Term Project

< MNIST datasets의 DNN 과정 시각화>

2019040973

소프트웨어학부

유은정

1. 중요 실험 코드 분석, 설명.

```
class MLP(nn.Module):
    def __init__(self):
        super().__init__()

        self.in_dim = 28*28
        self.out_dim = 10

        self.fc1 = nn.Linear(self.in_dim,512)
        self.fc2 = nn.Linear(512,256)
        self.fc3 = nn.Linear(256,128)
        self.fc4 = nn.Linear(128,64)
        self.fc5 = nn.Linear(64, self.out_dim)

        self.relu = nn.ReLU()
        self.log_softmax = nn.LogSoftmax()

    def forward(self,x):

        z1 = self.fc1(x.view(-1,self.in_dim))
        a1 = self.relu(self.fc1(x.view(-1,self.in_dim)))
        #print("a1",a1)
        z2 = self.fc2(a1)
        a2 = self.relu(self.fc2(a1))
        #print("a2",a2)
        a3 = self.relu(self.fc3(a2))
        #print("a3",a3)
        a4 = self.relu(self.fc4(a3))
        #print("a4",a4)
        logit = self.fc5(a4)
        return logit, z1, a1, z2, a2, a3, a4
```

<DNN model 구조>

- **MLP** : DNN model의 모양을 정의하는 class. Init과 forward로 나뉘어짐.

- **__init__** : model의 구조를 정의하는 함수.

  Self.In_dim : input의 크기를 정의

  Self.out_dim : output의 크기를 정의

  Self.fcN : linear layer. 좌측은 5개의 linear layer을 가진 구조.

  Self.relu : relu 함수. Activation function의 한 종류

  Log_softmax : softmax 함수. Activation function의 한 종류

- **Forward** : model이 어떤 순서로 진행될지를 정의하는 함수.

  z값을 구하고 relu를 적용하며 5개의 layer에 걸쳐 학습이 진행됨.

  마지막에는 relu를 적용하지 않음.

```
for epoch in range(10):
    running_loss=0.0
    for i, data in enumerate(train_loader,0):
        inputs, labels = data

        optimizer.zero_grad()

        outputs, z1, a1, z2, a2, a3, a4 = model(inputs)

        if(epoch == 0 and i == 0):
            print(epoch, i)
            z11 = z1
            z22 = z2
            a11 = a1
            a22 = a2
            label = labels

            a11 = a11.detach().numpy()
            a22 = a22.detach().numpy()
            z11 = z11.detach().numpy()
            z22 = z22.detach().numpy()
            label = label.detach().numpy()
            print(epoch, i)
        else:
          if(i % 1000 == 0) :
            print(i)
          a1 = a1.detach().numpy()
          a2 = a2.detach().numpy()
          z1 = z1.detach().numpy()
          z2 = z2.detach().numpy()
          label2 = labels.detach().numpy()

          z11 = tf.concat([z11,z1],0)
          z22 = tf.concat([z22,z2],0)
          a11 = tf.concat([a11,a1],0)
          a22 = tf.concat([a22,a2],0)
          label = tf.concat([label,label2],0)

        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()


        running_loss += loss.item()
        if (i+1) % 5000 ==0:
            print('[%d, %5d] loss: %.3f' %
                  (epoch + 1, i+1, running_loss/2000))
            running_loss = 0.0
```

<training 과정>

-10번의 epoch를 거치며 input data를 training하는 코드이다.

1. train_loader를 이용해 input과 labels를 batch size만큼씩 불러옴.
2. gradient를 zero로 초기화
3. model에 input을 넣은 후, 출력으로 outputs, z1, a1, z2, a2, a3, a4를 받음.
3. (epoch == 0 and I ==0) 인 경우, 가장 처음 시작되는 지점이므로 각 항목별로 데이터를 쌓아 저장할 배열(z11,z22,a11,a22,label) 선언, 초기화. NumPy 배열로 변환.
4. (epoch == 0 and I ==0)가 아닌 경우, NumPy 배열로 변환 후 저장용 배열(z11,z22,a11,a22,label)에 항목 추가.
5. loss를 계산한 후 backward와 optimizer를 차례대로 실행
6. running loss 갱신

```
X = z22[540000:600000]
y = label[540000:600000]

print(f'X.shape : {X.shape}')
print(f'y.shape : {y.shape}')
```

<원하는 데이터만 추출하여 X,y로 지정>

한 epoch당 60000개의 데이터를 training하므로 10epoch가 끝나면 저장용 배열에 600,000개의 data (1~10 epoch의 데이터)가 존재한다. 10epoch의 데이터는 [540000:600000]에 존재하므로 해당 범위의 데이터와 label을 추출한다. A1 데이터를 분석하고 싶다면 a11, a2 데이터를 분석하고 싶다면 a22값을 X에 대입한다.

```
N = 10000
df_subset = df.loc[rndperm[:N],:].copy()
data_subset = df_subset[feat_cols].values
pca = PCA(n_components = 2)
pca_result = pca.fit_transform(data_subset)
df_subset['pca-one'] = pca_result[:,0]
df_subset['pca-two'] = pca_result[:,1]
```

```
time_start = time.time()
tsne = TSNE(n_components=2,verbose=1,perplexity=40,n_iter=300)
tsne_results = tsne.fit_transform(data_subset)
df_subset['tsne-2d-one'] = tsne_results[:,0]
df_subset['tsne-2d-two'] = tsne_results[:,1]
```

<PCA, t-SNE>

앞서 주어진 X,y데이터를 가공한 df 값을 바탕으로 PCA, t-SNE 실행한다.

주성분 (n_component)는 2개로 설정.

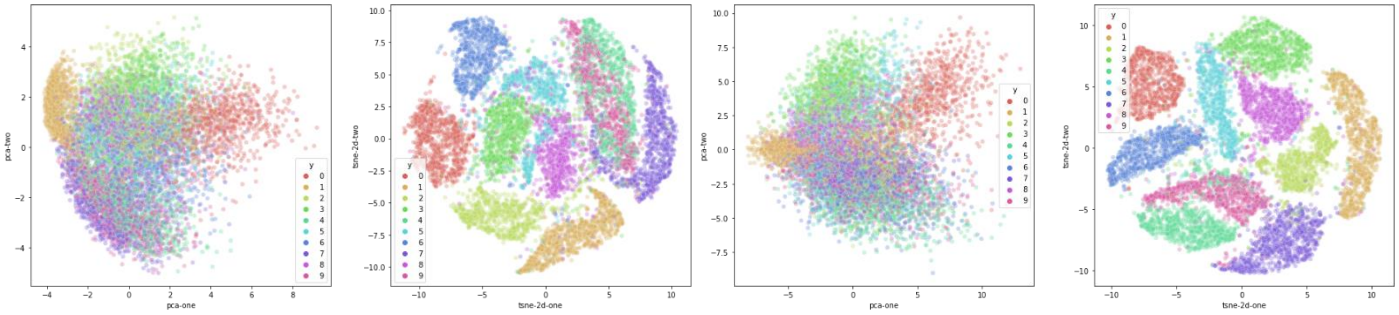
Data_subset을 표준화화에 pca_result에 저장한다.

```
plt.figure(figsize=(16,7))
ax1 = plt.subplot(1, 2, 1)
sns.scatterplot(
    x="pca-one", y="pca-two",
    hue="y",
    palette=sns.color_palette("hls", 10),
    data=df_subset,
    legend="full",
    alpha=0.3,
    ax=ax1
)
ax2 = plt.subplot(1, 2, 2)
sns.scatterplot(
    x="tsne-2d-one", y="tsne-2d-two",
    hue="y",
    palette=sns.color_palette("hls", 10),
    data=df_subset,
    legend="full",
    alpha=0.3,
    ax=ax2
)
```

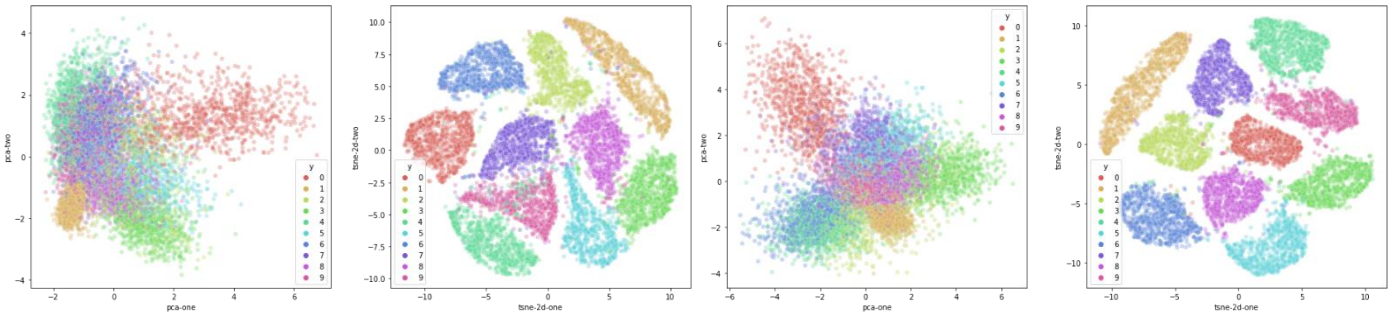
<PCA와 t-SNE 결과 plot으로 출력>

x축과 y축을 설정, 입력된 데이터의 항목별로 색상을 입힌 후 plot 출력.

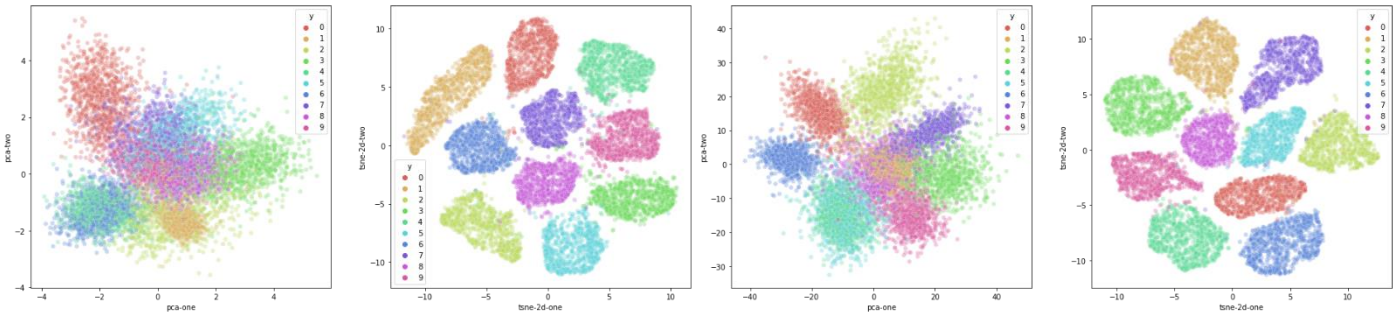
2. 결과 사진



<input image에 대한 PCA, t-SNE 결과 (a[0])> **< z[1] >**



< a[1] > **< z[2] >**



< a[2] > **< outputs >**

3. 결과 분석

- **PCA 방법과 t-SNE 방법의 공통된 특징**
  - **Layer를 지날수록 classify가 잘 됨.**

    더 깊은 layer에 도달할수록 PCA 방법과 t-SNE 방법에서 모두 classify가 더 잘 되는 것을 확인할 수 있었다.

    PCA 방법의 경우, input의 PCA 결과를 보면 각 항목이 제대로 구별되지 못하고 겹치고 섞어 있었지만, output의 PCA 결과를 보면 각 항목들이 중심점을 기준으로 모여 있는 것을 확인할 수 있었다.

    t-SNE 방법의 경우, input의 t-SNE 결과에서도 PCA보다는 비교적 덜하지만 각 항목이 구별되지 못하고 겹쳐 있는 부분들이 눈에 띄었다. 하지만 layer들을 거치며 구분이 잘 되는 것을 확인할 수 있었고, 마지막으로 나온 output의 t-SNE 결과를 보면 각 항목이 경계선을 가지고 명확히 구별 되어있는 것을 확인할 수 있었다.

  - **Epoch를 너무 작게 설정하면 loss가 커서 classify가 제대로 되지 않음.**

    Epoch를 1로 설정했을 때와 epoch를 10으로 설정했을 때의 각 layer별 결과를 비교해보면, PCA와 t-SNE 두 방법에서 모두 차이가 있는 것을 확인했다. Epoch가 1일 때는 epoch가 10일 때에 비해 classify가 덜 이루어진다. 하지만 epoch가 일정 횟수 이상이라면 이미 loss가 충분히 작고, epoch가 커지더라도 loss가 감소하는 속도도 더디기 때문에 결과에는 큰 차이가 없는 것을 확인할 수 있었다.

- **PCA 방법과 t-SNE 방법의 차이점**
  - **항목별 경계선의 유무**

    PCA 방법을 이용해 나타낸 각 layer의 plot에서는 t-SNE 방식에 비해 각 항목이 명확하게 구분되지 않는 것을 확인할 수 있다. Layer가 깊어질수록 구분이 더 잘되긴 하지만, 각 항목을 구분하는 명확한 경계선이 없이 구분되었다. 이는 PCA가 linear한 pattern은 잘 표현해내지만 non-linear한 pattern은 잘 표현해내지 못하기 때문에 일어난 현상이다.

    t-SNE 방법을 이용해 나타낸 각 layer의 plot에서는 각 항목들이 명확하게 구분된 것을 확인할 수 있었다. Input 이미지를 t-SNE 방법으로 나타낸 결과를 보면 명확히 구분되지 못한 부분이 보이긴 하지만 layer가 깊어질수록 각 항목들이 명확한 경계선을 가지고 구분되었다. T-SNE 방법은 각 데이터 포인트를 2차원에 무작위로 표현한 후에 원본 특성 공간에서 가까운 점은 가깝게, 멀리 떨어진 점은 멀어지게 만드는 과정을 수행한다. 따라서 각 항목들이 겹치는 공간이 많은 PCA 방법과 비교해서 t-SNE 방법은 각 항목끼리의 경계가 명확하고, 잘 구별되는 모습을 보여주는 것이다.

  - **실행 시간 차이**

    PCA 방법보다 t-SNE 방법에서 실행 속도가 확연히 느린 것을 확인할 수 있었다.

    이는 n개의 데이터가 있을 때 t-SNE의 계산 시간이 n^2이 되기 때문이다. 따라서 데이터가 늘어날수록 연산 시간은 급격히 늘어난다. 대용량의 데이터를 처리하기에는 부적합한 것을 확인할 수 있었다.