load the dataset, define the model architecture, specify the loss function and optimizer, and iterate through the training and validation phases. The key steps are as follows:

- **Data Loading:** The dataset is loaded using a custom data loader which splits the data into training and validation sets. The data is then fed into PyTorch DataLoader for batch processing.

- **Model Definition:** Depending on the specified model type (‘UNet‘ or ‘ResNet34_UNet‘), the corresponding model is instantiated.

- **Loss Function and Optimizer:** We use a combined loss function (Dice Loss + Binary Cross Entropy Loss) to optimize the model, and the Adam optimizer is employed for parameter updates.

- **Training Loop:** For each epoch, the model is trained on the training data and evaluated on the validation data. The loss and Dice score are computed for both the training and validation sets to monitor performance. The best-performing model based on the validation Dice score is saved.



```python
def train(args):
    # Load datasets
    print(f"Loading datasets from {args.data_path}...")
    train_dataset = load_dataset(args.data_path, mode="train")
    valid_dataset = load_dataset(args.data_path, mode="valid")

    # Create data loaders
    train_loader = DataLoader(train_dataset, batch_size=args.batch_size, shuffle=True)
    valid_loader = DataLoader(valid_dataset, batch_size=args.batch_size, shuffle=False)

    # Define model, loss function, and optimizer
    if args.model == 'unet':
        model = UNet(in_channels=3, out_channels=1)
    elif args.model == 'resnet34_unet':
        model = ResNet34UNet(in_channels=3, out_channels=1)
    else:
        raise ValueError(f"Unknown model type: {args.model}")

    criterion = CombinedLoss(weight_dice=1.0, weight_bce=1.0)  # Use CombinedLoss
    optimizer = optim.Adam(model.parameters(), lr=args.learning_rate)

    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    model.to(device)

    print(f"Starting training for {args.epochs} epochs with model {args.model}...")

    # Initialize best_dice
    best_dice = 0.0

    for epoch in range(args.epochs):
        print(f"Epoch {epoch + 1}/{args.epochs}")

        # Train one epoch
        model.train()
        running_loss = 0.0
        for batch_idx, batch in enumerate(train_loader):
            images = batch['image'].to(device)
            masks = batch['mask'].to(device)

            optimizer.zero_grad()
            outputs = model(images)
            loss = criterion(outputs, masks)  # Use CombinedLoss
            loss.backward()
            optimizer.step()
```

Figure 2: Training Code

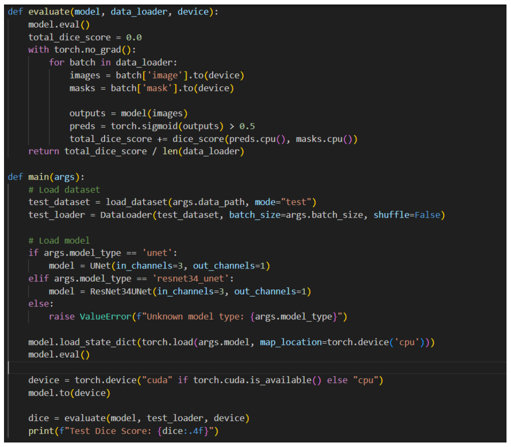
### 2.1.2 Evaluating Code

The evaluation code aims to assess the model performance on the test set by calculating the Dice score. The process includes:

- **Data Loading:** The test dataset is loaded similarly to the training dataset.

- **Model Loading:** The trained model weights are loaded, and the model is set to evaluation mode.

- **Evaluation Loop:** The model predictions are compared with the ground truth masks, and the Dice score is computed for each batch. The average Dice score over the entire test set is reported.



```python
def evaluate(model, data_loader, device):
    model.eval()
    total_dice_score = 0.0
    with torch.no_grad():
        for batch in data_loader:
            images = batch['image'].to(device)
            masks = batch['mask'].to(device)

            outputs = model(images)
            preds = torch.sigmoid(outputs) > 0.5
            total_dice_score += dice_score(preds.cpu(), masks.cpu())
    return total_dice_score / len(data_loader)

def main(args):
    # Load dataset
    test_dataset = load_dataset(args.data_path, mode="test")
    test_loader = DataLoader(test_dataset, batch_size=args.batch_size, shuffle=False)

    # Load model
    if args.model_type == 'unet':
        model = UNet(in_channels=3, out_channels=1)
    elif args.model_type == 'resnet34_unet':
        model = ResNet34UNet(in_channels=3, out_channels=1)
    else:
        raise ValueError(f"Unknown model type: {args.model_type}")

    model.load_state_dict(torch.load(args.model, map_location=torch.device('cpu')))
    model.eval()

    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    model.to(device)

    dice = evaluate(model, test_loader, device)
    print(f"Test Dice Score: {dice:.4f}")
```

Figure 3: Evaluation Code

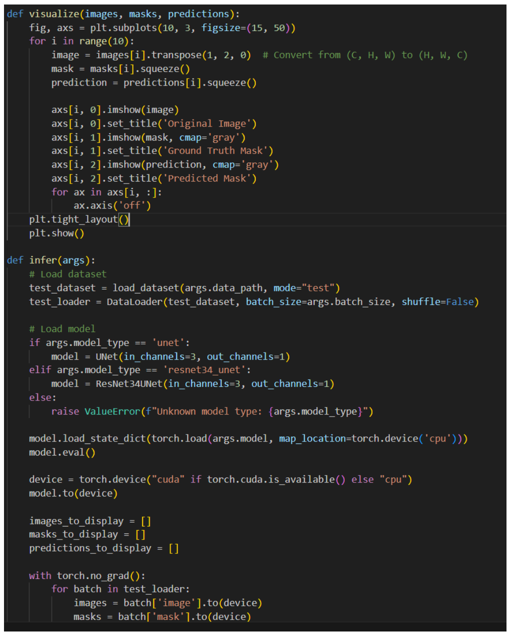
### 2.1.3 Evaluating Code

The inferencing code is designed to visualize the model predictions on unseen test images. The steps include:

- **Data Loading:** The test dataset is loaded.

- **Model Loading:** The trained model is loaded and set to evaluation mode.

- **Inference Loop:** For each batch of test images, the model predictions are generated, and a subset of images along with their predicted masks and ground truth masks are visualized.



```python
def visualize(images, masks, predictions):
    fig, axs = plt.subplots(10, 3, figsize=(15, 50))
    for i in range(10):
        image = images[i].transpose(1, 2, 0)  # Convert from (C, H, W) to (H, W, C)
        mask = masks[i].squeeze()
        prediction = predictions[i].squeeze()

        axs[i, 0].imshow(image)
        axs[i, 0].set_title('Original Image')
        axs[i, 1].imshow(mask, cmap='gray')
        axs[i, 1].set_title('Ground Truth Mask')
        axs[i, 2].imshow(prediction, cmap='gray')
        axs[i, 2].set_title('Predicted Mask')
        for ax in axs[i, :]:
            ax.axis('off')
    plt.tight_layout()
    plt.show()

def infer(args):
    # Load dataset
    test_dataset = load_dataset(args.data_path, mode="test")
    test_loader = DataLoader(test_dataset, batch_size=args.batch_size, shuffle=False)

    # Load model
    if args.model_type == 'unet':
        model = UNet(in_channels=3, out_channels=1)
    elif args.model_type == 'resnet34_unet':
        model = ResNet34UNet(in_channels=3, out_channels=1)
    else:
        raise ValueError(f"Unknown model type: {args.model_type}")

    model.load_state_dict(torch.load(args.model, map_location=torch.device('cpu')))
    model.eval()

    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    model.to(device)

    images_to_display = []
    masks_to_display = []
    predictions_to_display = []

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

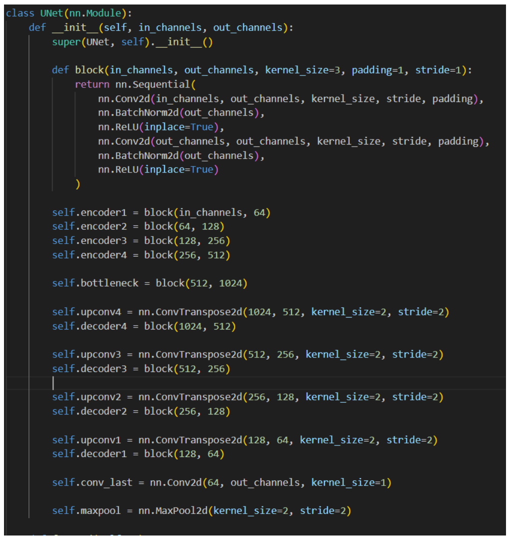
Figure 4: Inference Code

## 2.2 Details of the Models (UNet & ResNet34_UNet)

### 2.2.1 UNet

The UNet architecture is a fully convolutional network designed for image segmentation. It consists of an encoder-decoder structure with skip connections that allow high-resolution features from the encoder to be combined with upsampled features from the decoder. The key components are:

- **Encoder:** Composed of several convolutional blocks followed by max pooling layers, which reduce the spatial dimensions while increasing the feature dimensions.
- **Bottleneck:** A series of convolutional layers that further process the encoded features.
- **Decoder:** Uses transposed convolutions to upsample the features and concatenates them with the corresponding features from the encoder via skip connections. This helps in preserving spatial information.



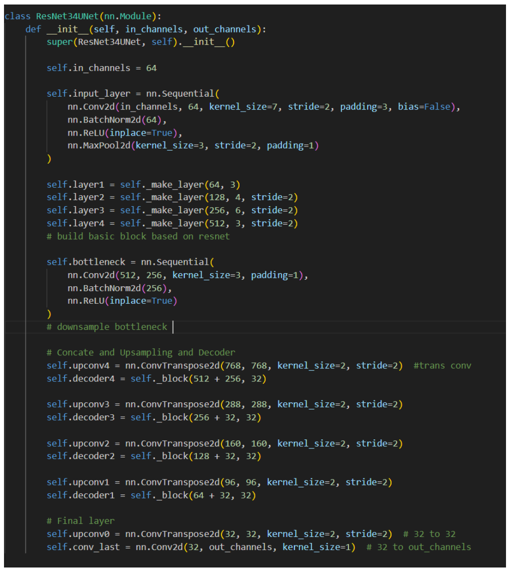
```python
class UNet(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(UNet, self).__init__()

        def block(in_channels, out_channels, kernel_size=3, padding=1, stride=1):
            return nn.Sequential(
                nn.Conv2d(in_channels, out_channels, kernel_size, stride, padding),
                nn.BatchNorm2d(out_channels),
                nn.ReLU(inplace=True),
                nn.Conv2d(out_channels, out_channels, kernel_size, stride, padding),
                nn.BatchNorm2d(out_channels),
                nn.ReLU(inplace=True)
            )

        self.encoder1 = block(in_channels, 64)
        self.encoder2 = block(64, 128)
        self.encoder3 = block(128, 256)
        self.encoder4 = block(256, 512)

        self.bottleneck = block(512, 1024)

        self.upconv4 = nn.ConvTranspose2d(1024, 512, kernel_size=2, stride=2)
        self.decoder4 = block(1024, 512)

        self.upconv3 = nn.ConvTranspose2d(512, 256, kernel_size=2, stride=2)
        self.decoder3 = block(512, 256)

        self.upconv2 = nn.ConvTranspose2d(256, 128, kernel_size=2, stride=2)
        self.decoder2 = block(256, 128)

        self.upconv1 = nn.ConvTranspose2d(128, 64, kernel_size=2, stride=2)
        self.decoder1 = block(128, 64)

        self.conv_last = nn.Conv2d(64, out_channels, kernel_size=1)

        self.maxpool = nn.MaxPool2d(kernel_size=2, stride=2)
```

Figure 5: Unet Structure

### 2.2.2 ResNet34_UNet

The ResNet34_UNet architecture combines a ResNet34 backbone as the encoder with a UNet-style decoder. This leverages the deep feature extraction capabilities of ResNet34 and the effective upsampling strategy of UNet. The key components are:

- **Encoder:** Utilizes the ResNet34 architecture up to the last convolutional block. This provides deep features at various scales.
- **Bottleneck:** A convolutional block that processes the features extracted by the ResNet encoder.
- **Decoder:** Similar to the UNet decoder, it uses transposed convolutions for upsampling and concatenates features from the encoder through skip connections.



```python
class ResNet34UNet(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(ResNet34UNet, self).__init__()

        self.in_channels = 64

        self.input_layer = nn.Sequential(
            nn.Conv2d(in_channels, 64, kernel_size=7, stride=2, padding=3, bias=False),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        )

        self.layer1 = self._make_layer(64, 3)
        self.layer2 = self._make_layer(128, 4, stride=2)
        self.layer3 = self._make_layer(256, 6, stride=2)
        self.layer4 = self._make_layer(512, 3, stride=2)
        # build basic block based on resnet

        self.bottleneck = nn.Sequential(
            nn.Conv2d(512, 256, kernel_size=3, padding=1),
            nn.BatchNorm2d(256),
            nn.ReLU(inplace=True)
        )
        # downsample bottleneck

        # Concate and Upsampling and Decoder
        self.upconv4 = nn.ConvTranspose2d(768, 768, kernel_size=2, stride=2)  #trans conv
        self.decoder4 = self._block(512 + 256, 32)

        self.upconv3 = nn.ConvTranspose2d(288, 288, kernel_size=2, stride=2)
        self.decoder3 = self._block(256 + 32, 32)

        self.upconv2 = nn.ConvTranspose2d(160, 160, kernel_size=2, stride=2)
        self.decoder2 = self._block(128 + 32, 32)

        self.upconv1 = nn.ConvTranspose2d(96, 96, kernel_size=2, stride=2)
        self.decoder1 = self._block(64 + 32, 32)

        # Final layer
        self.upconv0 = nn.ConvTranspose2d(32, 32, kernel_size=2, stride=2)  # 32 to 32
        self.conv_last = nn.Conv2d(32, out_channels, kernel_size=1)  # 32 to out_channels
```

Figure 6: ResNet34Unet Structure

## 2.3 Additional Details

- **Combined Loss Function:** We used a combination of Dice Loss and Binary Cross Entropy Loss to train the models. The Dice Loss ensures that the model focuses on the overlap between the predicted and ground truth masks, while Binary Cross Entropy Loss handles the pixel-wise classification.
- **Performance:** Both models were trained and evaluated on the Oxford-IIIT Pet Dataset, achieving Dice scores higher than 0.93, which indicates excellent segmentation performance.

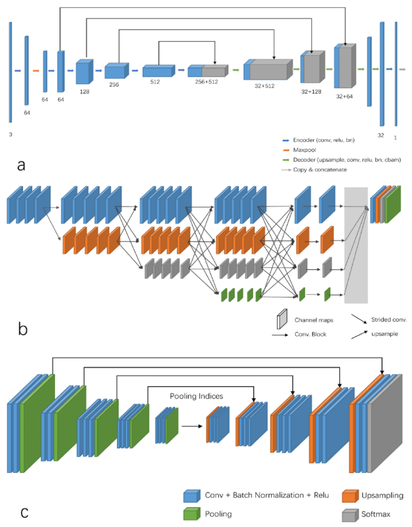


Figure 7: Illustraion of ResNet and Unet and their varients

## 3 Data Preprocessing (20%)

### 3.1 How You Preprocessed Your Data

The data preprocessing pipeline for our models involved several essential steps to ensure that the images and masks were adequately prepared for training, validation, and testing.

- **Loading the Dataset:** We utilized custom dataset classes, `OxfordPetDataset` and `SimpleOxfordPetDataset`, to load the Oxford-IIIT Pet Dataset. These classes efficiently read the images and their corresponding masks from disk, ensuring proper handling of the dataset.

- **Splitting the Dataset:** The dataset was split into training (90%), validation (10%), and test sets. This split ensures that the model is evaluated on unseen data, which is crucial for assessing generalization performance and preventing overfitting.

- **Resizing Images and Masks:** All images and masks were resized to a fixed resolution of $256 \times 256$ pixels. Resizing ensures uniformity across the dataset and compatibility with the network input requirements, which is essential for consistent training and evaluation.

- **Augmentation:** Data augmentation techniques were applied to the training set to increase variability and prevent overfitting. The augmentations included:
  - **Random Resized Crop:** This augmentation randomly crops and resizes the image, allowing the model to learn from different parts of the image, which enhances its ability to generalize to new images.
  - **Horizontal and Vertical Flips:** Flipping the images horizontally and vertically increases the diversity of the training set, enabling the model to be invariant to orientation changes.
  - **Shift, Scale, Rotate:** This transformation randomly shifts, scales, and rotates the images, helping the model become robust to geometric distortions.
  - **Color Jittering:** Adjusting the brightness, contrast, saturation, and hue of the images helps the model handle variations in lighting conditions.
  - **RGB Shift:** Shifting the RGB channels independently introduces color variations, improving the model's robustness to color changes.
  - **Gaussian Noise:** Adding noise to the images forces the model to learn more robust features, as it cannot rely solely on noise-free patterns.

- **Normalization:** The images were normalized using the mean and standard deviation of the ImageNet dataset: mean $(0.485, 0.456, 0.406)$ and standard deviation $(0.229, 0.224, 0.225)$. Normalization ensures that the pixel values are centered and scaled, facilitating faster convergence during training.

- **Mask Preprocessing:** The mask annotations were preprocessed to convert the multi-class labels into binary masks. Specifically, the pet regions (classes 1 and 3) were set to 1 (foreground), and the background (class 2) was set to 0. This conversion simplifies the segmentation task to binary classification, making it more tractable.

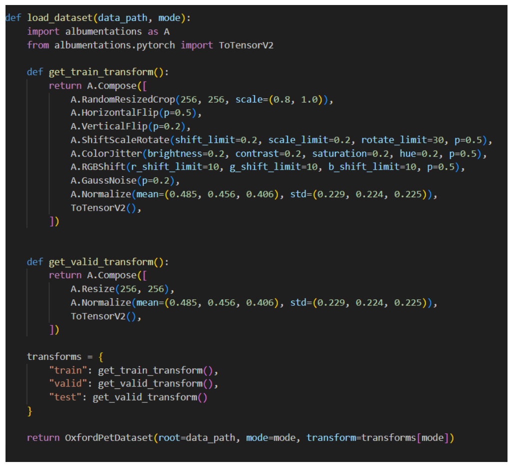
- **Transformations:** The dataset classes utilized the `albumentations` library for applying the augmentations and transformations. These transformations were applied on-the-fly during data loading, ensuring that the model sees different variations of the same image in each epoch, aiding generalization.

### 3.2 What Makes Your Method Unique

The uniqueness of our data preprocessing approach lies in the following aspects:

- **Comprehensive Augmentation:** Our augmentation strategy covers a wide range of transformations, significantly increasing the variability of the training data. This comprehensive approach prevents overfitting and improves the model's ability to generalize to new, unseen data.

- **Dynamic Loading and Transformations:** By using the `albumentations` library, we apply transformations dynamically during data loading. This ensures that the model sees a diverse set of augmented images in each epoch, which helps in learning more robust features.



```python
def load_dataset(data_path, mode):
    import albumentations as A
    from albumentations.pytorch import ToTensorV2

    def get_train_transform():
        return A.Compose([
            A.RandomResizedCrop(256, 256, scale=(0.8, 1.0)),
            A.HorizontalFlip(p=0.5),
            A.VerticalFlip(p=0.2),
            A.ShiftScaleRotate(shift_limit=0.2, scale_limit=0.2, rotate_limit=30, p=0.5),
            A.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2, hue=0.2, p=0.5),
            A.RGBShift(r_shift_limit=10, g_shift_limit=10, b_shift_limit=10, p=0.5),
            A.GaussNoise(p=0.2),
            A.Normalize(mean=(0.485, 0.456, 0.406), std=(0.229, 0.224, 0.225)),
            ToTensorV2(),
        ])


    def get_valid_transform():
        return A.Compose([
            A.Resize(256, 256),
            A.Normalize(mean=(0.485, 0.456, 0.406), std=(0.229, 0.224, 0.225)),
            ToTensorV2(),
        ])

    transforms = {
        "train": get_train_transform(),
        "valid": get_valid_transform(),
        "test": get_valid_transform()
    }

    return OxfordPetDataset(root=data_path, mode=mode, transform=transforms[mode])
```

Figure 8: Data Augmentation

## 4 Analyze on the Experiment Results

### 4.1 What Did You Explore During the Training Process?

During the training process, we encountered several challenges and made various observations that significantly impacted our model's performance:

- **Optimization of Learning Rate:** Due to the large size of the dataset and the high computational cost, we initially focused on optimizing the learning rate. Using Optuna, we conducted 50 trials to determine the best learning rate over 5 epochs. This optimization step was crucial as it allowed us to identify the learning rate that provides the fastest convergence while maintaining stability during training.

- **Extended Training:** With the best learning rate identified, we then extended the training process to 50 epochs. This extended training allowed the models to learn more robust features, leading to better performance. Despite the time-consuming nature of this process, it was necessary to achieve high accuracy and model generalization.

- **Combined Loss Function:** We employed a combined loss function, integrating Dice loss and Binary Cross-Entropy (BCE) loss. The combined loss function proved to be effective for both UNet and ResNet34UNet architectures. It balanced the pixel-wise accuracy (BCE) with the overall segmentation quality (Dice), leading to improved performance.

- **Model Performance:** Both models, UNet and ResNet34UNet, achieved impressive results with over 93% Dice score. However, the ResNet34UNet demonstrated a slightly better performance, particularly in handling complex shapes and edges. This indicates that the ResNet encoder's ability to capture more detailed features contributed to the improved accuracy.

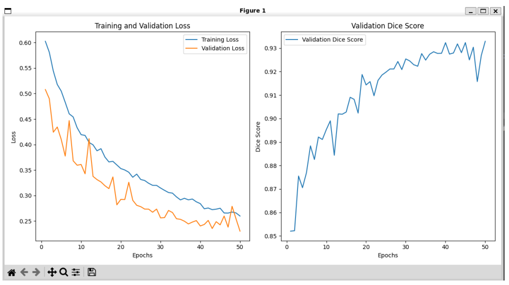


Figure 9: Unet Loss and Score

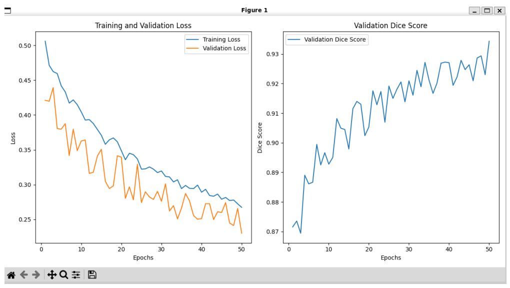


Figure 10: ResnetUnet Lss and Score

### 4.2 Found Any Characteristics of the Data?

Several characteristics of the Oxford-IIIT Pet Dataset were observed during the training process:

- **Class Imbalance:** The dataset exhibited a significant class imbalance, with a larger number of background pixels compared to foreground pixels (pets). This imbalance necessitated the use of a combined loss function to ensure that the model does not become biased towards predicting the background class.

- **Diverse Shapes and Sizes:** The pets in the images varied widely in shape, size, and orientation. This diversity required the model to be highly adaptable and capable of capturing fine details. The data augmentation techniques, such as random resizing, rotation, and flipping, were crucial in enabling the model to generalize well across different shapes and sizes.

- **Edge Detection:** One notable characteristic observed was the model's performance in edge detection. The ResNet34UNet, in particular, showed a strong ability to detect and segment round edges accurately. This is likely due to the deep ResNet encoder's capacity to capture high-level features, leading to more precise segmentation boundaries.

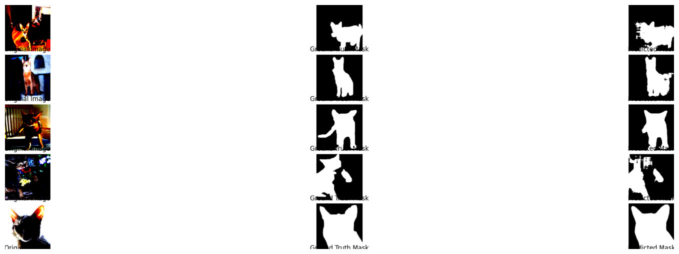


Figure 11: Sample of Unet Prediction (Left is Ground Truth, Right is Prediction)

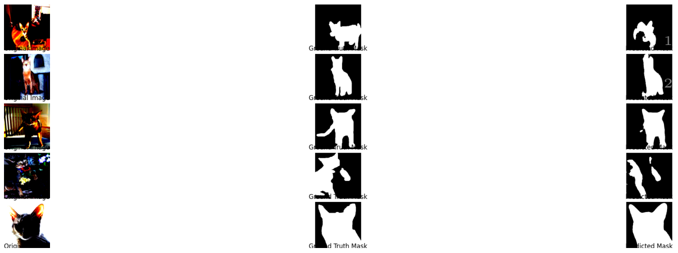


Figure 12: Sample of Res34Unet Prediction (Left is Ground Truth, Right is Prediction)

### 4.3 Anything More You Want to Mention

Additional insights and observations include:

- **Training Time and Computational Resources:** The extensive training time and high computational resources required were significant challenges. Efficient management of these resources and optimization techniques, such as learning rate tuning, were essential in achieving high performance within a reasonable timeframe.
- **Impact of Combined Loss Function:** The combined loss function not only improved the overall Dice score but also contributed to faster convergence. By balancing the pixel-wise accuracy and overall segmentation quality, it ensured that the model learned to segment both the foreground and background effectively.
- **Comparative Analysis of Models:** While both models achieved similar Dice scores, the qualitative analysis showed that the ResNet34UNet produced more visually appealing and accurate segmentation masks. The encoder's ability to capture detailed features made it particularly effective in handling complex segmentation tasks.

## 5 Execution Command (0%)

### 5.1 The Command and Parameters for the Training Process

**Caution: When using these commands, paste them in a text editor that supports long lines to ensure they stay on a single line. Directly pasting into the command shell may cause issues due to line breaks. And also please replace pth file corresponding to the pth name you want to test**

Training the models involves specifying the data path, model type, number of epochs, batch size, and learning rate. The commands used for training both the UNet and ResNet34_UNet models are as follows:

```
# Command to train the UNet model, learning rate is fine-tuned by
    optuna
python src/train.py --data_path dataset/oxford-iiit-pet --model
    unet --epochs 50 --batch_size 16 --learning_rate
    0.0003325313446284999
```

```
# Command to train the ResNet34_UNet model, learning rate is fine-
    tuned by optuna
python src/train.py --data_path dataset/oxford-iiit-pet --model
    resnet34_unet --epochs 50 --batch_size 16 --learning_rate
    0.00010623904863324497
```

### 5.2 The Command and Parameters for the Inference Process

For the inference process, the trained models are applied to the test dataset to generate predictions. The commands for running inference using both models are:

```
# Command to run inference using the UNet model
python src/inference.py --model saved_models/best_unet_model.pth --
    model_type unet --data_path dataset/oxford-iiit-pet --
    batch_size 16
```

```
# Command to run inference using the ResNet34_UNet model
python src/inference.py --model saved_models/
    best_resnet34_unet_model.pth --model_type resnet34_unet --
    data_path dataset/oxford-iiit-pet --batch_size 16
```

### 5.3 The Command and Parameters for the Evaluation Process

Evaluating the models involves calculating the Dice score on the test dataset to assess the segmentation performance. The commands for evaluating both models are:

```
# Command to evaluate the UNet model
python src/evaluate.py --model saved_models/best_unet_model.pth --
    model_type unet --data_path dataset/oxford-iiit-pet --
    batch_size 16
# UNet Test Dice Score: 0.9318
```

```
# Command to evaluate the ResNet34_UNet model
python src/evaluate.py --model saved_models/
    best_resnet34_unet_model.pth --model_type resnet34_unet --
    data_path dataset/oxford-iiit-pet --batch_size 16
# ResNet34_UNet Test Dice Score: 0.9314
```

The commands ensure that the correct model and dataset paths are used, along with appropriate batch sizes, to perform training, inference, and evaluation effectively.

## 6 Discussion

### 6.1 What architecture may bring better results?

In our experiments, both the UNet and ResNet34_UNet architectures performed remarkably well, achieving Dice scores above 93%. However, ResNet34_UNet showed a slight edge in handling complex edges and generalization. This can be attributed to several factors:

- **Residual Connections:** The residual connections in ResNet34 help in better gradient flow during backpropagation, allowing for deeper networks without suffering from the vanishing gradient problem. This enhances the network's ability to capture intricate details and complex patterns in the images.

- **Hierarchical Feature Extraction:** ResNet34's deeper architecture allows it to extract features at multiple levels of abstraction, from low-level edges to high-level semantic features. This multi-level feature extraction is crucial for accurate segmentation, especially for objects with complex boundaries.
- **Edge Detection:** The convolutional layers in ResNet34 are effective in detecting edges and textures, which is essential for segmenting objects with clear boundaries. The improved edge detection capability leads to more precise segmentation maps.
- **Generalization Ability:** The robustness of ResNet34's learned features contributes to better generalization across different samples and conditions. This means the model is less likely to overfit to the training data and can perform well on the validation and test datasets.

### 6.2 What are the potential research topics in this task?

The field of semantic segmentation is rich with research opportunities. Some potential research topics include:

- **Improved Loss Functions:** Developing new loss functions that better handle class imbalance and improve the accuracy of segmentation, especially in challenging conditions.
- **Domain Adaptation:** Exploring domain adaptation techniques to apply models trained on one dataset to different but related datasets, improving generalization and robustness.
- **Real-time Segmentation:** Focusing on optimizing models for real-time performance, which is crucial for applications like autonomous driving and medical imaging.
- **Unsupervised and Semi-supervised Learning:** Investigating methods for reducing the reliance on annotated data by utilizing unsupervised or semi-supervised learning techniques.
- **Explainability and Interpretability:** Enhancing the interpretability of segmentation models to understand their decision-making process, which is particularly important in fields like healthcare.

### 6.3 Anything more you want to mention

Several additional considerations and observations were made during our study:

- **Data Augmentation:** Effective data augmentation techniques played a crucial role in improving the generalization of our models. Augmentations like random cropping, flipping, and rotation helped in creating a more diverse training set.
- **Training Efficiency:** The choice of hyperparameters, especially the learning rate, was critical in ensuring efficient training. Using Optuna for hyperparameter tuning significantly streamlined this process.