|  |  |
| --- | --- |
| 학번 / 이름 | 2171262 박소희 |
| 전공 | IT융합공학부 사물인터넷트랙 |
| 주제 | homework #3 보고서 |

* 코드 설명

Cifar10이라는 오픈소스 데이터 셋을 불러와서 훈련 및 검증해보았다.

훈련시에는 50000장, test시에는 10000장의 영상(이미지)을 사용하였다.

이미지의 shape은 32x32x3이며 batch사이즈는 32로 설정하였다.

**[cnn\_cifar10.py]**

* 필요한 Library 및 모듈 import

import torch  
import torch.nn as nn  
import torchvision *#영상 관련 lib import*import torchvision.transforms as transforms *#데이터 증강을 위한 lib*import torch.optim as optim  
import os *#폴더 구성을 위한 lib*

* 정확도를 측정하는 함수

*#얼마나 정확하게 훈련되었는지 측정하는 함수*def accuracy(output, target, topk=(1, )):

*#tensor 메소드: 위~1등까지 순위를 매긴다* with torch.no\_grad():  
 maxk = max(topk)  
 batch\_size = target.size(0)  
  
 \_, pred = output.topk(maxk, 1, True, True)  
 pred = pred.t()  
 correct = pred.eq(target.view(1, -1).expand\_as(pred))  
  
 acc = []  
 num\_cor = []  
 for k in topk:  
 correct\_k = correct[:k].view(-1).float().sum(0, keepdim=True)  
 num\_cor.append(correct\_k.clone())  
 acc.append(correct\_k.mul(1/batch\_size))  
 return acc, num\_cor

정확도를 측정하는 accuracy 메소드를 따로 구현하여 모델로 예측한 엉뚱한 값과 실제 예측값 간의 정확도를 줄을 세워서 판단한다. 정확도와 정답의 수를 반환한다.

* 가중치를 초기화하는 함수

*#가중치를 초기화하는 함수*def init\_weights(m, init\_type='normal', init\_gain=0.02):

*# define the initialization function* classname = m.\_\_class\_\_.\_\_name\_\_  
 if hasattr(m, 'weight') and (classname.find('Conv') != -1 or classname.find('Linear') != -1):  
 if init\_type == 'normal': *#normal분포로 weight값을 초기화할게* nn.init.normal\_(m.weight.data, 0.0, init\_gain)  
 elif init\_type == 'xavier': *#xavier 성질로 weight값을 초기화할게* nn.init.xavier\_normal\_(m.weight.data, gain=init\_gain)  
 elif init\_type == 'kaiming':  
 nn.init.kaiming\_normal\_(m.weight.data, a=0, mode='fan\_in')  
 elif init\_type == 'orthogonal':

*#직교성질을 이용해서 weight값을 초기화할게* nn.init.orthogonal\_(m.weight.data, gain=init\_gain)  
 else:  
 raise NotImplementedError('initialization method [%s] is not implemented' % init\_type)  
 if hasattr(m, 'bias') and m.bias is not None:  
 nn.init.constant\_(m.bias.data, 0.0) *#bias값을 0으로 초기화할게* elif classname.find('BatchNorm2d') != -1: nn.init.normal\_(m.weight.data, 1.0, init\_gain)  
 nn.init.constant\_(m.bias.data, 0.0)

가중치를 초기화하는 함수로써 기본 속성은 normal값으로 가중치를 초기화하는 방법이다. 그 외에도 xavier, 직교 성질 등이 있고 속성값을 비교하여 bias값도 초기화해준다. 리턴값은 없다.

* Custom Layer 클래스 정의부 및 구현부

*#CNN Custom Layer 클래스 정의 및 구현부*class Network(nn.Module):  
 def \_\_init\_\_(self): *#생성자의 역할* super(Network, self).\_\_init\_\_() *#부모 메소드 상속 받고* self.conv1 = nn.Conv2d(in\_channels=3, out\_channels=32, kernel\_size=(7, 7), stride=(1, 1))  
 self.conv2 = nn.Conv2d(32, 64, (3,3), (1,1))

*#입력 채널 32, 커널 갯수 64, 커널 사이즈 3x3* self.conv3 = nn.Conv2d(64, 128, 3, 1)

*#입력 채널 64, 커널 갯수 128, 커널 사이즈 3x3* self.fc\_code = nn.Linear(in\_features=128, out\_features=128)

*#입력 특징 128, 출력도 128* self.fc\_output = nn.Linear(128, 10)

*#입력 128, 출력 10 -> 총 10개의 클래스로 분류* self.relu = nn.ReLU(inplace=True)

*#reLu activation 메소드를 이용하여 값자체를 바꿀게* self.maxpool = nn.MaxPool2d(2)

*#maxPool 방식을 이용하여 feature map의 크기를 줄임* self.dropout = nn.Dropout(p=0.5)

*#dropout응 이용하여 overfitting 방지* self.apply(init\_weights)

*#가중치 초기화 -> normal 가우시안 분포 구성* def forward(self, x): *# layer의 순서 부분* feature\_map = self.conv1(x) *#convolution 연산을 한다* activated = self.relu(feature\_map) *#활성화 함수를 씌운다* compressed = self.maxpool(activated)

*#pooling을 통해 feature map의 크기를 감소* x = self.maxpool(self.relu(self.conv2(compressed)))

*#한번에 연산한것* x = self.maxpool(self.relu(self.conv3(x)))

*#conv3을 통해 한번에 연산한 것* x = x.view(x.size(0), -1)

*#view로 사이즈를 바꾸어서 일자로 쭉 핀다* code = self.dropout(self.relu(self.fc\_code(x)))

*#dropout 계산하기* output= self.fc\_output(code)

*#128개 -> 10개 확률* return output, code

*#확률과 128개의 특징값 추출*

CustomLayer Class에서는 위와 같이 \_\_init\_\_메소드에서 부모 클래스를 상속받고 필요한 메서드와 Convolution연산을 모두 정의해준다. Forward()오버라이딩 함수를 정의하여 각 메서드의 순서를 정의해주고 튀어 나온 반환값(feature 128개, 확률)을 리턴한다.

* 메인함수 정의(디바이스 판단, 데이터 로더 선언, 정규화 등)

if \_\_name\_\_ == '\_\_main\_\_': *#메인함수* if not os.path.isdir("./ckpt\_0"): *#폴더가 없으면* os.makedirs("./ckpt\_0") *#해당 폴더를 만들어 주세요.* device = torch.device("cuda" if torch.cuda.is\_available() else "cpu") print(device)  
  
 transform\_train = transforms.Compose([transforms.RandomHorizontalFlip(), transforms.ToTensor(), transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2470, 0.2435, 0.2616))])  
 transform\_test = transforms.Compose([transforms.RandomHorizontalFlip(), transforms.ToTensor(), transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2470, 0.2435, 0.2616))])  
 trainset = torchvision.datasets.CIFAR10(root='./data', train=True, download=True, transform=transform\_train) *#데이터 셋 정의*   
 trainloader = torch.utils.data.DataLoader(trainset, batch\_size=32, shuffle=True, num\_workers=2) *#배치 사이즈 32로 해서 섞는다*  
 testset = torchvision.datasets.CIFAR10(root='./data', train=False, download=True, transform=transform\_test)  
 testloader = torch.utils.data.DataLoader(testset, batch\_size=32, shuffle=False, num\_workers=2) *#testLoader는 섞지 않는다.*  
  
 model = Network() *#customNetwork 초기화 및 선언* model.to(device) *#연산장치 위에 올리기* params = model.parameters() *#나의 모델의 파라미터 받기* optimizer = optim.Adam(params, lr=1e-3, betas=(0.9, 0.999), weight\_decay=0.001) *#Adam 활성화 함수* train\_criterion = nn.CrossEntropyLoss(reduction='mean')

*#평균값으로 계산할게요.*

- 메인함수에서는 연산할 디바이스 장치를 정하고 test, train을 위한 데이터 셋과 데이터 로더를 초기화해주었다. 그리고 앞서 정의한 NetWork()의 객체를 선언해주고 optimizer를 Adam을 통해 계산하겠다고 선언해주었다. Loss값은 nn 모듈의 CrossEntropy의 평균값들을 이용하여 계산할 것이다.

* 실제 학습하기

for epoch in range(1, 100): *#1~99까지 반복* model.train() *#dropout 때문에 반드시 명시해주어야 함* for step, (data, targets) in enumerate(trainloader): data = data.to(device, dtype=torch.float)

*#계산할때는 cpu나 gpu위에 올린다* targets = targets.to(device) *#target도 역시 올린다* optimizer.zero\_grad() *#optimizer 초기화* outputs, code = model(data) *#특징값과 확률을 뽑아서* loss = nn.CrossEntropyLoss(reduction='mean')(outputs, targets) *#손실값을 구한다.(output은 모델로 예측한 엉뚱한 값, targets은 원래 정답값)* loss.backward() *#gradient 계산* optimizer.step() *#w값 갱신* loss = loss.item() *#손실값 할당* acc, \_ = accuracy(outputs, targets)

*#ouput과 정답값 사이의 정확도 계산* acc = acc[0].item() *#정확도 할당* if step % 10 == 0: *#10번 돌 때마다* print('Epoch {} Step {}/{} Loss {:.4f} Accuracy {:.4f}'.format(epoch, step, len(trainloader), loss, acc))

실제 training은 지금까지 했던 방법인 이중 for문을 돌며 epoch 수만큼 반복하고, dataloader만큼 반복하도록 구성하였다. 학습하는 과정이므로 꼭 optimizer를 초기화해주어야 하고 gradient 값을 계산, w와 b값을 갱신하는 것은 동일하다. 10번 돌때마다 현재 loss값과 정확도를 출력해준다.

* 학습이 잘 되었는지 test하기

model.eval() *#여기서부터 test함*total\_cor = 0  
total\_samples = 0  
  
with torch.no\_grad(): *#gradient 적용X* for step, (data, targets) in enumerate(testloader):  
 data = data.to(device, dtype=torch.float)  
 targets = targets.to(device)  
 outputs, code = model(data) *#확률값과 feature값* \_, num\_cor = accuracy(outputs, targets) *#정확도 측정* num\_cor = num\_cor[0].item()  
 total\_samples += data.size(0)  
 total\_cor += num\_cor  
 acc = total\_cor / total\_samples  
 print('Epoch {} : Accuracy {:.4f}'.format(epoch, acc))

*#에폭당 정확도 출력*path = 'ckpt\_0/model\_state\_%d.st'%(epoch) *#모델의 가중치 w값 저장*torch.save(model.state\_dict(), path) *#해당 경로로 저장*

학습이 다 끝나면 모델을 가지고 예측을 하는 단계이다. 이 단계에서는 gradient 값을 적용하지 않고 dropout의 특징 때문에 model.eval()을 반드시 선언한다. 모델을 통해 리턴된 output값과 원래 target값을 비교하여 정확도를 계산하여 출력한다. 그리고 다음을 위하여 모델의 가중치 값을 앞서 만든 폴더의 파일로써 저장을 한다.

**[cnn\_cifar10\_checkmodel.py]**

코드가 동일한 부분 및 메서드는 재설명하지 않았습니다.

- 필요한 모듈 및 라이브러리 import

import torch  
import torch.nn as nn *#nn모듈 import*import torchvision  
import torchvision.transforms as transforms *#데이터 증강을 위한 lib*import numpy as np  
import matplotlib.pyplot as plt *#그림 그리는 lib*

* 그림 및 그래프 그리는 plot\_tensor 메서드 정의

*#그래프나 그림 그리는 함수*def plot\_tensor(tensor, mode=1, num\_col=None, label=None):  
 if mode == 1: *# 모드가 1이라면* tensor = tensor.detach()

*# 그림 그릴때는 텐서에서 분리한다* tensor = tensor.permute(0, 2, 3, 1)

*# (32, 3, 7, 7) -> (32, 7, 7, 3) 차원을 아예 바꾼다* npArr = tensor.cpu().numpy()

*# 텐서를 cpu에서 numpy로 변환* num\_row = int(npArr.shape[0] / num\_col) *# 행의 개수가 몇 개인지? shape[0] : 전체크기 / col(행의 크기)* fig, ax = plt.subplots(num\_row, num\_col)  
 idx = 0 *# idx 0 초기화* for r in range(num\_row): *# num\_row만큼 반복* for c in range(num\_col):  
 ax[r, c].imshow(npArr[idx,]) *# 이미지 보여주기* ax[r, c].set\_xticks([]) *# x축 설정하기* ax[r, c].set\_yticks([]) *# y축 설정하기* if label is not None:  
 ax[r, c].set\_title('label: {}\npred: {}'.  
 format(label[0][idx], label[1][idx]), fontsize=5) *#라벨, pred 제목 설정하기* idx+=1 *#반복문 돌때마다 인덱스 증가* plt.tight\_layout()  
 plt.show()  
 if mode == 2: *#모드가 2라면* tensor = tensor.detach().cpu() *# detach from tensor graph* img\_grid = torchvision.utils.make\_grid(tensor, nrow=num\_col, padding=2, pad\_value=1, normalize=True)  
 plt.imshow(img\_grid.permute((1, 2, 0)))

*#(w,h,c) 순서로 재배치하여 이미지 보이기* plt.axis('off') *#축은 설정하지 않을게요* plt.tight\_layout()  
 plt.show()

Plt를 이용하여 그림 그리는 메서드에서는 모드를 1과 2로 나누어 각 상황에 맞게 사용할 수 있도록 하였다. 모드 1에서는 4차원 텐서를 라벨과 함께 보여줄 때 사용하고 모드2는 3차원텐서를 축 없이 보여주고자 할 때 이용한다. plt라이브러를 이용하여 입력된 텐서의 사이즈를 이용하여 가로길이, 세로길이를 구하여 각 그림을 배치하였다.

* 메인함수 구성

if \_\_name\_\_ == '\_\_main\_\_': *#메인함수* device = torch.device("cuda" if torch.cuda.is\_available() else "cpu") *#디바이스 정하기* print(device) *#연산장치 프린트하기* transform\_train = transforms.Compose([transforms.RandomHorizontalFlip(), transforms.ToTensor(),  
 transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2470, 0.2435, 0.2616))])  
 transform\_test = transforms.Compose([transforms.RandomHorizontalFlip(), transforms.ToTensor(),  
 transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2470, 0.2435, 0.2616))])  
 trainset = torchvision.datasets.CIFAR10(root='./data', train=True, download=True, transform=transform\_train)  
 trainloader = torch.utils.data.DataLoader(trainset, batch\_size=32, shuffle=True, num\_workers=2)  
 testset = torchvision.datasets.CIFAR10(root='./data', train=False, download=True, transform=transform\_test)  
 testloader = torch.utils.data.DataLoader(testset, batch\_size=32, shuffle=False, num\_workers=2) *#앞 부분과 동일*   
  
 model = Network() *#CustomLayer 객체 생성 및 선언* model.to(device) *#model를 디바이스 위에 올린다* state\_dict = torch.load('./ckpt\_0/model\_state\_61.st')

*#61번째 모델을 로드할게요* model.load\_state\_dict(state\_dict, strict=True) *#model에 저장하기*

메인함수에서는 위와 같이 디바이스를 정하고 데이터를 증강하는 transform의 정규화 속성을 정의해주었다. 또한 앞서 정의했던 CustomLayer의 객체를 선언하고 이전과 다른 점은 모델을 저장하는 것이 아닌 저장한 모델을 다시 로드 해주었다는 점이다. 그리고 그 값을 모델의 상태 딕셔너리에 저장해주었다.

* 첫번째 커널 가시화 및 Test셋에 대한 추론 진행

weight1\_tensor = model.conv1.weight

*#(32, 3, 7, 7), 첫번째 커널(필터) 가중치값*plot\_tensor(weight1\_tensor, mode=2, num\_col=4)

*#plot\_tensor를 호출함으로써 가시화 #(32, 3, 7, 7)*classes = ('plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck') *#10개의 클래스 튜플 정의*model.eval() *#모델을 Test용도로 쓸게요*with torch.no\_grad(): *#gradient 적용 X* for step, (data, targets) in enumerate(testloader):

*#testloader만큼 반복하면서* data = data.to(device, dtype=torch.float) output, \_ = model(data) *#model(data)의 리턴값을 저장* \_, pred = output.topk(1, 1, True, True) *#topk로 줄 세우기* target\_cls = [classes[x] for x in targets.cpu().numpy()]

*#라벨을 위해 클래스 이름 값 할당* pred\_cls = [classes[x] for x in pred.cpu().squeeze().numpy()] *#예측값 라벨* print(target\_cls, pred\_cls, sep='\n')  
 plot\_tensor(data, mode=1, num\_col=4, label=[target\_cls, pred\_cls]) *#plot\_tensor호출 매개변수 전달* break

마지막으로 궁극적 목표였던 첫번째 커널을 plot\_tensor메서드를 호출하면서 가시화한다. 첫번째 커널을 가시화할 수 있는 이유는 (32, 3, 7, 7)의 shape를 갖기 때문이다. 2번째 커널은 (64, 32, 3, 3)의 모양을 가져서 가시화하는 것이 쉽지 않다. 첫번째 커널을 가시화한 결과는 아래의 모양을 가진다. 또한 테스트셋을 plot\_tensor를 통해 예측값과 정답값간의 차이를 그림으로써 보여주고 있다. 클래스 이름부분은 튜플 형식으로 10개의 이름을 선언해주었다.

스크린샷, 텍스트, 소프트웨어, 멀티미디어 소프트웨어이(가) 표시된 사진

자동 생성된 설명

**[첫번째 conv1 layer을 가시화한 결과]**

**[테스트 셋을 통한 추론 결과를 나타낸 모습]**

스크린샷, 콜라주, 텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트, 스크린샷, 디자인이(가) 표시된 사진

자동 생성된 설명

텍스트, 폰트, 스크린샷, 라인이(가) 표시된 사진

자동 생성된 설명

텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

자동 생성된 설명 <- cifar10.py 99까지 돌며 학습시킨 모습