<패턴인식 프로젝트 보고서>

융합전자공학과

황귀훈

201710885

**<프로젝트>**

수업시간에 배운 CNN MNIST의 모델을 변형하여 손 글씨 소문자 alphabet을 분류하는 것

**<Data set>**

데이터는 패턴인식 수업을 듣는 수강생들이 직접 데이터를 만들어 공유한 것을 사용

28x28 size, gray image, png형식으로 통일하여 알파벳 마다 50개씩 그려서 총 30000장 정도의 공유 데이터 셋을 모을 수 있었다.

Train : test = 5 : 1 로 나누어서 훈련 시켰고, 데이터 셋에 1채널이 아닌 데이터가 섞여 있어서 데이터를 불러올 때 GrayScale(1)함수를 사용했다.

다음은 공유데이터 셋의 정보이다. 여러 명의 데이터를 합쳐 잘못 만들어진 데이터를 제거하여 27000장 정도 수집되었다.

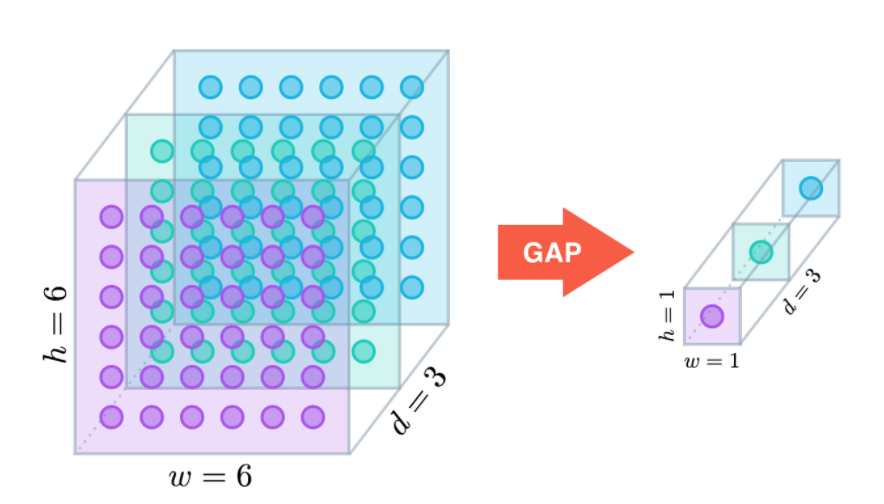
텍스트이(가) 표시된 사진

자동 생성된 설명

**<Model>**

class CNN(torch.nn.Module):  
 def \_\_init\_\_(self):  
 super(CNN, self).\_\_init\_\_()  
 self.layer1 = self.conv1(1, 128) # (\_, 14, 14, 128)  
 self.layer2 = self.conv2(128, 256) # (\_, 7, 7, 256)  
 self.layer3 = self.conv3(256, 512) # (\_, 4, 4, 512)  
 self.gap = self.global\_avg\_pool(512, 26) # (\_, 1, 1, 26)  
  
 def forward(self, x):  
 out = self.layer1(x)  
 out = self.layer2(out)  
 out = self.layer3(out)  
 out = self.gap(out)  
 out = out.view(-1, 26)  
 return out  
  
 def conv1(self, in\_ch, out\_ch):  
 return nn.Sequential(  
 nn.Conv2d(in\_ch, out\_ch, kernel\_size=5, padding=2, bias=False),  
 nn.BatchNorm2d(out\_ch),  
 nn.ReLU(inplace=True),  
 nn.MaxPool2d(kernel\_size=2, stride=2)  
 )  
  
 def conv2(self, in\_ch, out\_ch):  
 return nn.Sequential(  
 nn.Conv2d(in\_ch, out\_ch, kernel\_size=5, padding=2, bias=False),  
 nn.BatchNorm2d(out\_ch),  
 nn.ReLU(inplace=True),  
 nn.Dropout2d(p=0.4),  
 nn.MaxPool2d(kernel\_size=2, stride=2)  
 )  
  
 def conv3(self, in\_ch, out\_ch):  
 return nn.Sequential(  
 nn.Conv2d(in\_ch, out\_ch, kernel\_size=5, padding=2, bias=False),  
 nn.BatchNorm2d(out\_ch),  
 nn.ReLU(inplace=True),  
 nn.Dropout2d(p=0.4),  
 nn.MaxPool2d(kernel\_size=2, stride=2, padding=1)  
 )  
  
 def global\_avg\_pool(self, in\_ch, out\_ch):  
 return nn.Sequential(  
 nn.Conv2d(in\_ch, out\_ch, kernel\_size=5, padding=2, bias=False),  
 nn.BatchNorm2d(out\_ch),  
 nn.ReLU(inplace=True),  
 nn.Dropout(p=0.4),  
 nn.AdaptiveAvgPool2d(1)  
 )

훈련에 사용한 모델은 다음과 같다. 구글의 많은 논문을 찾아보던 중 마지막 layer에 fully connected layer을 사용하지 않고 Global Average Pooling을 사용할 경우 성능을 향상시킬 수 있다는 것을 보고 적용해 본 결과 Accuracy가 0.9934에서 0.9957로 상승했고 결과적으로 두번의 fc layer를 한번의 과정으로 줄일 수 있어서 layer수도 줄이는데 도움이 되었다.



Global Average Pooling은 위 그림처럼 작동한다. 마지막 Gap layer는 각 Feature Map상의 노드 값들의 평균을 뽑아낸다. layer집합을 input으로 하여 벡터를 output한다.

첫 layer에서 채널 수를 1에서 32나 64로 올리지 않고 바로 128로 올려 성능을 더 향상시킬 수 있었다. 0.9957 -> 0.9973

각 layer의 convolution의 kernel size를 3에서 5로 올렸을 때도 성능을 향상시킬 수 있었다. 0.9973 - >0.9980 더욱 성능을 향상시키기 위해 Batch normalization과 Dropout을 사용하였고 여러 번의 fine tuning으로 성능을 향상시켰다.

Batch normalization은 각 레이어마다 정규화 하는 레이어를 두어, 변형된 분포가 나오지 않도록 조절하게 하는 것이다.

Dropout은 확률 p로 은닉 유닛들을 p의 확률로 제거하는 것이다.

모든 layer의 activate function은 ReLU를 사용했다. 다른 activate function들도 사용해 봤지만 크게 성능향상이 되지 않아 ReLU를 사용했다.

ReLU는 양의 값에서는 Saturated되지 않고 계산 효율이 뛰어나다는 장점이 있다.

Layer를 거치며 data의 변화는 다음과 같다.

Layer1 ImgIn shape=(?, 28, 28, 1), Conv -> (?, 28, 28, 128), Pool -> (?, 14, 14, 128)

Layer2 ImgIn shape=(?, 14, 14, 128), Conv -> (?, 14, 14, 256), Pool -> (?, 7, 7, 256)

Layer3 ImgIn shape=(?, 7, 7, 256), Conv -> (?, 7, 7, 512), Pool -> (?, 4, 4, 512)

Layer4 (gap) ImgIn shape=(?, 4, 4, 512), Conv -> (?, 4, 4, 26), Pool -> (?, 1, 1, 26)

**<훈련과정>**

텍스트이(가) 표시된 사진

자동 생성된 설명

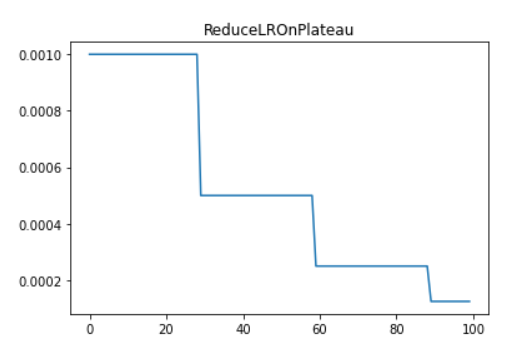
Cost function과 optimizer, scheduler는 위와 같다.

훈련 시 learning rate는 성능에 꽤 영향을 주기 때문에 처음부터 끝까지 같은 lr을 사용하지 않고 학습과정에서 lr을 조정하기 위해 lr\_scheduler를 사용했다.

Scheduler는 성능 향상이 없을 때 lr을 감소시킨다. 그렇기 때문에 validation loss나 metric을

Learnning rate step함수의 input으로 넣어주어야 한다. 그래서 metric이 향상되지 않을 때, epoch

만큼 참고 그 이후에 learning rate를 줄인다.



<https://pytorch.org/docs/stable/generated/torch.optim.lr_scheduler.ReduceLROnPlateau.html> 을 참

고하여 parameter를 수정하며 fine tune하였다.

모델은 매 epoch마다 accuracy를 측정하여 제일 높은 accuracy의 모델을 저장하도록 하였고 그렇게 저장된 모델을 다시 불러와서 parameter를 약간 수정하여 fine tuning을 하였다. 결과적으로 공유 데이터 셋에 대해 0.9987까지 올라갈 수 있었다.

데이터 셋이 한정적이라 다른 데이터 셋을 찾아봤는데 kaggle데이터 셋은 i, j, l 이 데이터가 너무 깨져 있었고 공유 데이터 셋으로 만든 best.pth로 kaggle데이터 셋을 inference한 결과 0.7782정도의 낮은 정확도를 보였다. 그래서 fine tuning을 할 때 적은 데이터로도 성능을 올리기 위해 데이터 전처리를 한번 하고, augmentation을 사용하여 계속 fine tuning하였다. 교수님의 test data set이 어떤 것일지 모르기 때문에 모델을 더 robust하게 만들기 위해 kaggle데이터 셋에서 괜찮은 데이터를 조금 뽑아서 훈련을 진행했다. 결과적으로 공유 데이터셋에 대해 0.9987이었던 정확도가 조금 내려갔지만 kaggle데이터 셋에 대해 0.7782정도 나왔던 모델이 0.95xxx까지 나오도록 향상시킬 수 있었다.

Augmentation은 다음을 참고하여 소문자 알파벳 데이터셋에 적용하기 좋은 것들을 골라 사용했다. <https://pytorch.org/vision/stable/transforms.html>

텍스트이(가) 표시된 사진

자동 생성된 설명

Accuracy를 계산하는 함수를 만들어 train과정에서 매 epoch마다 그 때의 모델로 계산한 accuracy와 그전 까지의 best accuracy를 비교해서 accuracy가 제일 높을 때의 model을 저장하였다.

텍스트이(가) 표시된 사진

자동 생성된 설명

**<Inference>**

**P201710885.py 에서 현재 디렉토리에 있는 “test”폴더를 테스트 파일로 로드하고 현재 디렉토리에 있는 모델 p201710885.pth를 load하여 inference한다.**

**한 디렉토리 내에 p201710885.py, p201710885.pth, test/를 같이 두고 p201710885.py를 실행시키면 된다.**

test코드에 필요한 라이브러리를 import한뒤 먼저 모델을 정의했고, device는 cuda를 사용할 수 있는 경우에는 ‘cuda’(GPU), 사용할 수 없는 경우에는 ‘cpu’로 설정했다.

random seed를 고정하고 현재 디렉토리의 테스트 데이터 셋을 ImageFolder()함수를 사용하여 불러왔는데 이 때 Totensor(), Grayscale(1)함수를 사용하여 데이터 셋을 transform했다.

Torch.utils.data의 DataLoader()를 사용하여 테스트 데이터 셋을 load했다. 이 때 batch size는 100을 주고 shuffle과 drop last는 false로 처리했다.

훈련된 모델을 torch.load()를 사용하여 불러오고 공유데이터 셋 에서 5:1로 나누었던 test data set에 대해 inference한 결과 아래와 같다. 시간 측정은 import부분 바로 밑에 시작 시간을 찍고 테스트가 끝나고 코드의 마지막 부분에서 끝 시간을 찍어 그 차이를 데이터셋 크기로 나눠 출력했다.

텍스트이(가) 표시된 사진

자동 생성된 설명

노트북에서 Pycharm을 사용하여 test한 결과 Accuracy: 0.9987, Time: 0.0017이 나왔다.

텍스트이(가) 표시된 사진

자동 생성된 설명

Jetson nano에서 inference한 결과 Accuracy: 0.9987, Time: 0.0065정도 나왔다.