Shri Vile Parle Kelavani Mandal's
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC Accredited with "A" Grade (CGPA : 3.18)

Department of Computer Science and Engineering (Data Science)

Subject: Image Processing and Computer Vision - II Laboratory (DJ19DSL702)

AY: 2024-25

Experiment 7

(Object Segmentation Using U-Net)

Name: Riya Chavan Sap: 60009220136 Batch: D2-1

Aim: To segment objects in an image using the U-Net architecture.

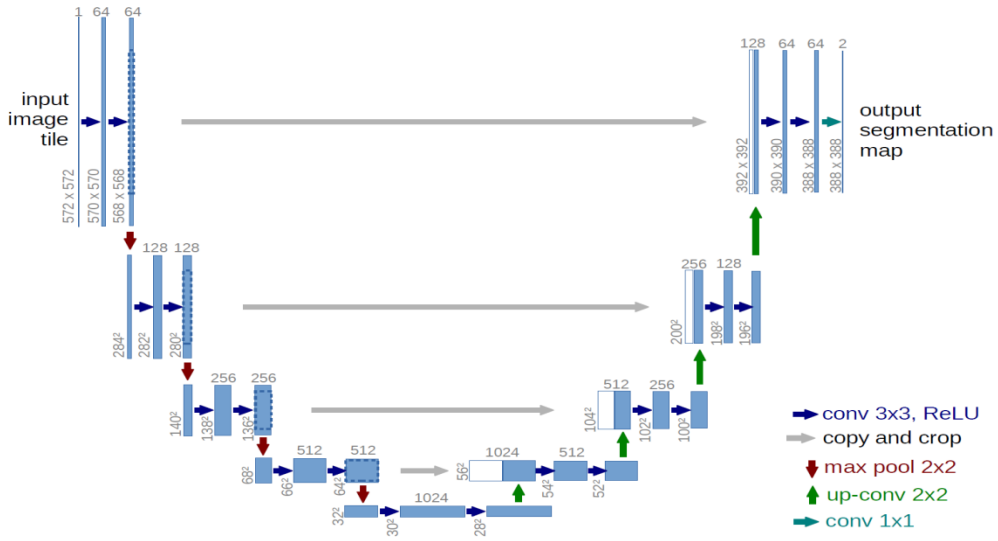
Theory:

1. Introduction

U-Net is known for its encoder-decoder structure and skip connections that help in segmenting biomedical and general images with limited data. The u-net is convolutional network architecture for fast and precise segmentation of images. Up to now it has outperformed the prior best method (a sliding-window convolutional network) on the ISBI challenge for segmentation of neuronal structures in electron microscopic stacks. It has won the Grand Challenge for Computer-Automated Detection of Caries in Bitewing Radiography at ISBI 2015, and it has won the Cell Tracking Challenge at ISBI 2015 on the two most challenging transmitted light microscopy categories (Phase contrast and DIC microscopy) by a large margin

2. Architecture

The U-Net includes a contracting path (encoder), expansive path (decoder), and skip connections that help retain spatial details.



Encoder Network

The encoder network acts as the feature extractor and learns an abstract representation of the input image through a sequence of the encoder blocks. Each encoder block consists of two 3x3 convolutions, where each convolution is followed by a ReLU (Rectified Linear Unit) activation function. The ReLU activation function introduces non-linearity into the network, which helps in the better generalization of the training data. The output of the ReLU acts as a skip connection for the corresponding decoder block.Next, follows a 2x2 max-pooling, where the spatial dimensions (height and width) of the feature maps are reduced by half. This reduces the computational cost by decreasing the number of trainable parameters.

Skip Connections

These skip connections provide additional information that helps the decoder to generate better semantic features. They also act as a shortcut connection that helps the indirect flow of gradients to the earlier layers without any degradation. In simple terms, we can say that skip connection helps in better flow of gradient while backpropagation, which in turn helps the network to learn better representation.

Bridge

The bridge connects the encoder and the decoder network and completes the flow of information. It consists of two 3x3 convolutions, where each convolution is followed by a ReLU activation function.

Decoder Network

The decoder network is used to take the abstract representation and generate a semantic segmentation mask. The decoder block starts with a 2x2 transpose convolution. Next, it is concatenated with the corresponding skip connection feature map from the encoder block. These skip connections provide features from earlier layers that are sometimes lost due to the depth of the network. After that, two 3x3 convolutions are used, where each convolution is followed by a ReLU activation function.

The output of the last decoder passes through a 1x1 convolution with sigmoid activation. The sigmoid activation function gives the segmentation mask representing the pixel-wise classification.

Dataset Preparation

Use the same dataset as PSPNet with masks one-hot encoded for multi-class segmentation.

Implementation Steps

1. Preprocess the dataset: normalize images and prepare segmentation masks.
2. Build the U-Net model with encoder-decoder blocks and skip connections.
3. Choose Binary Cross-Entropy or Categorical Cross-Entropy as the loss function.
4. Use Adam optimizer with learning rate scheduling.

5. Train and validate the model using accuracy and loss metrics.

6. Test the model and compare predictions with ground truth masks.

Result and Evaluation

Use IoU, and visualization plots to assess model performance.

Conclusion

U-Net effectively segments images by combining localization and context through skip connections.

```
# Import necessary libraries
import os
import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision.models as models
from torchvision import transforms
from torch.utils.data import Dataset, DataLoader
from PIL import Image
import numpy as np
import matplotlib.pyplot as plt

# Define U-Net architecture components
class DoubleConv(nn.Module):
    """(Conv => BN => ReLU) * 2"""
    def __init__(self, in_channels, out_channels, mid_channels=None):
        super().__init__()
        if not mid_channels:
            mid_channels = out_channels
        self.double_conv = nn.Sequential(
            nn.Conv2d(in_channels, mid_channels, kernel_size=3, padding=1,
bias=False),
            nn.BatchNorm2d(mid_channels),
            nn.ReLU(inplace=True),
            nn.Conv2d(mid_channels, out_channels, kernel_size=3,
padding=1, bias=False),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True)
        )
```

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

class Down(nn.Module):
    """Downscaling with maxpool then double conv"""
    def __init__(self, in_channels, out_channels):
        super().__init__()
        self.maxpool_conv = nn.Sequential(
            nn.MaxPool2d(2),
            DoubleConv(in_channels, out_channels)
        )

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

class Up(nn.Module):
    """Upscaling then double conv"""
    def __init__(self, in_channels, out_channels, bilinear=True):
        super().__init__()

        # if bilinear, use the normal convolutions to reduce the number of
channels
        if bilinear:
            self.up = nn.Upsample(scale_factor=2, mode='bilinear',
align_corners=True)
            self.conv = DoubleConv(in_channels, out_channels, in_channels
// 2)
        else:
            self.up = nn.ConvTranspose2d(in_channels, in_channels // 2,
kernel_size=2, stride=2)
            self.conv = DoubleConv(in_channels, out_channels)

    def forward(self, x1, x2):
        x1 = self.up(x1)
        # input is CHW
        diffY = x2.size()[2] - x1.size()[2]
        diffX = x2.size()[3] - x1.size()[3]

        x1 = F.pad(x1, [diffX // 2, diffX - diffX // 2,
                        diffY // 2, diffY - diffY // 2])
        x = torch.cat([x2, x1], dim=1)
```

```
        return self.conv(x)

class OutConv(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(OutConv, self).__init__()
        self.conv = nn.Conv2d(in_channels, out_channels, kernel_size=1)

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

# Define U-Net model
class UNet(nn.Module):
    def __init__(self, n_channels=3, n_classes=150, bilinear=True):
        super(UNet, self).__init__()
        self.n_channels = n_channels
        self.n_classes = n_classes
        self.bilinear = bilinear

        # Encoder path
        self.inc = DoubleConv(n_channels, 64)
        self.down1 = Down(64, 128)
        self.down2 = Down(128, 256)
        self.down3 = Down(256, 512)
        factor = 2 if bilinear else 1
        self.down4 = Down(512, 1024 // factor)

        # Decoder path with skip connections
        self.up1 = Up(1024, 512 // factor, bilinear)
        self.up2 = Up(512, 256 // factor, bilinear)
        self.up3 = Up(256, 128 // factor, bilinear)
        self.up4 = Up(128, 64, bilinear)
        self.outc = OutConv(64, n_classes)

    def forward(self, x):
        # Encoder path
        x1 = self.inc(x)
        x2 = self.down1(x1)
        x3 = self.down2(x2)
        x4 = self.down3(x3)
        x5 = self.down4(x4)
```

```
        # Decoder path with skip connections
        x = self.up1(x5, x4)
        x = self.up2(x, x3)
        x = self.up3(x, x2)
        x = self.up4(x, x1)
        logits = self.outc(x)

        return logits

# Define Dataset Loader with one-hot encoding for masks
class ADE20KDataset(Dataset):
    def __init__(self, root_dir, image_set='training', transform=None,
target_transform=None):
        self.image_dir = os.path.join(root_dir, 'images', image_set)
        self.mask_dir = os.path.join(root_dir, 'annotations', image_set)
        self.images = sorted(os.listdir(self.image_dir))
        self.masks = sorted(os.listdir(self.mask_dir))
        self.transform = transform
        self.target_transform = target_transform
        self.n_classes = 150

    def __len__(self):
        return len(self.images)

    def __getitem__(self, idx):
        image = Image.open(os.path.join(self.image_dir,
self.images[idx])).convert('RGB')
        mask = Image.open(os.path.join(self.mask_dir, self.masks[idx]))

        if self.transform:
            image = self.transform(image)

        if self.target_transform:
            mask = self.target_transform(mask)
        else:
            # Default resize to match model output
            mask = mask.resize((256, 256), resample=Image.NEAREST)
            mask = np.array(mask)

            # Set ignore label (150) to 255 for CrossEntropyLoss to ignore
            mask[mask >= 150] = 255
```

```
            mask = torch.tensor(mask, dtype=torch.long)

        return image, mask

# Define metrics for evaluation
def compute_iou(pred, target, n_classes=150, ignore_index=255):
    """Compute IoU for semantic segmentation"""
    pred = torch.argmax(pred, dim=1)

    # Create mask for valid pixels
    mask = (target != ignore_index)

    ious = []
    for cls in range(n_classes):
        pred_inds = (pred == cls) & mask
        target_inds = (target == cls) & mask

        intersection = (pred_inds & target_inds).sum().item()
        union = (pred_inds | target_inds).sum().item()

        if union == 0:
            ious.append(float('nan'))  # Class not present in ground truth
or prediction
        else:
            ious.append(intersection / union)

    return np.nanmean(ious)  # Average over classes

# Train the Model
def train_model():
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    print(f"Using device: {device}")

    # Data preprocessing
    transform = transforms.Compose([
        transforms.Resize((256, 256)),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229,
0.224, 0.225])
    ])
```

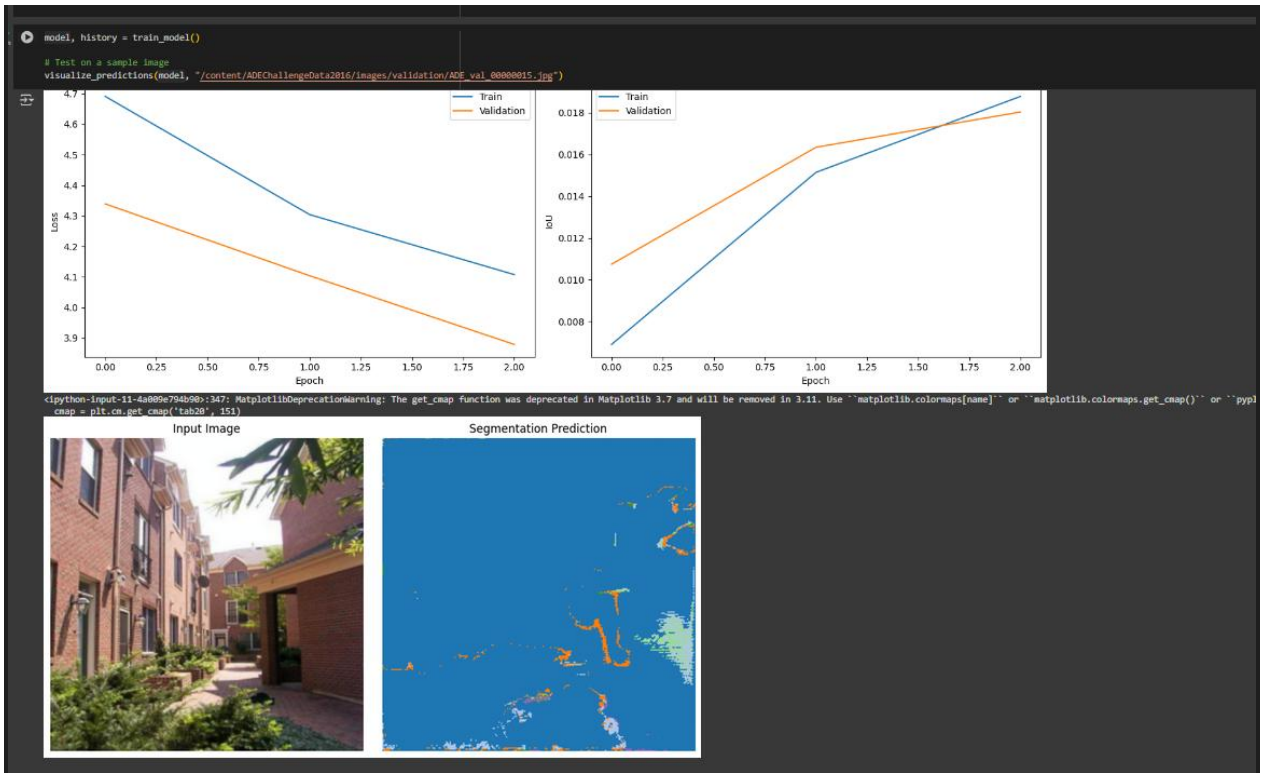
```
    # Create datasets
    train_dataset = ADE20KDataset(root_dir='ADEChallengeData2016',
image_set='training', transform=transform)
    val_dataset = ADE20KDataset(root_dir='ADEChallengeData2016',
image_set='validation', transform=transform)

    # Create dataloaders
    train_loader = DataLoader(train_dataset, batch_size=8, shuffle=True)
    val_loader = DataLoader(val_dataset, batch_size=8, shuffle=False)

    # Create model
    model = UNet(n_channels=3, n_classes=150, bilinear=True).to(device)

    # Define loss function and optimizer
    criterion = nn.CrossEntropyLoss(ignore_index=255)
    optimizer = torch.optim.Adam(model.parameters(), lr=1e-4)

    # Learning rate scheduler
    scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(
        optimizer, mode='min', factor=0.5, patience=3, verbose=True
    )

    # Training history
    history = {
        'train_loss': [],
        'train_iou': [],
        'val_loss': [],
        'val_iou': []
    }

    # Training loop
    num_epochs = 3
    for epoch in range(num_epochs):
        print(f"\nEpoch {epoch+1}/{num_epochs}")

        # Train
        model.train()
        train_loss = 0
        train_iou = 0
        batch_count = 0
```

```
        for images, masks in train_loader:
            images, masks = images.to(device), masks.to(device)

            # Forward pass
            outputs = model(images)
            loss = criterion(outputs, masks)

            # Backward pass
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            # Calculate IoU
            iou = compute_iou(outputs, masks)

            train_loss += loss.item()
            train_iou += iou
            batch_count += 1

            if batch_count % 10 == 0:
                print(f"Batch {batch_count}, Loss: {loss.item():.4f}, IoU:
{iou:.4f}")

        avg_train_loss = train_loss / batch_count
        avg_train_iou = train_iou / batch_count

        # Validation
        model.eval()
        val_loss = 0
        val_iou = 0
        batch_count = 0

        with torch.no_grad():
            for images, masks in val_loader:
                images, masks = images.to(device), masks.to(device)

                outputs = model(images)
                loss = criterion(outputs, masks)
                iou = compute_iou(outputs, masks)

                val_loss += loss.item()
```

```
                val_iou += iou
                batch_count += 1

        avg_val_loss = val_loss / batch_count
        avg_val_iou = val_iou / batch_count

        # Update learning rate
        scheduler.step(avg_val_loss)

        # Save history
        history['train_loss'].append(avg_train_loss)
        history['train_iou'].append(avg_train_iou)
        history['val_loss'].append(avg_val_loss)
        history['val_iou'].append(avg_val_iou)

        print(f"Epoch {epoch+1}, Train Loss: {avg_train_loss:.4f}, Train
IoU: {avg_train_iou:.4f}")
        print(f"Epoch {epoch+1}, Val Loss: {avg_val_loss:.4f}, Val IoU:
{avg_val_iou:.4f}")

    # Save model
    torch.save(model.state_dict(), "unet_ade20k.pth")
    print("☑ Model trained and saved as unet_ade20k.pth")

    # Plot training history
    plot_training_history(history)

    return model, history

# Visualization functions
def plot_training_history(history):
    """Plot training and validation loss and IoU"""
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 5))

    # Plot loss
    ax1.plot(history['train_loss'], label='Train')
    ax1.plot(history['val_loss'], label='Validation')
    ax1.set_title('Loss')
    ax1.set_xlabel('Epoch')
    ax1.set_ylabel('Loss')
    ax1.legend()
```

```
    # Plot IoU
    ax2.plot(history['train_iou'], label='Train')
    ax2.plot(history['val_iou'], label='Validation')
    ax2.set_title('IoU')
    ax2.set_xlabel('Epoch')
    ax2.set_ylabel('IoU')
    ax2.legend()

    plt.tight_layout()
    plt.show()

def visualize_predictions(model, image_path):
    """Visualize model predictions on a single image"""
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    model.eval()

    # Load and preprocess the image
    transform = transforms.Compose([
        transforms.Resize((256, 256)),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229,
0.224, 0.225])
    ])

    image = Image.open(image_path).convert('RGB')
    input_tensor = transform(image).unsqueeze(0).to(device)

    # Generate prediction
    with torch.no_grad():
        output = model(input_tensor)
        pred = torch.argmax(output, dim=1)[0].cpu().numpy()

    # Denormalize image for visualization
    input_np = input_tensor[0].cpu().numpy()
    input_np = np.transpose(input_np, (1, 2, 0))
    mean = np.array([0.485, 0.456, 0.406])
    std = np.array([0.229, 0.224, 0.225])
    input_np = input_np * std + mean
    input_np = np.clip(input_np, 0, 1)
```

```
    # Create a colormap for visualization
    cmap = plt.cm.get_cmap('tab20', 151)

    # Plot results
    plt.figure(figsize=(10, 5))

    plt.subplot(1, 2, 1)
    plt.imshow(input_np)
    plt.title('Input Image')
    plt.axis('off')

    plt.subplot(1, 2, 2)
    plt.imshow(pred, cmap=cmap, vmin=0, vmax=150)
    plt.title('Segmentation Prediction')
    plt.axis('off')

    plt.tight_layout()
    plt.show()



model, history = train_model()

# Test on a sample image
visualize_predictions(model,
"/content/ADEChallengeData2016/images/validation/ADE_val_00000015.jpg")
```


model, history = train_model()
# Test on a sample image
visualize_predictions(model, "/content/ADEChallengeData2016/images/validation/ADE_val_00000015.jpg")
Train
Validation
Loss
Epoch
IoU
Input Image
Segmentation Prediction