**소프트웨어융합학과**
**2020111406 이해현**

## 〔과제 내용〕

노이즈를 추가한 Mnist 데이터를 사용해서 3개의 서로다른 구조의 AucoEncoder 모델을 구현하여 이미지 분류 학습을 진행하고 결과 분석

- **모델 구성**

모델1: 오토인코더의 hidden layer에서 128개의 노드 수를 갖는 모델

모델2: 오토인코더의 encoder와 decoder에 각각 1개의 layer를 추가하여 encoder layer, 중앙 hidden layer, decoder layer에서 각각 256개, 128개, 256개의 노드 수를 갖는 모델

모델3: 합성곱 오토인코더

- loss function : MSE
- Optimizer : AdamOptimizer

## 〔실험 내용 및 결과 분석〕

### 1. 데이터 설명

실험에 사용한 MNIST 데이터는 0부터 9까지의 숫자들의 손글씨 데이터로, 해당 데이터셋에는 60000개의 train image와 10000개의 test이미지로 구성되어 있다. 각 이미지는 28x28 픽셀의 회색조 이미지 형태이고 이번 실험에서는 이미지 특징 추출을 강화하고, 과적합 방지를 위해 추가적으로 노이즈를 주어 실험을 진행하였다.

```
noise = init.normal_(torch.FloatTensor(batch_size, 1, 28, 28), 0, 0.1)
```

해당 코드와 같이 MNIST 이미지와 같은 크기의 텐서를 생성하고, 그 값을 평균이 0이고 표준편차가 0.1인 정규분포를 따르는 랜덤 값으로 설정한 뒤 MNIST이미지에 추가하는 방식으로 노이즈를 부여하였다.

### 2. 각 모델 별 네트워크 구조

### 2-1. 모델1 구조

```
# 3-1. 오토인코더의 hidden layer에서 128개의 노드 수를 갖는 모델
class Autoencoder(nn.Module):
  def __init__(self):
    super(Autoencoder, self).__init__()
    self.encoder = nn.Linear(28*28, 128) #입력을 128개 노드로 압축
    self.decoder = nn.Linear(128, 28*28) #128개 노드를 다시 원래 크기로 확장

  def forward(self, x):
    x = x.view(batch_size, -1)
    encoded = self.encoder(x)
    out = self.decoder(encoded).view(batch_size,1,28,28)
    return out
```

그림 1 모델 1 구현 코드

모델 1의 경우, 오토인코더의 hidden layer에서 128개의 노드 수를 갖는 모델로, encoder에서 입력데이터를 128개의 노드로 압축하여 hidden layer를 부여하였고, decoder에서 다시 128개의 노드를 원래 크기(28x28)로 확장하는 구조로 구현하였다.

### 2-2. 모델 2 구조

```
# 3-2. 모델 구현
class Autoencoder(nn.Module):
  def __init__(self):
    super(Autoencoder, self).__init__()
    # Encoder layers
    self.encoder = nn.Sequential(
        nn.Linear(28*28, 256),
        nn.ReLU(),
        nn.Linear(256, 128)

    )
    # Decoder layers
    self.decoder = nn.Sequential(
        nn.Linear(128, 256),
        nn.ReLU(),
        nn.Linear(256, 28*28)
    )

  def forward(self, x):
    x = x.view(batch_size, -1)
    encoded = self.encoder(x)
    out = self.decoder(encoded).view(batch_size,1,28,28)
    return out
```

그림 2 모델2 구현 코드

모델2의 경우, 오토인코더의 encoder와 decoder에 각각 1개의 layer를 추가하여 encoder layer, 중앙 hidden layer, decoder layer에서 각각 256개, 128개, 256개의 노드 수를 갖는 모델이므로 Encoder의 마지막 Linear 레이어에서 output 값을 128로 설정하고, Decoder의 첫 번째 Linear layer에서 input값을 128로 설정하는 방식으로 hidden layer를 구성하였다.

Decoder의 마지막 Linear layer에서 다시 256개의 노드수를 원래 이미지의 픽셀 크기수 만큼 돌려놓는 방식으로 오토인코더를 구현하였다.

### 2-3. 모델 3 구조

모델3: 합성곱 오토인코더

```
class Encoder(nn.Module):
  def __init__(self):
    super(Encoder, self).__init__()
    self.layer1 = nn.Sequential(
        nn.Conv2d(1,16,3, padding=1),
        nn.ReLU(),
        nn.BatchNorm2d(16),
        nn.Conv2d(16, 32, 3, padding=1),
        nn.ReLU(),
        nn.BatchNorm2d(32),
        nn.Conv2d(32, 64, 3, padding=1),
        nn.ReLU(),
        nn.BatchNorm2d(64),
        nn.MaxPool2d(2,2)
    )
    self.layer2 = nn.Sequential(
        nn.Conv2d(64, 128, 3, padding=1),
        nn.ReLU(),
        nn.BatchNorm2d(128),
        nn.MaxPool2d(2,2),
        nn.Conv2d(128, 256, 3, padding=1),
        nn.ReLU()
    )

  def forward(self, x):
    out = self.layer1(x)
    out = self.layer2(out)
    out = out.view(batch_size, -1)
    return out
```

그림 3 모델3 Encoder 구현

```
class Decoder(nn.Module):
  def __init__(self):
    super(Decoder, self).__init__()
    self.layer1 = nn.Sequential(
        nn.ConvTranspose2d(256,128,3,2,1,1),
        nn.ReLU(),
        nn.BatchNorm2d(128),
        nn.Conv2d(128,64,3,1,1),
        nn.ReLU(),
        nn.BatchNorm2d(64),
        )

    self.layer2 = nn.Sequential(
        nn.ConvTranspose2d(64, 16, 3, 1,1),
        nn.ReLU(),
        nn.BatchNorm2d(16),
        nn.ConvTranspose2d(16,1,3,2,1,1),
        nn.ReLU()
    )

  def forward(self, x):
    out = x.view(batch_size,256, 7, 7)
    out = self.layer1(out)
    out = self.layer2(out)
    return out
```

그림 4 모델3 Decoder 구현

모델3의 경우, 합성곱 연산을 수행하는 Encoder와 인코더와 대칭되는 형태로 전치 합성곱 연산을 진행하는 Decoder로 구성된 AutoEncoder로 구현하였다.
Encoder에서는 2개의 레이어로 구성되어있고, 첫 번째 레이어(layer1은) 3개의 합성곱 레이어로 구성되어있다. 각 합성곱 레이어 다음에는 ReLU()활성화 함수와 BatchNorm2d()를 통한 배치정규화를 진행하도록 하였다. 또한, 첫 번째 레이어 마지막에서 2x2 Max pooling을 사용하였다. 두 번째 레이어에서는 첫번째 합성곱 레이어와 BatchNorm 레이어를 통해 레이어 채널수를 128로 증가시켰고, MaxPooling 레이어를 거친 뒤 채널 수를 256으로 증가시키는 마지막 합성곱 레이어와 ReLU 활성화 함수를 통해 output을 출력하는 구조로 구현하였다.

Decoder 또한 2개의 레이어로 구성되어있고, 첫 번째 레이어(layer1)의 전치 합성곱 레이어에서는 256개의 입력 채널을 128개의 출력 채널로 변환하며, 커널 크기는 3x3, 스트라이드는 2, 패딩은 1, 출력 패딩도 1로 주어 차원을 확장하도록 하였다. 그 다음으로 ReLU 활성화 함수와 배치 정규화를 거쳐 128개의 입력 채널을 64개의 출력채널로 변환하는 일반 합성곱 레이어를 배치해였다.

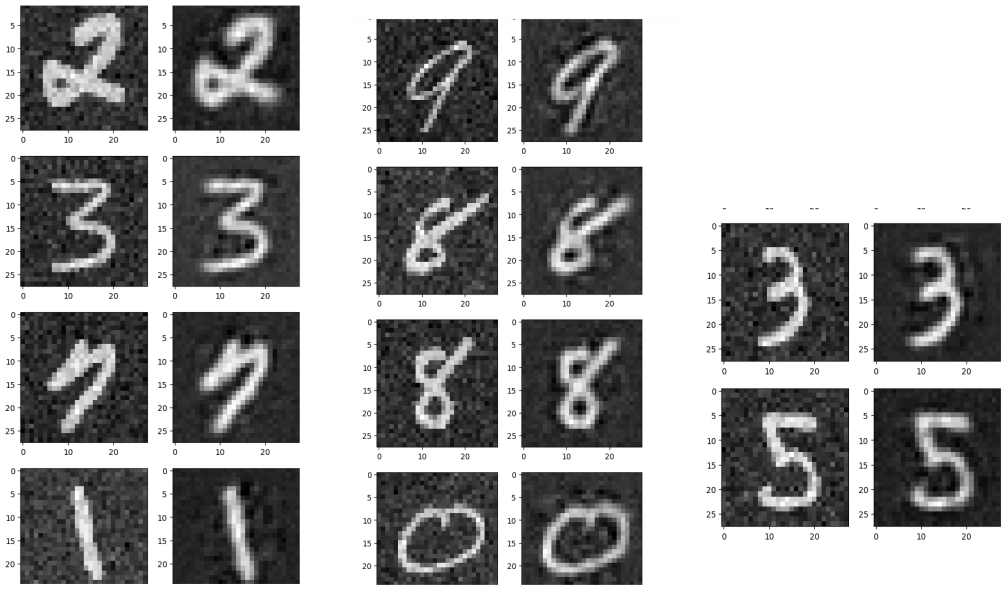
두 번째 레이어는 4개의 입력 채널을 16개의 출력 채널*(커널 크기 3x3, 스트라이드와 패딩 1)* 로 변환하는 전치 합성곱 레이어, ReLU, 배치정규화 레이어, 16개의 입력 채널을 1개의 출력 채널로 변환하는 전치 합성곱레이어, ReLU 순으로 구성하였다.

## 3. 결과 분석

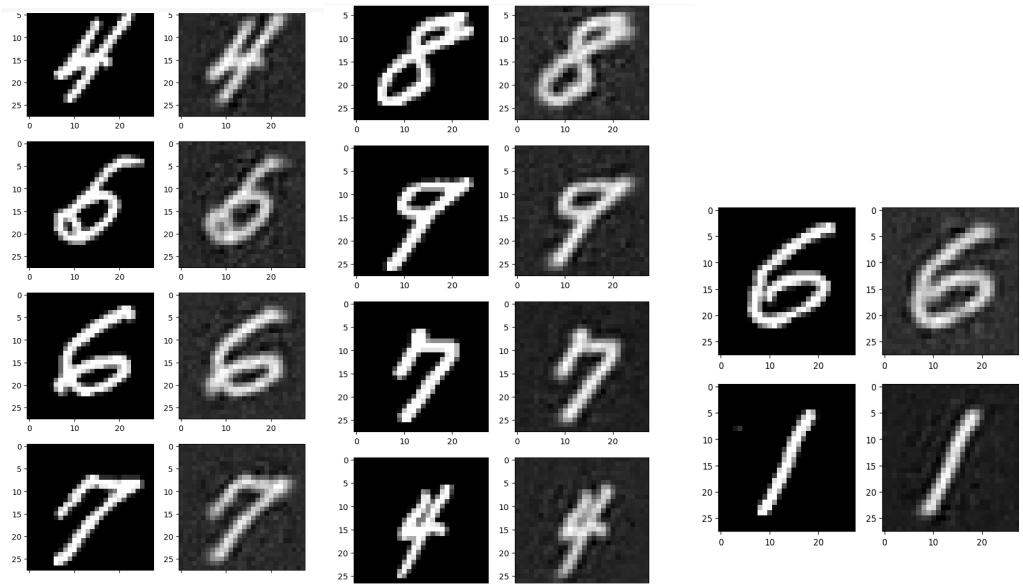
### 3-1. 모델 1

```
tensor(0.1378, device='cuda:0', grad_fn=<MseLossBackward0>)
tensor(0.0408, device='cuda:0', grad_fn=<MseLossBackward0>)
tensor(0.0279, device='cuda:0', grad_fn=<MseLossBackward0>)
tensor(0.0224, device='cuda:0', grad_fn=<MseLossBackward0>)
tensor(0.0194, device='cuda:0', grad_fn=<MseLossBackward0>)
tensor(0.0184, device='cuda:0', grad_fn=<MseLossBackward0>)
tensor(0.0172, device='cuda:0', grad_fn=<MseLossBackward0>)
tensor(0.0163, device='cuda:0', grad_fn=<MseLossBackward0>)
tensor(0.0157, device='cuda:0', grad_fn=<MseLossBackward0>)
tensor(0.0150, device='cuda:0', grad_fn=<MseLossBackward0>)
```

그림 5 모델 1 Loss 값 기록



모델 1 - train data를 통한 이미지 분류 결과 확인 (label 이미지, output 이미지)



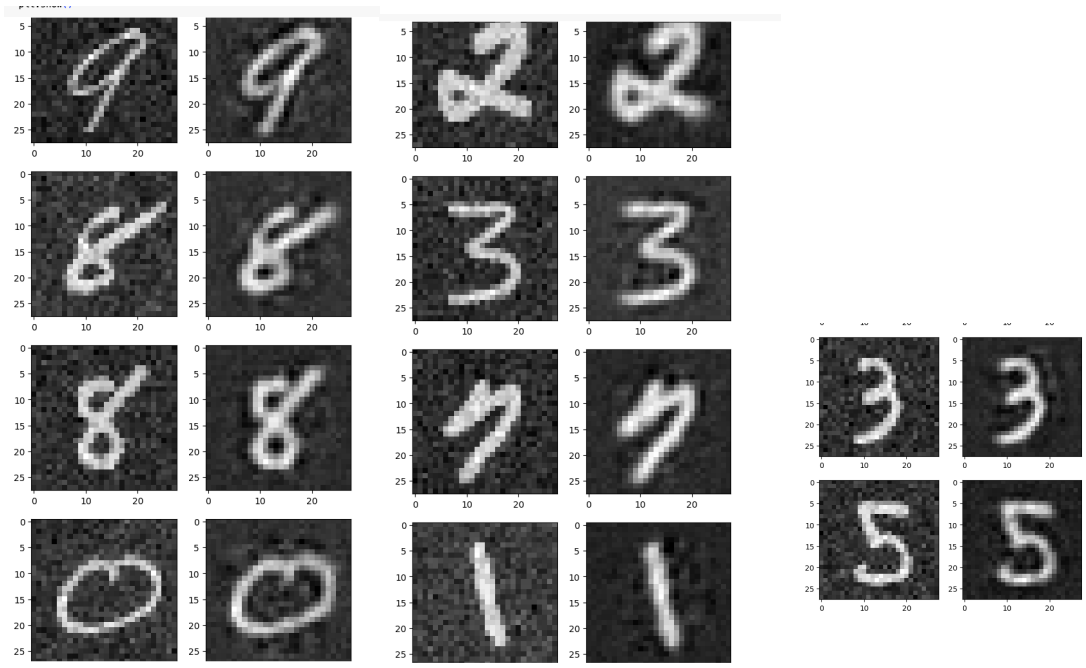
모델 1 - test data를 사용한 prediction 결과 (label 이미지, output 이미지)

모델1의 경우, loss 값은 약 0.015 까지 감소하였고, train data를 통한 이미지 분류 결과와 test data를 사용한 prediction 결과 이미지를 확인 해본 결과, 데이터에 노이즈를 주었음에도 불구하고 트레인 이미지 10개, 테스트 이미지 10개 모두 레이블 데이터와 같은 숫자 이미지를 정확하게 분류해냄을 확인할 수 있었다.

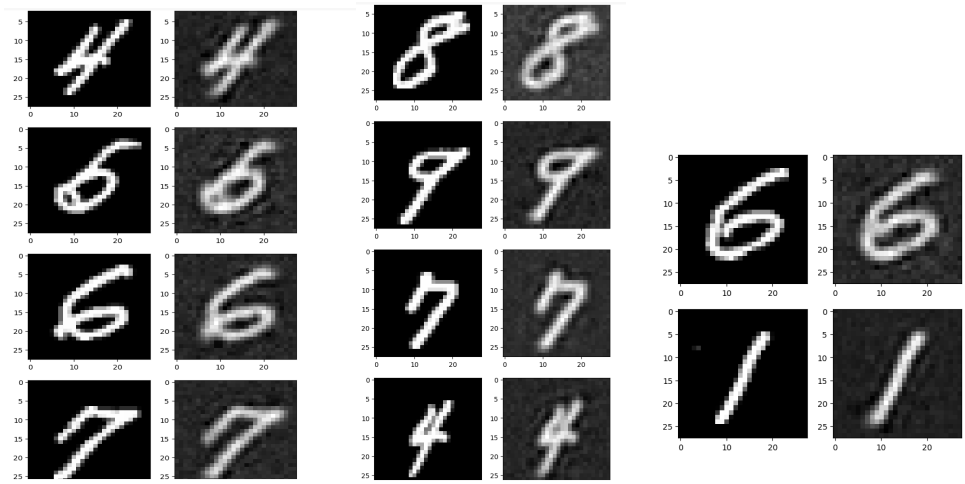
## 모델 2

```
tensor(0.1244, device='cuda:0', grad_fn=<MseLossBackward0>)
tensor(0.0363, device='cuda:0', grad_fn=<MseLossBackward0>)
tensor(0.0264, device='cuda:0', grad_fn=<MseLossBackward0>)
tensor(0.0226, device='cuda:0', grad_fn=<MseLossBackward0>)
tensor(0.0202, device='cuda:0', grad_fn=<MseLossBackward0>)
tensor(0.0189, device='cuda:0', grad_fn=<MseLossBackward0>)
tensor(0.0180, device='cuda:0', grad_fn=<MseLossBackward0>)
tensor(0.0172, device='cuda:0', grad_fn=<MseLossBackward0>)
tensor(0.0170, device='cuda:0', grad_fn=<MseLossBackward0>)
tensor(0.0160, device='cuda:0', grad_fn=<MseLossBackward0>)
```

그림 12 모델 2- loss 값 기록



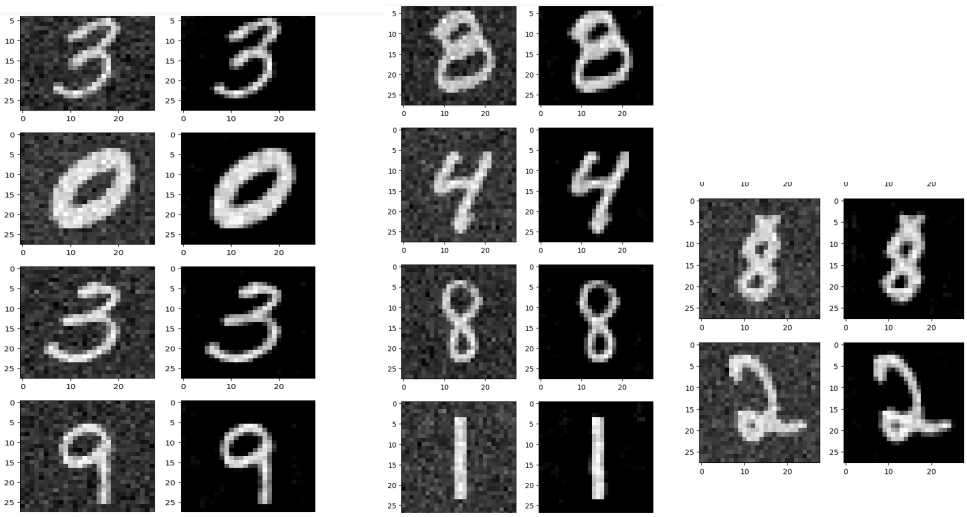
모델 2 - train data를 통한 이미지 분류 결과 확인 (label 이미지, output 이미지)



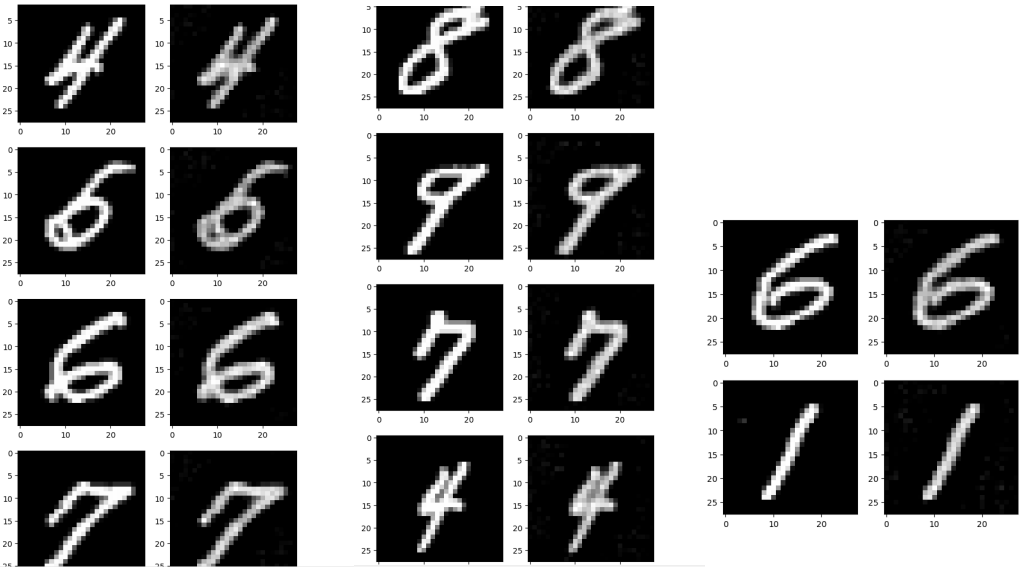
모델 2 - test data를 사용한 prediction 결과 (label 이미지, output 이미지)

모델2 의 경우, loss 값은 약 0.015 까지 감소하였고, train data를 통한 이미지 분류 결과와 test data를 사용한 prediction 결과 이미지를 확인 해본 결과, 모델 2 또한 트레인 이미지 10개, 테스트 이미지 10개 모두 레이블 데이터와 같은 숫자 이미지를 정확하게 분류해 냄을 확인할 수 있었다.

## 모델 3



모델 3 - train data를 통한 이미지 분류 결과 확인 (label 이미지, output 이미지)



모델 3 - test data를 사용한 prediction 결과 (label 이미지, output 이미지)

모델3 또한 loss 값은 약 0.01 까지 감소하였고, train data를 통한 이미지 분류 결과와 test data를 사용한 prediction 결과 이미지를 확인 해본 결과 또한 트레인 이미지 10개, 테스트 이미지 10개 모두 레이블 데이터와 같은 숫자 이미지를 정확하게 분류해냄을 확인할 수 있었다.

## 결론

모델1, 모델2, 모델3 순으로 복잡한 구조로 구성되었지만 세 모델 모두 MNIST 데이터 분류 문제에서는 비슷한 성능을 보였다. 이는 MNIST 데이터셋이 비교적 단순한 흑백 손글씨 이미지이기 때문에 hidden layer가 1개밖에 없는 모델1과 비교적 복잡한 구조를 가지고 있는 합성곱 오토인코더인 모델3이 비슷한 결과를 보여준 것 같다. 따라서, 복잡한 모델이 항상 최적의 결과를 보장하는 것은 아니기 때문에

데이터셋의 특성이나 복잡도와 같은 요소들도 고려하여 모델을 선택하는 것이 중요함을 알 수 있었다.