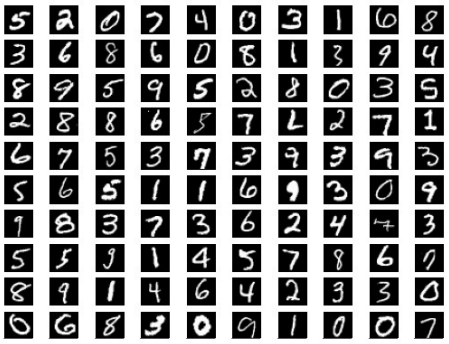
Summary Report [Week 12]. AutoEncoder

20190106 KimByungjun (김병준)

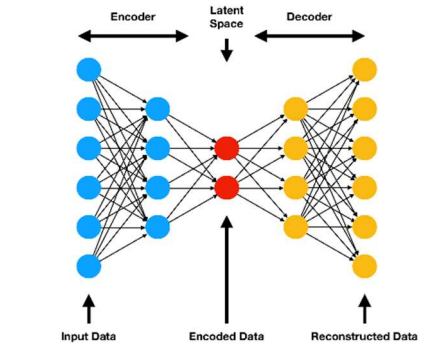
● Exercises

이번 Exercise 에선 AutoEncoder 를 구현해보고 visualization 과 model 의 performance 변화를 살펴보고자 한다. 일반적으로 AutoEncoder 란, encoder 를 통해 input data 를 compress 한 후, 그 과정에서 추출한 의미 있는 latent vector(혹은 latent representation)을 얻어내며 decoder 를 통해 원래 data 와 유사한 형태로 복원하는 model 을 말한다. AutoEncoder 는 여러 관점에서 사용될 수 있는데, 대표적으로 dimension reduction, feature extraction, visualization, 유사하면서 새로운 data 의 generation, anomaly detection 등에 유용하게 쓰인다. 해당 Exercise 에서는 MNIST hand-written digit image 를 가지고 linear autoencoder 과 convolutional autoencoder 에 대해 살펴본다.



[Fig 1] MNIST handwritten digits dataset

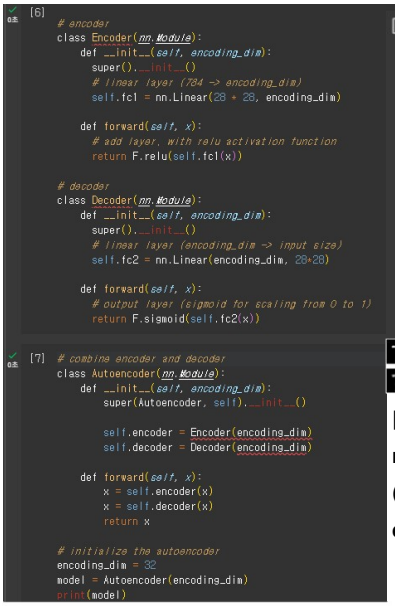
우선, **linear autoencoder model** 을 구현해보자. Linear autoencoder 은 **[Fig 2]**의 autoencoder model 에서 encoder 과 decoder 구현에 linear layer 를 사용한다. 예를 들어, encoder 에서 (input features, output features)=(28×28, 32) (* input image size 가 28×28 pixels 이다) linear layer 를, decoder 에서 (input features, output features)=(32, 28×28) 를 사용한다고 가정하면, **[Fig 3]**과 같이 구현할 수 있다. Training 은 decoder 를 통과한 reconstructed data 와 기존 original input data 간의 차이가 최소화되도록 진행된다. Model training 을 MSE loss function, Adam optimizer 를 기준으로 진행하였으며, **[Fig 4]**의 상단 그림처럼 training 이 잘 진행되었다.



[Fig 2] Autoencoder model representation

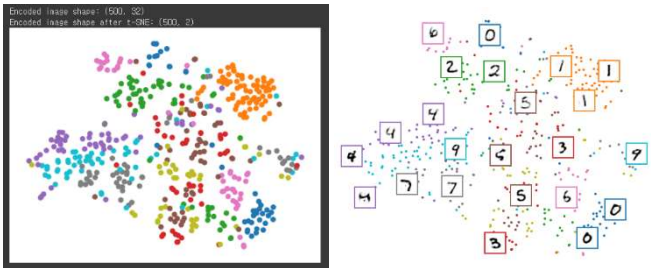
[Fig 4]의 아래 그림을 보자. 위 줄 은 model 적용 전 원본 data 이고, 아래 줄은 autoencoder 를 통과한 reconstructed data 이다. 잘 보면, 약간 흐려지기도 한 것 같고 edge 가 약간 뭉툭한 것 같아 보인다. 즉, autoencoder 을 통해 기존 data 와 output 간의 loss 가 생겼음을 인지할 수 있다.

Autoencoder 를 통해 구한 latent vector 혹은 latent representation 을 가시화(visualization)해보자. Encoding 된 image 는 우리가 흔히 아는 MNIST image 가 아님에 주의하자. Encoding 된 image 의 shape 은 **[Fig 5]** compile 결과에서 볼 수 있듯이 (500, 32)이며, 32 차원의 data 는 visualizing 이 어렵다. 이를 해결하기 위해, t-SNE(t-distributed Stochastic Neighbor Embedding) 기법을 통해 32 차원 data 를 2D space 내 표현이 가능하게 transform 한다. 이 때, 2D space point 간의 거리가 가까울수록, 원래 32 차원에서 data 의 구조가 비슷함을 의미한다. 그렇다면, 당연히 같은 digit 을 나타내는 data 들은 가깝게 모여서 하나의 cluster 를 형성하려 할 것이며, 두 digit 을 애매하게 닮은 image 를 나타내는 point 역시 어느 정도 가깝게 분포하려고 할 것이다. 해당 상황을 표현한 게 **[Fig 5]**이다.



[Fig 3] Linear Autoencoder Implementation

[Fig 4] (Above) Training loss of model
(Below) Comparison of Original data Reconstructed data image



[Fig 5] Visualization of latent representation

허나, linear autoencoder 처럼 simple 한 structure 의 경우, loss 가 상대적으로 크며 performance 가 낮기도 하다. 이보다 복잡하지만 performance 향상을 위해 convolutional autoencoder 을 구현해보도록 하자.

1) Implement an Encoder and Decoder to have the exact same network structure as shown in the [Fig 6].

<Explanation>

Note. [Fig 6]은 Autoencoder 에 쓰이는 구성 요소들이며, Encoder 에서는 conv1 → maxpool → conv2 → maxpool 순서로 encoding 이, Decoder 에서는 t_conv1 → t_conv2 순서로 decoding 이 진행된다.

```
ConvAutoencoder(
  (encoder): Encoder(
    (conv1): Conv2d(1, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (conv2): Conv2d(16, 4, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (decoder): Decoder(
    (t_conv1): ConvTranspose2d(4, 16, kernel_size=(2, 2), stride=(2, 2))
    (t_conv2): ConvTranspose2d(16, 1, kernel_size=(2, 2), stride=(2, 2))
  )
)
```

[Fig 6] Structure of Convolutional Autoencoder

위 순서대로 진행하기 위해, Encoder class 와 Decoder class 를 정의하면, **[Fig 7]**와 같이 implement 하게 된다. Batch 내 단일 train data 의 Shape (channel 수, row, column) = (1, 28, 28)인 상태에서 conv1 적용 시 (16, 28, 28), kernel_size 및 stride 가 2 인 maxpool 1 회 적용 후 (16, 14, 14), 그 뒤 conv2 적용 시 (4, 14, 14), 마지막 maxpool 적용이 끝나면 (4, 7, 7)이 된다. 단순히 Encoder 상에서 maxpool layer 을 2 번 적용하였으므로 28 x 28 image 는 7 x 7 로 축소한다고 볼 수 있다.

```
# encoder
class Encoder(nn.Module):
    def __init__(self):
        super().__init__()
        """ ### Implement the code yourself ### """
        # conv layer (depth from 1 --> 16), 3x3 kernels
        self.conv1 = nn.Conv2d(1,16, kernel_size = (3, 3), stride=(1, 1), padding=(1, 1))
        # conv layer (depth from 16 --> 4), 3x3 kernels
        self.conv2 = nn.Conv2d(16,4, kernel_size = (3, 3), stride=(1, 1), padding=(1, 1))
        # pooling layer to reduce x-y dims by two; kernel and stride of 2
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)

    def forward(self, x):
        x = self.conv1(x)
        x = self.pool(x)
        x = self.conv2(x)
        x = self.pool(x)
        return x
```

```
# decoder
class Decoder(nn.Module):
    def __init__(self):
        super().__init__()
        """ ### Implement the code yourself ### """
        # a kernel of 2 and a stride of 2 will increase the spatial dims by 2
        self.t_conv1 = nn.ConvTranspose2d(4,16, kernel_size=(2,2), stride=(2,2))
        self.t_conv2 = nn.ConvTranspose2d(16,1, kernel_size=(2,2), stride=(2,2))
    def forward(self, x):
        x = self.t_conv1(x)
        x = self.t_conv2(x)
        return F.sigmoid(x)
```

[Fig 7] Code for Exercise

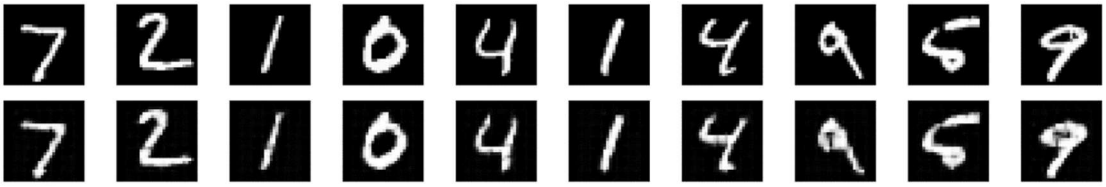
7 x 7 인 data 가 Decoding 후 원래 size 인 28 x 28 로 복원되어야 하므로, kernel size 및 stride 가 2 인 Transposed convolution layer 를 2 번 적용한다. 이 때, 각 layer 는 in_channel, out_channel 값이 다름에 주의하자. Decoder 를 통과한 data 는 sigmoid activation function 을 마지막으로 거쳐가는 데, 이는 본래 MNIST Dataset 이 0 에서 1 사이의 값을 가지는 점을 감안한 적절한 activation function 이라고 볼 수 있다.

```
Epoch: 1        Training Loss: 0.410049
Epoch: 2        Training Loss: 0.267915
Epoch: 3        Training Loss: 0.245315
Epoch: 4        Training Loss: 0.239953
Epoch: 5        Training Loss: 0.237116
Epoch: 6        Training Loss: 0.235215
Epoch: 7        Training Loss: 0.233849
Epoch: 8        Training Loss: 0.232747
Epoch: 9        Training Loss: 0.231955
Epoch: 10       Training Loss: 0.231199
Epoch: 11       Training Loss: 0.230556
Epoch: 12       Training Loss: 0.229859
Epoch: 13       Training Loss: 0.229240
Epoch: 14       Training Loss: 0.228534
Epoch: 15       Training Loss: 0.227520
Epoch: 16       Training Loss: 0.225447
Epoch: 17       Training Loss: 0.216162
Epoch: 18       Training Loss: 0.203375
Epoch: 19       Training Loss: 0.198207
Epoch: 20       Training Loss: 0.196060
Epoch: 21       Training Loss: 0.194744
Epoch: 22       Training Loss: 0.193741
Epoch: 23       Training Loss: 0.192816
Epoch: 24       Training Loss: 0.192187
Epoch: 25       Training Loss: 0.191648
Epoch: 26       Training Loss: 0.191212
Epoch: 27       Training Loss: 0.190794
Epoch: 28       Training Loss: 0.190453
Epoch: 29       Training Loss: 0.190184
Epoch: 30       Training Loss: 0.189918
```

[Fig 8] Training loss for convolutional autoencoder

앞서 구현한 linear autoencoder 과 convolutional autoencoder 를 비교해보자. Training loss 측면에서 비교해보면, linear autoencoder 의 경우(**[Fig 4]**) 마지막 두 epochs 의 training loss 에서 약 0.28 의 값을 보인다. **[Fig 8]**의 convolutional autoencoder 에서는 초기 0.41 로 시작했던 training loss 가 대략 0.19 까지 감소했음을 알 수 있으며, 해당 결과를 통해 convolutional autoencoder 가 training set 에 대해 보다 더 훈련이 잘 진행되었음으로 추측할 수 있다.

실제 test data 를 기반으로 두 autoencoder 간의 image output 차이를 확인해보도록 하자. Convolutional Autoencoder model 을 통과한 data 는 **[Fig 9]**와 같은 reconstructed data(아래 줄)를 반환한다. 이 때, **[Fig 4]**의 linear autoencoder 와 비교했을 때, convolutional autoencoder 의 경우가 original image 와 더 유사한 output 을 보임을 눈에 띄게 알 수 있으며, 이는 input 과 output 간의 loss 가 더 작다는 것을 의미한다. 즉, convolutional autoencoder 가 더 좋은 performance 를 보임을 알 수 있다.



[Fig 9] Comparison of Original data Reconstructed data image in convolutional autoencoder

● References

✓ https://www.compthree.com/blog/autoencoder/