

Technical Report-Assignment11

컴퓨터공학 류다현(2376087)

Assignment11에서는 기본 UNet과 ResNet을 encoder로 사용하는 UNet을 구현하며 Pascal VOC 2012 데이터셋으로 모델을 학습시키고 이미지 segmentation을 수행한다.

UNet.py

UNet.py는 UNet을 구현한다. UNet은 encoder, decoder와 이들을 잇는 skip connection으로 구성된 네트워크이며, 이때 encoder는 pooling 레이어와 convolution으로 이루어져 다운샘플링을 수행하고 decoder는 pooling 레이어와 convolution으로 업샘플링을 수행한다. 구현할 UNet의 전체적인 구조는 아래와 같다.

Question 1 : Implement the UNet model code.

1 conv(in_channels, out_channels)

```
def conv(in_channels, out_channels):  
    return nn.Sequential(  
        nn.Conv2d(in_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)  
    )
```

conv()는 UNet에서 사용할 convolution 블록을 [3 x 3 Conv] → [BN] → [ReLU] → [3 x 3 Conv] → [BN] → [ReLU] 구조로 정의한다.

2 class Unet(nn.Module)

Unet 클래스에서는 UNet 전체 구조를 정의하고 forward() 함수를 구현해 순전파를 정의한다.

2.1 __init__(self, in_channels, out_channels)

__init__()은 UNet의 생성자로, conv() 함수를 호출해 앞서 서술한 UNet 구현에 사용할 convDown 레이어들과 convUp 레이어, max pooling 레이어와 up-convolution 레이어를 정의한다.

```
super(Unet, self).__init__()  
  
self.convDown1 = conv(in_channels, 64)  
self.convDown2 = conv(64, 128)  
self.convDown3 = conv(128, 256)  
self.convDown4 = conv(256, 512)  
self.convDown5 = conv(512, 1024)
```

```
self.maxpool = nn.MaxPool2d(2, stride=2)
self.upsample = nn.Upsample(scale_factor=2, mode='bilinear',
align_corners=True)
```

encoder 구현은 위와 같다. convDown1부터 convDown5까지는 encoder용 convolution 레이어로, 입력 feature map의 채널은 64채널 → 128채널 → 256채널 → 512채널 → 1024채널 순으로 2배씩 증가한다. 이후 nn.MaxPool2d를 사용해 stride=2로 하여 해상도를 2배 줄이는 다운샘플링용 max pooling 레이어 maxpool을 정의하고, 아래 decoder에서 사용하기 위해 nn.Upsample로 scale factor=2로 하여 해상도를 2배 늘리는 업샘플링용 레이어 upsample을 정의한다.

```
self.convUp4 = conv(1536, 512) # convDown4 -> 512 + 1024 = 1536
self.convUp3 = conv(768, 256) # convDown3 -> 256 + 512 = 768
self.convUp2 = conv(384, 128) # convDown2 -> 128 + 256 = 384
self.convUp1 = conv(192, 64) # convDown1 -> 64 + 128 = 192
self.convUp_fin = nn.Conv2d(64, out_channels, 1)
```

decoder 구현은 위와 같다. decoder에서는 skip connection을 이용해 encoder에서 다운샘플링했던 feature map 중 동일한 레벨의 것을 연결(concatenate)한다.

convUp4에서는 1024채널 입력 feature map에 convDown4의 512채널 출력 feature map을 연결한 1024+512=1536채널 feature map을 입력으로 받아 512채널 feature map을 출력하므로 self.convUp4 = conv(1536, 512), convUp3에서는 512채널 입력 feature map에 convDown3의 256채널 출력 feature map을 연결한 512+256=768채널 feature map을 입력으로 받아 256채널 feature map을 출력하므로 self.convUp3 = conv(768, 256)이다. convUp2와 convUp1 또한 동일한 원칙에 따라 채널 수를 계산해 정의한다. 이처럼 UNet에서는 low level의 feature가 high level의 feature와 합쳐질 수 있게 함으로써 성능을 향상할 수 있다. 이후 convUp1 결과에 적용할 1x1 convolution 레이어 convUp_fin을 정의한다.

2.2 forward(self, x)

forward()에서는 UNet의 순전파 과정을 정의한다. UNet의 대략적인 구조는 아래와 같으므로, 이에 맞추어 각 단계를 구현한다.

Encoder

```
[3×3 Conv → BN → ReLU] → [3×3 Conv → BN → ReLU] → [2×2 MaxPool]
[3×3 Conv → BN → ReLU] → [3×3 Conv → BN → ReLU] → [2×2 MaxPool]
[3×3 Conv → BN → ReLU] → [3×3 Conv → BN → ReLU] → [2×2 MaxPool]
[3×3 Conv → BN → ReLU] → [3×3 Conv → BN → ReLU] → [2×2 MaxPool]
[3×3 Conv → BN → ReLU] → [3×3 Conv → BN → ReLU]
```

Decoder

```
[2×2 upConv] → concat → [3×3 Conv → BN → ReLU] → [3×3 Conv → BN → ReLU]
[2×2 upConv] → concat → [3×3 Conv → BN → ReLU] → [3×3 Conv → BN → ReLU]
```

[2×2 upConv] → concat → [3×3 Conv → BN → ReLU] → [3×3 Conv → BN → ReLU]
 [2×2 upConv] → concat → [3×3 Conv → BN → ReLU] → [3×3 Conv → BN → ReLU]
 [1x1 Conv]

```
conv1 = self.convDown1(x)
x = self.maxpool(conv1)

conv2 = self.convDown2(x)
x = self.maxpool(conv2)

conv3 = self.convDown3(x)
x = self.maxpool(conv3)

conv4 = self.convDown4(x)
x = self.maxpool(conv4)

conv5 = self.convDown5(x)
```

encoder의 순전파는 위와 같으며, 위에서 서술한 구조대로 3x3 convolution과 2x2 max pooling을 반복하며 채널과 해상도를 절반씩 줄여 나간다. 매 단계 max pooling한 결과 feature map들은 추후 decoder에서 concat하기 위해 conv1, conv2 등으로 분리해 저장한다.

```
x = self.upsample(conv5)
x = torch.cat([x, conv4], dim=1)
x = self.convUp4(x)

x = self.upsample(x)
x = torch.cat([x, conv3], dim=1)
x = self.convUp3(x)

x = self.upsample(x)
x = torch.cat([x, conv2], dim=1)
x = self.convUp2(x)

x = self.upsample(x)
x = torch.cat([x, conv1], dim=1)
x = self.convUp1(x)
```

decoder의 순전파는 위와 같다. 위에서 서술한 구조대로 2x2 up-convolution을 통한 업샘플링과 3x3 convolution을 반복하며 채널과 해상도를 두 배씩 키워 나가며, 매 upConv-Conv 단계마다 torch.cat() 함수를 호출해 현재 encoder feature map과 대응하는 decoder feature map을 연결한다. 예를 들어, unsample()로 conv5를 업샘플링한 것에 torch.cat([x, conv4], dim=1)하여 conv4(encoder에서 convDown4()한 결과)와 feature map을 연결하고, 그 결과를 convUp4()에 전달한다.

```
out = self.convUp_fin(x)
return out
```

이후 convUp1의 결과 feature map에 1x1 convolution하여 최종 결과 feature map을 생성

한 뒤 이를 반환하고 함수를 종료한다.

resnet_encoder_unet.py

resnet_encoder_unet.py에서는 위 UNet의 ResNet-50을 UNet의 encoder로 사용하는 모델을 구현한다. ResNet 구현은 이전 과제10의 resnet50_full.py의 코드를 대체로 재사용한다.

1 conv1x1(in_channels, out_channels, stride, padding) & conv3x3(in_channels, out_channels, stride, padding)

```
def conv1x1(in_channels, out_channels, stride, padding):
    model = nn.Sequential(
        nn.Conv2d(in_channels, out_channels, kernel_size=1, stride=stride,
                  padding=padding),
        nn.BatchNorm2d(out_channels)
    )
    return model

def conv3x3(in_channels, out_channels, stride, padding):
    model = nn.Sequential(
        nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=stride,
                  padding=padding),
        nn.BatchNorm2d(out_channels)
    )
    return model
```

conv1x1()과 conv3x3()함수는 각각 1x1 convolution 레이어와 3x3 Conv 레이어를 정의하고, 과제 10에 구현된 것과 동일하다.

2 class ResidualBlock(nn.Module)

```
class ResidualBlock(nn.Module):
    def __init__(self, in_channels, middle_channels, out_channels,
                 downsample=False):
        super(ResidualBlock, self).__init__()
        self.downsample = downsample

        if self.downsample:
            self.layer = nn.Sequential(
                conv1x1(in_channels, middle_channels, stride=2, padding=0),
                conv3x3(middle_channels, middle_channels, stride=1, padding=1),
                conv1x1(middle_channels, out_channels, stride=1, padding=0)
            )
            self.downsize = conv1x1(in_channels, out_channels, 2, 0)

        else:
            self.layer = nn.Sequential(
                conv1x1(in_channels, middle_channels, stride=1, padding=0),
                conv3x3(middle_channels, middle_channels, stride=1, padding=1),
                conv1x1(middle_channels, out_channels, stride=1, padding=0)
            )
            self.make_equal_channel = conv1x1(in_channels, out_channels, 1, 0)
            self.activation = nn.ReLU(inplace=True)
```

ResidualBlock 클래스 또한 과제10의 ResNet 구현과 대체로 유사하며, skip connection을 포함하는 residual block을 정의한다.

2.1 forward(self, x)

forward()에서는 입력에 residual block을 적용한 순전파 결과를 반환한다. 마찬가지로 과제 10 ResidualBlock 클래스의 forward()와 유사하나 일부 변경된 부분이 존재한다.

forward()에서는 입력에 residual block을 적용한 순전파 결과를 반환한다. 마찬가지로 과제 10 ResidualBlock 클래스의 forward()와 유사하나 일부 변경된 부분이 존재한다.

```
def forward(self, x):
    if self.downsample:
        out = self.layer(x)
        x = self.downsize(x)
        return out + x
    else:
        out = self.layer(x)
        if x.size() is not out.size():
            x = self.make_equal_channel(x)
        return out + x
```

위는 과제10 ResidualBlock 클래스의 forward()이며 convolution 결과 out에 원본 입력 x를 더한 out+x를 반환한다.

```
def forward(self, x):
    if self.downsample:
        out = self.layer(x)
        x = self.downsize(x)
        return self.activation(out + x)
    else:
        out = self.layer(x)
        if x.size() is not out.size():
            x = self.make_equal_channel(x)
        return self.activation(out + x)
```

반면 과제11 ResidualBlock 클래스의 forward는 (out+x)가 아닌 activation(out+x)를 반환함으로써 convolution 결과와 원본을 더한 것에 ReLU 활성화 함수를 한번 더 적용한 뒤 반환한다.

Question 2 : Implement the forward function of Resnet_encoder_UNet.

3 conv(in_channels, out_channels)

```
def conv(in_channels, out_channels):
    return nn.Sequential(
        nn.Conv2d(in_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)
```

)

ResNet-50을 encoder로 가지는 UNet 모델에서 사용할 convolution 블록을 정의한다. UNet.py에서 구현한 바와 같이 [3 x 3 Conv] → [BN] → [ReLU] → [3 x 3 Conv] → [BN] → [ReLU] 구조를 사용한다.

이때 ReLU()에서 inplace를 True로 설정하면 ReLU는 별도의 출력 할당 없이 원본 텐서를 직접 조정함으로써 메모리를 절약한다.

4 class UNetWithResnet50Encoder(nn.Module)

UNetWithResnet50Encoder 클래스에서는 ResNet-50을 encoder로 사용하는 UNet의 전체 구조를 정의하고 forward() 함수를 구현해 순전파를 정의한다. 과제10 ResNet50_layer4 클래스와 전반적으로 유사하다.

4.1 __init__(self, n_classes=22) & forward(self, x, with_output_feature_map=False)

__init__()은 UNetWithResnet50Encoder 클래스의 생성자로 앞서 정의한 ResidualBlock() 함수를 호출해 UNet의 encoder 및 decoder에서 사용할 블록들을 구현한다.

```
super().__init__()
self.n_classes = n_classes
self.layer1 = nn.Sequential(
    nn.Conv2d(3, 64, kernel_size=7, stride=2, padding=3),
    nn.BatchNorm2d(64),
    nn.ReLU(inplace=True),
)
self.pool = nn.MaxPool2d(3, 2, 1, return_indices=True)

self.layer2 = nn.Sequential(
    ResidualBlock(64, 64, 256, False),
    ResidualBlock(256, 64, 256, False),
    ResidualBlock(256, 64, 256, True)
)

self.layer3 = nn.Sequential(
    ResidualBlock(256, 128, 512, False),
    ResidualBlock(512, 128, 512, False),
    ResidualBlock(512, 128, 512, False),
    ResidualBlock(512, 128, 512, False)
)
```

layer1, layer2, layer3은 과제10에 구현된 것과 동일하다. ResidualBlock()을 반복적으로 호출하여 ResNet50의 레이어 구조를 구현하지만 과제10에서 layer4까지 구현했던 것과 달리 layer3까지의 코드만을 재사용하고 이후 구현은 달리한다. 정확한 설명을 위해 구현된 decoder 레이어와 해당 레이어가 순전파 함수 forward()에서 사용되는 방식을 비교하며 설명한다.

```
self.bridge = conv(512, 512)
self.UnetConv1 = conv(512, 256)
self.UpConv1 = nn.Conv2d(512, 256, 3, padding=1)
```

이때 bridge는 ResNet-50 encoder와 기존 UNet decoder를 연결하는 역할을 한다. 위 layer3을 호출한 결과, 즉 encoder의 최종 출력인 512 채널의 feature map을 입력받으므로 self.bridge = conv(512, 512)로 호출하여 두 번의 3x3 convolution과 BN, ReLU를 적용한다. UNetConv1은 bridge의 결과를 256 채널로 줄여 convolution하며 이것이 decoder에 전달 된다.

```
out1 = self.layer1(x)
out1, indices = self.pool(out1)
out2 = self.layer2(out1)
out3 = self.layer3(out2)
x = self.bridge(out3)
x = self.UpConv1(x)
```

위는 forward()에서 out1, out2, out3은 입력 x에 대해 encoder residual block에서 계산한 단계별 결과이다. layer3의 출력 out3에 bridge를 적용해 decoder에 진입하며, UpConv1(x) 결과 x는 256채널 feature map이다.

```
self.upconv2_1 = nn.ConvTranspose2d(256, 256, 3, 2, 1)
self.upconv2_2 = nn.Conv2d(256, 64, 3, padding=1)
```

upconv2_1은 nn.ConvTranspose2d()를 호출함으로써 stride=2인 transpose convolution으로 spatial resolution을 2배 늘려 업샘플링하며, upconv2_2는 256채널 입력에 convolution을 적용해 64채널 feature map을 출력한다.

```
x = torch.cat([x, out2], dim=1)
x = self.UnetConv1(x)
x = self.upconv2_1(x, output_size=torch.Size([x.size(0), 256, 64, 64]))
x = self.upconv2_2(x)
```

forward()에서는 이들을 위와 같이 사용한다. UNet 모델이므로 대응하는 encoder 출력과 concat하게 되며, 위의 기본 UNet과 동일하게 torch.cat()을 사용해 x와 out2를 연결하면 x와 out2가 모두 256채널이므로 연결 결과 512 채널의 feature map이 생성된다. UnetConv1(x)하여 x의 채널을 256채널로 줄이고 upconv2_1(x)로 2배 업샘플링하며 upconv2_2(x) 함으로써 그 결과를 64채널 feature map으로 변환한다.

```
self.unpool = nn.MaxUnpool2d(3, 2, 1)
self.UnetConv2_1 = nn.ConvTranspose2d(64, 64, 3, 2, 1)
self.UnetConv2_2 = nn.ConvTranspose2d(128, 128, 3, 2, 1)
self.UnetConv2_3 = nn.Conv2d(128, 64, 3, padding=1)
```

unpool은 nn.MaxUnpool2d()를 호출해 stride=2인 max unpooling으로 spatial resolution을 2배 늘려 업샘플링하고, UnetConv2_1, UnetConv2_2는 각각 stride=2인 transpose convolution을 두 번 적용해 spatial resolution을 각각 2배씩 총 4배 늘려 업샘플링하며 UnetConv2_3은 128채널 입력에 convolution하여 64채널 feature map을 출력한다.

```
x = torch.cat([x, out1], dim=1)
x = self.UnetConv2_2(x, output_size=torch.Size([x.size(0), 128, 128, 128]))
x = self.UnetConv2_2(x, output_size=torch.Size([x.size(0), 128, 256, 256]))
x = self.UnetConv2_3(x)
```

forward()에서는 위와 같이 활용한다. 마찬가지로 64채널 x를 대응하는 encoder 출력인 64채널 out1과 연결하므로 연결 결과 128채널의 feature map이 생성되며, 이에 UnetConv2_2를 두 번 적용하여 spatial resolution을 총 4배 줄인 뒤 UnetConv2_3(x)하여 128채널에서 64채널로 변경한다.

```
self.UnetConv3 = nn.Conv2d(64, self.n_classes, kernel_size=1, stride=1)
```

UnetConv3은 64채널 입력 feature map에 1x1 convolution한다.

```
x = self.UnetConv3(x)
return x
```

forward()에서 UnetConv3(x)함으로써 기본 Unet의 마지막 1x1 convolution을 구현하며, 최종 출력 x를 반환하고 함수를 종료한다.

Question 3 : Implement the train/test module.

modules.py

modules.py는 앞서 구현한 모델을 학습 및 평가하기 위한 모듈이다.

- 1 def train_model(trainloader, model, criterion, optimizer, scheduler, device)
train_model()에서는 모델을 훈련하며, 학습 과정에서 사용할 trainloader, 모델, 손실 함수, optimizer, scheduler, 디바이스를 파라미터로 전달받는다.

```
def train_model(trainloader, model, criterion, optimizer, scheduler, device):
    model.train()

    for i, (inputs, labels) in enumerate(trainloader):
        from datetime import datetime

        inputs = inputs.to(device)
        labels = labels.to(device=device, dtype=torch.int64)
        criterion = criterion.cuda()

        optimizer.zero_grad() # init grad
        outputs = model(inputs) # forward prop
        loss = criterion(outputs, labels) # calc loss
        loss.backward() # backprop
        optimizer.step() # update param
        scheduler.step() # update scheduler
```

trainlaoder의 (input, label)쌍에 대해 for문으로 순회하며 가중치 초기화→순전파→손실 계산→역전파→파라미터 갱신→스케줄러 갱신 과정을 거친다. 코드별로 수행하는 작업은 주석과 같다.

- 2 accuracy_check(label, pred)

```
def accuracy_check(label, pred):
    ims = [label, pred]
    np_ims = []
    for item in ims:
        item = np.array(item)
```



```

    np_ims.append(item)
    compare = np.equal(np_ims[0], np_ims[1])
    accuracy = np.sum(compare)
    return accuracy / len(np_ims[0].flatten())

```

accuracy_check()는 ground truth와 모델 예측값을 파라미터로 받아 예측값 pred의 정확도를 계산한다. label, pred 쌍을 ims 배열에 저장하고 ims의 원소 전체를 순회하며 label과 예측값을 비교하여 결과가 일치할 경우 compare에 저장한다. 이후 np.sum(compare)하여 label과 예측값이 일치한 개수를 카운트한 뒤 이를 전체 픽셀 수로 나누어 정확도를 반환한다.

3 accuracy_check_for_batch(labels, preds, batch_size)

```

def accuracy_check_for_batch(labels, preds, batch_size):
    total_acc = 0
    for i in range(batch_size):
        total_acc += accuracy_check(labels[i], preds[i])
    return total_acc/batch_size

```

accuracy_check_for_batch()는 accuracy_chack()와 유사하게 label과 예측값을 비교해 정확도를 계산하며, 배치 사이즈 단위로 정확도를 평균한 것을 반환한다.

4 get_loss_train(model, trainloader, criterion, device)

get_loss_train()은 데이터셋에 대한 손실과 정확도를 계산한다.

```

model.eval()
total_acc = 0
total_loss = 0

```

모델을 evaluation 모드로 전환하고 정확도와 손실 변수를 초기화한다.

```

for batch, (inputs, labels) in enumerate(trainloader):
    with torch.no_grad():
        inputs = inputs.to(device)
        labels = labels.to(device = device, dtype = torch.int64)
        inputs = inputs.float()

        outputs = model(inputs) # forward prop
        loss = criterion(outputs, labels) # calc loss

        outputs = np.transpose(outputs.cpu(), (0,2,3,1))
        preds = torch.argmax(outputs, dim=3).float()
        acc = accuracy_check_for_batch(labels.cpu(), preds.cpu(),
                                       inputs.size()[0]) # calc accuracy for batch

        total_acc += acc # accumulate accuracy
        total_loss += loss.cpu().item() # accumulate loss

    return total_acc/(batch+1), total_loss/(batch+1)

```

이후 trainloader 내 입력과 label 쌍에 대해 for문으로 순회하며 순전파→손실 계산→정확도 계산 과정을 거친다. train_model()에서 모델을 학습시켰으므로 오차를 역전파하고 파라미터를 업데이트한 것과 달리, 현재 evaluation 모드에서는 gradient를 계산하고 있지 않으므로 파라미터를 업데이트하지 않는다. 따라서 파라미터 초기화(zero_grad), 역전파(backward()), 파라미터 업데이

트(step()) 등을 모두 삭제해 손실과 정확도만 계산한다. 코드별 수행하는 작업은 주석과 같으며, 계산된 총 정확도와 총 손실을 반환하고 함수를 종료한다.

5 val_model(model, valloader, criterion, device, dir)

val_model()은 테스트 데이터셋에 대해 모델이 예측한 결과의 손실과 정확도를 계산하고, 결과를 저장한다.

```
cls_invert = {0: (0, 0, 0), 1: (128, 0, 0), 2: (0, 128, 0),
              # 0:background, 1:aeroplane, 2:bicycle
              3: (128, 128, 0), 4: (0, 0, 128), 5: (128, 0, 128),
              # 3:bird, 4:boat, 5:bottle
              6: (0, 128, 128), 7: (128, 128, 128), 8: (64, 0, 0),
              # 6:bus, 7:car, 8:cat
              9: (192, 0, 0), 10: (64, 128, 0), 11: (192, 128, 0),
              # 9:chair, 10:cow, 11:diningtable
              12: (64, 0, 128), 13: (192, 0, 128), 14: (64, 128, 128),
              # 12:dog, 13:horse, 14:motorbike
              15: (192, 128, 128), 16: (0, 64, 0), 17: (128, 64, 0),
              # 15:person, 16:pottedplant, 17:sheep
              18: (0, 192, 0), 19: (128, 192, 0), 20: (0, 64, 128),
              # 18:sofa, 19:train, 20:tvmonitor
              21: (224, 224, 192)}

total_val_loss = 0
total_val_acc = 0
n=0
```

cls_invert 딕셔너리에서 클래스 인덱스가 어떤 RGB 색상으로 매핑되는지 지정한다. 0: (0, 0, 0), 1: (255, 0, 0)로 설정한다면 결과 이미지에서 인덱스가 0인 클래스(위 경우 배경)는 검은색, 인덱스가 1인 클래스(위 경우 비행기)는 빨간색으로 표시된다. 이후 총 validation 손실과 정확도로 0으로 초기화되며, n은 이미지 번호로 추후 for문을 순회함에 따라 1씩 증가한다.

```
for batch, (inputs, labels) in enumerate(valloader):
    with torch.no_grad():
        ...
```

valloader 내의 (입력, label)쌍에 대해 for문을 순회하며 정확도와 손실을 계산한다.

get_loss_train()의 경우와 마찬가지로 evaluation 모드이므로 gradient를 계산하지 않으며 파라미터를 업데이트하지 않는다. 따라서 파라미터 초기화(zero_grad), 역전파(backward()), 파라미터 업데이트(step()) 등을 모두 삭제해 손실과 정확도만 계산한다.

```
inputs = inputs.to(device)
labels = labels.to(device=device, dtype=torch.int64)

outputs = model(inputs)
loss = criterion(outputs, labels)
outputs = np.transpose(outputs.cpu(), (0, 2, 3, 1))
preds = torch.argmax(outputs, dim=3).float()
acc = accuracy_check_for_batch(labels.cpu(), preds.cpu(), inputs.size()[0])
total_val_acc += acc
total_val_loss += loss.cpu().item()
```

이후 결과를 예측하고 그 결과에 대해 손실과 정확도를 계산하며, 계산한 것을 total_val_acc와 total_val_loss에 누적한다.

```
for i in range(preds.shape[0]):
    temp = preds[i].cpu().data.numpy()
    temp_l = labels[i].cpu().data.numpy()
    temp_rgb = np.zeros((temp.shape[0], temp.shape[1], 3))
    temp_label = np.zeros((temp.shape[0], temp.shape[1], 3))
    for j in range(temp.shape[0]):
        for k in range(temp.shape[1]):
            temp_rgb[j, k] = cls_invert[int(temp[j, k])]
            temp_label[j, k] = cls_invert[int(temp_l[j, k])]

    img = inputs[i].cpu()
    img = np.transpose(img, (2, 1, 0))

    img_print = Image.fromarray(np.uint8(temp_label))
    mask_print = Image.fromarray(np.uint8(temp_rgb))

    img_print.save(dir + str(n) + 'label' + '.png')
    mask_print.save(dir + str(n) + 'result' + '.png')

    n += 1
```

이후 모델 예측값 preds 배열과 labels 배열의 값들을 컬러 이미지로 변환하고 파일로 저장한다. 내부의 이중 for문에서 앞서 정의한 cls_invert 딕셔너리에 따라 temp_rgb[j, k]에 예측된 클래스의 색상, temp_label[j, k]에 실제 label 색상을 저장한 뒤 for문을 벗어난 후 저장된 것에 따라 Image.fromarray() 배열→이미지 변환한 뒤 결과를 .png 파일로 저장한다. 이때 n을 사용하여 1씩 증가하는 순차적인 파일명으로 저장하며, 다음 반복으로 넘어가기 전에 n을 1 늘려 다음 파일명을 갱신한다.

datasets.py

dataests.py에서는 데이터셋을 불러오고 사용하기 위한 Dataset 클래스를 구현한다. _check_exists() 함수에서 데이터셋 저장 경로가 유효한지 확인하고 VOCdataloader()에서 이미지와 label 파일을 전처리해 반환하며, __len__() 함수와 __getitem__() 함수에서는 각각 데이터셋의 이미지 개수를 반환하고 이미지와 마스크 튜플을 반환한다.

__init__()에서는 지정된 경로로부터 데이터를 읽어 그 중 80%를 훈련용 데이터셋으로, 나머지 20%를 테스트용 데이터셋으로 분리한다.

Question 4 : Implement the main code.

main.py

main.py에서는 실행 환경을 설정하고 데이터를 로드해 전처리한 뒤 구현한 모델들을 사용해 학습과 모델 평가를 수행한다.

```
batch_size = 16
```

```
learning_rate = 0.001
data_dir = "../VOCdevkit/VOC2012"
resize_size = 256
```

코드 상단에서 외부 라이브러리와 모듈을 import한 뒤 위와 같이 hyperparameter를 설정한다. 배치 크기는 16, 학습률은 0.001, 이미지 resize 크기=256으로 설정하였으며 VOC 데이터셋 저장 경로는 상대 경로로 사용하였다.

```
transforms = transforms.Compose([
    transforms.ToPILImage(),
    transforms.Resize([resize_size,resize_size], PIL.NEAREST),
    transforms.ToTensor(),
    transforms.Normalize((0.485, 0.456, 0.406), (0.229, 0.224, 0.225))
])
print("trainset")
trainset = Loader(data_dir, flag='train', resize = resize_size, transforms =
transforms)
print("valset")
valset = Loader(data_dir, flag = 'val', resize = resize_size, transforms =
transforms)

print("tainLoader")
trainLoader = DataLoader(trainset, batch_size = batch_size, shuffle=True)
print("valLoader")
validLoader = DataLoader(valset, batch_size = batch_size, shuffle=True)
```

이후 데이터를 전처리하고 앞서 정의한 dataloader를 사용해 훈련 데이터셋과 테스트 데이터셋을 생성한다.

```
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
scheduler = StepLR(optimizer, step_size=4, gamma=0.1)

epochs = 40
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
model = model.to(device)
```

학습에 사용할 손실 함수, 활성화 함수, 스케줄러를 지정하고 학습할 epoch를 설정한다. 위에서는 cross entropy loss와 Adam optimizer, StepLR 스케줄러를 사용해 40 epoch 동안 학습하는 것으로 설정하였다. 이후 파라미터 체크포인트를 불러오고 결과를 저장할 각종 폴더를 생성한 뒤 모델을 학습하고, 학습 과정의 손실과 정확도를 plot하며 모델이 예측한 값을 이미지로 시각화해 label segmentation 결과와 예측 segmentation 결과를 .png 이미지로 저장한다.

```
if epoch % 4 == 0:
    savepath2 = savepath1 + str(epoch) + ".pth"
    torch.save(model.state_dict(), savepath2)
    print(f"Saved checkpoint: {savepath2}")
```

이때 위와 같이 구현하여 epoch가 4의 배수일 때마다 체크포인트를 저장하도록 하였다.

결과 분석

PyCharm 로컬 환경에서 CPU를 사용해 학습한 결과는 아래와 같다.

```
C:\Users\Wrdh08\AppData\Local\Programs\Python\Python311\python.exe
C:\Users\Wrdh08\PycharmProjects\WopenSWproject\Wassignment11\main.py
```

```
trainset
valset
tainLoader
valLoader
pretrained checkpoint loaded
Training

iters 146
[Train] lter 0/146, loss = 1.4836
[Train] lter 1/146, loss = 1.0413
[Train] lter 2/146, loss = 1.2559
[Train] lter 3/146, loss = 1.0150
[Train] lter 4/146, loss = 0.9525
[Train] lter 5/146, loss = 1.2491
[Train] lter 6/146, loss = 0.9684
[Train] lter 7/146, loss = 0.8903
[Train] lter 8/146, loss = 0.7450
[Train] lter 9/146, loss = 1.1689
[Train] lter 10/146, loss = 0.9646
[Train] lter 11/146, loss = 1.2412
[Train] lter 12/146, loss = 0.8406
[Train] lter 13/146, loss = 1.6459
[Train] lter 14/146, loss = 1.0160
[Train] lter 15/146, loss = 0.8003
[Train] lter 16/146, loss = 1.2013
[Train] lter 17/146, loss = 1.1702
[Train] lter 18/146, loss = 1.2633
[Train] lter 19/146, loss = 0.6963
[Train] lter 20/146, loss = 0.8728
```

종료 코드 -1(으)로 완료된 프로세스

Pretrained parameter를 사용하였으므로 1 epoch동안 실행하였다. modules.py의 train_model()에 코드를 추가하여 총 iteration 횟수 iters를 표시하고 1 iteration마다 중간 결과 loss를 출력하도록 하였으나, 실행 시간이 지나치게 길어져 20 iteration에서 임의로 종료하고 구글 colab 환경으로 변경해 재 실행하였다. Colab에서 사용하기 위해 변경한 코드와 실행 결과는 아래와 같다.

```
!mkdir -p ../VOCdevkit

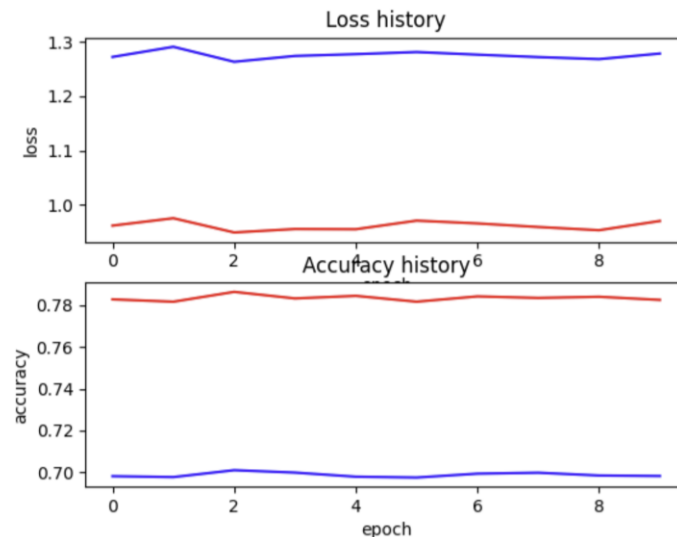
!wget -q http://host.robots.ox.ac.uk/pascal/VOC/voc2012/VOCtrainval_11-May-2012.tar
!tar xf VOCtrainval_11-May-2012.tar -C ../VOCdevkit
```

```
import os
os.chdir('/content/drive/MyDrive/assignment11')
```

```
%run main.py
```


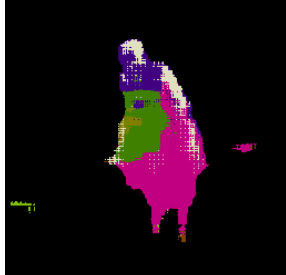
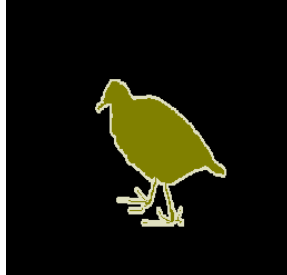
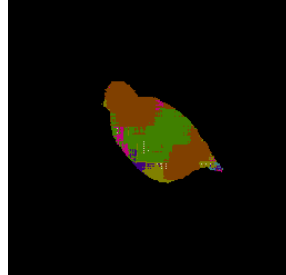

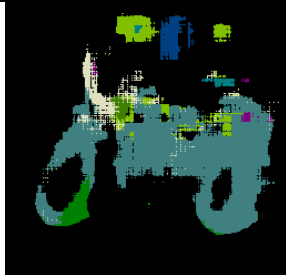

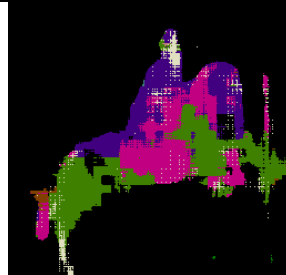
```
trainset
valset
tainLoader
valLoader
pretrained checkpoint loaded
Training
epoch 1 train loss : 0.9618364927703387 train acc : 0.7828649874790519
epoch 1 val loss : 1.2721719113556114 val acc : 0.6980825286727768
Saved checkpoint: ./output/model12025-06-12(06-01)/0.pth
epoch 2 train loss : 0.9754330144353109 train acc : 0.7817249351539469
epoch 2 val loss : 1.2908254845722302 val acc : 0.697657335958882
epoch 3 train loss : 0.9492595032469867 train acc : 0.786449844483866
epoch 3 val loss : 1.2632520698212288 val acc : 0.7009560181213929
epoch 4 train loss : 0.9555706957431689 train acc : 0.7833028089064886
epoch 4 val loss : 1.2740222006230741 val acc : 0.6998084386189778
epoch 5 train loss : 0.9551617143905327 train acc : 0.7845371194081764
epoch 5 val loss : 1.2773069320498287 val acc : 0.6978384310060793
Saved checkpoint: ./output/model12025-06-12(06-01)/4.pth
epoch 6 train loss : 0.9710144616969644 train acc : 0.7817379617750422
epoch 6 val loss : 1.2811482178198326 val acc : 0.6974442542136252
epoch 7 train loss : 0.965878909581328 train acc : 0.7842791641634399
epoch 7 val loss : 1.2763869842967472 val acc : 0.6992977331350515
epoch 8 train loss : 0.9592858610087878 train acc : 0.7835432987284393
epoch 8 val loss : 1.2719008165436823 val acc : 0.699748606295199
epoch 9 train loss : 0.9533549869713718 train acc : 0.7840993674576358
epoch 9 val loss : 1.2681595715316567 val acc : 0.6984144674765097
Saved checkpoint: ./output/model12025-06-12(06-01)/8.pth
epoch 10 train loss : 0.9703835089729257 train acc : 0.7826051872367431
epoch 10 val loss : 1.2783952999759365 val acc : 0.6981655842549092
Finish Training
Fin
```

PyCharm에서 작성한 코드를 동일하게 사용하되 .ipynb 파일에서 main.py를 실행하였으며, 구글 드라이브에 mount하여 필요한 데이터셋을 저장하였다. 매 epoch마다 훈련 정확도와 validation 정확도가 출력되며, 4 epoch 간격으로 폴더에 checkpoint가 저장되는 것을 확인할 수 있다. Colab GPU 환경에서 10 epoch만큼 학습한 손실 및 정확도 곡선은 다음과 같다.



손실 곡선과 정확도 곡선이 뚜렷하게 감소하거나 증가하지 않는 것을 확인할 수 있는데, 이는 epoch를 작게 하여 모델을 충분히 학습시키지 않았기 때문이라 추측된다.

history/result2025-06-12(06-01)/predicted 폴더에 epoch0부터 epoch9까지 이미지 label과 segmentation 결과가 저장되어 있으며 그중 Epoch9 폴더에서 segmentation 결과가 우수한 것을 몇 개 나열하면 아래와 같다.

10label.png	10result.png	16label.png	16result.png
			
32label.png	32result.png	48label.png	48result.png
			
225label.png	225result.png	398label.png	398result.png
