Auto-Encoder (AE) 기반 차원 축소

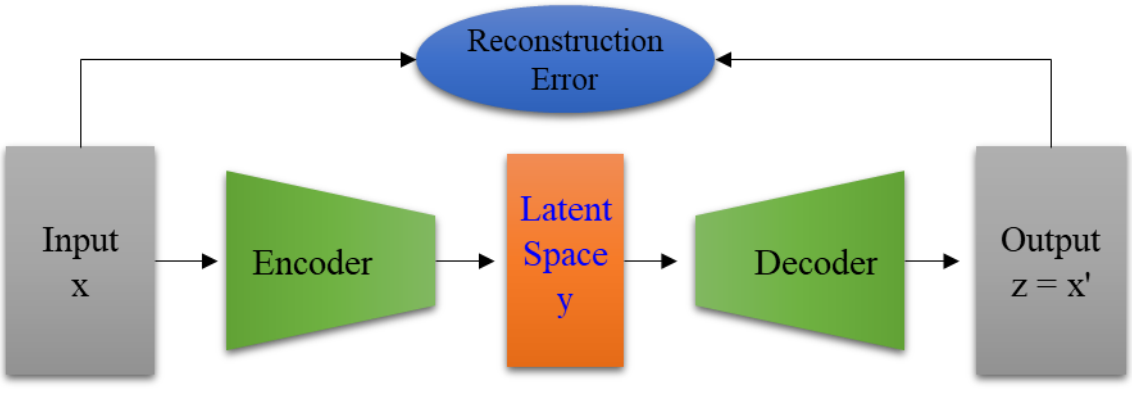
◆ 개요

- 차원 축소를 위한 AE의 구성
- TensorFlow를 통한 코드 구현
- AE 성능 확인
- 느낀 점

◆ 차원 축소를 위한 AE의 구성

- Unsupervised Learning 모델
- Input Data와 Out Data 구조가 동일
- Reconstruction Loss가 최소화 되도록 모델 학습 진행
- Input Data의 특징이 압축된 Latent Space가 핵심
- Latent Space라는 개념을 통해 차원 축소 가능

AE는 **고차원의 정보를 압축해 주는 Encoder**와 **압축된 정보를 다시 원래 정보로 돌려주는 Decoder**로 이루어져 있다.



【 Auto-Encoder (AE) 구조 】

AE 모델은 Encoder-Decoder의 결합으로 구성된다.

응용 방안으로는 학습이 완료된 Encoder-Decoder 구조에서 Decoder만 따로 사용이 가능하다. 압축된 정보를 Decoder의 입력으로 사용하면 알아서 원본 이미지와 유사한 가짜 이미지를 만드는 것이 가능한다.

- Encoder

$$y = f_\theta(x) = s(Wx + b)$$

- Decoder

$$z = g_{\theta'}(y) = s(W'y + b')$$

- 손실 함수

$$\theta^*, \theta'^* = \arg min \frac{1}{N} \sum_{i=1}^{n} L\left(x^{(i)}, z^{9i)}\right)$$

기본적인 Auto-Encoder 의 Weight를 학습하는 방법은 Backpropagation (BP) 알고리즘과 Stochastic Gradient Descent (SGD) 알고리즘을 활용하였다. Input으로 주어진 값에 대하여, hidden과 input 간의 weight 값을 계산하여, sigmoid 함수에 넣고, 다시 그 값을 한번 더 hidden과 output에 연결되어 있는 weight와 계산하여 추정된 값을, 최초 input data와 지속적을 비교하여 Reconstruction Error에 대해 update를 진행하는 Unsupervised Learning이다.

◆ 논문

Auto-encoderBasedDimensionalityReduction(2016, Neurocomputing)

해당 논문은 “Building Block”인 AE의 차원 축소 능력에 중점을 두고 있다. AE의 원래 입력 노드 수보다 적은 수의 히든 레이어 노드를 제한하면 원하는 차원 축소 효과를 얻을 수 있다.

차원 축소 과정에서 일부 차원을 버리면 필연적으로 정보 손실이 발생한다. 따라서 해결해야 할 주요 문제는 원본 데이터의 주요 및 중요한 특성을 가능한 한 많이 유지하는 것이다. 차원 축소 과정은 원래 데이터 특성과 밀접한 관련이 있다고 말한다. 고차원 데이터의 중요한 고유 특성인 latent space는 데이터의 필수 차원을 잘 반영 할 수 있다. 고차원 공간의 샘플 데이터는 일반적으로 전체 공간에서 확산 될 수 없다. 그것들은 실제 고차원 공간에 내장 된 저 차원 다양체에 놓여 있으며 다양체의 차원은 데이터의 고유한 차원이다.

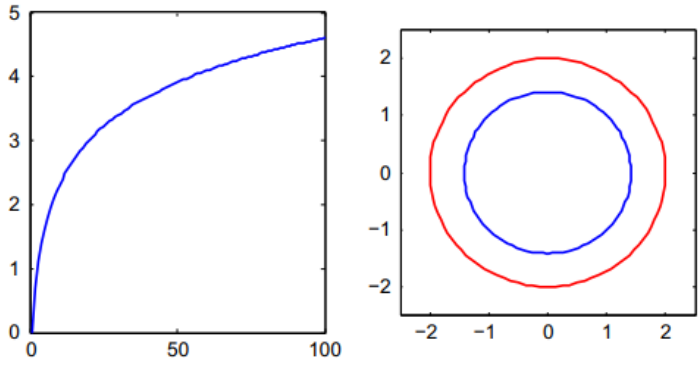


Fig. 3. Two cases from two-dimensional space.

MNIST와 Olivetti 얼굴 데이터 세트로 평가를 진행하였다.
대표적인 차원 축소 방법인 PCA, LDA, LLR, Isomap에 대한 분류 성능 비교 수행하였다.
- 선형 방법 : PCA, LDA
- 비선형 방법 : LLE, Isomap

MNIST 데이터 셋은 60,000개의 학습 이미지와 10,000개의 테스트 이미지가 포함된 필기 숫자의 데이터 세트이다. 모든 이미지는 정규화 되고 통합된 크기의 28x28의 흑백 이미지다.

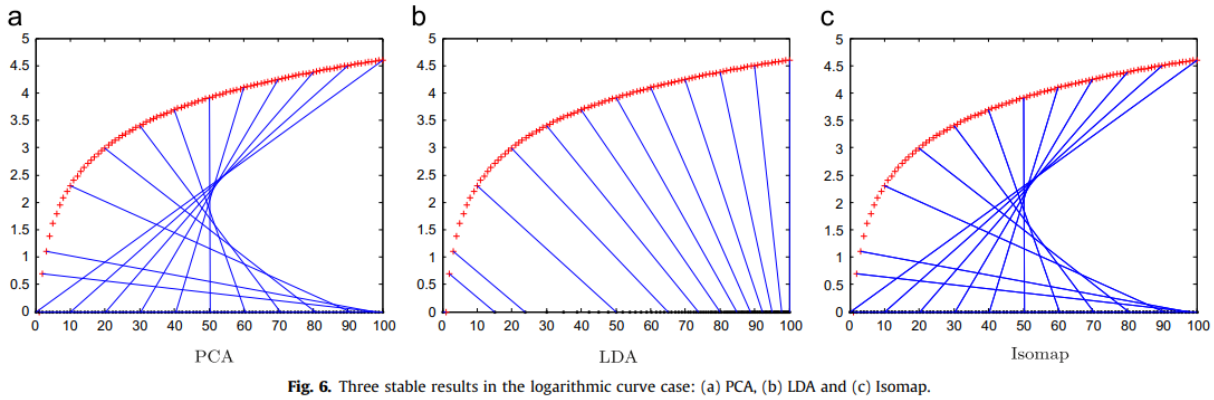


Fig. 6. Three stable results in the logarithmic curve case: (a) PCA, (b) LDA and (c) Isomap.

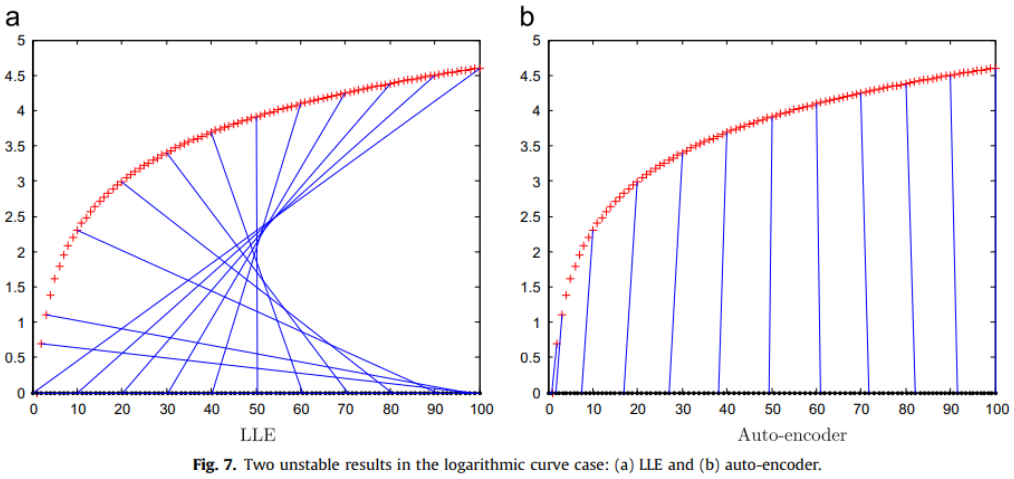


Fig. 7. Two unstable results in the logarithmic curve case: (a) LLE and (b) auto-encoder.

◆ Auto-Encoder 구현(using Torch)

Input 데이터 784(28x28)의 feature를 Latent Space Dim을 3개로 추출한 code이다.
AE의 Encoder를 4개의 hidden layer로 구성하고 Decoder도 역으로 동일하게 구성해주었다.
Decoder의 마지막은 픽셀을 0과 1 사이로 출력하기 위해 sigmoid를 적용하였다.

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

        self.encoder = nn.Sequential(
            nn.Linear(28*28, 128),
            nn.Relu(),
            nn.Linear(128, 64),
            nn.Relu(),
            nn.Linear(64,12),
            nn.Relu(),
            nn.Linear(12,3),          #3차원 특징 추출
        )

        self.decoder = nn.Sequential(
            nn.Linear(3, 12),
            nn.Relu(),
            nn.Linear(12, 64),
            nn.Relu(),
            nn.Linear(64,128),
            nn.Relu(),
            nn.Linear(128, 28*28)             #3차원 특징 추출
            nn.Sigmoid(),
        )

    def forward(self, x):
        encoded = self.encoder(x)
        decoded = self.decoder(encoded)
        return encoded, decoded
```

Optimizer는 'adam'을 사용하였으며, Loss Function으로는 MSE를 적용하였다.

```
autoencoder = Autoencoder().to(DEVICE)
optimizer = torch.optim.Adam(autoencoder.parameters(), lr=0.005)
criterion = nn.MESLoos()
```

◆ Auto-Encoder 구현(using Tensorflow)

- 코드 환경

Python Version	3.8.3
Tensor Flow Version	2.3.0

Auto-Encoder의 Latent Space 구성에 따른 성능 차이 확인을 중점적으로 진행하기 위하여 구성을 단순화하여 실험하였다.

Dataset은 MNIST를 keras를 통해 load하여 진행하였다. MNIST 데이터 셋을 불러와서 train과 test로 나눈다.(train 60K, test 10K)

```
import tensorflow as tf

mnist = tf.keras.datasets.mnist
(x_train, _), (x_test, _) = mnist.load_data()

x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.

print (x_train.shape)
print (x_test.shape)
```

Optimizer는 'adam'을 사용하였으며, Loss Function으로는 MSE를 적용하였다.
또한 epoch은 10으로 사용하여 실험을 진행하였다.

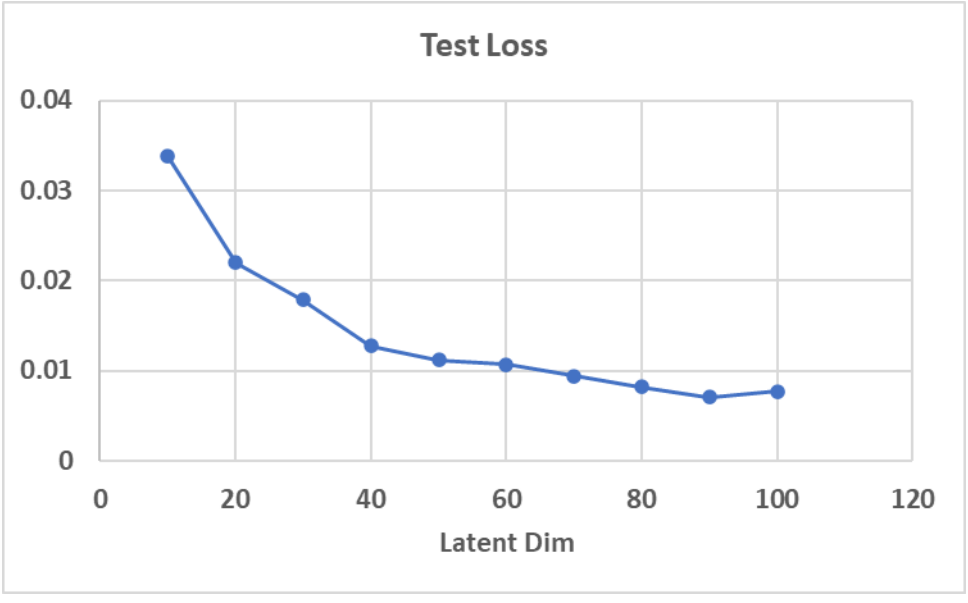
```
autoencoder.compile(optimizer='adam', loss=losses.MeanSquaredError())

autoencoder.fit(x_train, x_train,
                epochs=10,
                shuffle=True,
                validation_data=(x_test, x_test))
```

Latent Space의 차원에 대해 변경하면서 실험을 진행한다.
Latent Space 차원에 따라서 hidden layer에서의 input 데이터에 대한 압축하는 양을 조절할 수 있고, 해당 압축에 따라서 입력 이미지의 압축에 대한 데이터 손실을 확인할 수 있다.
결국 Latent 의 dim.에 의한 성능에 집중하였다.

```
latent_dim = 10

class Autoencoder(Model):
  def __init__(self, latent_dim):
    super(Autoencoder, self).__init__()
    self.latent_dim = latent_dim
    self.encoder = tf.keras.Sequential([
      layers.Flatten(),
      layers.Dense(latent_dim, activation='relu'),
    ])
    self.decoder = tf.keras.Sequential([
      layers.Dense(784, activation='relu'),
      layers.Reshape((28, 28))
    ])

  def call(self, x):
    encoded = self.encoder(x)
    decoded = self.decoder(encoded)
    return decoded

autoencoder = Autoencoder(latent_dim)
```

◆ Latent Dim에 따른 생성 이미지 비교

Latent Dim	Train Loss	Test Loss
10	0.0341	0.0338
20	0.0224	0.0220
30	0.0183	0.0179
40	0.0136	0.0128
50	0.0115	0.0112
60	0.0110	0.0107
70	0.0096	0.0094
80	0.0084	0.0082
90	0.0072	0.0071
100	0.0080	0.0077



Latent Space의 차원에 따른 Test Loss를 확인한 결과 위와 같이 Latent의 차원이 커질수록 Loss는 줄어 들고 있음을 확인 할 수 있다.

Latent Dim이 커진다는 것은 입력 이미지에 대한 압축을 적게 수행한다는 것을 의미하고 해당 과정에서 정보의 손실이 적어진다는 것을 의미한다.

반대로, Latent Dim이 작아진다는 것은 입력 이미지에 대한 압축을 많이 수행하게 되고 이는 정보의 손실을 의미한다고 볼 수 있다.

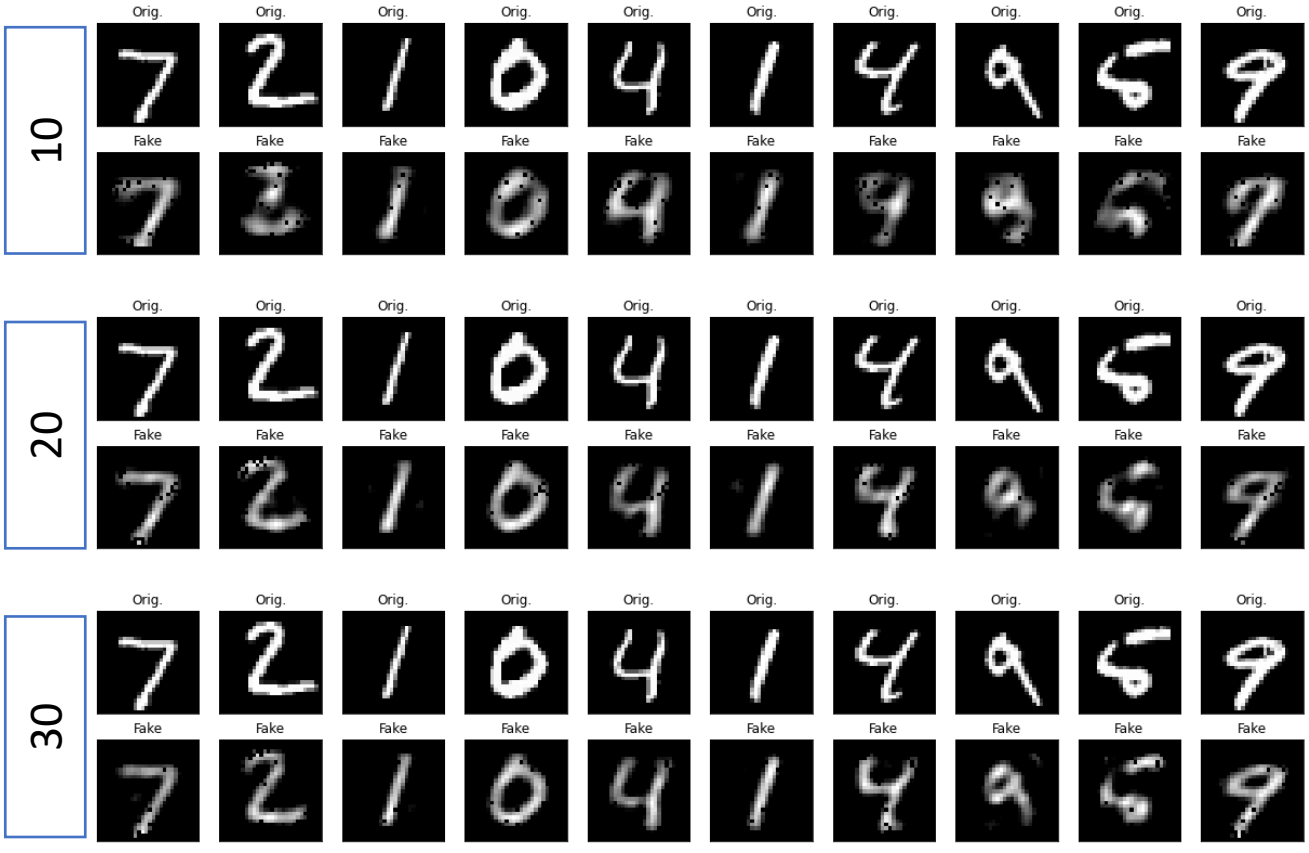
우리는 다차원의 데이터를 차원 축소가 목적이었으므로, 차원과 정보량의 손실 사이의 trade-off 관계에서 최적을 찾을 필요가 있다.

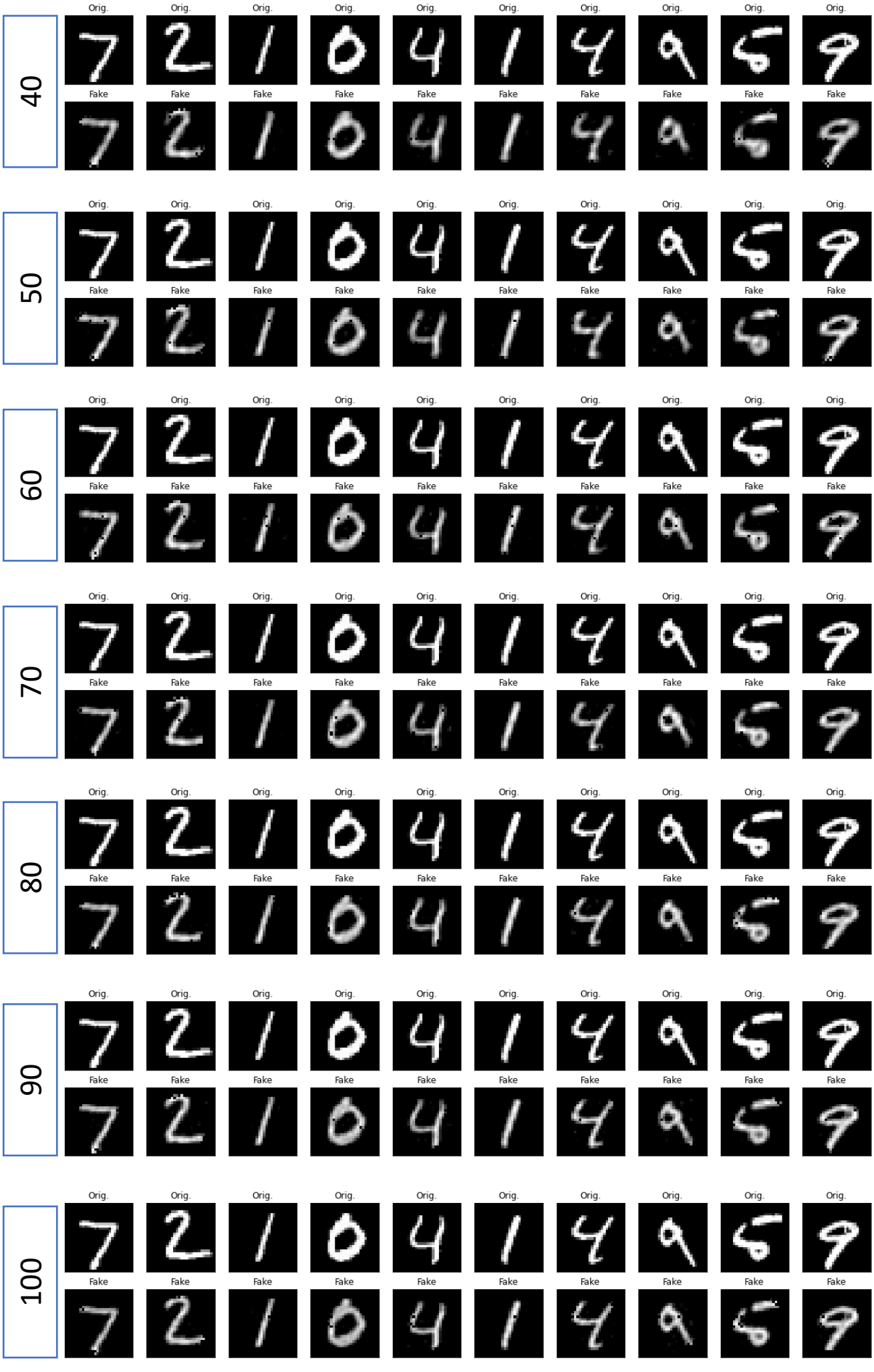
그렇기 때문에 해당 MNIST 데이터 셋에 대해서는 test loss 그래프에서 어느정도 saturation이 되었다고 판단되는 latent dim이 50 이상인 지점정도라고 판단할 수 있다.

기능이나 목적에 있어서 데이터 손실을 조금 더 가져오더라도 분류 정확도에 문제가 되지 않는다면 차원을 더 줄이는 것도 가능하다.

이어서 각 Latent Space Dim에 따른 복원된 이미지에 대해 육안을 통해 확인해본다.

* Latent Dim별 AE를 통해 복원된 MNIST 이미지 비교





Loss를 통해 확인한 것처럼 50이상부터는 눈에 띄는 이미지의 개선은 없는 것으로 보인다. 이는 데이터 셋에 대한 최적의 Latent Dim이 존재 하고 이는 hidden layer와 관계가 있음을 의미한다.

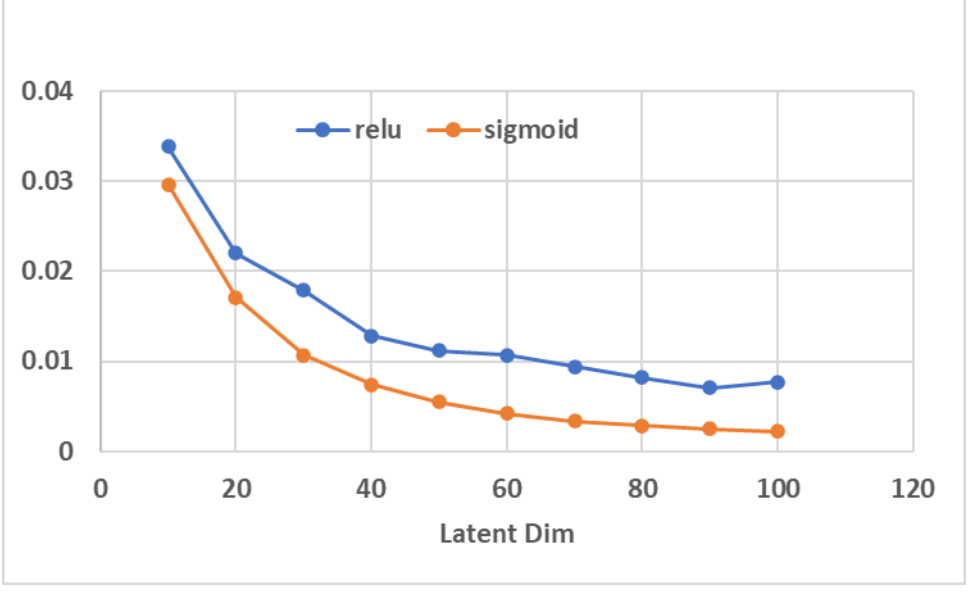
◆ Decoder의 Activation에 의한 차이(relu vs sigmoid)

앞서 AE의 Decoder는 relu를 통해 구성된 모델이었다. 사실, 정확히 말하자면 sigmoid를 적용하고 싶었으나, 복붙의 폐해로 인해 relu가 적용되어서 실험이 진행되었다. 그래서 해당 과오를 정정하기 위해 이번에는 Sigmoid를 적용하여 실험을 다시 하였다.

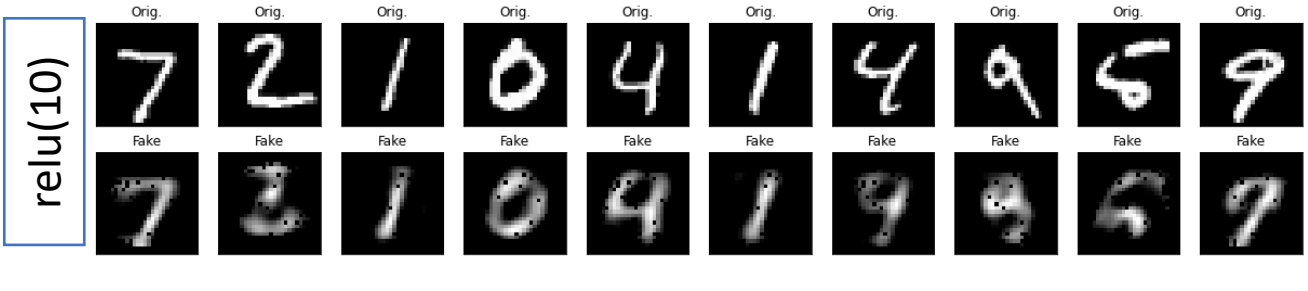
```
latent_dim = 10      # 10 ~ 100

class Autoencoder(Model):
  def __init__(self, latent_dim):
    super(Autoencoder, self).__init__()
    self.latent_dim = latent_dim
    self.encoder = tf.keras.Sequential([
      layers.Flatten(),
      layers.Dense(latent_dim, activation='relu'),
    ])
    self.decoder = tf.keras.Sequential([
      layers.Dense(784, activation='sigmoid'),
      layers.Reshape((28, 28))
    ])

  def call(self, x):
    encoded = self.encoder(x)
    decoded = self.decoder(encoded)
    return decoded

autoencoder = Autoencoder(latent_dim)
```

```
self.decoder = tf.keras.Sequential([
  layers.Dense(784, activation='relu'),
  layers.Reshape((28, 28))
```

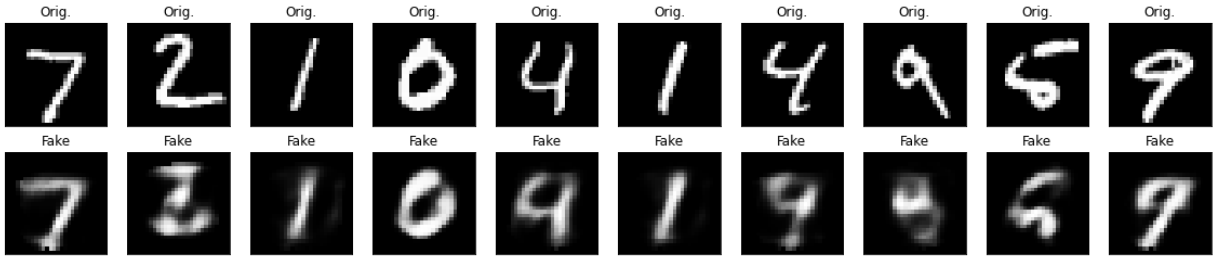
Decoder의 Activation으로 sigmoid를 사용 하였을 경우가 relu 보다 loss값이 더 적을 것을 알 수 있다. 하지만 relu를 사용하였을 때와 동일하게 미세하게 우 하향 하고는 있지만, saturation 되는 구간은 확인됨을 알 수 있다.



* 이미지를 통한 비교(relu vs sigmoid)
아래는 Latent dim을 10 기준으로 확인한 이미지의 비교이다.
Loss에서도 유사하였듯이 흐릿한 이미지인 것을 확인할 수 있다.

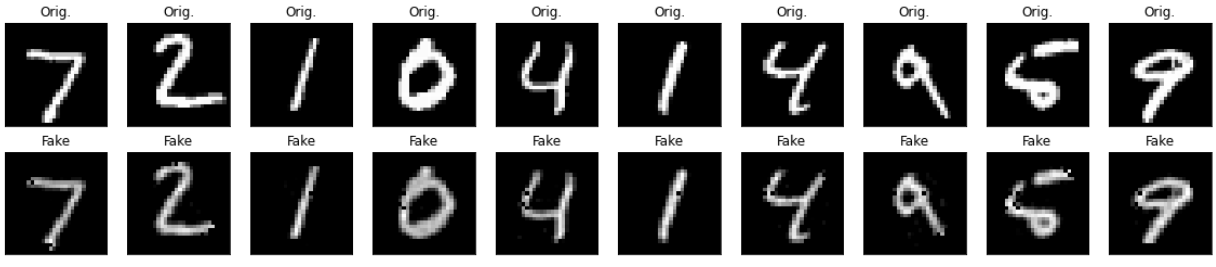


sigmoid(10)

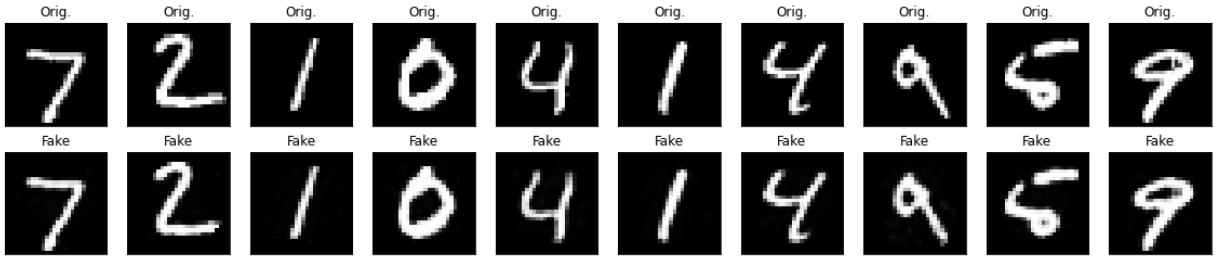


Latent Dim을 100으로 하여 압축 손실을 최소화 하여 비교 하였을 때는 이미지의 차이가 확연히 비교된다. 일정 값을 기준으로 0인지 1인지 구분하는 sigmoid를 decoder의 activation으로 적용한 경우가 더욱 선명한 이미지의 확보가 가능함을 확인할 수 있었다.

relu(100)



sigmoid(100)



◆ 마치며..

AE를 통하여 차원 축소하는 모델을 코드를 통하여 구현해보았다.
Latent Space를 통해 많은 입력 이미지에 대한 차원을 축소할 수 있음을 확인하였고,
Hyperparameter인 Latent Space의 차원을 통해 정보가 압축됨을 실험을 통해 확인하였다.
AE의 Latent Space는 절대적인 것이 아니라 데이터 셋에 따라 최적의 Latent dim이 다를 수 있음을 알 수 있었다.

그리고 의도한바는 아니지만 코드 실수로 인해 Activation 에 의한 모델의 성능 차이도 확인 할 수 있는 기회였다.