Technical report-Assignment10

컴퓨터공학 류다현(2376087)

vgg16_full.py

VGG는 작은 3*3 conv 필터를 더 많이 연결하여 네트워크 깊이를 깊게 한 모델로, 자세한 구조는 아래와 같다.

Input: 3 x 224 x 224

[3 x 3 Conv, 64] → [3 x 3 Conv, 64] → MaxPool
[3 x 3 Conv, 128] → [3 x 3 Conv, 128] → MaxPool
[3 x 3 Conv, 256] → [3 x 3 Conv, 256] → [3 x 3 Conv, 256] → MaxPool
[3 x 3 Conv, 512] → [3 x 3 Conv, 512] → [3 x 3 Conv, 512] → MaxPool
[3 x 3 Conv, 512] → [3 x 3 Conv, 512] → [3 x 3 Conv, 512] → MaxPoo

→ [FC 4096] → ReLU → Dropout
→ [FC 4096] → ReLU → Dropout
→ [FC 1000] → Softmax

vgg16_full.py에서는 위 구조에 따라 VGG16을 구현하고 이를 이용해 CIFAR-10 데이터셋에서 이미지 분류를 수행한다.

1 class VGG(nn.Module)

VGG 클래스는 VGG16 모델의 전체 구조를 정의한다.

1.1 __init__(self, features)

```
super(VGG, self).__init__()
```

__init__()은 VGG 클래스의 생성자로, super(VGG, self).__init__()을 통해 모델을 초기화하며 시작한다.

```
self.features = features
self.classifier = nn.Sequential(
       nn.Dropout(),
       nn.Linear(512, 512),
       nn.BatchNorm1d(512),
       nn.ReLU(True),
       nn.Dropout(),
       nn.Linear(512, 10),
)
```

features는 convolution layer들의 묶음이며, classifier는 nn.Sequential 모듈을 사용해 FC layer 부분을 정의한 것이다. 위와 같이 구현하였으므로 순전파 시 convolution 결과 벡터는

dropout

512차원 입력 텐서를 512차원 출력 텐서로 선형 변환

batch normalization

ReLU 적용

Dropout

512차원 입력 텐서를 10차원 출력 텐서로 선형 변환

의 과정을 거쳐 최종 값이 반환된다.

```
for m in self.modules():
            if isinstance(m, nn.Conv2d):
                n = m.kernel_size[0] * m.kernel_size[1] * m.out_channels
                m.weight.data.normal_(0, math.sqrt(2. / n))
                m.bias.data.zero_()
```

이후 위와 같이 가중치를 초기화한다.

1.2 forward(self, x)

forward()는 모델의 순전파 과정을 정의한다.

```
x = self.features(x)
x = x.view(x.size(0), -1)
x = self.classifier(x)
return x
```

위에서 정의한 features() 함수를 호출해 x를 features()에 전달함으로써 입력을 convolution layer에 통과시키고 생성된 feature map을 저장하며, view()를 사용해 그 결과를 flatten한 뒤 이를 classifier()에 전달해 FC layer에 통과시킨다.

2 make_layers(cfg, batch_norm=False)

make_layers()는 파라미터 cfg에 따라 convolution 블록의 레이어들을 생성한다.

```
layers = []
in_channels = 3
```

layers는 레이어를 저장할 빈 리스트이며, RGB 이미지를 사용하므로 입력 채널 in_channels는 3으로 설정한다.

```
for v in cfg:
        ...
```

이후 for문을 순회하며 cfg에 정의된 레이어들을 순서대로 layers에 추가한다.

```
if v == 'M':
    layers += [nn.MaxPool2d(kernel_size=2, stride=2)]
else:
    conv2d = nn.Conv2d(in_channels, v, kernel_size=3, padding=1)
    if batch_norm:
        layers += [conv2d, nn.BatchNorm2d(v), nn.ReLU(inplace=True)]
    else:
```

```
            layers += [conv2d, nn.ReLU(inplace=True)]
        in_channels = v
```

v가 M이라면 max pooling 레이어이므로 layers에 MaxPool2d를 추가하고, v가 숫자라면 in_channels x v 크기인 Conv2D 레이어를 생성해 conv2d에 저장한다. 배치 정규화 여부인 batch_norm 파라미터가 True라면 layers에 conv2d, BatchNorm2d, ReLU 순서대로 레이어를 추가하고 False라면 BatchNorm2d를 제외해 추가한다. 이후 다음 입력 채널을 v로 업데이트해 그 다음 레이어로 이어질 수 있게 하며, cfg의 모든 원소에 대해 위 과정을 반복한다.

```
    return nn.Sequential(*layers)
```

for문이 종료되면 사용할 레이어들이 layers 배열에 저장 완료되므로, nn.Sequential()에 layers를 전달해 레이어들을 묶어 반환한다.

3 vgg16()

vgg16()은 VGG16 모델을 생성한다.

```
cfg = [64, 64, 'M', 128, 128, 'M', 256, 256, 256, 'M', 512, 512, 512, 'M', 512, 512, 512, 'M']
return VGG(make_layers(cfg))
```

vgg16() 에서는 모델에서 사용할 레이어 구조를 정의해 cfg 배열에 저장하고, 위에서 정의한 make_layers() 함수에 cfg를 전달해 이를 다시 VGG() 에 넘김으로써 전체 모델을 생성한다.

resnet50_full.py

ResNet은 입력을 출력에 그대로 전달해 더할 수 있게 하는 skip connection을 추가한 residual block을 사용함으로써 레이어가 깊어짐에 따라 발생하는 vanishing gradient 문제를 해결하는 모델이다. resnet50_full.py에서는 resnset50을 구현하고, 이를 이용해 CIFAR-10 데이터셋에서 이미지 분류를 수행한다.

1 conv1x1(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),
        nn.ReLU(inplace=True)
    )
    return model
```

추후 사용될 1 x 1 convolutional block을 정의한다. 입력 파라미터는 다음과 같다.

in_channels: 입력 채널
out_channels: 출력 채널
stride: 필터를 슬라이드할 때 사용할 stride 크기
padding: 사용할 padding 크기

nn.Conv2d(in_channels, out_channels, kernel_size=1, stride=stride, padding=padding)이므로, 1 x 1 크기 필터를 사용해 입력의 채널 수를 출력 채널 수로 변환한다. 이후 convolution한 결과에 배치 정규화를 수행하고, ReLU 함수를 적용한다. 이들을 nn.Sequential()로 묶은 model을 반환하므로, conv1x1()을 호출할 때마다 이와 같이 구현된 convolutional block을 얻을 수 있다.

2 conv3x3(in_channels, out_channels, stride, padding)

```
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),
        nn.ReLU(inplace=True)
    )
    return model
```

추후 사용될 3 x 3 convolutional block을 정의한다. 입력 파라미터는 다음과 같다.

in_channels: 입력 채널
out_channels: 출력 채널
stride: 필터를 슬라이드할 때 사용할 stride 크기
padding: 사용할 padding 크기

nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=stride, padding=padding)이므로, 3 x 3 크기 필터를 사용해 입력의 채널 수를 출력 채널 수로 변환한다. 이후 convolution한 결과에 배치 정규화를 수행하고 ReLU 함수를 적용한다. 이들을 nn.Sequential()로 묶은 model을 반환하므로, conv3x3()을 호출할 때마다 이와 같이 구현된 convolutional block을 얻을 수 있다.

Question 1 : Implement the "bottle neck building block" part.

3 class ResidualBlock(nn.Module)

ResidualBlock 클래스는 skip connection을 포함하는 residual block을 정의한다.

3.1 __init__(self, in_channels, middle_channels, out_channels, downsample=False)

__init__()은 ResidualBlock 클래스의 생성자로, 사용하는 파라미터는 다음과 같다.

in_channels: 입력 채널
middle_channels: bottleneck 구조의 중간 채널
out_channels: 출력 채널
downsample: 다운샘플링 여부

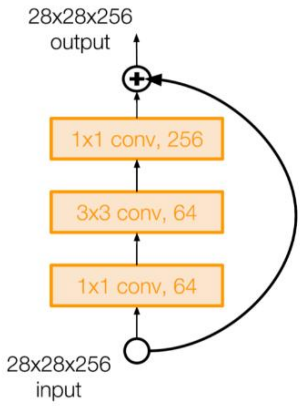
```
super(ResidualBlock, self).__init__()
self.downsample = downsample
```

super(ResidualBlock, self).__init__()을 통해 모델을 초기화하며 시작하고, 다운샘플링 여부에

따라 다르게 처리한다.

```
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)
```

downsample=True일 때 위와 같이 구현한다. Residual block의 구조는 그림과 같고 이 과정을 수식으로 나타내면 원본 입력 x와 x가 convolution layer를 통과한 결과 F(x)에 대해 residual block의 결과 H(x)는 H(x)=F(x)+x이며, 앞서 정의한 conv1x1()과 conv3x3()을 이용해 [1 x 1 conv] → [3 x 3 conv] → [1 x 1 conv] 구조와 skip connection을 구현한다. 다운샘플링을 수행하므로 첫 번째 conv1x1()을 호출할 때 stride=2로 전달하고, conv3x3()과 conv1x1()을 순서대로 호출하며 이들은 stride=1로 설정한다. 이 과정에서 해상도가 절반으로 줄어들었으므로, skip connection에 사용할 downsize 또한 stride=2로 설정해 convolution 결과와 해상도를 통일한다.




```
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)
```

downsample=False일 때 위와 같이 구현한다. 앞서 정의한 conv1x1()과 conv3x3()을 이용해 [1 x 1 conv] → [3 x 3 conv] → [1 x 1 conv] 구조와 skip connection을 구현하는 것은 동일하나, 다운샘플링하지 않으므로 모든 호출에서 stride=1로 전달한다. F(x)와 x의 채널 수가 다른 경우의 skip connection에 사용하기 위해 downsize와 별개인 make_equal_channel을 선언하고 stride=1로 호출한 conv1x1을 저장한다. stride=1로 1x1 convolution을 수행하면 이미지 크기를 유지한 채 채널만 변경할 수 있으므로 이와 같이 구현한다.

4 forward(self, x)

forward()에서는 입력에 residual block을 적용한 결과를 반환한다. 앞서 구현한 내용을 바탕으로, 입력 x는 세 개의 convolution layer를 통과한 뒤 skip connection으로 전달된 원본 x와 합산된다. 3에서 다운샘플링 여부에 따라 skip connection을 달리 구현하였으므로, 마찬가지로 downsample=True인 경우와 downsample=False인 경우를 분리한다.

```
if self.downsample:
```

```
    out = self.layer(x)
    x = self.downsize(x)
    return out + x
```

다운샘플링하는 경우라면 3에서 선언한 downsize를 사용해 x의 크기를 줄여 out의 크기와 동일하게 한다. 이후 convolution 결과인 out에 원본 입력 x를 더한 것을 반환한다.

```
else:
    out = self.layer(x)
    if x.size() is not out.size():
            x = self.make_equal_channel(x)
    return out + x
```

다운샘플링하지 않는다면 out = self.layer(x)는 동일하게 계산하지만, x와 out의 크기가 다를 경우, 즉 H(x)=F(x)+x에서 x와 F(x)의 차원이 다를 경우 x를 make_equal_channel(x)로 설정해 x의 채널을 F(x)에 맞춘 뒤 out+x를 반환한다.

Question 2 : Implement the "class, ResNet50_layer4" part.

5 ResNet50_layer4(nn.Module)

ResNet50_layer4 클래스에서는 ResNet50의

Layer1: [7 x 7 Conv, 64/2] → MaxPool
Layer2: [1x1(64->64), 3x3(64), 1x1(64->256)] x2
Layer3: [1x1(256->128), 3x3(128), 1x1(128->512)] x3
Layer4: [1x1(512->256), 3x3(256), 1x1(256->1024)] x6

4계층 구조를 정의하고 그 뒤에 FC layer와 AvgPool을 연결한다.

5.1 __init__(self, num_classes=10)

```
def __init__(self, num_classes=10:
super(ResNet50_layer4, self).__init__()
```

__init__() 에서 ResNet50의 layer1~4 구조를 구현한다. 훈련에 사용할 CIFAR-10 데이터셋의 클래스가 10개이므로 num_classes=10으로 설정한다. 이후 super()를 호출해 모듈을 초기화한다.

```
self.layer1 = nn.Sequential(
            nn.Conv2d(3, 64, 7, 2, 3),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(3, 2, 1)
)
```

Layer1은 [7 x 7 Conv] → BN → ReLU → MaxPool 구조이다. nn.Conv2d(3, 64, 7, 2, 3)으로 호출해 stride=2이므로 convolution layer를 통과하며 입력 크기가 절반으로 줄어드는데, 이때 CIFAR-10의 이미지 크기는 32 x 32, 채널 크기는 3이므로 convolution 결과 이미지 크기는 16 x 16으로 감소하며 채널 수는 3에서 64로 증가한다. 이 convolution 결과에 대해

차례로 배치 정규화와 ReLU 함수를 적용한 뒤 stride=2로 max pooling한다. Layer1의 결과 이미지와 채널 크기는 8 x 8 x 64이다.

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

Layer2는 ResidualBlock()을 세 번 호출함으로써 세 개의 residual block을 쌓는다. Layer1에서 8 x 8 x 64 데이터가 전달되어 첫 번째 rediual block은 ResidualBlock(64, 64, 256, False)으로 호출한다. 64개의 입력 채널이 동일한 크기의 중간 채널을 거쳐 256개로 증가하고, 이때 downsample=False로 하여 크기를 유지하므로 결과는 8 x 8 x 256이다. 이후 ResidualBlock(256, 64, 256, False)로 다운샘플링 없이 두 번째 residual block을 호출하며, 그 결과는 8 x 8 x 256이다. 마지막으로 ResidualBlock(256, 64, 256, True)를 호출해 세 번째 residual block을 쌓는데, 이때 stride=2로 다운샘플링하므로 최종 출력은 4 x 4 x 256이다.

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

Layer3에서는 ResidualBlock()을 네 번 호출해 네 개의 residual block을 쌓는다. Layer2에서 4 x 4 x 256 데이터가 전달되므로 첫 번째 residual block은 ResidualBlock(256, 128, 512, False)로 호출하고, 다운샘플링하지 않으므로 결과는 4 x 4 x 512이다. 이후 다운샘플링 없이 ResidualBlock(512, 128, 512, False), ResidualBlock(512, 128, 512, False)를 순차적으로 호출해 4 x 4 x 512 데이터를 출력하고, 마지막으로 다운샘플링한 ResidualBlock(512, 128, 512, True)을 호출함으로써 최종적으로 2 x 2 x 512 이미지를 출력한다.

```
self.layer4 = nn.Sequential(
            ResidualBlock(512, 256, 1024, False),
            ResidualBlock(1024, 256, 1024, False),
            ResidualBlock(1024, 256, 1024, False),
            ResidualBlock(1024, 256, 1024, False),
            ResidualBlock(1024, 256, 1024, False),
            ResidualBlock(1024, 256, 1024, False)
)
```

Layer4에서는 ResidualBlock()을 여섯 번 호출해 여섯 개의 residual block을 쌓는다. ResidualBlock(512, 256, 1024, False)을 호출하고, 이어서 ResidualBlock(1024, 256, 1024, False)을 다섯 번 호출하며 모든 경우 다운샘플링을 수행하지 않으므로 Layer4 최종 출력은 2 x 2 x 1024 크기이다.

```
self.fc = nn.Linear(1024, 10)
self.avgpool = nn.AvgPool2d(2, 2)
```

이후 순전파에서 사용할 FC layer와 AvgPool 레이어를 정의한다. Layer4 결과 추출된

feature map 채널 수가 1024이므로 입력 파라미터는 1024, 출력 파라미터는 10으로 설정해 10개 클래스를 가지는 CIFAR-10 분류에 사용할 수 있도록 한다. avgpool은 2 x 2 필터를 stride=2로 적용하게 설정하며, 따라서 2 x 2 x 1024인 입력에 대해 pooling 결과 1 x 1 x 1024가 출력된다.

```
for m in self.modules():
	if isinstance(m, nn.Linear):
		nn.init.xavier_uniform_(m.weight.data)
	elif isinstance(m, nn.Conv2d):
		nn.init.xavier_uniform_(m.weight.data)
```

이후 위와 같이 모든 모듈을 순회하며 가중치를 초기화한다. 모듈이 nn.Linear()이거나 nn.Conv2d이면 xavier uniform 초기화한다.

6 forward(self, x)

forward()에서는 입력 x를 위에서 정의한 Layer1~Layer4와 AvgPool, FC layer에 순차적으로 전달하여 최종 출력을 계산한다.

```
out = self.layer1(x)
out = self.layer2(out)
out = self.layer3(out)
out = self.layer4(out)
out = self.avgpool(out)
out = out.view(out.size()[0], -1)
out = self.fc(out)

return out
```

위와 같이 구현하며, layer1(), layer2(), layer3(), layer4(), avgpool(), view(), fc()를 차례대로 호출해 계산하는 것을 확인할 수 있다. 이후 최종 출력 out을 반환한다.

main.py

main에서는 CIRAF-10 데이터셋을 불러오고, 앞서 구현한 VGG16 또는 ResNet50 모델을 사용해 학습한 후 이를 평가한다.

1 전처리

```
transform_train = transforms.Compose([
    transforms.RandomCrop(32, padding=4),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)),
])

transform_test = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)),
])
```

위와 같이 학습용 데이터셋과 테스트용 데이터셋에 적용할 변환(transform)을 정의하며, 학습용 데이터에는 crop과 flip을 적용해 data augmentation을 수행한다.

```
# CIFAR-10 Dataset
train_dataset = torchvision.datasets.CIFAR10(root='../osproj/data/',
                                             train=True,
                                             transform=transform_train,
                                             download=True)

test_dataset = torchvision.datasets.CIFAR10(root='../osproj/data/',
                                            train=False,
                                            transform=transform_test)

# data loader
train_loader = torch.utils.data.DataLoader(dataset=train_dataset,
                                           batch_size=100,
                                           shuffle=True)
test_loader = torch.utils.data.DataLoader(dataset=test_dataset,
                                          batch_size=100,
                                          shuffle=False)
```

CIFAR-10 데이터셋을 불러온 뒤 앞서 정의한 transform_train과 transform_test에 따라 각각 전처리해 train_dataset과 test_dataset에 저장하고, 이후 DataLoader를 사용해 이들을 로드하여 train_loader와 test_loader에 저장한다. 이때 batch_size=100이므로 학습 시 미니배치 단위로 순회할 수 있다.

```
model = ResNet50_layer4().to(device)
PATH = 'resnet50_epoch285.ckpt'  # test acc would be almost 80

# model = vgg16().to(device)
# PATH = './vgg16_epoch250.ckpt'  # test acc would be almost 85
# checkpoint = torch.load(PATH)
checkpoint = torch.load(PATH, map_location=torch.device('cpu'))

model.load_state_dict(checkpoint)
```

앞서 구현한 ResNet50_layer4()와 vgg16() 중 사용할 모델을 선택해 model을 설정하며, torch.load()로 미리 저장한 체크포인트 파일을 불러오고 model.load_state_dict()로 가중치를 로드한다. 코드 상단에서 device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')로 설정하였으므로 개발 환경에 따라 GPU와 CPU를 모두 사용할 수 있다.

2 학습 및 평가

```
# Hyper-parameters
num_epochs = 1  # students should train 1 epoch because they will use cpu
learning_rate = 0.001

# Loss and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
```

학습 전에 하이퍼파라미터를 설정하고 훈련에 사용할 손실 함수와 optimizer를 지정한다. 빠른

실행을 위해 한 epoch만 수행하도록 num_epoch를 1로 설정하였으며, 학습률은 0.001로 한다. 손실 함수로는 cross-entropy 손실 함수를 사용하고 optimizer는 Adam이다.

```
for epoch in range(num_epochs):

    model.train()
    train_loss = 0

    for batch_index, (images, labels) in enumerate(train_loader):
        …
```

이후 epoch만큼 학습하며, 각 학습 루프 내에서 각 배치마다 순전파, 손실 계산, 역전파 과정을 거치며 가중치를 계산한다. 앞서 batch size=10, epoch=1로 설정했으므로 1 epoch동안 100번의 배치마다 수행한다. 배치마다 수행되는 자세한 내용은 다음과 같다.

```
for batch_index, (images, labels) in enumerate(train_loader):
        # print(images.shape)
        images = images.to(device)  # "images" = "inputs"
        labels = labels.to(device)  # "labels" = "targets"

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

        # Backward and optimize
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        train_loss += loss.item()

        if (batch_index + 1) % 100 == 0:
            print("Epoch [{}/{}], Step [{}/{}] Loss: {:.4f}"
                  .format(epoch + 1, num_epochs, batch_index + 1, total_step,
                          train_loss / (batch_index + 1)))
```

outputs=model(images)로 순전파를 수행하고 그 결과에 대해 criterion(outputs, labels)로 cross-entropy 손실 함수를 사용해 손실을 계산한 것을 loss에 저장한다. 이후 optimizer.zero_grad(), loss.backward(), optimizer.step()을 순서대로 호출해 가중치를 갱신하고 train_loss에 계산한 것을 추가한다. For문 내에서 100번의 배치마다 평균 손실을 출력하도록 해 학습 과정을 확인할 수 있게 한다.

```
# Decay learning rate
if (epoch + 1) % 20 == 0:
        current_lr /= 3
        update_lr(optimizer, current_lr)
        torch.save(model.state_dict(), './resnet50_epoch' + str(epoch+1)+'.ckpt')
```

위와 같이 학습률 감소(learning rate decay)를 구현하여 매 epoch마다 학습률을 1/3으로 줄이지만, 현재 epoch=1로 설정하였으므로 실행 시 실질적으로 반영되지 않는다.

```
torch.save(model.state_dict(), 'resnet50_final.ckpt')
```

```
model.eval()
with torch.no_grad():
    correct = 0     total = 0     for images, labels in test_loader:
        images = images.to(device)
        labels = labels.to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

    print('Accuracy of the model on the test images: {} %'.format(100 * correct /
      total))
```

학습이 완전히 종료되면 .ckpt 파일에 최종 모델 상태를 저장한 뒤 model.eval()하여 평가 상태로 전환하고 테스트 데이터를 사용해 모델의 정확도를 계산한다.

3 결과 분석

모든 경우 CPU를 사용해 CIFAR-10 데이터셋으로 학습 및 테스트하며, cross-entropy 손실 함수와 Adam optimizer를 사용하고 학습률은 0.001로 설정해 1 epoch동안 학습한다..

3.1 VGG16

```
model = vgg16().to(device)
PATH = './vgg16_epoch250.ckpt'
```

으로 하여 실행한 전체 결과는 다음과 같다.

```
C:₩Users₩rdh08₩AppData₩Local₩Programs₩Python₩Python311₩python.exe
C:₩Users₩rdh08₩PycharmProjects₩openSW_project₩assignment10₩main.py

Epoch [1/1], Step [100/500] Loss: 0.1807
Epoch [1/1], Step [200/500] Loss: 0.1736
Epoch [1/1], Step [300/500] Loss: 0.1813
Epoch [1/1], Step [400/500] Loss: 0.1877
Epoch [1/1], Step [500/500] Loss: 0.1883

Accuracy of the model on the test images: 86.16 %


종료 코드 0(으)로 완료된 프로세스
```

실행 결과 테스트 정확도가 86.16% 로 과제 조건인 85% 이상을 만족하는 것을 확인할 수 있다.

3.2 ResNet50

```
model = ResNet50_layer4().to(device)
PATH = 'resnet50_epoch285.ckpt'
```

으로 하여 실행한 전체 결과는 다음과 같다.

```
C:₩Users₩rdh08₩AppData₩Local₩Programs₩Python₩Python311₩python
```

```
C:₩Users₩rdh08₩PycharmProjects₩openSW project₩assignment10₩main.py

Epoch [1/1], Step [100/500] Loss: 0.2752
Epoch [1/1], Step [200/500] Loss: 0.2846
Epoch [1/1], Step [300/500] Loss: 0.2972
Epoch [1/1], Step [400/500] Loss: 0.2952
Epoch [1/1], Step [500/500] Loss: 0.3008

Accuracy of the model on the test images: 82.96 %


종료 코드 0(으)로 완료된 프로세스
```

실행 결과 테스트 정확도가 82.96% 로 과제 조건인 80% 이상을 만족하는 것을 확인할 수 있다.