Open Software Project (01)
컴퓨터공학과 1771046 이혜진

Technical Report

Assignment09 : CNN Architecture

1. main.py

(1) Code analysis

```
(PyTorch_env) C:₩WINDOWS₩system32>conda install pytorch torchvision cpuonly -c pytorch
Collecting package metadata (current_repodata.json): done
Solving environment: done
```

처음엔 pip install torch 명령어를 사용하니 제대로 해당 라이브러리를 받을 수 없었다. torch 홈페이지에 들어가 명령어를 확인한 후 주어진 명령어대로 해당 라이브러리를 설치하니, 설치가 성공적으로 완료되었다. 이후 pip install torchvision을 입력하였더니, 이 라이브러리도 정상적으로 설치가 완료되었다.

```
# CIFAR-10 Dataset
train_dataset = torchvision.datasets.CIFAR10(root='../osproj/data/',
                                             train=True,
                                             transform=transform_train,
                                             download=False) # Change Download-flag "True" at the first excution.

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

먼저 앞으로의 CNN 과정에서 사용할 데이터 셋을 다운받는다. download 변수를 True로 세팅한 후, 진행하면 해당 디렉토리에 데이터가 저장된다. 이 데이터 셋은 10개의 클래스로 분류되어 있는 데이터들을 담고 있다. 학습을 위한 training data와 테스트를 위한 test data를 각각 할당한다.

```
# Choose model
# 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
```

모델을 선택하는 부분이다. 먼저 VGG 모델을 이용하기 위해 위의 ResNet과 관련된 코드를 주석처리 하고, 아래의 두 줄만 남겨놓았다.

```
#checkpoint = torch.load(PATH)
checkpoint = torch.load(PATH, map_location=lambda storage, loc: storage)
model.load_state_dict(checkpoint)
```

모델을 읽어오는 함수이다. 하지만 기존에 skeleton code에서 제공되는 함수에서 알 수 없는 오류가 발생하였고 이에 대한 문제를 해결하는 과정에 아래의 사이트들이 많은 도움이 되었다.

https://jangjy.tistory.com/319
https://discuss.pytorch.org/t/on-a-cpu-device-how-to-load-checkpoint-saved-on-gpu-device/349/2

```
# Save the model checkpoint
torch.save(model.state_dict(), './resnet50_final.ckpt')
```

파일 외의 공간에서 선언된 함수들을 불러와 처리를 마친 후 해당 모델을 체크 포인트에 저장한다.

```
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))
```

최종 모델에서 테스트 데이터에 대한 정확도를 측정해 출력한다. 현재 main 함수에서 실제 작성할 코드는 없었다.

2. vgg16_full.py

(1) Code analysis

```
def vgg16():
    # cfg shows 'kernel size'
    # 'M' means 'max pooling'
    cfg = [64, 64, 'M', 128, 128, 'M', 256, 256, 256, 'M', 512, 512, 512, 'M', 512, 512, 512, 'M']
    return VGG(make_layers(cfg))
```

현재 해당 파일에서는 실제 코드를 짤 부분은 없다. 따라서 간단한 설명 및 결과만 덧붙인다. 커널 사이즈와 max pooling 장소를 정해 VGG 함수에 이러한 정보들을 전달한다.

```
class VGG(nn.Module):
    def __init__(self, features):
        super(VGG, self).__init__()
        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),
        )
```

위에서 이용한 VGG 함수는 크게 이러한 구조로 구성되어 있으며, 이후 weight(W)를 시작해 계산을 이어나간다.

이후의 중간과정은 생략한다.

(2) Result analysis

```
C:\ProgramData\Anaconda3\envs\PyTorch_env\python.exe C:/Users/HEAJIN/PycharmProjects/Lec13_Practice/main.py
Epoch [1/1], Step [100/500] Loss: 0.1745
Epoch [1/1], Step [200/500] Loss: 0.1895
Epoch [1/1], Step [300/500] Loss: 0.1907
Epoch [1/1], Step [400/500] Loss: 0.1921
Epoch [1/1], Step [500/500] Loss: 0.1910
Accuracy of the model on the test images: 86.03 %
```

해당 모델을 이용해 순차적으로 학습을 완료하였을 경우 얻게 되는 결과이다. 총 500번의 스탭을 거쳐 86.03%의 정확도를 얻었으며 이는 요구사항인 85%를 조금 넘는 정확도이다. 따라서 test image를 통해 추정한 결과를 통해, training image가 올바르게 학습되었음을 의미한다고 할 수 있다.

또한 Loss를 살펴보면 거의 증가하는 추세임을 알 수 있다. 현재 CPU를 이용해 학습을 진행하였는데 많은 계산 과정이 들어 있어서 그런지 학습에 꽤 많은 시간이 소요되었다.

3. resnet50_skeleton.py

(1) Code analysis

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

먼저 main 함수에서 ResNet50 모델을 이용하기 위해 특정 부분을 주석처리하고, 코드를 변경한다.

```
# 1x1 convolution
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
```

1x1 컨볼루션 이기 때문에 커널 사이즈는 1로 설정한다. 마찬가지로 3x3 컨볼루션에서는 커널 사이즈를 3으로만 변경하면 된다. 컨볼루션을 진행하고 batch normalization을 진행하고, ReLU를 진행한다. 이는 모두 한 개의 conv 함수 내에서 진행된다.

```
##########################################################################
# Question 1 : Implement the "bottle neck building block" part.
# Hint : Think about difference between downsample True and False. How we make the difference by code?
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(
                ########################################
                ############### fill in here (20 points)
                # Hint : use these functions (conv1x1, conv3x3)
                ########################################
                conv1x1(in_channels, middle_channels, 2, 0),
                conv3x3(middle_channels, middle_channels, 1, 1),
                conv1x1(middle_channels, out_channels, 1, 0)
            )
            self.downsize = conv1x1(in_channels, out_channels, 2, 0)
```

CNN practice 27p의 내용을 기반으로 해당 코드를 작성하였다. 이 부분에서는 bottle neck building block 부분의 함수에 대하여 작성하였다. 컨볼루션

을 총 3회 진행하는데, stride와 padding 사이즈를 알맞게 conv 함수의 인자로 전달한다. 결과적으로 커널의 사이즈를 줄일 수 있다.

```
else:
    self.layer = nn.Sequential(
        ##########################################
        ############## fill in here (20 points)
        ##########################################
        conv1x1(in_channels, middle_channels, 1, 0),
        conv3x3(middle_channels, middle_channels, 1, 1),
        conv1x1(middle_channels, out_channels, 1, 0)
    )
    self.make_equal_channel = conv1x1(in_channels, out_channels, 1, 0)
```

유사하게 아래의 코드를 작성할 수 있다. 커널의 채널 수를 그대로 이용한다.

```
##############################################################################
# Question 2 : Implement the "class, ResNet50_layer4" part.
# Understand ResNet architecture and fill in the blanks below. (25 points)
# (blank : #blank#, 1 points per blank )
# Implement the code.
class ResNet50_layer4(nn.Module):
    def __init__(self, num_classes= 10 ): # Hint : How many classes in Cifar-10 dataset?
        super(ResNet50_layer4, self).__init__()
        self.layer1 = nn.Sequential(
            nn.Conv2d(in_channels=3, out_channels=64, kernel_size=7, stride=2, padding=3),
                # Hint : Through this conv-layer, the input image size is halved.
                #        Consider stride, kernel size, padding and input & output channel sizes.
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2, padding=0)
        )
```

CNN practice 28p의 내용을 기반으로 해당 빈칸을 채울 수 있다. 현재 데이터 셋이 10개의 클래스로 구분하고 있는 데이터이기 때문에 num_classes는 10이 된다. 아래에도 각각 올바른 변수를 집어넣는다. 총 4개의 layer 중에서, 첫 번째 layer에서 일어나는 일을 담고 있다.

Layer 1	7x7 conv, channel = 64, stride = 2 3x3 max pool, stride = 2	8 x 8

```
self.layer2 = nn.Sequential(
    # in_channels, middle_channels, out_channels, downsample=False
    ResidualBlock(in_channels=64, middle_channels=64, out_channels=256, downsample=False),
    ResidualBlock(in_channels=256, middle_channels=64, out_channels=256, downsample=False),
    ResidualBlock(in_channels=256, middle_channels=64, out_channels=256, downsample=True)
)
```

2번 째 layer에서 일어나는 일들을 담고 있다. 2+1=3이므로 총 3번의 과정이 실행되어야 함을 알 수 있다.

Layer 2	[1x1 conv, channel = 64, 3x3 conv, channel = 64, 1x1 conv, channel = 256] x 2 [1x1 conv, channel = 64, stride = 2 3x3 conv, channel = 64, 1x1 conv, channel = 256] x 1	4 x 4

```
self.layer3 = nn.Sequential(
    ##########################################
    ############## fill in here (20 points)
    ####### you can refer to the 'layer2' above
    ##########################################
    ResidualBlock(in_channels=256, middle_channels=128, out_channels=512, downsample=False),
    ResidualBlock(in_channels=512, middle_channels=128, out_channels=512, downsample=False),
    ResidualBlock(in_channels=512, middle_channels=128, out_channels=512, downsample=False),
    ResidualBlock(in_channels=512, middle_channels=128, out_channels=512, downsample=True)
)
```

3번 째 layer에서 일어나는 일들을 담고 있다.

Layer 3	[1x1 conv, channel = 128, 3x3 conv, channel = 128, 1x1 conv, channel = 512] x 3 [1x1 conv, channel = 128, stride = 2 3x3 conv, channel = 128, 1x1 conv, channel = 512] x 1	2 x 2

```
self.layer4 = nn.Sequential(
    ########################################
    ############# fill in here (20 points)
    ####### you can refer to the 'layer2' above
    ########################################
    ResidualBlock(in_channels=512, middle_channels=256, out_channels=1024, downsample=False),
    ResidualBlock(in_channels=1024, middle_channels=256, out_channels=1024, downsample=False),
    ResidualBlock(in_channels=1024, middle_channels=256, out_channels=1024, downsample=False),
    ResidualBlock(in_channels=1024, middle_channels=256, out_channels=1024, downsample=False),
    ResidualBlock(in_channels=1024, middle_channels=256, out_channels=1024, downsample=False),
    ResidualBlock(in_channels=1024, middle_channels=256, out_channels=1024, downsample=False)
)
```

4번 째 layer에서 일어나는 일들을 담고 있다.

Layer 4	[1x1 conv, channel = 256, 3x3 conv, channel = 256, 1x1 conv, channel = 1024] x 6	2 x 2

```
self.fc = nn.Linear(1024, 10) # Hint : Think about the reason why fc layer is needed
self.avgpool = nn.AvgPool2d(2, stride=1)
```

4번째 layer까지 처리를 마치면 fully connected를 적용시키고, AvgPool을 적용시킨다. 해당 pdf에는 AvgPool이 우선적으로 적용되어 있었으나, skeleton code에는 fully connected가 우선적으로 작성되어 있어 그대로 사용하였다.

	AvgPool	1 x 1
	Fully connected layer	?

```
def forward(self, x):

    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
```

작성한 코드를 바탕으로 layer들을 학습시켜 나간다.

(2) Result analysis

```
C:\ProgramData\Anaconda3\envs\PyTorch_env\python.exe C:/Users/HEAJIN/PycharmProjects/Lec13_Practice/main.py
Epoch [1/1], Step [100/500] Loss: 0.3651
Epoch [1/1], Step [200/500] Loss: 0.3496
Epoch [1/1], Step [300/500] Loss: 0.3479
Epoch [1/1], Step [400/500] Loss: 0.4090
Epoch [1/1], Step [500/500] Loss: 0.4092
Accuracy of the model on the test images: 81.15 %
```

해당 모델을 이용해 순차적으로 학습을 완료하였을 경우 얻게 되는 결과이다. 총 500번의 스탭을 거쳐 81.15%의 정확도를 얻었으며 이는 요구사항인 80%를 조금 넘는 정확도이다. 따라서 test image를 통해 추정한 결과를 통해, training image가 올바르게 학습되었음을 의미한다고 할 수 있다.