알고리즘 응용(00) Lab07

201802161 조은빈

1. Goal

- Data 전처리
- 딥러닝 네트워크 구현하기
- 학습하기
- Kaggle에 결과 제출하기

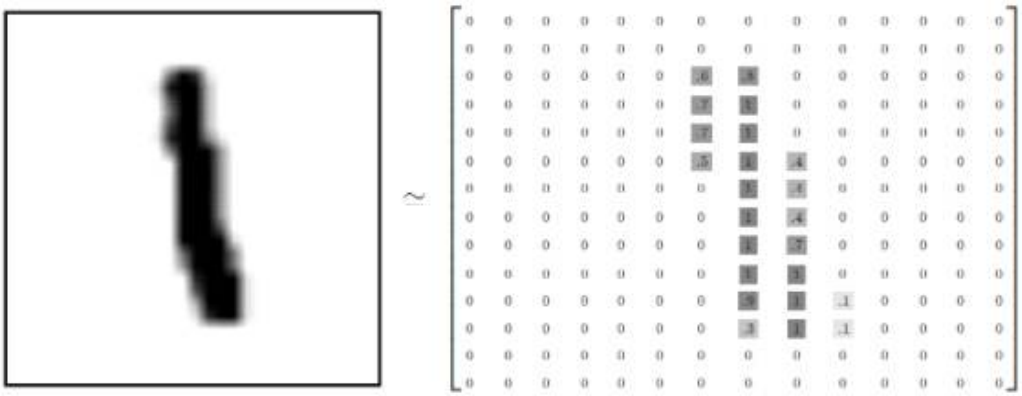
2. 코드 분석

```
1    import numpy as np
2    import pandas as pd
3    import torch.nn as nn
4    import torch
5    from torch import optim
6    import matplotlib.pyplot as plt
```

코드 구현에 필요한 라이브러리 패키지를 import 한다.

```
8    def read_data(train_path, test_path):
9
10       # train 파일 읽어온다
11       train = pd.read_csv(train_path)
12
13       # 정답 항목인 label 데이터를 y_train에 저장
14       y_train = train['label']
15
16       # 전체 데이터 개수는 42000개이고 0~9의 숫자를 구분하므로
17       # shape이 42000, 10이고 0으로 채워진 np 어레이를 생성
18       y_list = np.zeros(shape=(y_train.size, 10))
19
20       # One-Hot Encoding. y_train 값이 1이면 0 1 0 0 0 0 0 0 0 0
21       for i, y in enumerate(y_train):
22           y_list[i][y] = 1
23
24       # y_train을 One-Hot Encoding 한 결과를 다시 y_train에 저장
25       y_train = y_list
26
27       # train에서 label 삭제
28       del train['label']
29
30       # 0~255 범위의 데이터 값들이 0~1 사이에 존재하도록 스케일링
31       x_train = train.to_numpy() / 255
32
33       # test 파일 읽어온다
34       test = pd.read_csv(test_path)
35
36       x_test = test.to_numpy() / 255
37
38       return x_train, y_train, x_test
```

read_data는 train 파일과 test 파일을 읽어와 데이터를 전처리 하는 함수이다. train 파일은 label과 784개의 pixel 데이터들로 이루어져 있다. label은 이미지가 나타내는 숫자가 어떤 숫자인지 나타내는 데이터로 0~9까지의 숫자로 이루어져 있다.



이미지는 위와 같이 28x28로 구성되어 있는데 이 이차원 행렬을 일차원으로 변환한 데이터가 pixel 데이터이다.

pixel 데이터로 맞춰야하는 정답이 label 데이터이므로 label 데이터를 y_train에 저장한다. 전체 데이터 개수는 42000개이고 0~9로 구분해야하므로 shape이 (42000, 10)인 np 어레이를 생성한다.

y_train의 값들을 One-Hot Encoding 해서 다시 y_train에 저장한다. One-Hot Encoding 하면 y_train 값이 1일 경우 [0, 1, 0, 0, 0, 0, 0, 0, 0, 0]으로 저장된다.

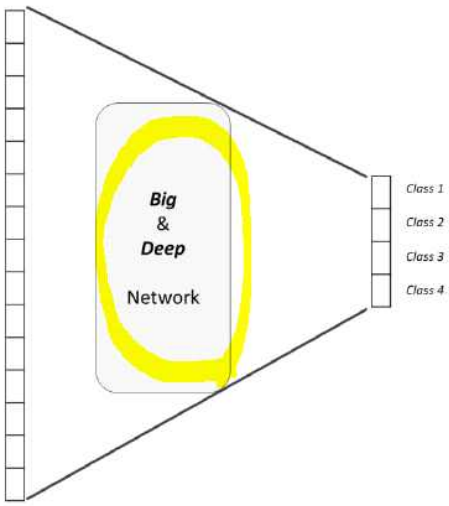
train에서 label 항목을 삭제하고 값들을 스케일링한다. train에 저장된 데이터의 범위는 0~255이므로 255로 나눠서 0~1 사이에 모든 값들이 존재하도록 스케일링 해준다.

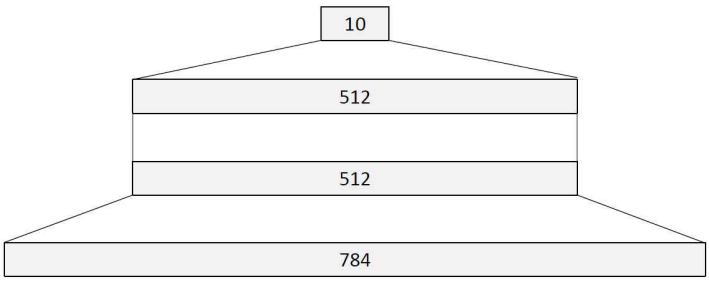
test 파일을 읽어 와서 똑같이 스케일링 해준다.

```
40      # network model 정의
41  class MNISTModel(nn.Module):
42      def __init__(self):
43          super(MNISTModel, self).__init__()
44          self.fc1 = nn.Linear(784, 512)  # 784 차원 --> 512 차원
45          self.fc2 = nn.Linear(512, 512)  # 512 차원 --> 512 차원
46          self.fc3 = nn.Linear(512, 10)   # 512 차원 --> 10 차원
47
48      def forward(self, x):
49          x1 = torch.relu(self.fc1(x))
50          x2 = torch.relu(self.fc2(x1))
51          x3 = self.fc3(x2)
52          return x3
```

MNISTModel 클래스는 데이터를 가공해 차원을 줄여주는 Network model을 정의한 클래스이다.

한 이미지의 픽셀 개수는 784개이므로 784 차원의 데이터에서 시작해서 784->512->512->10으로 줄여나갈 것이다. 우리가 맞춰야할 정답의 범위가 0~9의 숫자이므로 최종적으로 10차원 데이터에 도달해야 한다.






```python
def train(x_train, y_train, batch, lr, epoch):

    # model 생성
    model = MNISTModel()

    # model을 훈련시킬 때 model.train() 사용
    model.train()

    # 평균 제곱 오차(MSE)를 loss_function으로 사용
    # reduction="mean" 옵션은 모든 에러값들의 평균 도출
    loss_function = nn.MSELoss(reduction="mean")

    # Adam 알고리즘으로 모델을 최적화, 파라미터 업데이트.
    optimizer = optim.Adam(model.parameters(), lr=lr)

    # data 처리. 텐서 자료형으로 변환.
    x = torch.from_numpy(x_train).float()
    y = torch.from_numpy(y_train).float()

    # data를 batch 크기로 분리
    data_loader = torch.utils.data.DataLoader(list(zip(x, y)), batch, shuffle = True)

    epoch_loss =[]  # loss 리스트
    epoch_acc = []  # accuracy 리스트
```

train은 MNISTModel 클래스로 model을 생성해서 학습시키는 함수이다.

model.train()은 model을 훈련시키겠다고 선언하는 부분이다.

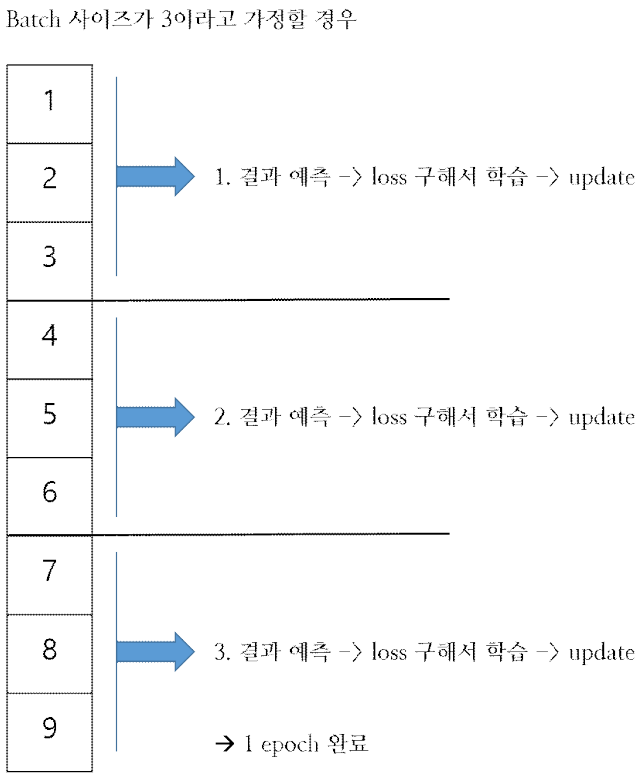
예측값과 정답 사이의 차이, 즉 에러 값을 계산하기 위해 loss_function을 정의한다. loss_function으로는 평균 제곱 오차(MSE)를 사용한다. reduction="mean" 옵션은 모든 에러 값들의 평균을 도출하겠다는 의미이다.

optimizer는 Adam 알고리즘으로 파라미터를 업데이트하면서 모델을 최적화시킨다.

x_train과 y_train을 텐서 자료형으로 변환하고 data를 batch 크기로 쪼갠다. data_loader는 하나의 커다란 data를 일정한 사이즈로 쪼개서 볼 수 있게 해준다.

```python
    for e in range(epoch):
        total_loss = 0
        total_acc = 0
        for data in data_loader:
            x_data, y_data = data

            # x_data를 model에 넣어서 결과 예측
            pred = model(x_data)

            # 예측한 결과와 정답 간의 차이를 구한다
            loss = loss_function(pred, y_data)

            # 이전의 학습 결과를 리셋
            optimizer.zero_grad()

            # 학습
            loss.backward()

            # update. 학습된 결과를 반영
            optimizer.step()

            total_loss += loss.item()  # loss 값들을 더해준다
            total_acc += get_acc(pred, y_data)  # accuracy 값들을 더해준다

        epoch_loss.append(total_loss / len(data_loader))    # 1 epoch이 끝나면 리스트에 loss 저장
        epoch_acc.append(total_acc / len(data_loader))    # 1 epoch이 끝나면 리스트에 accuracy 저장
        print("Epoch [%d] Loss: %.3f\tAcc: %.3f"% (e+1, epoch_loss[e], epoch_acc[e]))

    return model, epoch_loss, epoch_acc
```

지정한 epoch만큼 data를 돈다. data_loader로 일정하게 쪼개진 데이터를 돌아가면서 본다. data를 x_data와 정답 데이터인 y_data로 나눈다. x_data를 model에 넣어서 결과를 예측한다. loss_function에 예측값인 pred와 정답 데이터인 y_data를 넣고 둘 사이의 loss를 구한다. loss 값으로 학습시킨 후 step 함수를 통해 학습된 결과를 반영한다. 나온 loss 값과 accuracy 값들은 각각 total_loss, total_acc에 더해준다. 위 과정이 1 epoch이다. 그림으로 간단하게 나타내면 아래와 같다.



1 epoch이 완료되면 loss 확률과 accuracy 확률을 구해 각각의 리스트에 저장한다. epoch을 반복할수록 loss는 감소하고 accuracy는 증가하는 것을 이후 결과에서 확인할 수 있다.

```
54  def get_acc(pred, answer):  # 예측값과 정답을 받아서 정확도 측정
55      correct = 0
56      for p, a in zip(pred, answer):
57          pv, pi = p.max(0)
58          av, ai = a.max(0)
59          if pi == ai:
60              correct += 1
61      return correct / len(pred)
```

get_acc는 하나의 mini batch를 돌고나서 accuracy를 구하는 함수이다. 예측값과 정답을 받아서 얼마나 일치하는지 측정한다. 예측 데이터 p, 정답 데이터 a를 비교하는데 우선 각각 max로 최댓값과 그 인덱스를 반환받는다. p와 a는 One-Hot Encoding 되어있기 때문이다. 만약 [0, 1, 0, 0, 0, 0, 0, 0, 0, 0]이면 max 값은 1이므로 1과 인덱스 1이 반환될 것이다. 즉 이 리스트가 가리키는 값은 1이라는 의미이다.

p와 a에서 max 값의 인덱스를 추출해 서로 동일하면 예측값이 정답이라는 의미가 된다.

```
119 def test(model, x_test, batch): # model로 test 해본다
120
121     # model을 테스트할 때 model.eval() 사용
122     model.eval()
123
124     # data 처리, 텐서 자료형으로 변환.
125     x = torch.from_numpy(x_test).float()
126
127     # data를 batch 크기로 분리
128     data_loader = torch.utils.data.DataLoader(x, batch, shuffle=False)
129
130     preds = []
131     for data in data_loader:    # epoch 없이 data_loader 돌려서 결과 예측
132         pred = model(data)
133         for p in pred:
134             pv, pi = p.max(0)   # p는 One-Hot Encoding 되어 있는 데이터
135             preds.append(pi.item()) # p의 max 값을 찾아서 preds에 저장
136
137     return preds
```

test 함수는 학습시킨 model에 test data를 넣어서 확인해보는 함수이다. model을 테스트할 경우 model.eval()을 선언해준다. data_loader로 data를 batch 사이즈로 분리한다. 이때는 epoch 없이 data_loader를 돌려서 결과를 예측한다. 결과를 예측하고 예측값들을 preds 배열에 담아 반환한다.

```
140 def draw_graph(data):
141     plt.plot(data)
142     plt.show()
143
144 def save_pred(save_path, preds):
145     # 예측된 결과를 submission 형식에 맞게 작성
146     submission = pd.read_csv('sample_submission.csv', index_col='ImageId')
147     submission["Label"] = preds
148
149     # 결과를 csv 파일로 저장
150     submission.to_csv(save_path)
```

draw_graph 함수는 그래프를 그리고 시각화하는 함수이다.

save_pred는 예측값을 submission 형식에 맞게 작성해서 csv 파일로 저장하는 함수이다.

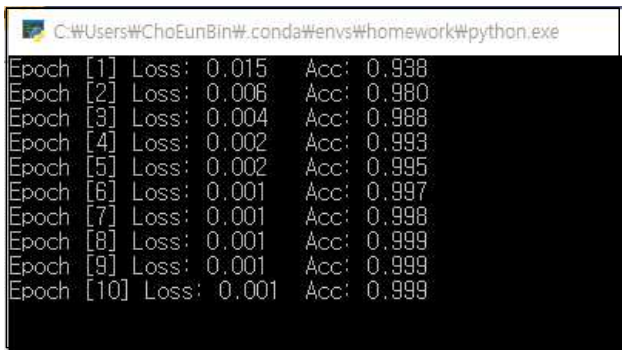
```python
def main():
    train_path = 'train.csv'
    test_path = 'test.csv'
    save_path = 'my_submission.csv'

    batch = 128
    lr = 0.001  # learning rate
    epoch = 10

    x_train, y_train, x_test = read_data(train_path, test_path)
    model, epoch_loss, epoch_acc = train(x_train, y_train, batch, lr, epoch)
    preds = test(model, x_test, batch)

    # 저장
    save_pred(save_path, preds)

    draw_graph(epoch_loss)
    draw_graph(epoch_acc)

if __name__ == '__main__':
    main()
```

batch, lr, epoch을 설정한다. batch는 한 번에 확인할 데이터의 양, lr은 learning rate로 한 번 학습할 때 얼마만큼 학습해야 하는지 학습 양을 의미한다. learning rate가 너무 크면 한 번에 학습하는 양이 커서 학습할 때마다 결과가 크게 변하고 너무 작으면 거의 갱신되지 않기 때문에 적절한 값으로 설정한다.

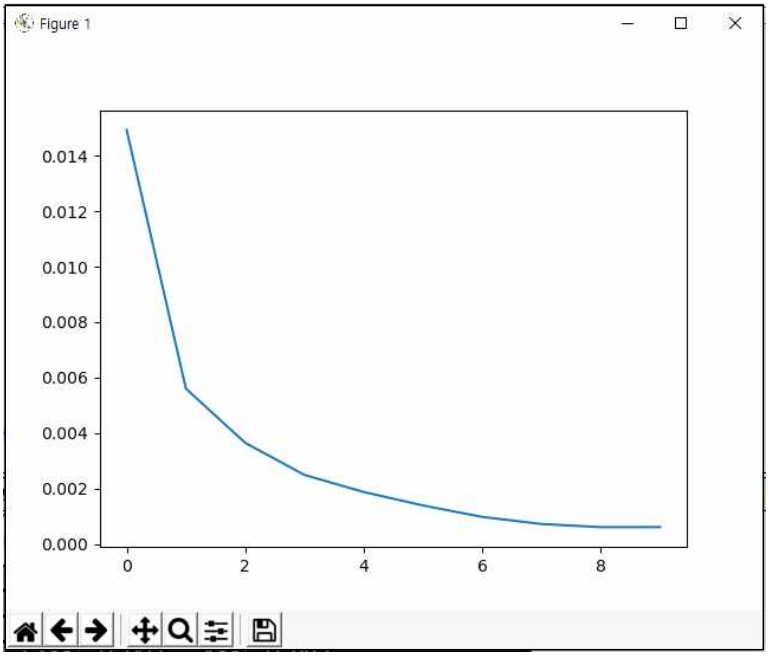
3. 실행 결과

- Kaggle 결과

- 각 Epoch 별 Loss 값과 Accuracy 값. Epoch을 거듭할수록 Loss 값은 줄어들고 Accuracy 값은 증가한다.



- Loss 그래프. Loss가 줄어드는 것을 직관적으로 확인 가능하다.



- Accuracy 그래프. Accuracy가 증가하는 것을 직관적으로 확인 가능하다.