

1. Implementation Details

a. Model Architecture

We implemented two models: UNet and ResNet34_UNet for the image segmentation task.

UNet:

The UNet architecture can be divided into the following four components:

i. Encoder:

- The encoder consists of multiple convolutional layers and pooling layers, which aims to progressively extract features from the image while reducing image size.
- Each convolution operation uses a 3x3 kernel and is followed by a ReLU activation function. After each convolution, a pooling operation is applied to reduce the image dimensions.

ii. Bottleneck:

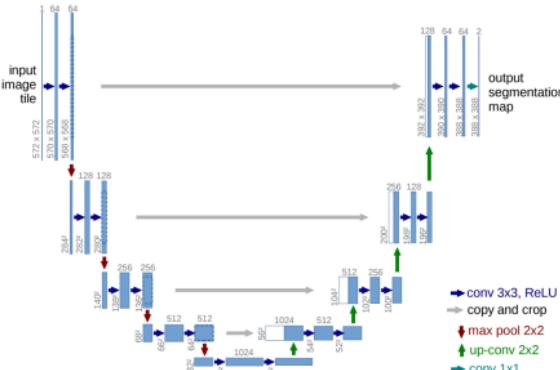
- The bottleneck layer is responsible for the deepest abstraction of features, which increasing the number of channels and processing high-dimensional feature maps from the encoder.
- We set the number of channels to 1024.

iii. Decoder:

- The decoder's main function is to restore the feature map to the original image size.
- The decoder typically includes upsampling layers (ex: transposed convolutions) to increase the image size, and connects feature maps from corresponding layers of encoder (skip connections).

iv. Output:

- The final output layer uses a 1x1 convolution to map the final feature map to the output channels.



```

import torch
import torch.nn as nn
import torch.nn.functional as F

class DoubleConv(nn.Module):
    # 兩次 conv + ReLU，用於 Encoder 與 Decoder 中
    def __init__(self, in_channels, out_channels):
        super().__init__()
        self.double_conv = nn.Sequential(
            nn.Conv2d(in_channels, out_channels, 3, padding=1), # 保持尺寸
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True),
            nn.Conv2d(out_channels, out_channels, 3, padding=1),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True),
        )

    def forward(self, x):
        return self.double_conv(x)

class UNet(nn.Module):
    def __init__(self, in_channels=3, out_channels=1, base_c=64):
        super().__init__()

        # Encoder
        self.enc1 = DoubleConv(in_channels, base_c)
        self.enc2 = DoubleConv(base_c, base_c * 2)
        self.enc3 = DoubleConv(base_c * 2, base_c * 4)
        self.enc4 = DoubleConv(base_c * 4, base_c * 8)

        self.pool = nn.MaxPool2d(2)

        # Bottleneck
        self.bottleneck = DoubleConv(base_c * 8, base_c * 16)

        # Decoder
        self.up4 = nn.ConvTranspose2d(base_c * 16, base_c * 8, 2, stride=2)
        self.dec4 = DoubleConv(base_c * 16, base_c * 8)

        self.up3 = nn.ConvTranspose2d(base_c * 8, base_c * 4, 2, stride=2)
        self.dec3 = DoubleConv(base_c * 8, base_c * 4)

        self.up2 = nn.ConvTranspose2d(base_c * 4, base_c * 2, 2, stride=2)
        self.dec2 = DoubleConv(base_c * 4, base_c * 2)

        self.up1 = nn.ConvTranspose2d(base_c * 2, base_c, 2, stride=2)
        self.dec1 = DoubleConv(base_c * 2, base_c)

        # Output layer
        self.out_conv = nn.Conv2d(base_c, out_channels, 1) # 輸出單通道

```

```

def forward(self, x):
    # Encoder
    x1 = self.enc1(x)
    x2 = self.enc2(self.pool(x1))
    x3 = self.enc3(self.pool(x2))
    x4 = self.enc4(self.pool(x3))

    # Bottleneck
    x5 = self.bottleneck(self.pool(x4))

    # Decoder with skip connections
    d4 = self.up4(x5)
    d4 = self.dec4(torch.cat([d4, x4], dim=1))

    d3 = self.up3(d4)
    d3 = self.dec3(torch.cat([d3, x3], dim=1))

    d2 = self.up2(d3)
    d2 = self.dec2(torch.cat([d2, x2], dim=1))

    d1 = self.up1(d2)
    d1 = self.dec1(torch.cat([d1, x1], dim=1))

    out = self.out_conv(d1)
    return out

```

ResNet34_UNet:

The ResNet34_UNet architecture can be divided into the following four components:

i. Encoder (ResNet34 Encoder):

- The encoder uses ResNet34 as the feature extraction part and progressively reducing the size of the image while increasing the depth (number of channels) of the feature maps through multiple convolution layers.
- It includes four main convolution layers with the channel numbers being 64, 128, 256, and 512 respectively, there's MaxPool layers between each layer to reduce spatial dimensions.

ii. Bottleneck:

- The bottleneck located between the encoder and decoder, it further expands the feature maps from 512 to 1024 channels and then reduces back to 512 channels.
- This layer is for feature reorganization at the deepest level.

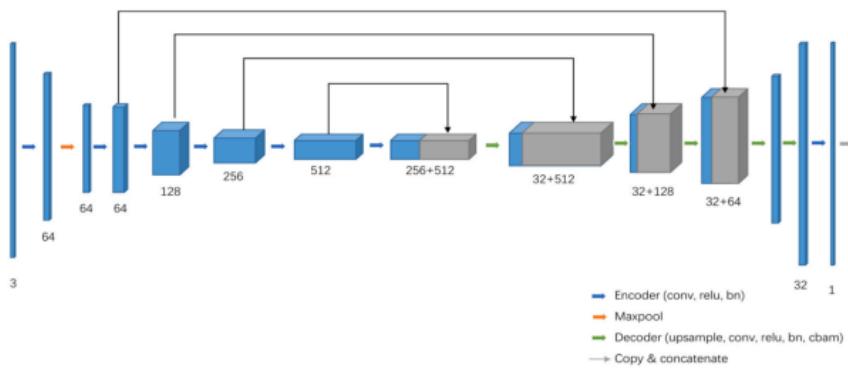
iii. Decoder (U-Net Decoder):

- The decoder uses upsampling layers (ConvTranspose2d) to progressively restore the spatial dimensions of the image.
- Each decoder layer connects the output of the previous layer with corresponding features from the encoder via skip connections.
- Each upsampling step uses 2x2 convolutions to process the feature maps.

iv. Output Layer:

- A 1x1 convolution layer compresses the final feature map into a single-channel (grayscale) image to output the segmentation result.

ResNet34 (Encoder) + UNet (Decoder)



```

import torch
import torch.nn as nn
import torch.nn.functional as F

class BasicBlock(nn.Module):
    # ResNet
    def __init__(self, in_channels, out_channels, stride=1, downsample=None):
        super().__init__()
        self.conv1 = nn.Conv2d(in_channels, out_channels, 3, stride, 1, bias=False)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.relu = nn.ReLU(inplace=True)

        self.conv2 = nn.Conv2d(out_channels, out_channels, 3, 1, 1, bias=False)
        self.bn2 = nn.BatchNorm2d(out_channels)

        self.downsample = downsample # 如果需要跳接維度，使用 1x1 conv

    def forward(self, x):
        identity = x

        out = self.relu(self.bn1(self.conv1(x)))
        out = self.bn2(self.conv2(out))

        if self.downsample:
            identity = self.downsample(x)

        out += identity
        out = self.relu(out)
        return out

def make_layer(in_channels, out_channels, num_blocks, stride=1):
    layers = []
    # 第一個 block 可能需要 downsample
    downsample = None
    if stride != 1 or in_channels != out_channels:
        downsample = nn.Sequential(
            nn.Conv2d(in_channels, out_channels, 1, stride, bias=False),
            nn.BatchNorm2d(out_channels)
        )

    layers.append(BasicBlock(in_channels, out_channels, stride, downsample))
    # 其他 block stride = 1
    for _ in range(1, num_blocks):
        layers.append(BasicBlock(out_channels, out_channels))

    return nn.Sequential(*layers)

class ResNet34Encoder(nn.Module):
    def __init__(self):
        super().__init__()
        self.initial = nn.Sequential(
            nn.Conv2d(3, 64, 7, stride=2, padding=3, bias=False),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(3, stride=2, padding=1)
        )
        self.layer1 = make_layer(64, 64, num_blocks=3)
        self.layer2 = make_layer(64, 128, num_blocks=4, stride=2)
        self.layer3 = make_layer(128, 256, num_blocks=6, stride=2)
        self.layer4 = make_layer(256, 512, num_blocks=3, stride=2)

    def forward(self, x):
        x0 = self.initial(x)
        x1 = self.layer1(x0)
        x2 = self.layer2(x1)
        x3 = self.layer3(x2)
        x4 = self.layer4(x3)
        return x1, x2, x3, x4

class Up(nn.Module):
    """上採樣 + conv"""
    def __init__(self, in_channels, skip_channels, out_channels):
        super().__init__()
        self.up = nn.ConvTranspose2d(in_channels, out_channels, 2, stride=2)
        self.conv = nn.Sequential(
            nn.Conv2d(out_channels + skip_channels, out_channels, 3, padding=1),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True),
            nn.Conv2d(out_channels, out_channels, 3, padding=1),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True),
        )

    def forward(self, x, skip):
        x = self.up(x)

        # 使用 F.interpolate 與 skip 大小需匹配
        x = F.interpolate(x, size=skip.size()[2:], mode='bilinear', align_corners=True)

        x = torch.cat([x, skip], dim=1)
        return self.conv(x)

```

```

class ResNet34_UNet(nn.Module):
    def __init__(self, in_channels=3, out_channels=1):
        super().__init__()
        self.encoder = ResNet34Encoder()
        self.bottleneck = nn.Sequential(
            nn.Conv2d(512, 1024, 3, padding=1),
            nn.BatchNorm2d(1024),
            nn.ReLU(inplace=True),
            nn.Conv2d(1024, 512, 3, padding=1),
            nn.BatchNorm2d(512),
            nn.ReLU(inplace=True),
        )

        # Decoder
        self.up1 = Up(512, 256, 256)
        self.up2 = Up(256, 128, 128)
        self.up3 = Up(128, 64, 64)
        self.up4 = Up(64, 64, 64)

        # 輸出層
        self.out_conv = nn.Conv2d(64, out_channels, kernel_size=1)

    def forward(self, x):
        x1, x2, x3, x4 = self.encoder(x)  # skip connections
        x = self.bottleneck(x4)

        # 確保解碼過程的尺寸一致
        x = self.up1(x, x3)
        x = self.up2(x, x2)
        x = self.up3(x, x1)
        x = self.up4(x, x1)  # 重複使用 x1，因為最初只有一層

        # 確保最後尺寸匹配
        x = F.interpolate(x, size=(256, 256), mode='bilinear', align_corners=True)

        out = self.out_conv(x)
        return out

```

b. Training.py

Training Setup

- Begins by selecting the model architecture, either UNet or ResNet34_UNet.
- Loading the training and validation datasets using the load_dataset function, and the data is loaded into DataLoader objects.
- If CUDA is available, the model will be moved to the GPU(faster), otherwise, it will remain on the CPU.
- The loss function is BCEWithLogitsLoss, the optimizer is Adam.

Training Loop

- The model is set to training mode.
- Predicted output compares with the ground truth masks using the loss function.
- The optimizer updates the model parameters.
- Validation step, calculates the validation loss and dice coefficient in each batch, then prints average values.

Model Saving

- The best model is identified based on the highest Dice coefficient value, and its weights are stored in a file named `best_{args.model}.pth` in the `saved_models` folder.

Training History Plotting

- Plots the average training loss, average validation loss, and average validation Dice score of every epoch.

Testing

- Loading the testing dataset and evaluates the best model, then calculates the Dice score.

Arguments

- The path to the dataset (`--data_path`). Default data path is the same as the format.
- The model to use (`--model`, either 'unet' or 'resnet'). **Needed!**
- The number of epochs for training (`--epochs`). Default is 10.
- The batch size (`--batch_size`). Default is 4.
- The learning rate (`--learning-rate`). Default is 1e-4.

```

import argparse
import os
import torch
import torch.nn as nn
from torch.utils.data import DataLoader
from tqdm import tqdm
import matplotlib.pyplot as plt
from oxford_pet import load_dataset
from models.unet import UNet
from models.resnet34_unet import ResNet34_UNet

def dice_score(preds, targets, threshold=0.5, eps=1e-7):
    preds = torch.sigmoid(preds)
    preds = (preds > threshold).float()
    intersection = (preds * targets).sum(dim=(1,2,3))
    union = preds.sum(dim=(1,2,3)) + targets.sum(dim=(1,2,3))
    dice = (2 * intersection + eps) / (union + eps)
    return dice.mean().item()

def train(args):
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    print(f" 使用装置: {device}")
    # Load datasets
    train_dataset = load_dataset(args.data_path, mode="train")
    val_dataset = load_dataset(args.data_path, mode="valid")

    train_loader = DataLoader(train_dataset, batch_size=args.batch_size, shuffle=True)
    val_loader = DataLoader(val_dataset, batch_size=1, shuffle=False)

    # Choose model
    if args.model == "unet":
        model = UNet(in_channels=3, out_channels=1)
    elif args.model == "resnet":
        model = ResNet34_UNet(in_channels=3, out_channels=1)
    else:
        raise ValueError("Invalid model: choose 'unet' or 'resnet'")

    model.to(device)

    # Loss & Optimizer
    criterion = nn.BCEWithLogitsLoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=args.learning_rate)

    best_dice = 0.0
    save_model_dir = os.path.join(os.path.dirname(os.path.dirname(__file__)), "saved_models")
    os.makedirs(save_model_dir, exist_ok=True)
    # os.makedirs("saved_models", exist_ok=True)
    train_loss_history = []
    val_loss_history = []
    val_dice_history = []

```

```

for epoch in range(args.epochs):
    model.train()
    running_loss = 0.0
    for batch in tqdm(train_loader, desc=f"Epoch {epoch+1}/{args.epochs}"):
        images = batch["image"].to(device, dtype=torch.float)
        masks = batch["mask"].to(device, dtype=torch.float)

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

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        running_loss += loss.item()

    avg_train_loss = running_loss / len(train_loader)

    # Validation
    model.eval()
    val_loss = 0.0
    val_dice = 0.0
    with torch.no_grad():
        for batch in val_loader:
            images = batch["image"].to(device, dtype=torch.float)
            masks = batch["mask"].to(device, dtype=torch.float)

            preds = model(images)
            loss = criterion(preds, masks)
            val_loss += loss.item()
            val_dice += dice_score(preds, masks)
    avg_val_loss = val_loss / len(val_loader)
    avg_val_dice = val_dice / len(val_loader)

    print(f"\nEpoch {epoch+1}: Train Loss={avg_train_loss:.4f} | Val Loss={avg_val_loss:.4f} | Val Dice={avg_val_dice:.4f}")

    # Save best model
    if avg_val_dice > best_dice:
        best_dice = avg_val_dice
        model_path = os.path.join(save_model_dir, f"best_{args.model}.pth")
        torch.save(model.state_dict(), model_path)
        print(f"Saved new best model (dice={best_dice:.4f})"

    train_loss_history.append(avg_train_loss)
    val_loss_history.append(avg_val_loss)
    val_dice_history.append(avg_val_dice)

plot_training_history(train_loss_history, val_loss_history, val_dice_history)

print("\n Evaluating on test set with best model...")
# model.load_state_dict(torch.load(f"saved_models/best_{args.model}.pth", map_location=device))
# model.load_state_dict(torch.load(f"saved_models/best_{args.model}.pth", map_location=device, weights_only=True))
model.load_state_dict(torch.load(os.path.join(save_model_dir, f"best_{args.model}.pth"), map_location=device, weights_only=True))

test_dataset = load_dataset(args.data_path, mode="test")
test_loader = DataLoader(test_dataset, batch_size=1, shuffle=False)

model.eval()
test_dice = 0.0
with torch.no_grad():
    for batch in test_loader:
        images = batch["image"].to(device, dtype=torch.float)
        masks = batch["mask"].to(device, dtype=torch.float)

        preds = model(images)
        test_dice += dice_score(preds, masks)
    test_dice /= len(test_loader)
print(f"\n Test Dice Score = {test_dice:.4f}")

def get_args():
    parser = argparse.ArgumentParser(description='Train a segmentation model')
    default_data_path = os.path.abspath(os.path.join(os.path.dirname(__file__), "..", "dataset"))
    parser.add_argument('--data_path', type=str, default=default_data_path, help='path of the input data')
    parser.add_argument('--model', type=str, choices=['unet', 'resnet'], required=True, help='Model type: unet or resnet')
    parser.add_argument('--epochs', '-e', type=int, default=10, help='Number of training epochs')
    parser.add_argument('--batch_size', '-b', type=int, default=4, help='Batch size')
    parser.add_argument('--learning-rate', '-lr', type=float, default=1e-4, help='Learning rate')
    return parser.parse_args()

def plot_training_history(train_loss_history, val_loss_history, val_dice_history):
    epochs = range(1, len(train_loss_history) + 1)

    # Plot Train Loss
    plt.figure(figsize=(15, 5))

    plt.subplot(1, 3, 1)
    plt.plot(epochs, train_loss_history, label='Train Loss')
    plt.title('Train Loss')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.legend()

    # Plot Validation Loss
    plt.subplot(1, 3, 2)
    plt.plot(epochs, val_loss_history, label='Validation Loss', color='orange')
    plt.title('Validation Loss')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.legend()

    # Plot Validation Dice Score
    plt.subplot(1, 3, 3)
    plt.plot(epochs, val_dice_history, label='Validation Dice', color='green')
    plt.title('Validation Dice Score')
    plt.xlabel('Epoch')
    plt.ylabel('Dice Score')
    plt.legend()

    plt.tight_layout()
    plt.show()

if __name__ == "__main__":
    args = get_args()
    train(args)

```

c. Evaluate.py

- Loads the model (UNet or ResNet34_UNet).
- The dataset is loaded using the load_dataset function, the mode is either test or valid.
- Evaluated in evaluation mode (net.eval()).
- Dice_score function calculates the Dice coefficient. (loads from utils.py)
- Arguments same as train.py
- After evaluation, it prints the average Dice score for test or validation dataset.

```
import os
import torch
from torch.utils.data import DataLoader
from oxford_pet import load_dataset
from models.unet import UNet
from models.resnet34_unet import ResNet34_UNet
from utils import dice_score # 從 utils 引入 dice 計算

def evaluate(net, data, device):
    net.eval()
    total_dice = 0.0
    count = 0

    with torch.no_grad():
        for batch in data:
            image = batch["image"].to(device, dtype=torch.float)
            mask = batch["mask"].to(device, dtype=torch.float)

            pred = net(image)
            dice = dice_score(pred, mask)
            total_dice += dice
            count += 1

    avg_dice = total_dice / count
    return avg_dice

if __name__ == "__main__":
    import argparse

    parser = argparse.ArgumentParser()
    parser.add_argument('--data_path', type=str, required=True)
    parser.add_argument('--model_path', type=str, required=True)
    parser.add_argument('--model_type', type=str, choices=["unet", "resnet"], required=True)
    parser.add_argument('--mode', type=str, choices=["test", "valid"], default="test")
    parser.add_argument('--batch_size', type=int, default=1)

    args = parser.parse_args()

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

    # Load model
    if args.model_type == "unet":
        net = UNet(in_channels=3, out_channels=1)
    elif args.model_type == "resnet":
        net = ResNet34_UNet(in_channels=3, out_channels=1)
    net.load_state_dict(torch.load(args.model_path, map_location=device))
    net.to(device)

    # Load dataset
    dataset = load_dataset(args.data_path, mode=args.mode)
    dataloader = DataLoader(dataset, batch_size=args.batch_size, shuffle=False)

    # Evaluate
    avg_dice = evaluate(net, dataloader, device)
    print(f"[{args.model_type.upper()}] Average Dice Score on {args.mode} set: {avg_dice:.4f}")
```

d. Inference.py

It used to load a trained model and perform inference, it generates segmentation masks for input images and allows the user to save the predicted masks to a specified directory.

```

import argparse
import torch
from torch.utils.data import DataLoader
from oxford_pet import load_dataset
from models.unet import UNet
from models.resnet34_unet import ResNet34_UNet
from utils import tensor_to_image, show_prediction
import os
import matplotlib.pyplot as plt

def get_args():
    parser = argparse.ArgumentParser(description='Predict masks from input images')
    parser.add_argument('--model', required=True, help='path to the stored model weight (.pth)')
    parser.add_argument('--model_type', choices=['unet', 'resnet'], required=True, help='Model type')
    parser.add_argument('--data_path', type=str, required=True, help='path to the input data')
    parser.add_argument('--batch_size', '-b', type=int, default=1, help='batch size')
    parser.add_argument('--mode', type=str, choices=['test', 'valid'], default='test', help='dataset split')
    parser.add_argument('--num_samples', type=int, default=5, help='Number of images to predict')
    parser.add_argument('--save_dir', type=str, default=None, help='Folder to save masks (optional)')
    parser.add_argument('--no_show', action='store_true', help='If set, do not display images')

    return parser.parse_args()

def inference(net, dataloader, device, save_dir=None, show=True):
    net.eval()
    os.makedirs(save_dir, exist_ok=True) if save_dir else None

    with torch.no_grad():
        for idx, batch in enumerate(dataloader):
            image = batch["image"].to(device, dtype=torch.float)
            orig_image = image[0].detach().cpu()
            pred = net(image)
            pred = torch.sigmoid(pred)
            pred = (pred > 0.5).float()

            img_np = tensor_to_image(orig_image)
            mask_np = tensor_to_image(pred[0])

            if show:
                show_prediction(orig_image, pred[0], title=f"Sample {idx}")

            if save_dir:
                plt.imsave(os.path.join(save_dir, f"sample_{idx}.png"), mask_np, cmap="Reds")

if __name__ == '__main__':
    args = get_args()
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

    # Load model
    if args.model_type == "unet":
        model = UNet(in_channels=3, out_channels=1)
    elif args.model_type == "resnet":
        model = ResNet34_UNet(in_channels=3, out_channels=1)
    else:
        raise ValueError("Unknown model type")

    model.load_state_dict(torch.load(args.model, map_location=device))
    model.to(device)

    # Load dataset
    dataset = load_dataset(args.data_path, mode=args.mode)
    subset = torch.utils.data.Subset(dataset, range(args.num_samples))
    dataloader = DataLoader(subset, batch_size=1, shuffle=False)

    inference(model, dataloader, device, save_dir=args.save_dir, show=not args.no_show)

```

e. Utils.py

- dice_score:

Computes the Dice similarity coefficient between the predicted binary mask and the ground truth mask. The Dice coefficient measures the similarity, ranging from 0 to 1.

- tensor_to_image:

Converts a tensor format image to a NumPy array and adjusts the dimensions.

$$(C, H, W) \rightarrow (H, W, C)$$

- show_prediction:

Displays the original image with the predicted mask. The mask is laid on the image in red transparent color.

```

import torch
import numpy as np
import matplotlib.pyplot as plt

def dice_score(pred_mask, gt_mask, threshold=0.5, eps=1e-7):
    pred_mask = torch.sigmoid(pred_mask)
    pred_mask = (pred_mask > threshold).float()

    intersection = (pred_mask * gt_mask).sum(dim=(1, 2, 3))
    union = pred_mask.sum(dim=(1, 2, 3)) + gt_mask.sum(dim=(1, 2, 3))

    dice = (2 * intersection + eps) / (union + eps)
    return dice.mean().item()

def tensor_to_image(tensor):

    array = tensor.detach().cpu().numpy()
    if array.shape[0] == 1:
        array = array.squeeze(0) # (1, H, W) → (H, W)
    else:
        array = np.moveaxis(array, 0, -1) # (C, H, W) → (H, W, C)
    return array

def show_prediction(image_tensor, pred_mask_tensor, title="Prediction"):

    image = tensor_to_image(image_tensor)
    mask = tensor_to_image(pred_mask_tensor)

    plt.figure(figsize=(8, 4))
    plt.subplot(1, 2, 1)
    plt.imshow(image)
    plt.axis("off")
    plt.title("Image")

    plt.subplot(1, 2, 2)
    plt.imshow(image)
    plt.imshow(mask, alpha=0.4, cmap="Reds")
    plt.axis("off")
    plt.title(title)
    plt.tight_layout()
    plt.show()

```

2. Data Preprocessing

```

def get_transform(mode, to_tensor=True):
    import albumentations as A
    from albumentations.pytorch import ToTensorV2

    transforms = []

    if mode == "train":
        transforms += [
            A.HorizontalFlip(p=0.5),
            A.RandomBrightnessContrast(p=0.5),
            A.ColorJitter(p=0.5),
            A.Rotate(limit=15, p=0.5),
        ]

        transforms += [
            A.Resize(256, 256),
            A.Normalize(mean=(0.5, 0.5, 0.5), std=(0.5, 0.5, 0.5)),
        ]

        if to_tensor:
            transforms += [ToTensorV2()]

    return A.Compose(transforms)

```

In the Data Preprocessing section, I use the Albumentations package. The main advantage of this package is that it allows applying same transformations to image and mask at the same time.

In train mode:

a. A.HorizontalFlip($p=0.5$):

Randomly horizontally flips the image with a 50% probability. This transformation helps improve the model's generalization.

b. A.RandomBrightnessContrast(p=0.5):

Randomly changes the brightness and contrast with a 50% probability. This helps the model handle variations in brightness and contrast.

c. A.ColorJitter(p=0.5):

Randomly adjusts the color properties with a 50% probability. This improves model's ability to adapt to different color environments.

d. A.Rotate(limit=15, p=0.5):

Randomly rotates the image with an angle between -15 and 15 degrees with a 50% probability. Making the model learn to handle angle variations.

In all modes:

e. A.Resize(256, 256):

Ensures all input images be same size, which is necessary for the neural network.

f. A.Normalize(mean=(0.5, 0.5, 0.5), std=(0.5, 0.5, 0.5)):

Normalizes the image by adjusting pixel values with mean = 0.5 and std = 0.5. This helps center them around zero. This helps accelerate training and stabilizes the data distribution.

For each channel (R, G, B), the pixel values are first scaled by subtracting the mean and

dividing by the standard deviation: $\text{normalized_pixel} = \frac{\text{pixel}-\text{mean}}{\text{std}}$.

g. ToTensorV2():

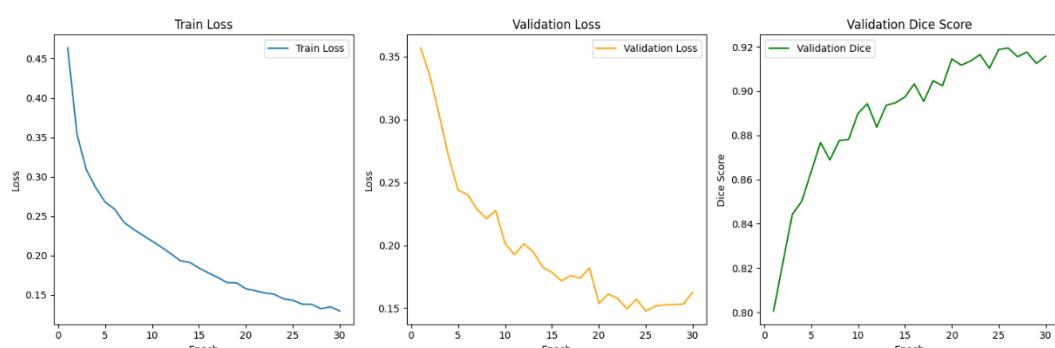
Converts the image into a PyTorch tensor format.

What makes my preprocessing method unique?

Standard preprocessing methods usually focus on basic normalization and resizing, but I use more transformations such as random horizontal flipping, random brightness and contrast adjustments, color jitter, and random rotations. These augmentations let the model adapt to changes in orientation, lighting, and color variations. I set the transformations in train mode with a 50% probability, so the model can have more diverse set of training examples which helps it generalize better when deployed in real-world situations.

3. Analyze the experiment results

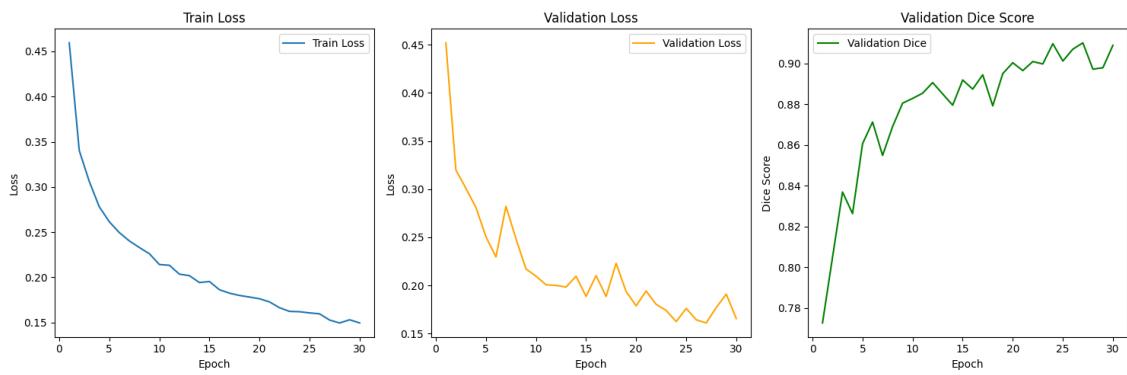
a. UNet



Evaluating on test set with best model...
Test Dice Score = 0.9202

```
PS C:\Users\annie95162\Desktop\Lab2_Binary_Semantic_Segmentation_2025\src> python train.py --model unet --epochs 30 --batch_size 8 --learning_rate 1e-4
    使用装置: cuda
Epoch 1/30: 100%
Epoch 1: Train Loss=0.4634 | Val Loss=0.3569 | Val Dice=0.8006
    Saved new best model (dice=0.8006)
Epoch 2/30: 100%
Epoch 2: Train Loss=0.3533 | Val Loss=0.3344 | Val Dice=0.8226
    Saved new best model (dice=0.8226)
Epoch 3/30: 100%
Epoch 3: Train Loss=0.3087 | Val Loss=0.3029 | Val Dice=0.8442
    Saved new best model (dice=0.8442)
Epoch 4/30: 100%
Epoch 4: Train Loss=0.2859 | Val Loss=0.2700 | Val Dice=0.8502
    Saved new best model (dice=0.8502)
Epoch 5/30: 100%
Epoch 5: Train Loss=0.2677 | Val Loss=0.2438 | Val Dice=0.8634
    Saved new best model (dice=0.8634)
Epoch 6/30: 100%
Epoch 6: Train Loss=0.2598 | Val Loss=0.2405 | Val Dice=0.8768
    Saved new best model (dice=0.8768)
Epoch 7/30: 100%
Epoch 7: Train Loss=0.2422 | Val Loss=0.2287 | Val Dice=0.8689
Epoch 8/30: 100%
Epoch 8: Train Loss=0.2335 | Val Loss=0.2213 | Val Dice=0.8776
    Saved new best model (dice=0.8776)
Epoch 9/30: 100%
Epoch 9: Train Loss=0.2259 | Val Loss=0.2276 | Val Dice=0.8781
    Saved new best model (dice=0.8781)
Epoch 10/30: 100%
Epoch 10: Train Loss=0.2184 | Val Loss=0.2015 | Val Dice=0.8899
    Saved new best model (dice=0.8899)
Epoch 11/30: 100%
Epoch 11: Train Loss=0.2107 | Val Loss=0.1926 | Val Dice=0.8943
    Saved new best model (dice=0.8943)
Epoch 12/30: 100%
Epoch 12: Train Loss=0.2024 | Val Loss=0.2012 | Val Dice=0.8837
Epoch 13/30: 100%
Epoch 13: Train Loss=0.1935 | Val Loss=0.1949 | Val Dice=0.8935
Epoch 14/30: 100%
Epoch 14: Train Loss=0.1915 | Val Loss=0.1827 | Val Dice=0.8947
    Saved new best model (dice=0.8947)
Epoch 15/30: 100%
Epoch 15: Train Loss=0.1842 | Val Loss=0.1784 | Val Dice=0.8973
    Saved new best model (dice=0.8973)
Epoch 16/30: 100%
Epoch 16: Train Loss=0.1781 | Val Loss=0.1717 | Val Dice=0.9032
    Saved new best model (dice=0.9032)
Epoch 17/30: 100%
Epoch 17: Train Loss=0.1722 | Val Loss=0.1760 | Val Dice=0.8954
Epoch 18/30: 100%
Epoch 18: Train Loss=0.1657 | Val Loss=0.1739 | Val Dice=0.9047
    Saved new best model (dice=0.9047)
Epoch 19/30: 100%
Epoch 19: Train Loss=0.1654 | Val Loss=0.1820 | Val Dice=0.9024
Epoch 20/30: 100%
Epoch 20: Train Loss=0.1578 | Val Loss=0.1539 | Val Dice=0.9145
    Saved new best model (dice=0.9145)
Epoch 21/30: 100%
Epoch 21: Train Loss=0.1554 | Val Loss=0.1612 | Val Dice=0.9117
Epoch 22/30: 100%
Epoch 22: Train Loss=0.1527 | Val Loss=0.1577 | Val Dice=0.9136
Epoch 23/30: 100%
Epoch 23: Train Loss=0.1510 | Val Loss=0.1495 | Val Dice=0.9165
    Saved new best model (dice=0.9165)
Epoch 24/30: 100%
Epoch 24: Train Loss=0.1453 | Val Loss=0.1573 | Val Dice=0.9103
Epoch 25/30: 100%
Epoch 25: Train Loss=0.1433 | Val Loss=0.1478 | Val Dice=0.9188
    Saved new best model (dice=0.9188)
Epoch 26/30: 100%
Epoch 26: Train Loss=0.1384 | Val Loss=0.1516 | Val Dice=0.9195
    Saved new best model (dice=0.9195)
Epoch 27/30: 100%
Epoch 27: Train Loss=0.1383 | Val Loss=0.1526 | Val Dice=0.9155
Epoch 28/30: 100%
Epoch 28: Train Loss=0.1326 | Val Loss=0.1530 | Val Dice=0.9176
Epoch 29/30: 100%
Epoch 29: Train Loss=0.1350 | Val Loss=0.1533 | Val Dice=0.9125
Epoch 30/30: 100%
Epoch 30: Train Loss=0.1297 | Val Loss=0.1627 | Val Dice=0.9158
    Evaluating on test set with best model...
    Test Dice Score = 0.9202
```

b. ResNet34_UNet



Evaluating on test set with best model...
Test Dice Score = 0.9133

```
使用装置: cuda
Epoch 1/30: 100%
Epoch 1: Train Loss=0.4595 | Val Loss=0.4520 | Val Dice=0.7726 | 414/414 [00:42<00:00, 9.70it/s]
Saved new best model (dice=0.7726)
Epoch 2/30: 100%
Epoch 2: Train Loss=0.3404 | Val Loss=0.3200 | Val Dice=0.8048 | 414/414 [00:55<00:00, 7.49it/s]
Saved new best model (dice=0.8048)
Epoch 3/30: 100%
Epoch 3: Train Loss=0.3062 | Val Loss=0.3009 | Val Dice=0.8369 | 414/414 [00:44<00:00, 9.26it/s]
Saved new best model (dice=0.8369)
Epoch 4/30: 100%
Epoch 4: Train Loss=0.2781 | Val Loss=0.2811 | Val Dice=0.8263 | 414/414 [00:43<00:00, 9.57it/s]
Epoch 5/30: 100%
Epoch 5: Train Loss=0.2615 | Val Loss=0.2508 | Val Dice=0.8606 | 414/414 [00:51<00:00, 8.09it/s]
Saved new best model (dice=0.8606)
Epoch 6/30: 100%
Epoch 6: Train Loss=0.2496 | Val Loss=0.2296 | Val Dice=0.8713 | 414/414 [00:50<00:00, 8.21it/s]
Saved new best model (dice=0.8713)
Epoch 7/30: 100%
Epoch 7: Train Loss=0.2403 | Val Loss=0.2823 | Val Dice=0.8549 | 414/414 [00:53<00:00, 7.75it/s]
Epoch 8/30: 100%
Epoch 8: Train Loss=0.2330 | Val Loss=0.2486 | Val Dice=0.8691 | 414/414 [00:54<00:00, 7.60it/s]
Epoch 9/30: 100%
Epoch 9: Train Loss=0.2262 | Val Loss=0.2172 | Val Dice=0.8806 | 414/414 [00:54<00:00, 7.56it/s]
Saved new best model (dice=0.8806)
Epoch 10/30: 100%
Epoch 10: Train Loss=0.2142 | Val Loss=0.2097 | Val Dice=0.8822 | 414/414 [00:54<00:00, 7.56it/s]
Saved new best model (dice=0.8822)
Epoch 11/30: 100%
Epoch 11: Train Loss=0.2134 | Val Loss=0.2008 | Val Dice=0.8855 | 414/414 [00:48<00:00, 8.53it/s]
Saved new best model (dice=0.8855)
Epoch 12/30: 100%
Epoch 12: Train Loss=0.2036 | Val Loss=0.2001 | Val Dice=0.8907 | 414/414 [00:55<00:00, 7.51it/s]
Saved new best model (dice=0.8907)
Epoch 13/30: 100%
Epoch 13: Train Loss=0.2019 | Val Loss=0.1984 | Val Dice=0.8851 | 414/414 [00:54<00:00, 7.58it/s]
Epoch 14/30: 100%
Epoch 14: Train Loss=0.1943 | Val Loss=0.2096 | Val Dice=0.8796 | 414/414 [00:43<00:00, 9.47it/s]
Epoch 15/30: 100%
Epoch 15: Train Loss=0.1954 | Val Loss=0.1886 | Val Dice=0.8920 | 414/414 [00:43<00:00, 9.48it/s]
Saved new best model (dice=0.8920)
Epoch 16/30: 100%
Epoch 16: Train Loss=0.1863 | Val Loss=0.2103 | Val Dice=0.8875 | 414/414 [00:47<00:00, 8.63it/s]
Epoch 17/30: 100%
Epoch 17: Train Loss=0.1825 | Val Loss=0.1886 | Val Dice=0.8945 | 414/414 [00:43<00:00, 9.41it/s]
Saved new best model (dice=0.8945)
Epoch 18/30: 100%
Epoch 18: Train Loss=0.1800 | Val Loss=0.2231 | Val Dice=0.8793 | 414/414 [00:49<00:00, 8.32it/s]
Epoch 19/30: 100%
Epoch 19: Train Loss=0.1782 | Val Loss=0.1937 | Val Dice=0.8951 | 414/414 [00:41<00:00, 10.08it/s]
Saved new best model (dice=0.8951)
Epoch 20/30: 100%
Epoch 20: Train Loss=0.1763 | Val Loss=0.1789 | Val Dice=0.9004 | 414/414 [00:40<00:00, 10.19it/s]
Saved new best model (dice=0.9004)
Epoch 21/30: 100%
Epoch 21: Train Loss=0.1728 | Val Loss=0.1943 | Val Dice=0.8966 | 414/414 [00:40<00:00, 10.13it/s]
Epoch 22/30: 100%
Epoch 22: Train Loss=0.1664 | Val Loss=0.1804 | Val Dice=0.9010 | 414/414 [00:40<00:00, 10.22it/s]
Saved new best model (dice=0.9010)
Epoch 23/30: 100%
Epoch 23: Train Loss=0.1623 | Val Loss=0.1741 | Val Dice=0.8998 | 414/414 [00:40<00:00, 10.21it/s]
Epoch 24/30: 100%
Epoch 24: Train Loss=0.1620 | Val Loss=0.1625 | Val Dice=0.9098 | 414/414 [00:40<00:00, 10.23it/s]
Saved new best model (dice=0.9098)
Epoch 25/30: 100%
Epoch 25: Train Loss=0.1607 | Val Loss=0.1763 | Val Dice=0.9013 | 414/414 [00:40<00:00, 10.28it/s]
Epoch 26/30: 100%
Epoch 26: Train Loss=0.1596 | Val Loss=0.1644 | Val Dice=0.9071 | 414/414 [00:40<00:00, 10.26it/s]
Epoch 27/30: 100%
Epoch 27: Train Loss=0.1528 | Val Loss=0.1611 | Val Dice=0.9102 | 414/414 [00:40<00:00, 10.26it/s]
Saved new best model (dice=0.9102)
Epoch 28/30: 100%
Epoch 28: Train Loss=0.1495 | Val Loss=0.1771 | Val Dice=0.8973 | 414/414 [00:40<00:00, 10.23it/s]
Epoch 29/30: 100%
Epoch 29: Train Loss=0.1530 | Val Loss=0.1910 | Val Dice=0.8988 | 414/414 [00:40<00:00, 10.22it/s]
Epoch 30/30: 100%
Epoch 30: Train Loss=0.1496 | Val Loss=0.1656 | Val Dice=0.9090 | 414/414 [00:40<00:00, 10.22it/s]
Evaluating on test set with best model...
Test Dice Score = 0.9133
```

Comparisons:

I trained each for 30 epochs, and both the training losses and validation losses decreased steadily, the dice score gradually increased. From the experimental results, I noticed that UNet compared to ResNet34_UNet, showed signs of overfitting particularly in the fluctuations of the validation loss. However, both UNet and ResNet34_UNet achieved a dice score of **over 0.9** on the test set.

The dice score on test set showed that UNet performed slightly better than ResNet34_UNet. I think it is because UNet can effectively fuse features from different layers through skip connections and it is better at capturing local details, which lead to perform better on relatively small datasets and with fewer epochs. I think if the epochs of ResNet34_UNet increased, the dice score would steadily improve.

Observations:

- I used another learning rate of 1e-3, but the result is worse than 1e-4.
- Both UNet and ResNet34_UNet have a large number of parameters, so this time the training takes much longer.

Characteristics of the data:

UNet	ResNet34_UNet	
 <small>Predicted Mask Dice Score: 0.9676</small>  <small>Ground Truth Mask</small> 	 <small>Predicted Mask Dice Score: 0.9794</small>  <small>Ground Truth Mask</small> 	
 <small>Predicted Mask Dice Score: 0.6125</small>  <small>Ground Truth Mask</small> 	 <small>Predicted Mask Dice Score: 0.8960</small>  <small>Ground Truth Mask</small> 	
 <small>Predicted Mask Dice Score: 0.6854</small>  <small>Ground Truth Mask</small> 	 <small>Predicted Mask Dice Score: 0.7307</small>  <small>Ground Truth Mask</small> 	

A simple monochromatic background is easier for distinction between the pet and background. However, when the pet's color is similar to the background color, and the background is a complex indoor scene, it becomes more difficult to distinguish.

Both UNet and ResNet34_UNet have Pros and Cons: UNet performs better in irregular situations, shown as cat's head in image three, but it may also produce irregular shapes, shown as image two. ResNet34_UNet performs better with smoother objects, shown as image one.

Challenges of the dataset:

In some images, pets are partially blocked by furniture or other objects which increases the difficulty of accurately segmenting the pet's full outline. Additionally, there is a variation in the fur characteristics of different breeds, such as long hair, short hair, or curly hair. Segmenting the fur edges accurately is a challenge especially for long-haired breeds.

4. Execution steps

First execute: `pip install -r requirements.txt`

requirements.txt: (five must requirements:)

torch

torchvision

albumentations

matplotlib

tqdm

Then enter src dir: `cd src`

UNet:

```
python train.py --model unet --epochs 30 --batch_size 8 --learning-rate 1e-4
```

ResNet34_UNet:

```
python train.py --model resnet --epochs 30 --batch_size 8 --learning-rate 1e-4
```

- If the device is CUDA-enabled, it will run faster.
- If I want to show the specified image to see the result:

UNet: `python test_models.py --model unet --image_index 9`

ResNet34_UNet: `python test_models.py --model resnet --image_index 9`

(you can change the index you want)

5. Discussion

Alternative architectures that may potentially yield better results:

- 3D UNet

3D UNet is an extension of the UNet architecture, unlike 2D UNet, 3D UNet uses 3D convolutions to capture spatial relationships in volumetric data, which not only retains more spatial information but also enables more accurate segmentation, particularly in recognizing small regions in volumetric data.

- FPN (Feature Pyramid Network)

FPN is suitable for detecting and segmenting objects at different scales. In traditional CNN, the feature maps become smaller as the network depth increases. FPN can generate high-quality multi-scale features at different levels through a pyramid structure to improve performance in detecting small objects. This method enhances performance when dealing with objects of varying sizes.

- Multi-task Learning

Multi-task learning can let the model learn multiple related tasks simultaneously, improving generalization and reducing overfitting. In image segmentation tasks, in addition to segmenting target regions, other related tasks (such as boundary detection or object classification) can be learned simultaneously, helping the model capture more useful feature information. It can enhance the collaboration between different tasks, leading to more precise and stable predictions.

Potential research directions:

- Unsupervised and Self-supervised Learning

Most image segmentation methods rely on labeled data, but labeling is costly especially for objects like pets with subtle differences. Researching unsupervised or self-supervised learning methods can learn without a large amount of labeled data, could be valuable for future applications.

- Ensemble Learning

Combining multiple models such as using an ensemble of binary classifiers could enhance overall performance: can achieve better generalization, reducing overfitting, and increasing robustness, especially in binary classification tasks where the decision boundary is hard to define.

- Real-time Segmentation

Investigating methods can speed up segmentation models, like using lighter models or improving inference pipelines, can help in areas like autonomous driving or live video processing.