

Autoencoder & Transposed Convolution

2017.12.02

최건호

01

Autoencoder

- 정의
- 이유
- 활용

02

Convolution
Transposed

- 필요성
- 연산과정
- 실제활용

03

Convolutional
Autoencoder

- 전체적 구조
- Denoising
CAE

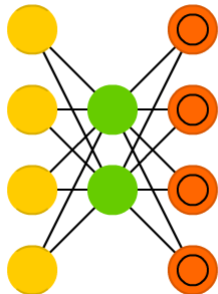
04

Variational
Autoencoder

- Intuition
 - Variational
Inference
-

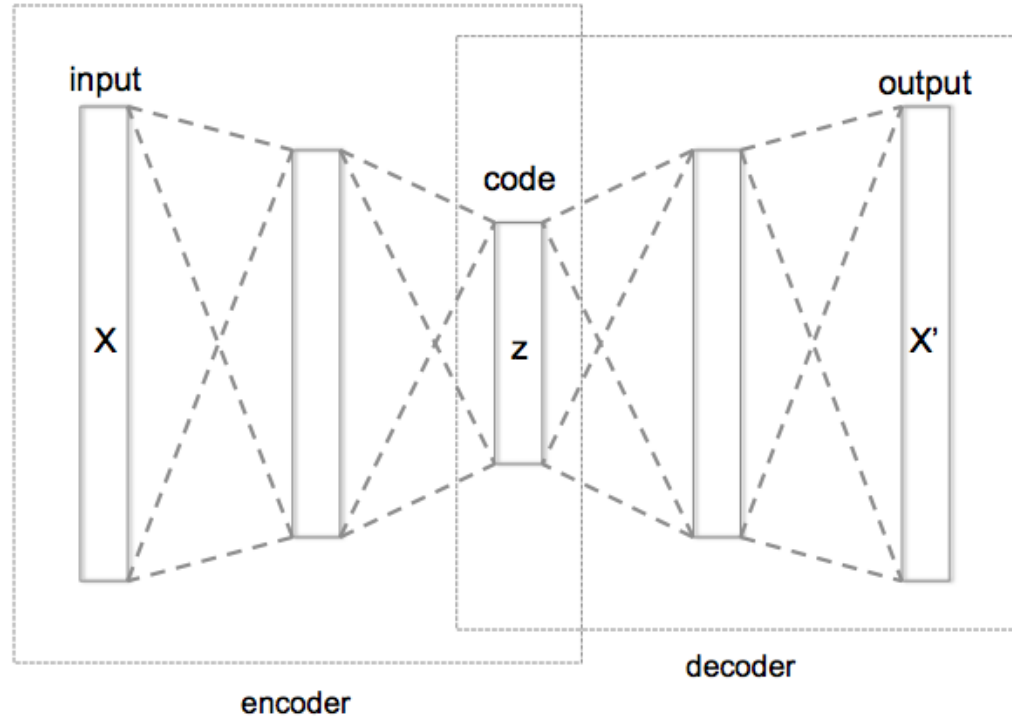
Autoencoder

Auto Encoder (AE)

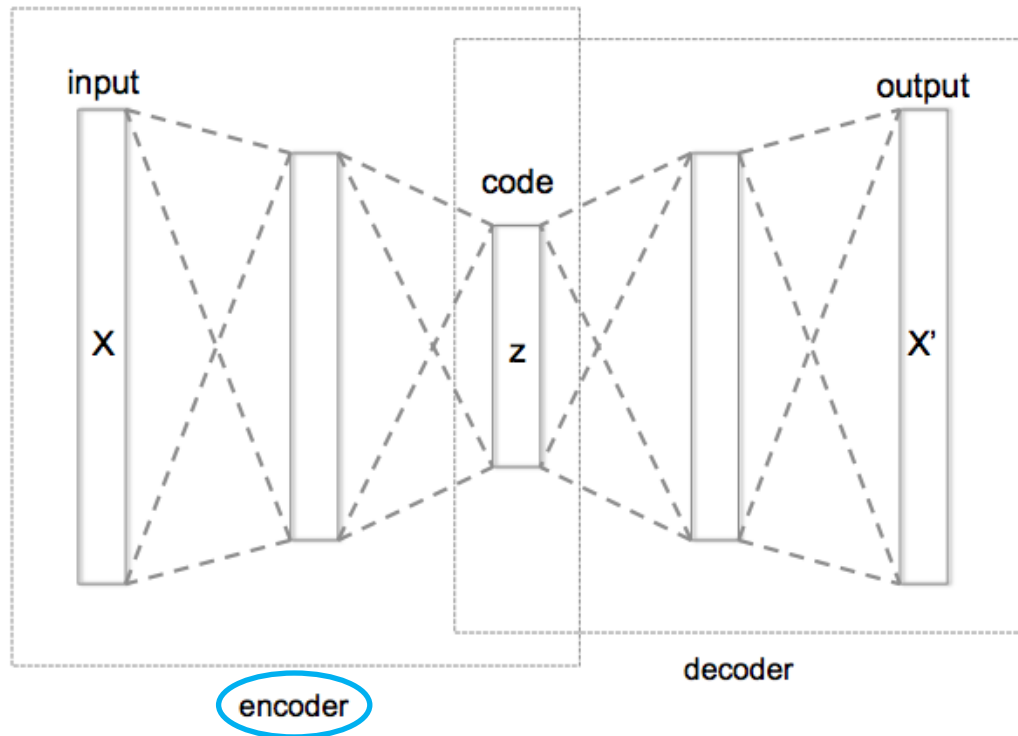


- Unsupervised Learning
- Feature Learning
- Representation Learning
- Efficient Coding
- Dimensionality Reduction

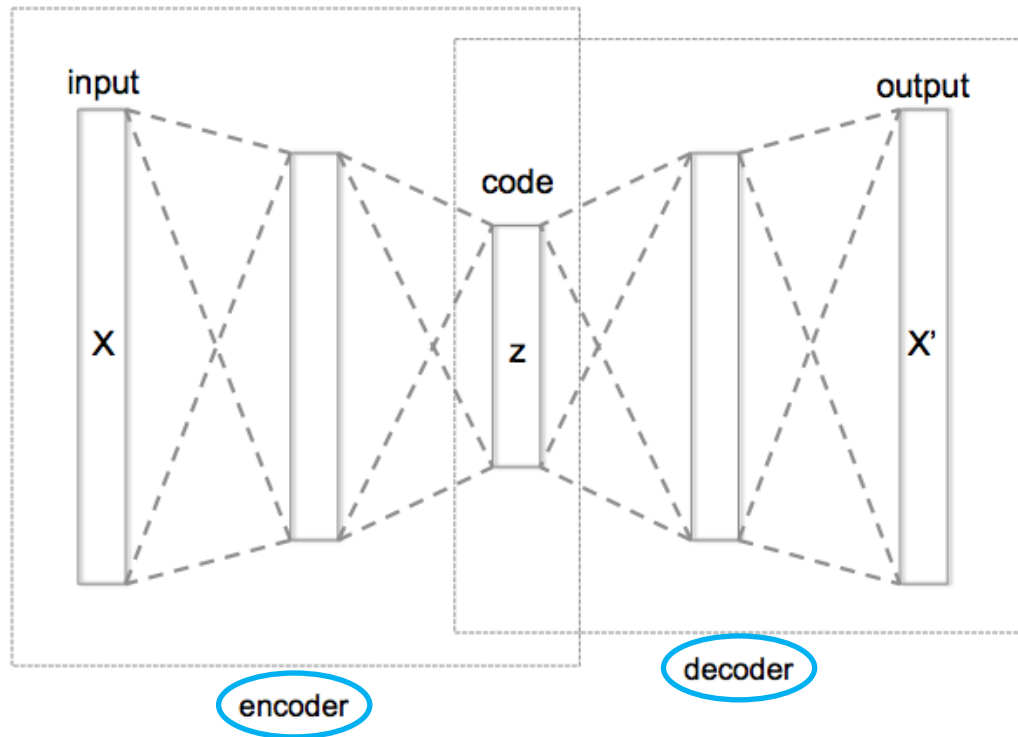
Autoencoder



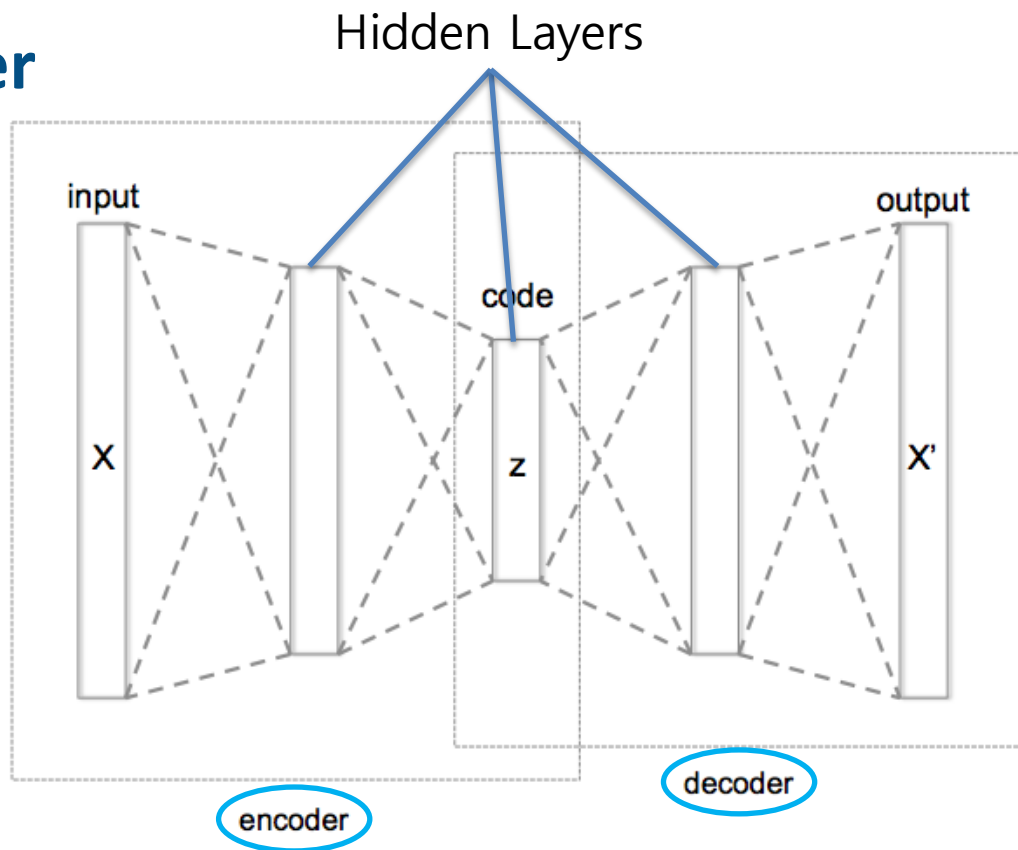
Autoencoder



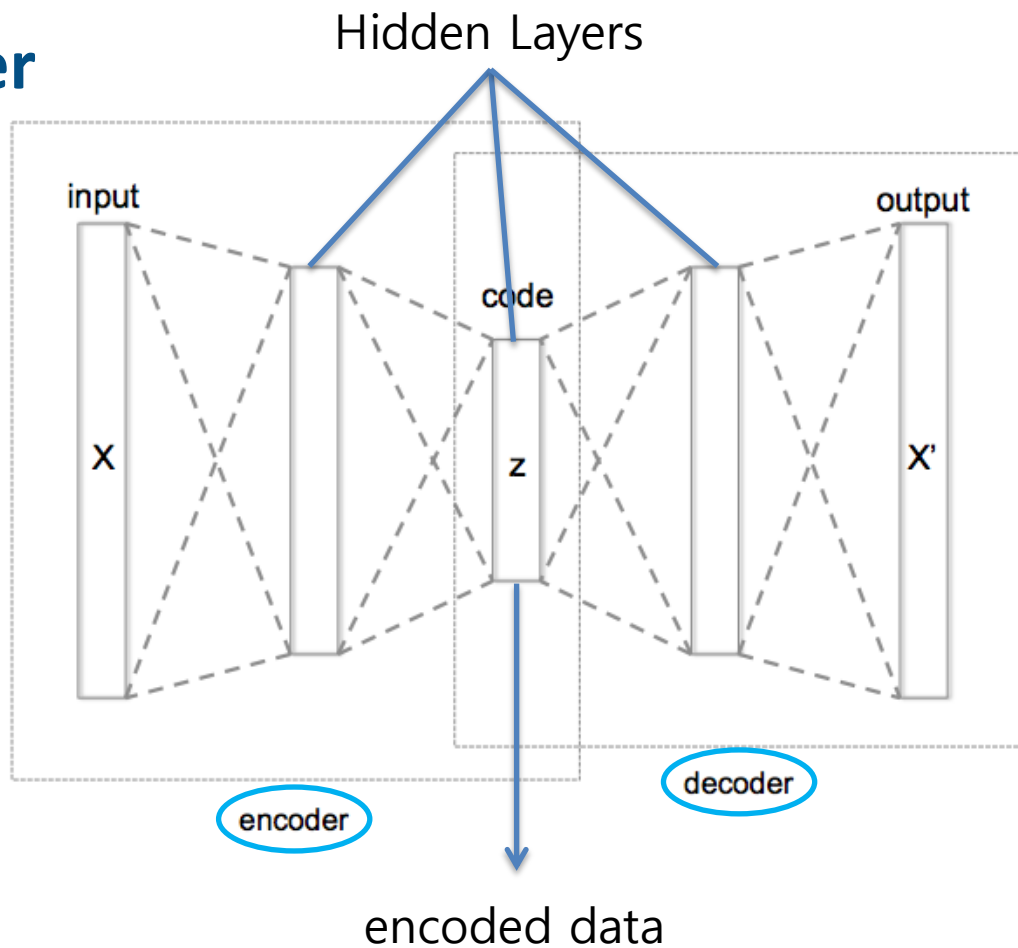
Autoencoder



Autoencoder

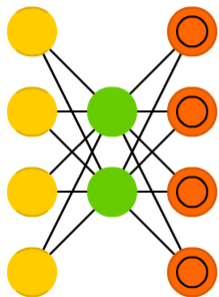


Autoencoder

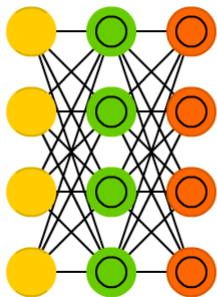


Autoencoder

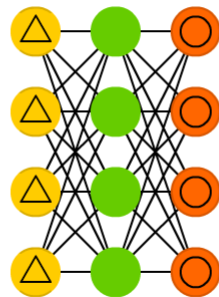
Auto Encoder (AE)



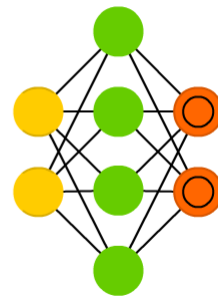
Variational AE (VAE)



Denoising AE (DAE)



Sparse AE (SAE)



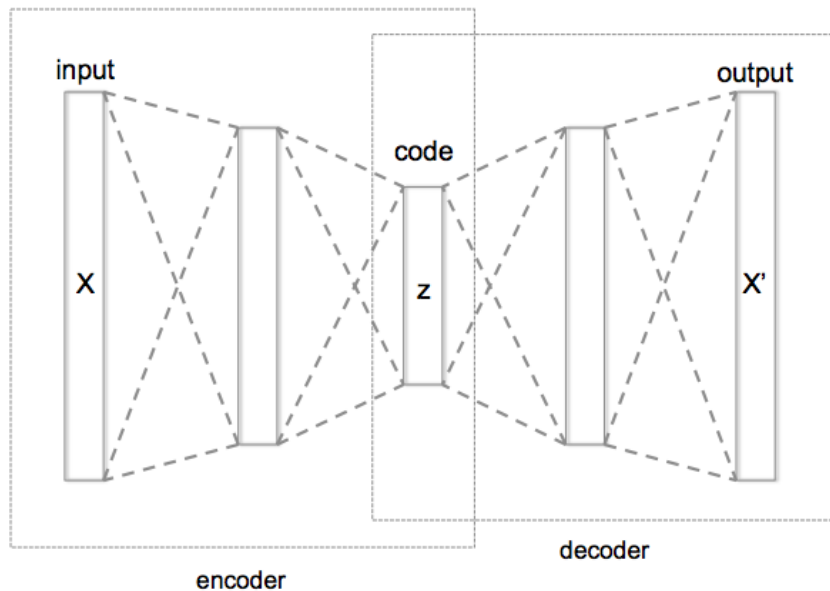
다양한 형태가 있음

Autoencoder

Loss는 어떻게 계산할까?

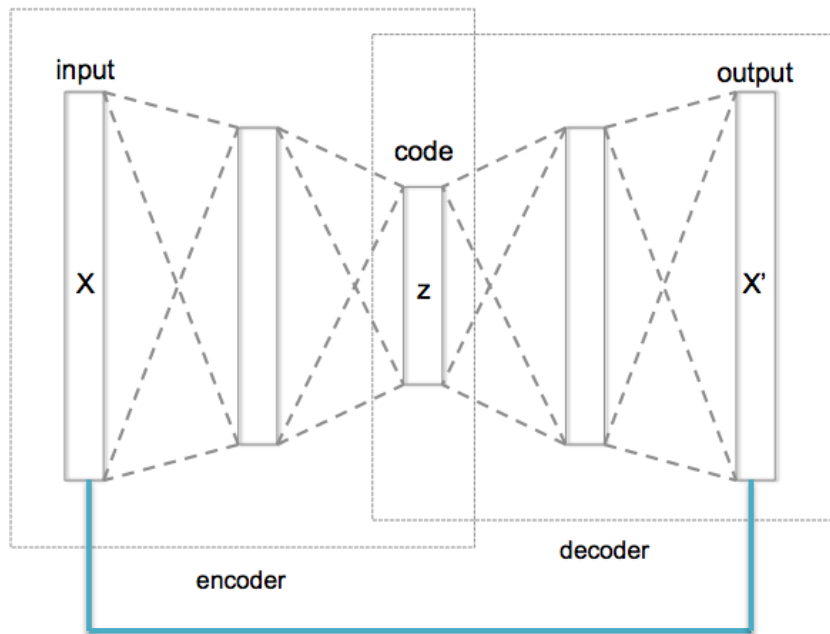
Autoencoder

Loss는 어떻게 계산할까?



Autoencoder

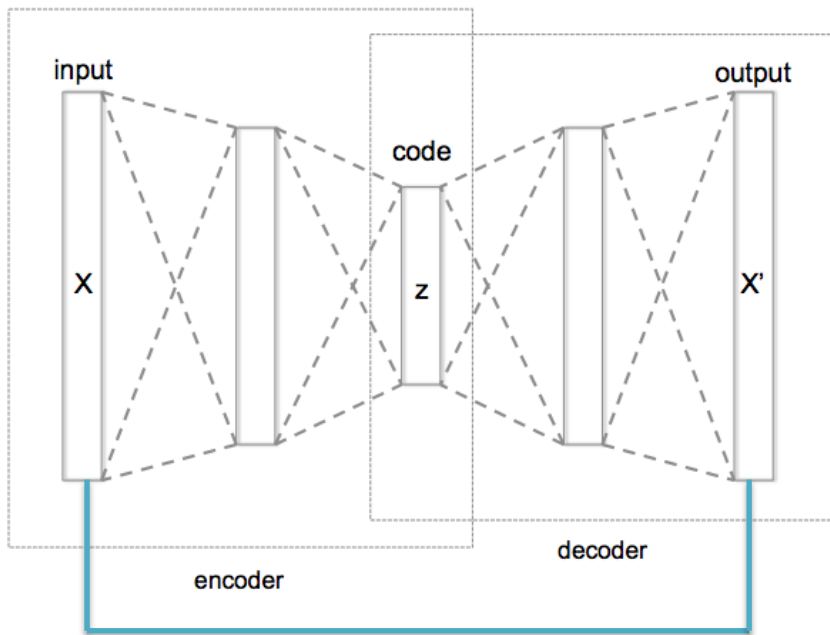
Loss는 어떻게 계산할까?



$$\|x - x'\|$$

Autoencoder

Loss는 어떻게 계산할까?

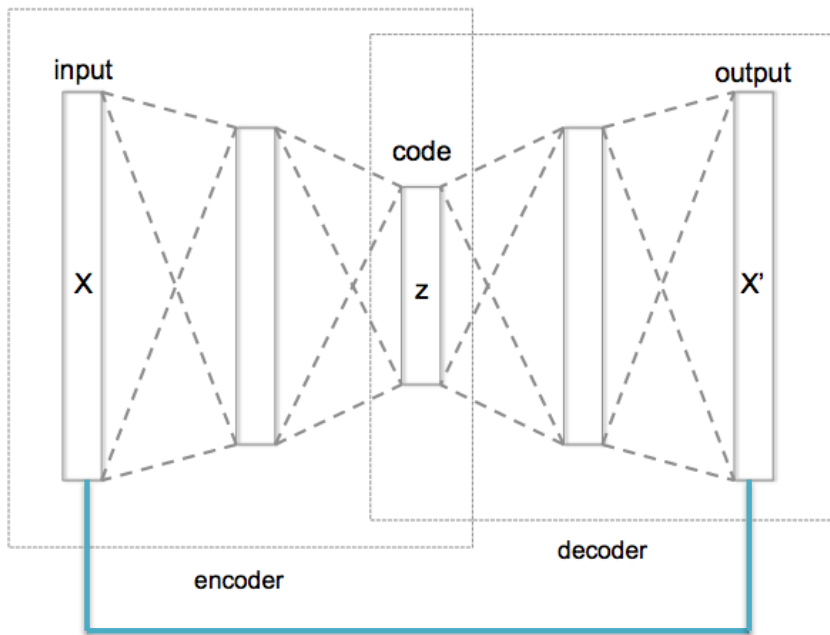


원본 데이터 x 자체가 라벨의 역할을 하여 reconstructed 된 데이터 또는 decoded 데이터와의 차이로 loss를 계산.

$$\|x - x'\|$$

Autoencoder

Loss는 어떻게 계산할까?



$$\|x - x'\|$$

원본 데이터 x 자체가 라벨의 역할을 하여 reconstructed 된 데이터 또는 decoded 데이터와의 차이로 loss를 계산.

L1 loss나 L2 loss를 많이 사용함

```
class Autoencoder(nn.Module):
    def __init__(self):
        super(Autoencoder, self).__init__()
        self.encoder = nn.Linear(28*28, 50)
        self.decoder = nn.Linear(50, 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)

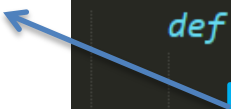
        return out

model = Autoencoder().cuda()
```

Autoencoder

MNIST data

MNIST 데이터는 28x28
이기 때문에 784개의
숫자를 50짜리 latent
feature로 encode



```
class Autoencoder(nn.Module):
    def __init__(self):
        super(Autoencoder, self).init ()
        self.encoder = nn.Linear(28*28, 50)
        self.decoder = nn.Linear(50, 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

model = Autoencoder().cuda()
```


Autoencoder

MNIST data

MNIST 데이터는 28x28
이기 때문에 784개의
숫자를 50짜리 latent
feature로 encode

50개의 latent feature에서
다시 784개로 decode

```
class Autoencoder(nn.Module):
    def __init__(self):
        super(Autoencoder, self). init ()
        self.encoder = nn.Linear(28*28, 50)
        self.decoder = nn.Linear(50, 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

model = Autoencoder().cuda()
```

Autoencoder

MNIST data

MNIST 데이터는 28x28
이기 때문에 784개의
숫자를 50짜리 latent
feature로 encode

50개의 latent feature에서
다시 784개로 decode

모델 구조에 알맞게 데이
터를 reshape하여 전달

```
class Autoencoder(nn.Module):  
    def __init__(self):  
        super(Autoencoder, self). init ()  
        self.encoder = nn.Linear(28*28, 50)  
        self.decoder = nn.Linear(50, 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  
  
model = Autoencoder().cuda()
```

Autoencoder

```
loss_func = nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

loss_arr = []

for i in range(num_epoch):
    for j, [image, label] in enumerate(train_loader):
        x = Variable(image).cuda()

        optimizer.zero_grad()
        output = model.forward(x)
        loss = loss_func(output, x)
        loss.backward()
        optimizer.step()

    if j % 1000 == 0:
        print(loss)
        loss_arr.append(loss.cpu().data.numpy()[0])
```

Autoencoder

Loss function $\frac{0}{2}$ Mean Squared Error

```
loss_func = nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

loss_arr = []

for i in range(num_epoch):
    for j, [image, label] in enumerate(train_loader):
        x = Variable(image).cuda()

        optimizer.zero_grad()
        output = model.forward(x)
        loss = loss_func(output, x)
        loss.backward()
        optimizer.step()

    if j % 1000 == 0:
        print(loss)
        loss_arr.append(loss.cpu().data.numpy()[0])
```

Autoencoder

Loss function은 Mean Squared Error

원본 데이터 x와 모델의 output간의 차이를 통해 loss를 구하고 Back Prop.

```
loss_func = nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

loss_arr = []

for i in range(num_epoch):
    for j,[image,label] in enumerate(train_loader):
        x = Variable(image).cuda()

        optimizer.zero_grad()
        output = model.forward(x)
        loss = loss_func(output,x)
        loss.backward()
        optimizer.step()

    if j % 1000 == 0:
        print(loss)
        loss_arr.append(loss.cpu().data.numpy()[0])
```

Convolution Transposed

이미지 데이터를 사용할 때는 Encoding, Decoding을 Fully connected layer 말고 Convolution으로 할 수는 없을까?

Convolution Transposed

이미지 데이터를 사용할 때는 Encoding, Decoding을 Fully connected layer 말고 Convolution으로 할 수는 없을까?



Encoding 부분은 기존의 convolution 연산으로 가능한데 Decoding 부분은 어떻게 하지?

Convolution Transposed

이미지 데이터를 사용할 때는 Encoding, Decoding을 Fully connected layer 말고 Convolution으로 할 수는 없을까?

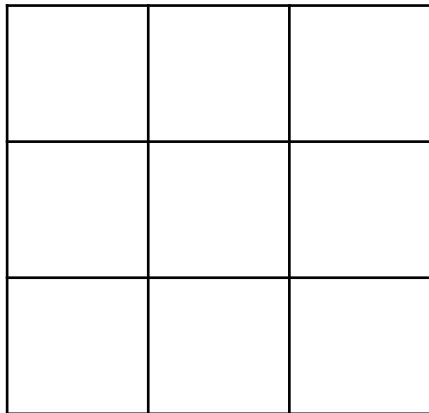


Encoding 부분은 기존의 convolution 연산으로 가능한데 Decoding 부분은 어떻게 하지?



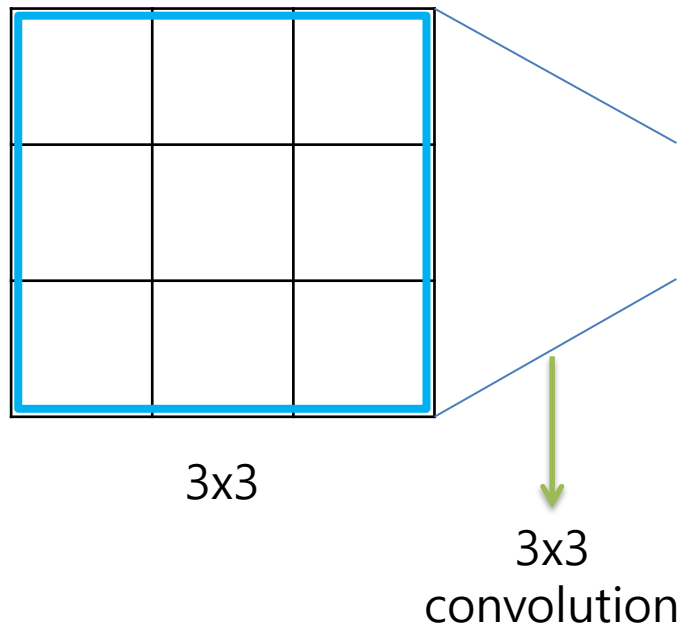
Transposed Convolution!!

Convolution Transposed

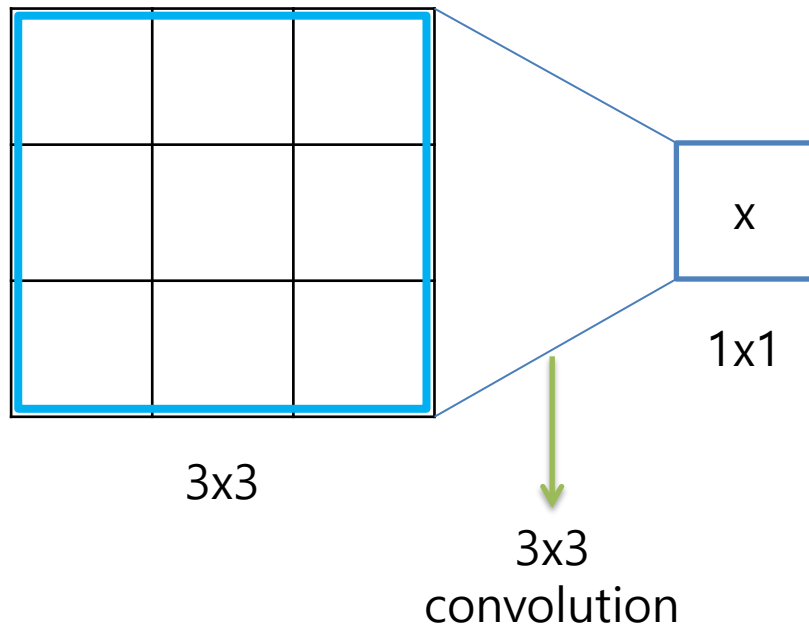


3x3

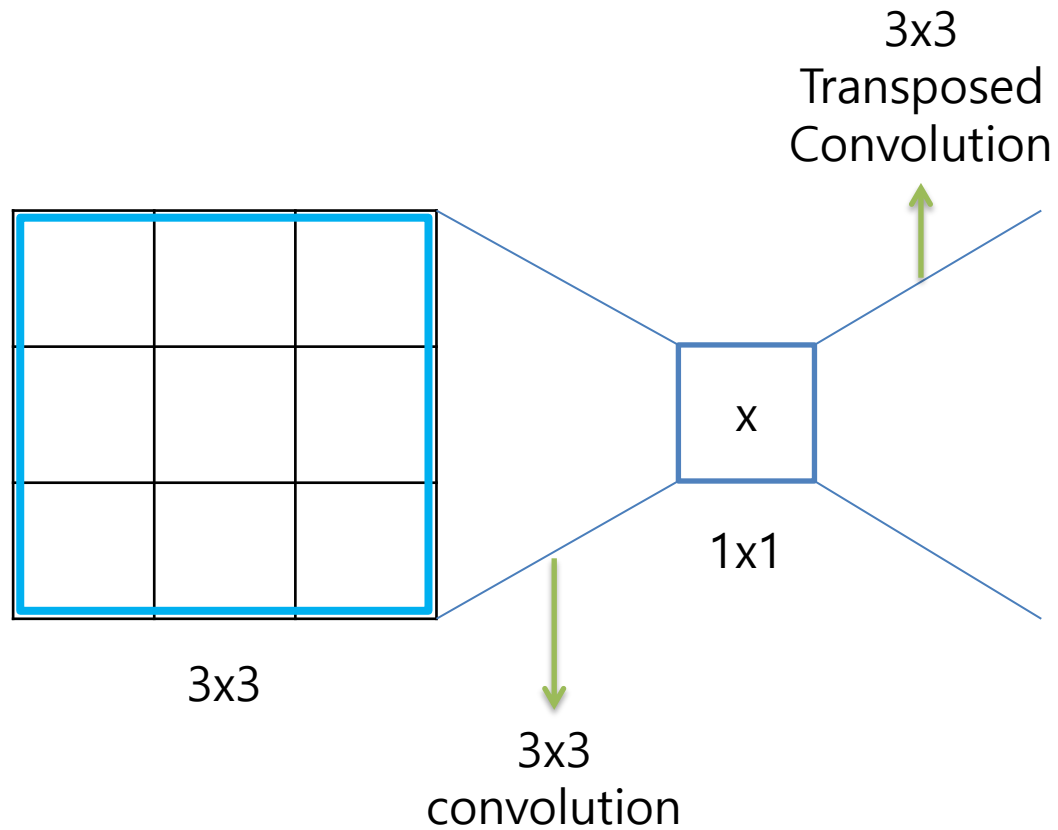
Convolution Transposed



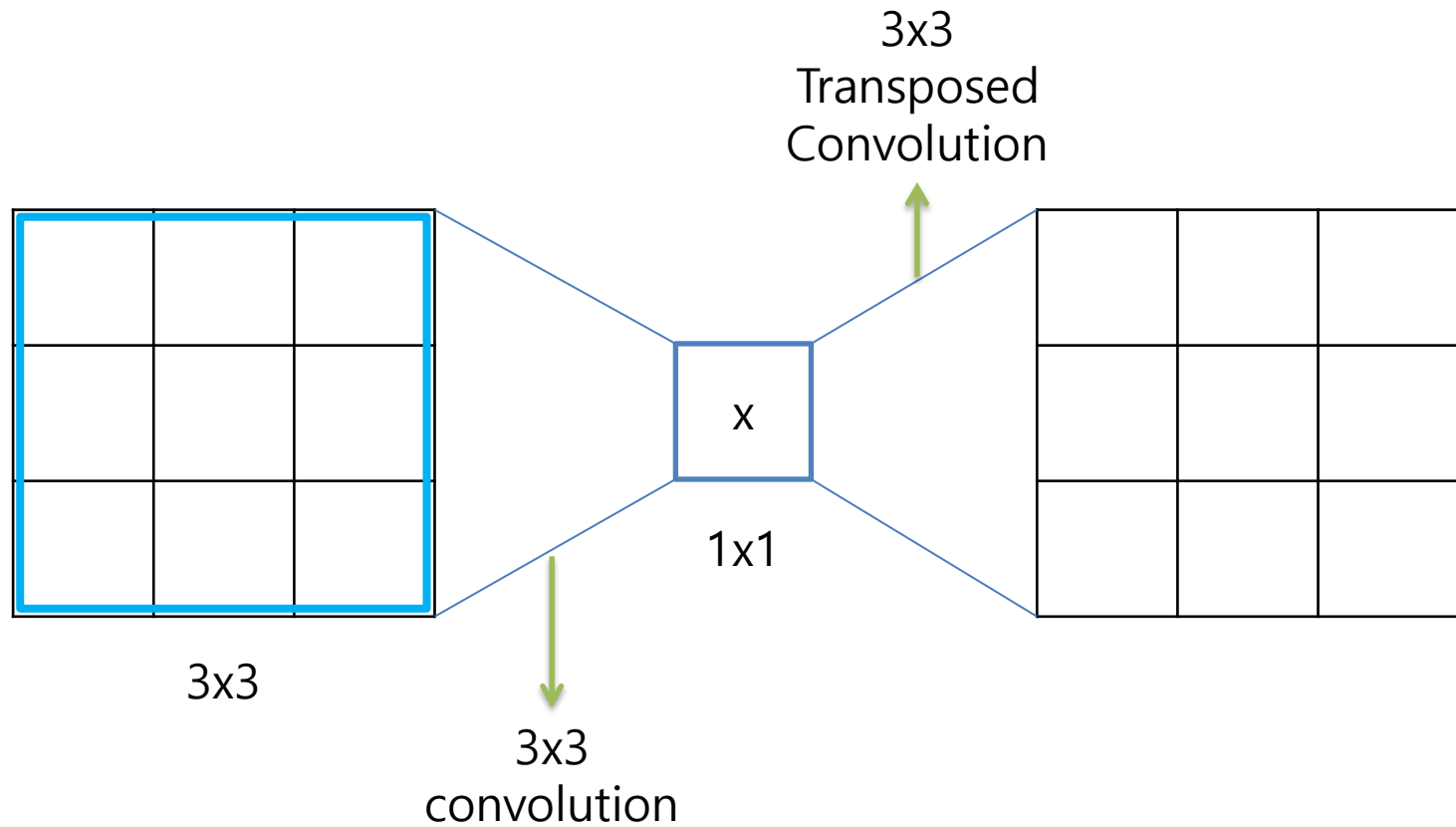
Convolution Transposed



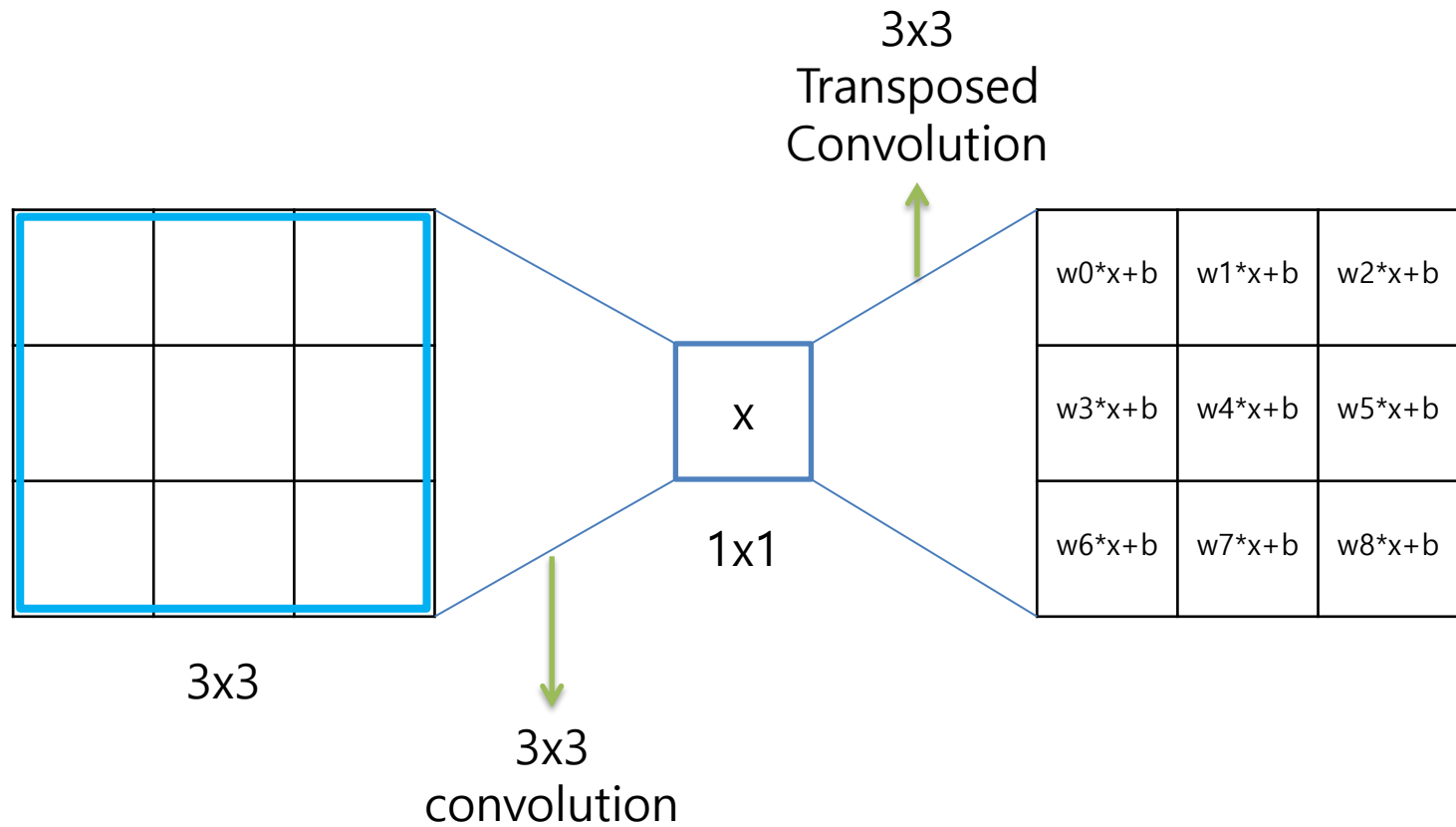
Convolution Transposed



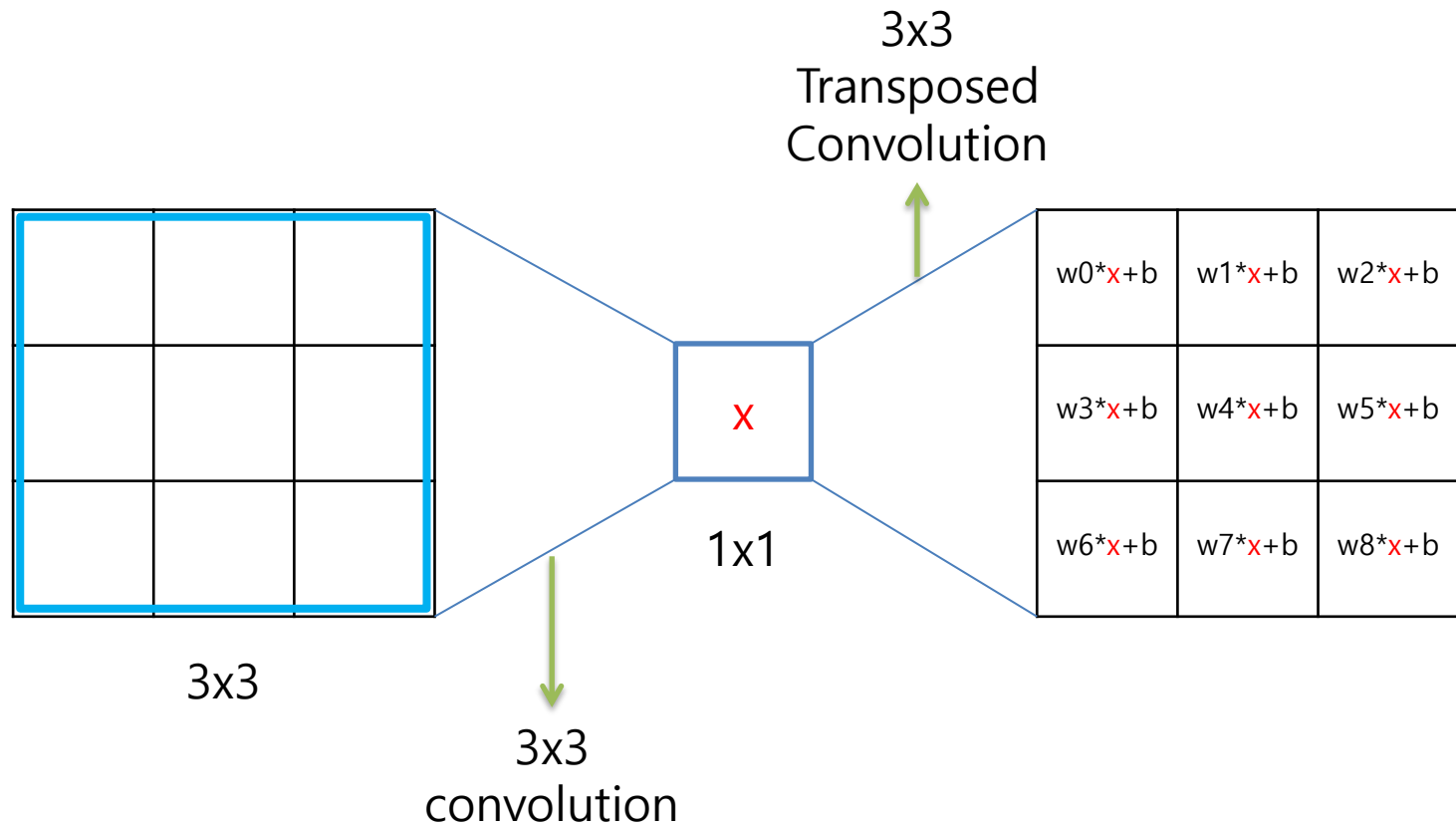
Convolution Transposed



Convolution Transposed

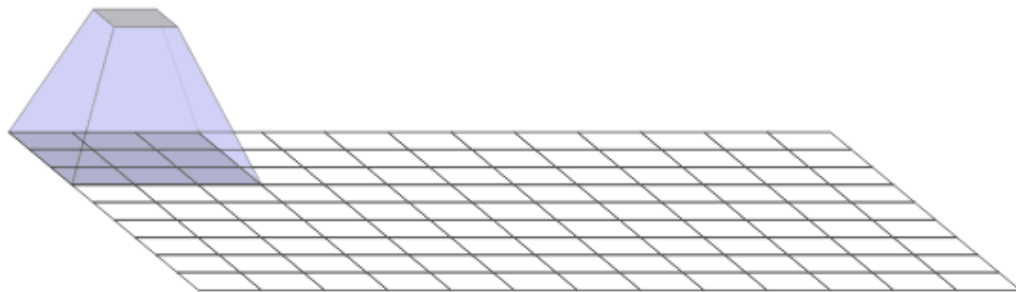


Convolution Transposed



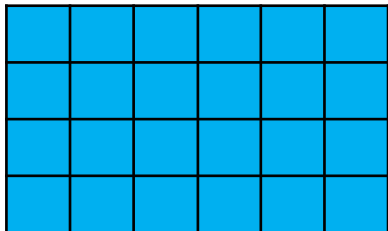
Convolution Transposed

Kernel size 3x3
stride 2



Convolution Transposed

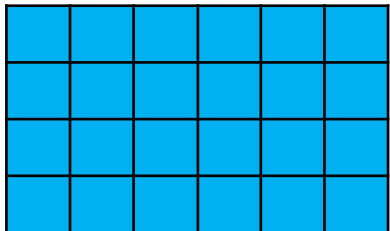
Kernel size 3x3
stride 2



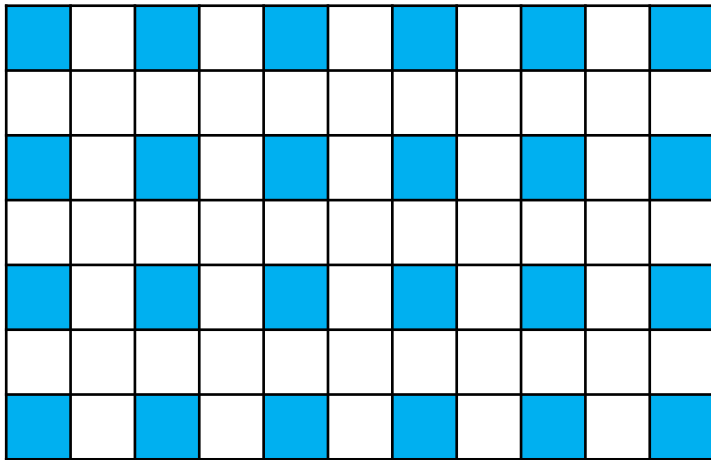
6x4 image

Convolution Transposed

Kernel size 3x3
stride 2



6x4 image

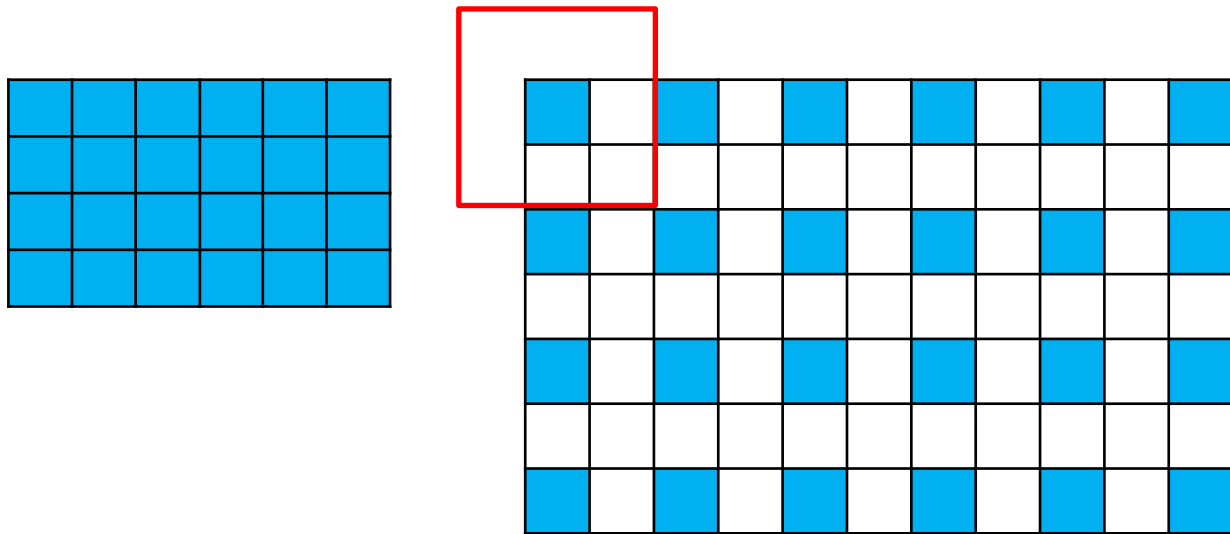


11x7 image

stride 2에
맞춰 펼치기

Convolution Transposed

Kernel size 3x3
stride 2



6x4 image



11x7 image



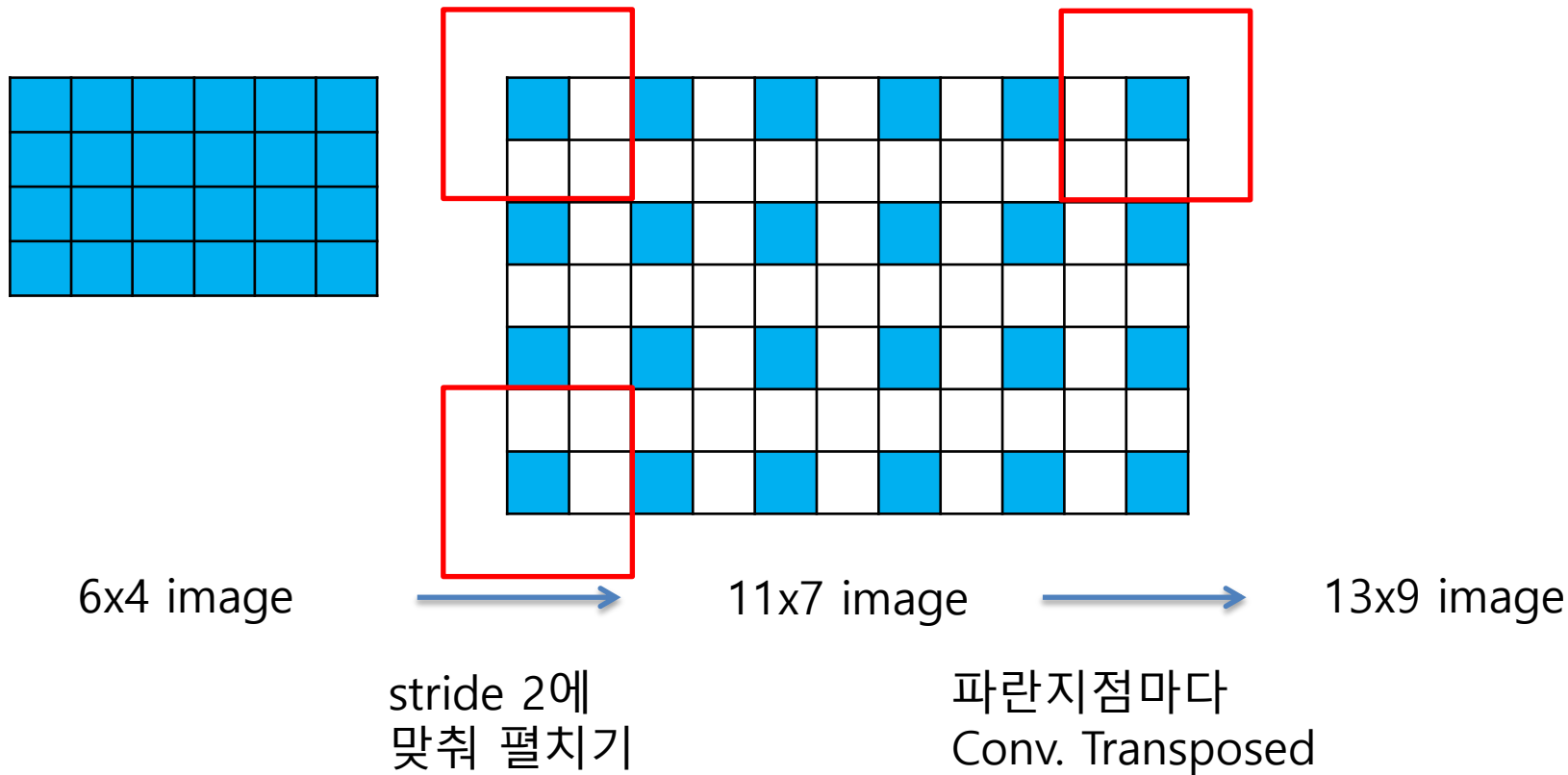
13x9 image

stride 2에
맞춰 펼치기

파란지점마다
Conv. Transposed

Convolution Transposed

Kernel size 3x3
stride 2



Convolution Transposed

```
class torch.nn.ConvTranspose2d(in_channels, out_channels, kernel_size, stride=1, padding=0, output_padding=0, groups=1, bias=True, dilation=1) \[source\]
```

Parameters:

- `in_channels` (*int*) – Number of channels in the input image
- `out_channels` (*int*) – Number of channels produced by the convolution
- `kernel_size` (*int or tuple*) – Size of the convolving kernel
- `stride` (*int or tuple, optional*) – Stride of the convolution
- `padding` (*int or tuple, optional*) – Zero-padding added to both sides of the input
- `output_padding` (*int or tuple, optional*) – Zero-padding added to one side of the output
- `groups` (*int, optional*) – Number of blocked connections from input channels to output channels
- `bias` (*bool, optional*) – If True, adds a learnable bias to the output
- `dilation` (*int or tuple, optional*) – Spacing between kernel elements

Shape:

- Input: $(N, C_{in}, H_{in}, W_{in})$
- Output: $(N, C_{out}, H_{out}, W_{out})$ where
$$H_{out} = (H_{in} - 1) * stride[0] - 2 * padding[0] + kernel_size[0] + output_padding[0]$$
$$W_{out} = (W_{in} - 1) * stride[1] - 2 * padding[1] + kernel_size[1] + output_padding[1]$$

Convolutional Autoencoder

```
class Encoder(nn.Module):
    def __init__(self):
        super(Encoder, self).__init__()
        self.layer1 = nn.Sequential(
            nn.Conv2d(1, 16, 3, padding=1), # batch x 16 x 28 x 28
            nn.ReLU(),
            nn.BatchNorm2d(16),
            nn.Conv2d(16, 32, 3, padding=1), # batch x 32 x 28 x 28
            nn.ReLU(),
            nn.BatchNorm2d(32),
            nn.Conv2d(32, 64, 3, padding=1), # batch x 32 x 28 x 28
            nn.ReLU(),
            nn.BatchNorm2d(64),
            nn.MaxPool2d(2, 2) # batch x 64 x 14 x 14
        )
        self.layer2 = nn.Sequential(
            nn.Conv2d(64, 128, 3, padding=1), # batch x 64 x 14 x 14
            nn.ReLU(),
            nn.BatchNorm2d(128),
            nn.MaxPool2d(2, 2),
            nn.Conv2d(128, 256, 3, padding=1), # batch x 64 x 7 x 7
            nn.ReLU()
        )

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

encoder = Encoder().cuda()
```

Convolutional Autoencoder

일반적인 CNN model

```
class Encoder(nn.Module):
    def __init__(self):
        super(Encoder, self).__init__()
        self.layer1 = nn.Sequential(
            nn.Conv2d(1, 16, 3, padding=1), # batch x 16 x 28 x 28
            nn.ReLU(),
            nn.BatchNorm2d(16),
            nn.Conv2d(16, 32, 3, padding=1), # batch x 32 x 28 x 28
            nn.ReLU(),
            nn.BatchNorm2d(32),
            nn.Conv2d(32, 64, 3, padding=1), # batch x 32 x 28 x 28
            nn.ReLU(),
            nn.BatchNorm2d(64),
            nn.MaxPool2d(2, 2) # batch x 64 x 14 x 14
        )
        self.layer2 = nn.Sequential(
            nn.Conv2d(64, 128, 3, padding=1), # batch x 64 x 14 x 14
            nn.ReLU(),
            nn.BatchNorm2d(128),
            nn.MaxPool2d(2, 2),
            nn.Conv2d(128, 256, 3, padding=1), # batch x 64 x 7 x 7
            nn.ReLU()
        )

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

encoder = Encoder().cuda()
```

Convolutional Autoencoder

```
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.ConvTranspose2d(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

decoder = Decoder().cuda()
```


Convolutional Autoencoder

Shape:

- Input: $(N, C_{in}, H_{in}, W_{in})$
- Output: $(N, C_{out}, H_{out}, W_{out})$ where
$$H_{out} = (H_{in} - 1) * stride[0] - 2 * padding[0] + kernel_size[0] + output_padding[0]$$
$$W_{out} = (W_{in} - 1) * stride[1] - 2 * padding[1] + kernel_size[1] + output_padding[1]$$

```
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.ConvTranspose2d(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

decoder = Decoder().cuda()
```

Convolutional Autoencoder

[batch,256,7,7] -> [batch,128,14,14]

Shape:

- Input: $(N, C_{in}, H_{in}, W_{in})$
- Output: $(N, C_{out}, H_{out}, W_{out})$ where
$$H_{out} = (H_{in} - 1) * stride[0] - 2 * padding[0] + kernel_size[0] + output_padding[0]$$
$$W_{out} = (W_{in} - 1) * stride[1] - 2 * padding[1] + kernel_size[1] + output_padding[1]$$

```
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.ConvTranspose2d(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

decoder = Decoder().cuda()
```

Convolutional Autoencoder

[batch,256,7,7] -> [batch,128,14,14]

[batch,128,14,14] -> [batch,64,14,14]

```
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.ConvTranspose2d(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

decoder = Decoder().cuda()
```

Shape:

- Input: $(N, C_{in}, H_{in}, W_{in})$
- Output: $(N, C_{out}, H_{out}, W_{out})$ where
$$H_{out} = (H_{in} - 1) * stride[0] - 2 * padding[0] + kernel_size[0] + output_padding[0]$$
$$W_{out} = (W_{in} - 1) * stride[1] - 2 * padding[1] + kernel_size[1] + output_padding[1]$$

Convolutional Autoencoder

[batch,256,7,7] -> [batch,128,14,14]

[batch,128,14,14] -> [batch,64,14,14]

[batch,64,14,14] -> [batch,16,14,14]

Shape:

- Input: $(N, C_{in}, H_{in}, W_{in})$
- Output: $(N, C_{out}, H_{out}, W_{out})$ where
$$H_{out} = (H_{in} - 1) * stride[0] - 2 * padding[0] + kernel_size[0] + output_padding[0]$$
$$W_{out} = (W_{in} - 1) * stride[1] - 2 * padding[1] + kernel_size[1] + output_padding[1]$$

```
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.ConvTranspose2d(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

decoder = Decoder().cuda()
```

Convolutional Autoencoder

[batch,256,7,7] -> [batch,128,14,14]

[batch,128,14,14] -> [batch,64,14,14]

[batch,64,14,14] -> [batch,16,14,14]

[batch,16,14,14] -> [batch,1,28,28]

Shape:

- Input: $(N, C_{in}, H_{in}, W_{in})$
- Output: $(N, C_{out}, H_{out}, W_{out})$ where
$$H_{out} = (H_{in} - 1) * stride[0] - 2 * padding[0] + kernel_size[0] + output_padding[0]$$
$$W_{out} = (W_{in} - 1) * stride[1] - 2 * padding[1] + kernel_size[1] + output_padding[1]$$

```
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.ConvTranspose2d(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

decoder = Decoder().cuda()
```

Convolutional Autoencoder

```
parameters = list(encoder.parameters()) + list(decoder.parameters())
loss_func = nn.MSELoss()
optimizer = torch.optim.Adam(parameters, lr=learning_rate)

# train encoder and decoder


try:
    encoder, decoder = torch.load('./model/autoencoder.pkl')
    print("\n-----model restored-----\n")
except:
    pass

for i in range(epoch):
    for image, label in train_loader:
        image = Variable(image).cuda()
        optimizer.zero_grad()
        output = encoder(image)
        output = decoder(output)
        loss = loss_func(output, image)
        loss.backward()
        optimizer.step()

    if i % 2 == 0:
        torch.save([encoder, decoder], './model/autoencoder.pkl')
        print(loss)
```

Convolutional Autoencoder

encoder, decoder의 파라미터를
list로 묶어서 학습하도록 전달.
loss는 Mean Squared Loss



```
parameters = List(encoder.parameters()) + List(decoder.parameters())  
loss_func = nn.MSELoss()  
optimizer = torch.optim.Adam(parameters, lr=learning_rate)
```

```
# train encoder and decoder
```

```
try:
```

```
    encoder, decoder = torch.load('./model/autoencoder.pkl')  
    print("\n-----model restored-----\n")
```

```
except:
```

```
    pass
```

```
for i in range(epoch):
```

```
    for image, label in train_loader:
```

```
        image = Variable(image).cuda()
```

```
        optimizer.zero_grad()
```

```
        output = encoder(image)
```

```
        output = decoder(output)
```

```
        loss = loss_func(output, image)
```

```
        loss.backward()
```

```
        optimizer.step()
```

```
if i % 2 == 0:
```

```
    torch.save([encoder, decoder], './model/autoencoder.pkl')
```

```
    print(loss)
```

Convolutional Autoencoder

encoder, decoder의 파라미터를
list로 묶어서 학습하도록 전달.
loss는 Mean Squared Loss

```
parameters = list(encoder.parameters()) + list(decoder.parameters())  
loss_func = nn.MSELoss()  
optimizer = torch.optim.Adam(parameters, lr=learning_rate)
```

```
# train encoder and decoder
```

```
try:  
    encoder, decoder = torch.load('./model/autoencoder.pkl')  
    print("\n-----model restored-----\n")  
except:  
    pass
```

```
for i in range(epoch):  
    for image, label in train_loader:  
        image = Variable(image).cuda()  
        optimizer.zero_grad()  
        output = encoder(image)  
        output = decoder(output)  
        loss = loss_func(output, image)  
        loss.backward()  
        optimizer.step()  
  
    if i % 2 == 0:  
        torch.save([encoder, decoder], './model/autoencoder.pkl')  
        print(loss)
```

기존에 학습시켜 놓은 모델이 있
다면 불러오고 아니면 pass

Convolutional Autoencoder

encoder, decoder의 파라미터를
list로 묶어서 학습하도록 전달.
loss는 Mean Squared Loss

```
parameters = list(encoder.parameters()) + list(decoder.parameters())  
loss_func = nn.MSELoss()  
optimizer = torch.optim.Adam(parameters, lr=learning_rate)
```

```
# train encoder and decoder
```

```
try:  
    encoder, decoder = torch.load('./model/autoencoder.pkl')  
    print("\n-----model restored-----\n")  
except:  
    pass
```

기존에 학습시켜 놓은 모델이 있
다면 불러오고 아니면 pass

```
for i in range(epoch):  
    for image, label in train_loader:  
        image = Variable(image).cuda()  
        optimizer.zero_grad()  
        output = encoder(image)  
        output = decoder(output)  
        loss = loss_func(output, image)  
        loss.backward()  
        optimizer.step()
```

원본 이미지를 encoding,
decoding하여 결과값과
원본의 차이를 계산

```
if i % 2 == 0:  
    torch.save([encoder, decoder], './model/autoencoder.pkl')  
    print(loss)
```

Convolutional Autoencoder

encoder, decoder의 파라미터를
list로 묶어서 학습하도록 전달.
loss는 Mean Squared Loss

```
parameters = list(encoder.parameters()) + list(decoder.parameters())  
loss_func = nn.MSELoss()  
optimizer = torch.optim.Adam(parameters, lr=learning_rate)
```

```
# train encoder and decoder
```

```
try:  
    encoder, decoder = torch.load('./model/autoencoder.pkl')  
    print("\n-----model restored-----\n")  
except:  
    pass
```

기존에 학습시켜 놓은 모델이 있
다면 불러오고 아니면 pass

원본 이미지를 encoding,
decoding하여 결과값과
원본의 차이를 계산

```
for i in range(epoch):  
    for image, label in train_loader:  
        image = Variable(image).cuda()  
        optimizer.zero_grad()  
        output = encoder(image)  
        output = decoder(output)  
        loss = loss_func(output, image)  
        loss.backward()  
        optimizer.step()
```

일정 기간마다 모델을 저장

```
if i % 2 == 0:  
    torch.save([encoder, decoder], './model/autoencoder.pkl')  
    print(loss)
```

Convolutional Autoencoder

```
parameters = list(encoder.parameters()) + list(decoder.parameters())
loss_func = nn.MSELoss()
optimizer = torch.optim.Adam(parameters, lr=learning_rate)

# train encoder and decoder

try:
    encoder, decoder = torch.load('./model/autoencoder.pkl')
    print("\n-----model restored-----\n")
except:
    pass

for i in range(epoch):
    for image, label in train_loader:
        image = Variable(image).cuda()
        optimizer.zero_grad()
        output = encoder(image)
        output = decoder(output)
        loss = loss_func(output, image)
        loss.backward()
        optimizer.step()

    if i % 2 == 0:
        torch.save([encoder, decoder], './model/autoencoder.pkl')
        print(loss)
```

Convolutional Autoencoder

Denoising Autoencoder 같은 경우에는 원본 이미지에 noise 를 추가하여 autoencoder를 통과 하면 노이즈가 제거된 원본 이미지가 나오도록 학습함.

```
parameters = list(encoder.parameters()) + list(decoder.parameters())
loss_func = nn.MSELoss()
optimizer = torch.optim.Adam(parameters, lr=learning_rate)

# train encoder and decoder

try:
    encoder, decoder = torch.load('./model/autoencoder.pkl')
    print("\n-----model restored-----\n")
except:
    pass

for i in range(epoch):
    for image, label in train_loader:
        image = Variable(image).cuda()
        optimizer.zero_grad()
        output = encoder(image)
        output = decoder(output)
        loss = loss_func(output, image)
        loss.backward()
        optimizer.step()

    if i % 2 == 0:
        torch.save([encoder, decoder], './model/autoencoder.pkl')
        print(loss)
```

Convolutional Autoencoder

Denoising Autoencoder 같은 경우에는 원본 이미지에 noise를 추가하여 autoencoder를 통과하면 노이즈가 제거된 원본 이미지가 나오도록 학습함.

이렇게 되면 학습 이후 좀 지저분한 데이터가 들어오더라도 정제할 수 있음

```
parameters = list(encoder.parameters()) + list(decoder.parameters())
loss_func = nn.MSELoss()
optimizer = torch.optim.Adam(parameters, lr=learning_rate)

# train encoder and decoder

try:
    encoder, decoder = torch.load('./model/autoencoder.pkl')
    print("\n-----model restored-----\n")
except:
    pass

for i in range(epoch):
    for image, label in train_loader:
        image = Variable(image).cuda()
        optimizer.zero_grad()
        output = encoder(image)
        output = decoder(output)
        loss = loss_func(output, image)
        loss.backward()
        optimizer.step()

    if i % 2 == 0:
        torch.save([encoder, decoder], './model/autoencoder.pkl')
        print(loss)
```

Convolutional Autoencoder

Denoising Autoencoder 같은 경우에는 원본 이미지에 noise를 추가하여 autoencoder를 통과하면 노이즈가 제거된 원본 이미지가 나오도록 학습함.

이렇게 되면 학습 이후 좀 지저분한 데이터가 들어오더라도 정제할 수 있음



```
parameters = list(encoder.parameters()) + list(decoder.parameters())
loss_func = nn.MSELoss()
optimizer = torch.optim.Adam(parameters, lr=learning_rate)

# train encoder and decoder

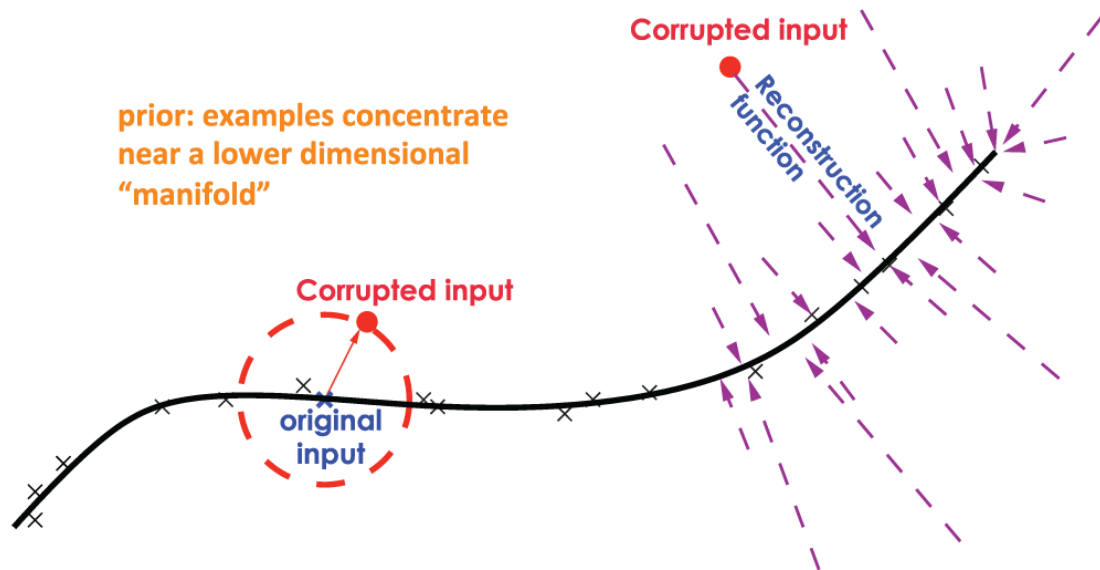
try:
    encoder, decoder = torch.load('./model/autoencoder.pkl')
    print("\n-----model restored-----\n")
except:
    pass

for i in range(epoch):
    for image, label in train_loader:
        image = Variable(image).cuda()
        optimizer.zero_grad()
        output = encoder(image)
        output = decoder(output)
        loss = loss_func(output, image)
        loss.backward()
        optimizer.step()

    if i % 2 == 0:
        torch.save([encoder, decoder], './model/autoencoder.pkl')
        print(loss)
```

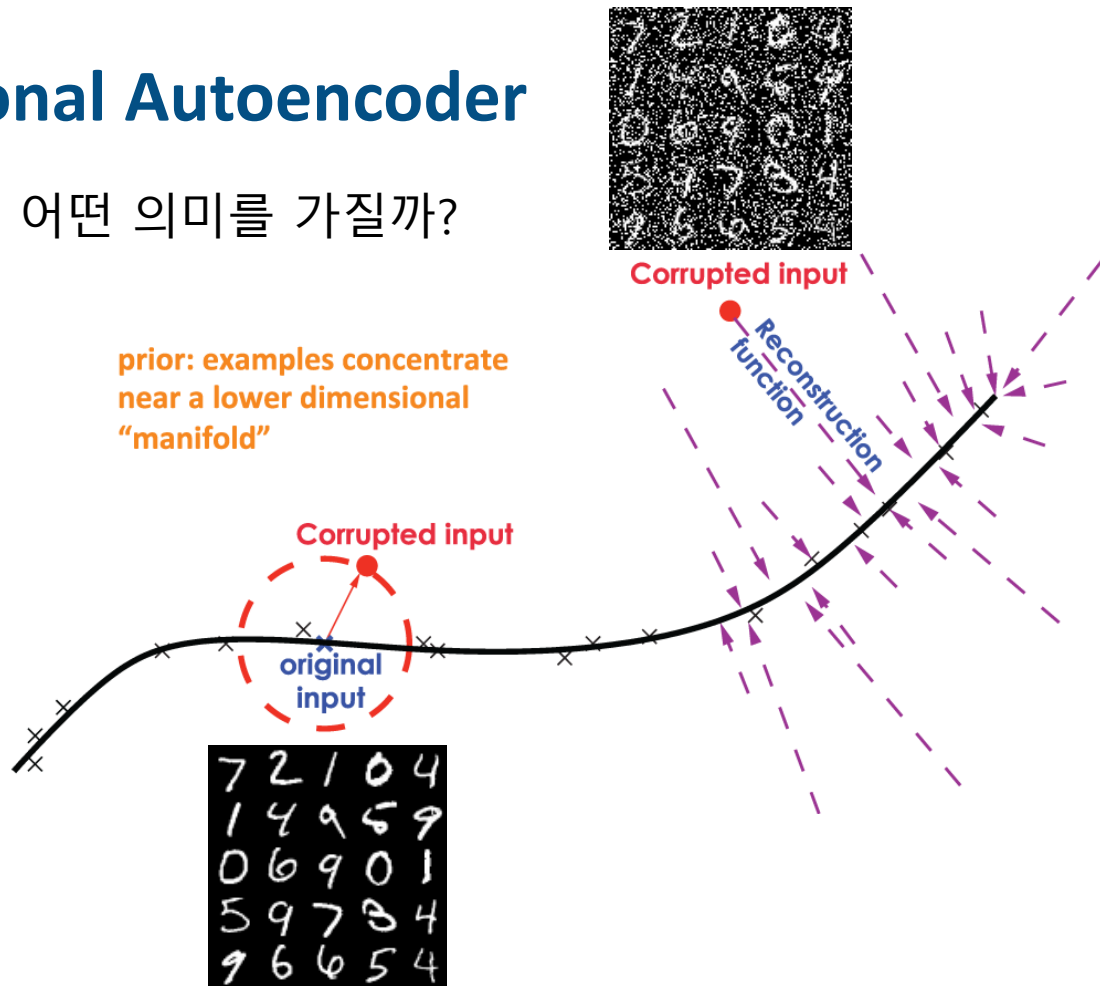
Convolutional Autoencoder

Denoising은 어떤 의미를 가질까?

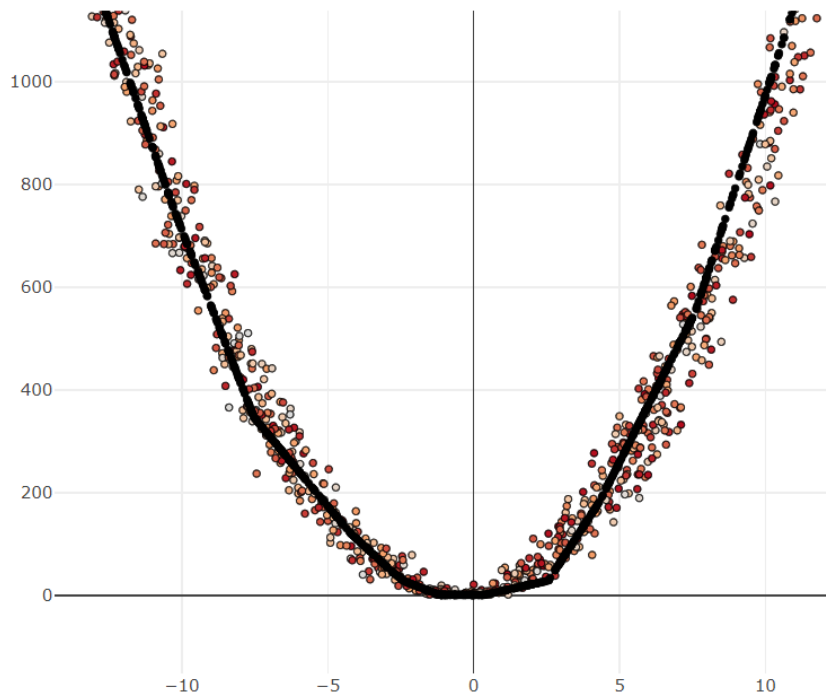


Convolutional Autoencoder

Denoising은 어떤 의미를 가질까?

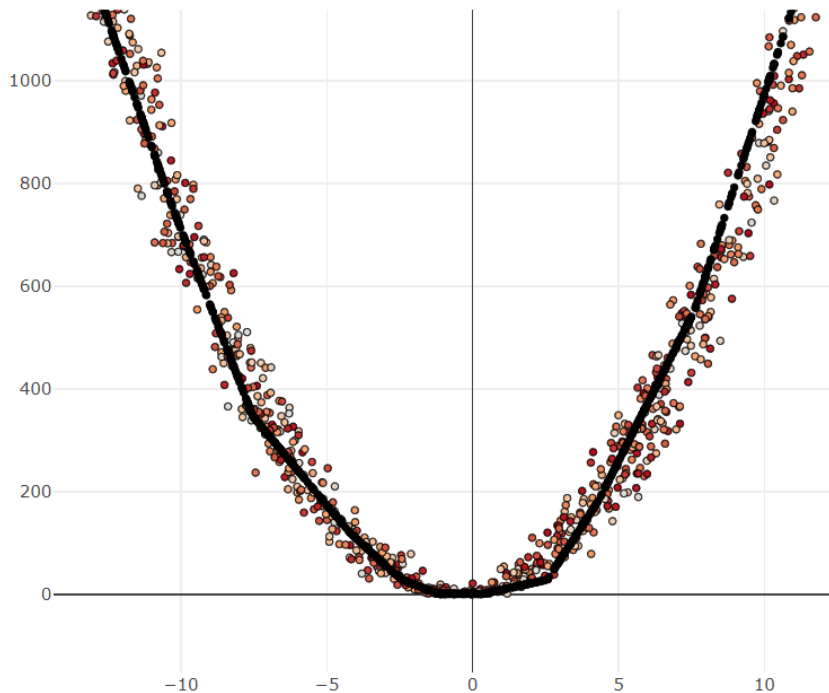


Convolutional Autoencoder



Convolutional Autoencoder

사실 2번째 강의에서 2차 함수를
근사 할 때 noise가 있는 데이터
-를 input으로 사용했던 것도 같은
의미

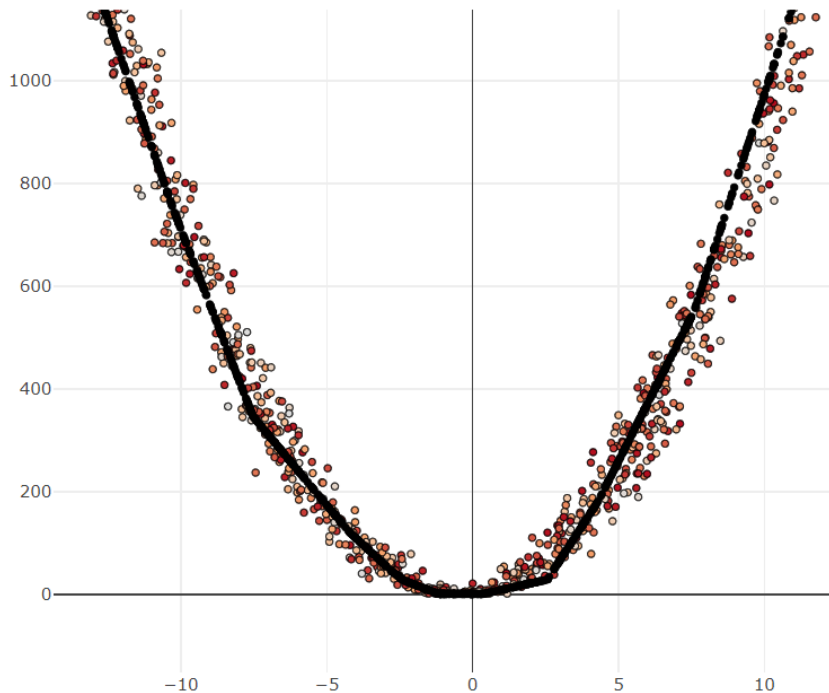


Convolutional Autoencoder

사실 2번째 강의에서 2차 함수를
근사 할 때 noise가 있는 데이터
-를 input으로 사용했던 것도 같은
의미



Noise에 강하게(robust) 학습됨
Filter들도 Clean 데이터보다 더
선명하게 생성됨



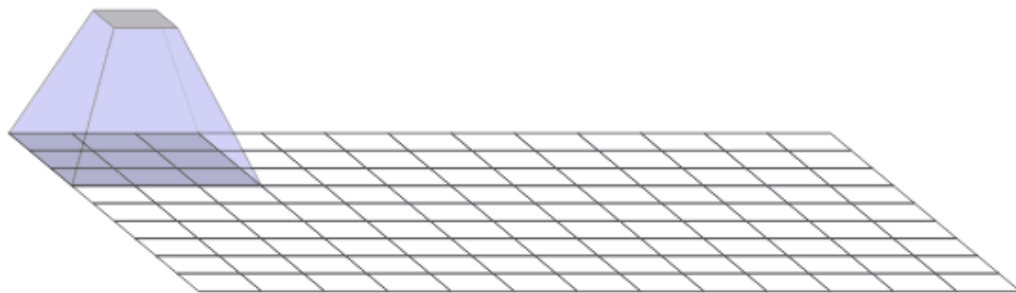
Convolutional Autoencoder

Checkerboard Artifacts



Convolution Transposed 를 사용하면 인위적인 형태들이 생긴다?

Convolutional Autoencoder



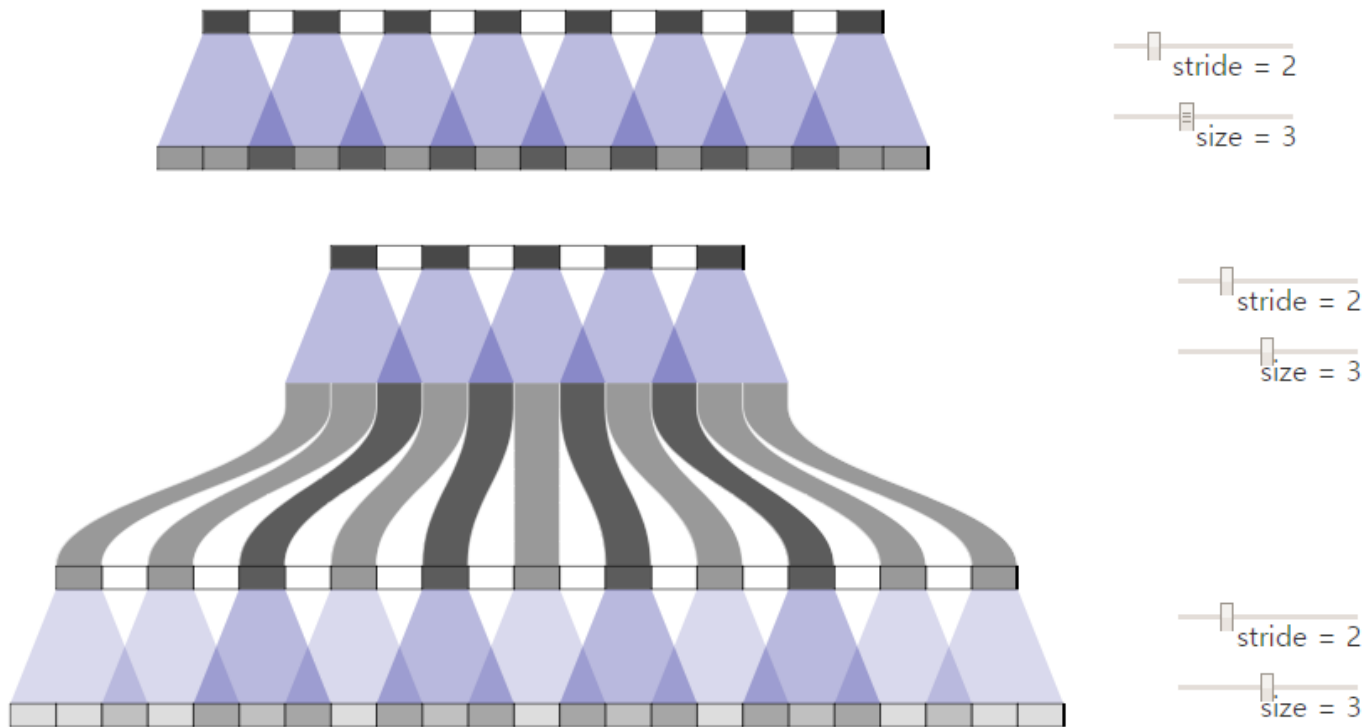
작동원리에 의해 여러 번 값들이 겹치는 부분들이 생겨나고
이러한 옆 픽셀과의 차이가 체커보드 형태로 나타남



어떻게 없앨 수 있을까?

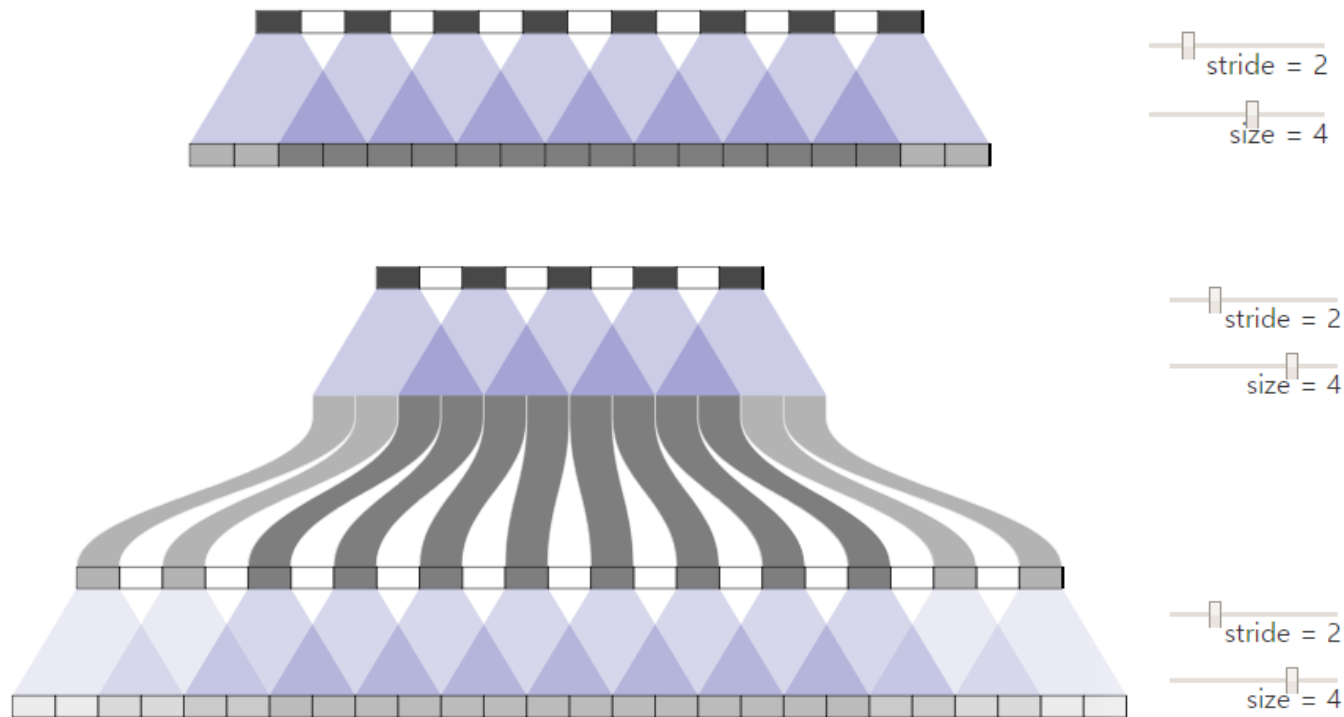
Convolutional Autoencoder

kernel size를 4로 바꾸면?

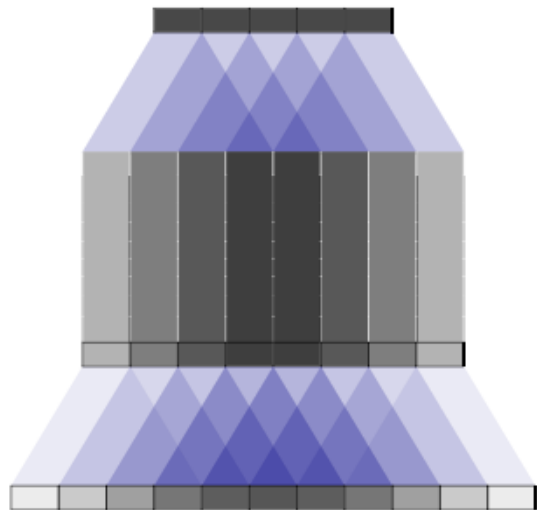


Convolutional Autoencoder

어느 정도는 균등해지지만
완전 없앨 수는 없음



Convolutional Autoencoder



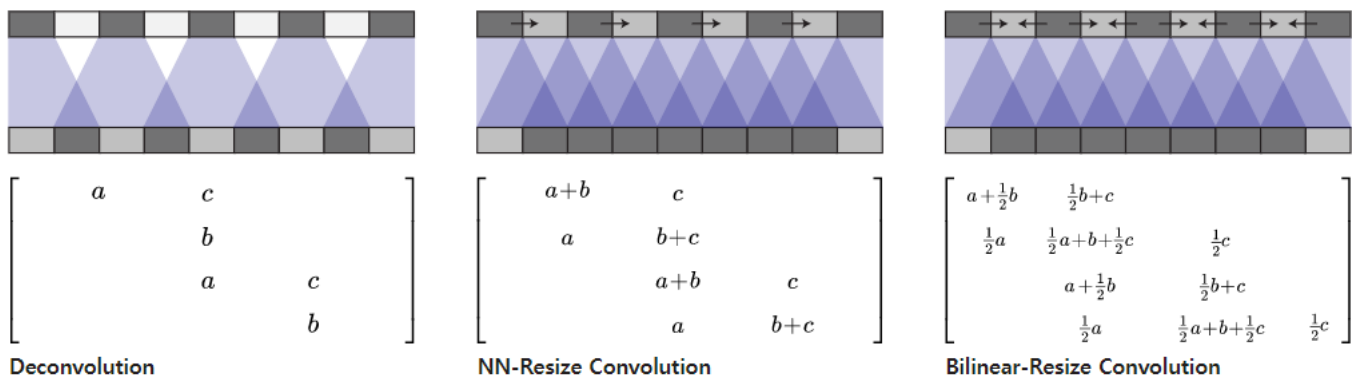
stride = 1
size = 4

stride = 1
size = 4

이에 비해 stride를 1로 주면 비교적 균등한 결과값을 얻을 수 있다.

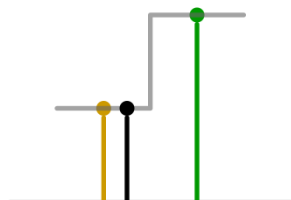
하지만 그냥 stride 1을 쓰면 이미지의 크기를 stride 2때처럼 키울 수 없음

Convolutional Autoencoder

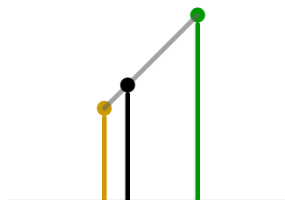


기존에 사용하던 upsampling 방법들을 써서 크기를 키우고
여기에 stride 1 짜리 convolution을 적용하자!

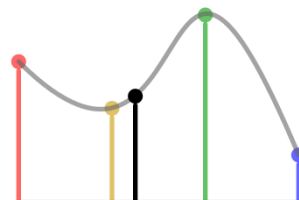
Convolutional Autoencoder



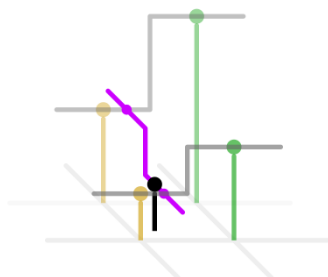
1D nearest-neighbour



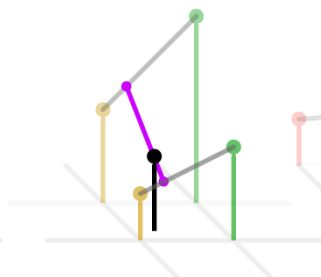
Linear



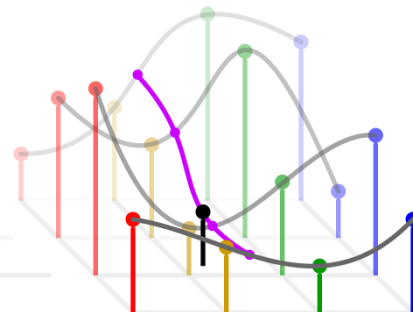
Cubic



2D nearest-neighbour



Bilinear



Bicubic

Convolutional Autoencoder

```
class torch.nn.UpsamplingNearest2d(size=None, scale_factor=None) \[source\]
```

Applies a 2D nearest neighbor upsampling to an input signal composed of several input channels.

To specify the scale, it takes either the `size` or the `scale_factor` as it's constructor argument.

When *size* is given, it is the output size of the image (h, w).

- Parameters:
- *size* (*tuple*, *optional*) – a tuple of ints (H_out, W_out) output sizes
 - *scale_factor* (*int*, *optional*) – the multiplier for the image height / width

```
class torch.nn.UpsamplingBilinear2d(size=None, scale_factor=None) \[source\]
```

Applies a 2D bilinear upsampling to an input signal composed of several input channels.

To specify the scale, it takes either the `size` or the `scale_factor` as it's constructor argument.

When *size* is given, it is the output size of the image (h, w).

- Parameters:
- *size* (*tuple*, *optional*) – a tuple of ints (H_out, W_out) output sizes
 - *scale_factor* (*int*, *optional*) – the multiplier for the image height / width
-

Convolutional Autoencoder



Deconv in last two layers.
Other layers use resize-convolution.
Artifacts of frequency 2 and 4.

Deconv only in last layer.
Other layers use resize-convolution.
Artifacts of frequency 2.

All layers use resize-convolution.
No artifacts.

Convolutional Autoencoder



Using deconvolution.
Heavy checkerboard artifacts.

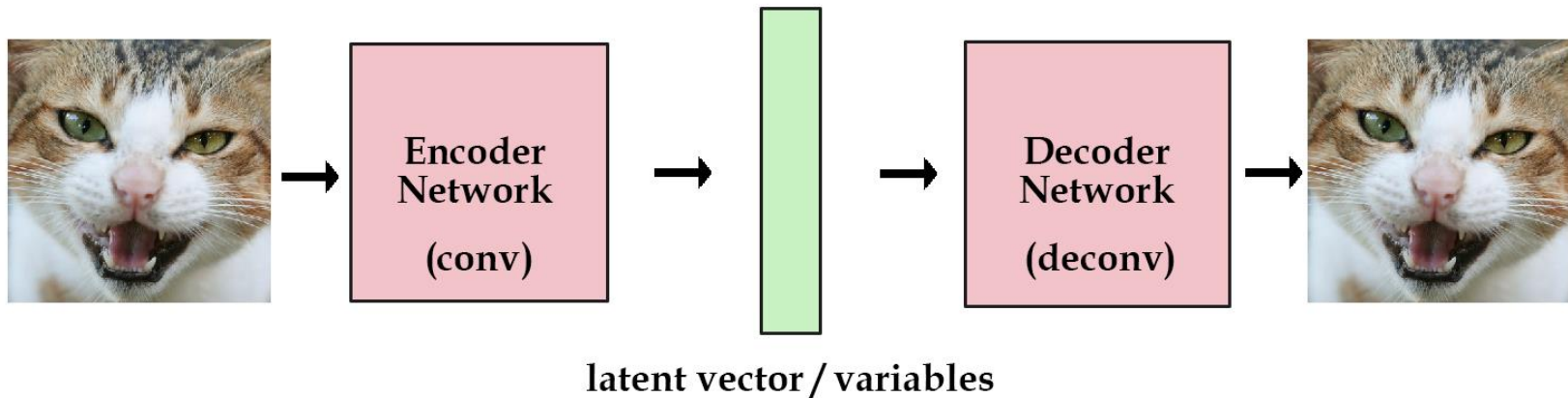


Using resize-convolution.
No checkerboard artifacts.

Interpolation -> Convolution Transposed로 해결할 수 있다.

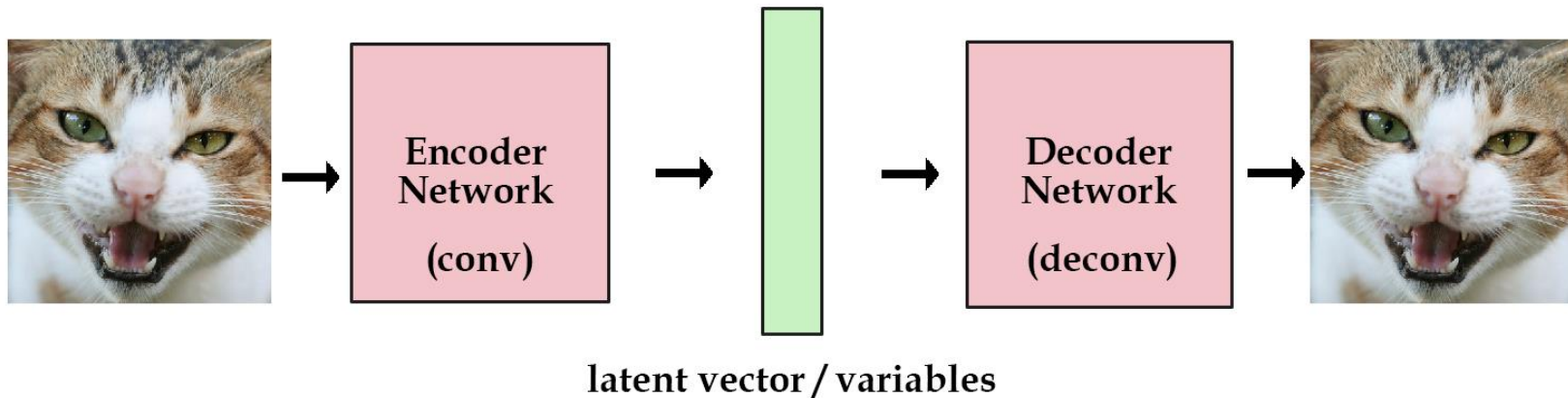
Variational Autoencoder

기존의 autoencoder



Variational Autoencoder

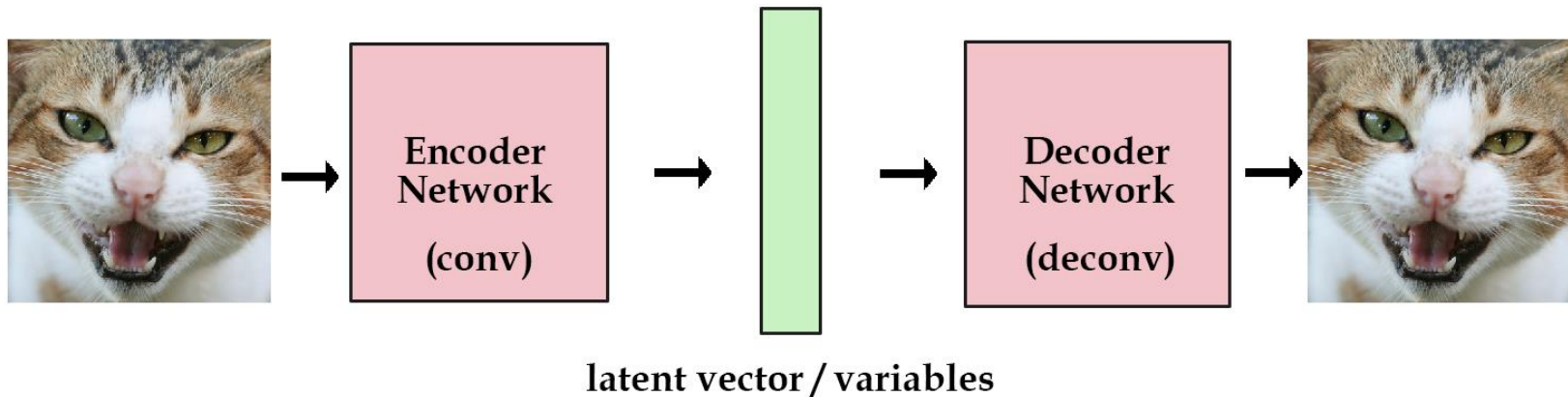
기존의 autoencoder



latent vector가 어떻게 분포되어 있는지 알 수 없음

Variational Autoencoder

기존의 autoencoder

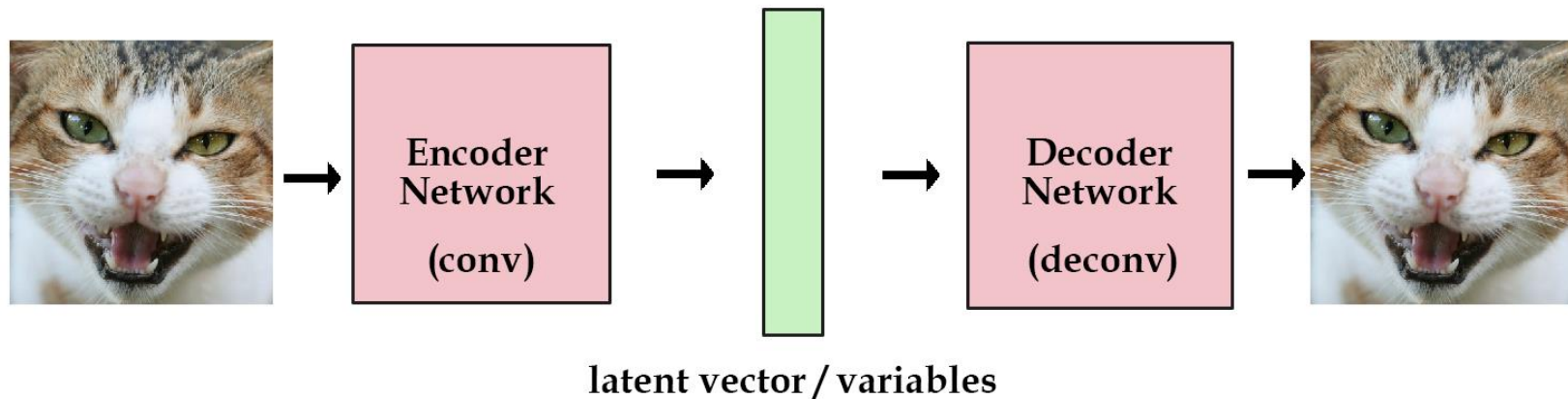


latent vector가 어떻게 분포되어 있는지 알 수 없음

고양이라는 개념이 어떻게 mapping되는지 모르기 때문에
decoder만으로 새로운 데이터를 generate하기 어려움

Variational Autoencoder

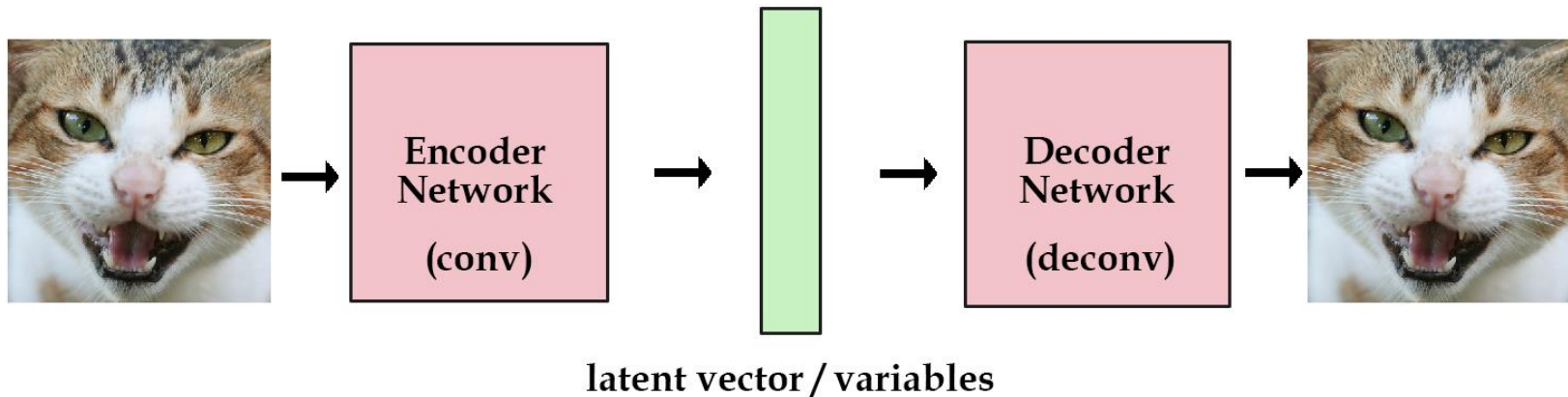
기존의 autoencoder



그렇다면 latent vector가 우리가 알고 있는 특정 분포의 모양을 가지고 있고 거기서부터 sampling 하게 하는 건 어떨까?

Variational Autoencoder

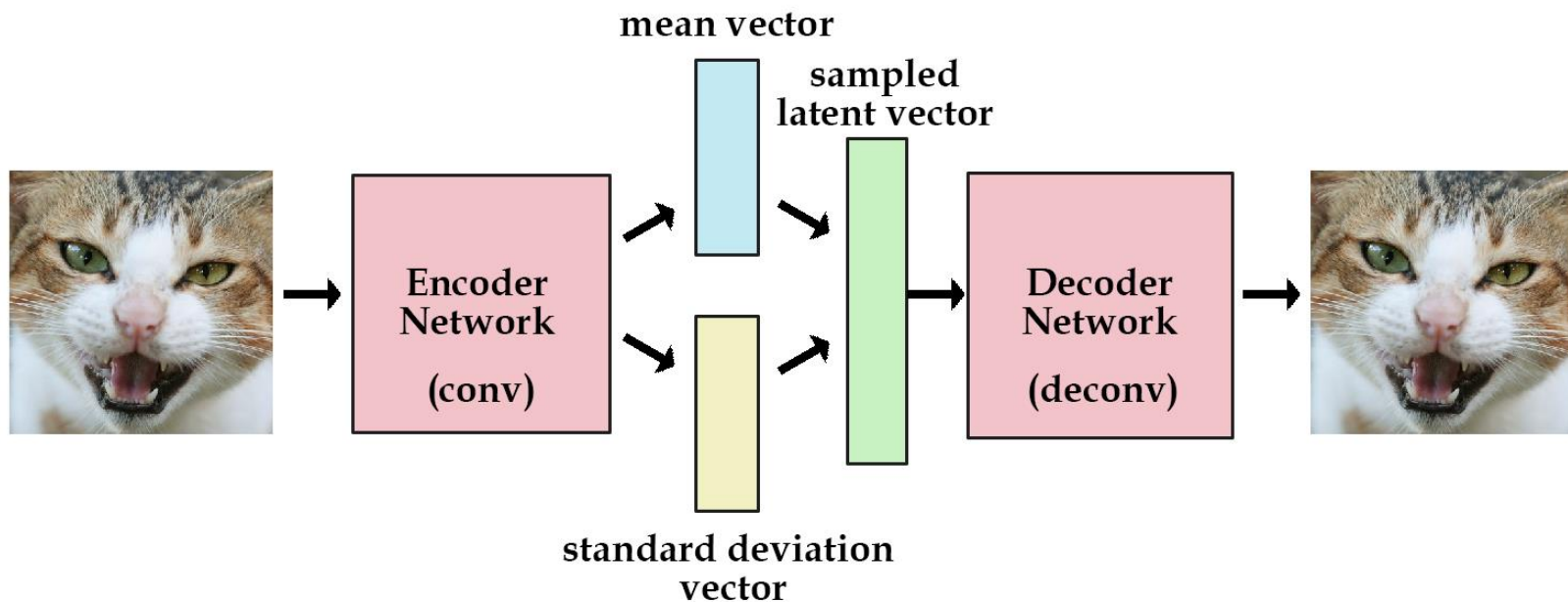
기존의 autoencoder



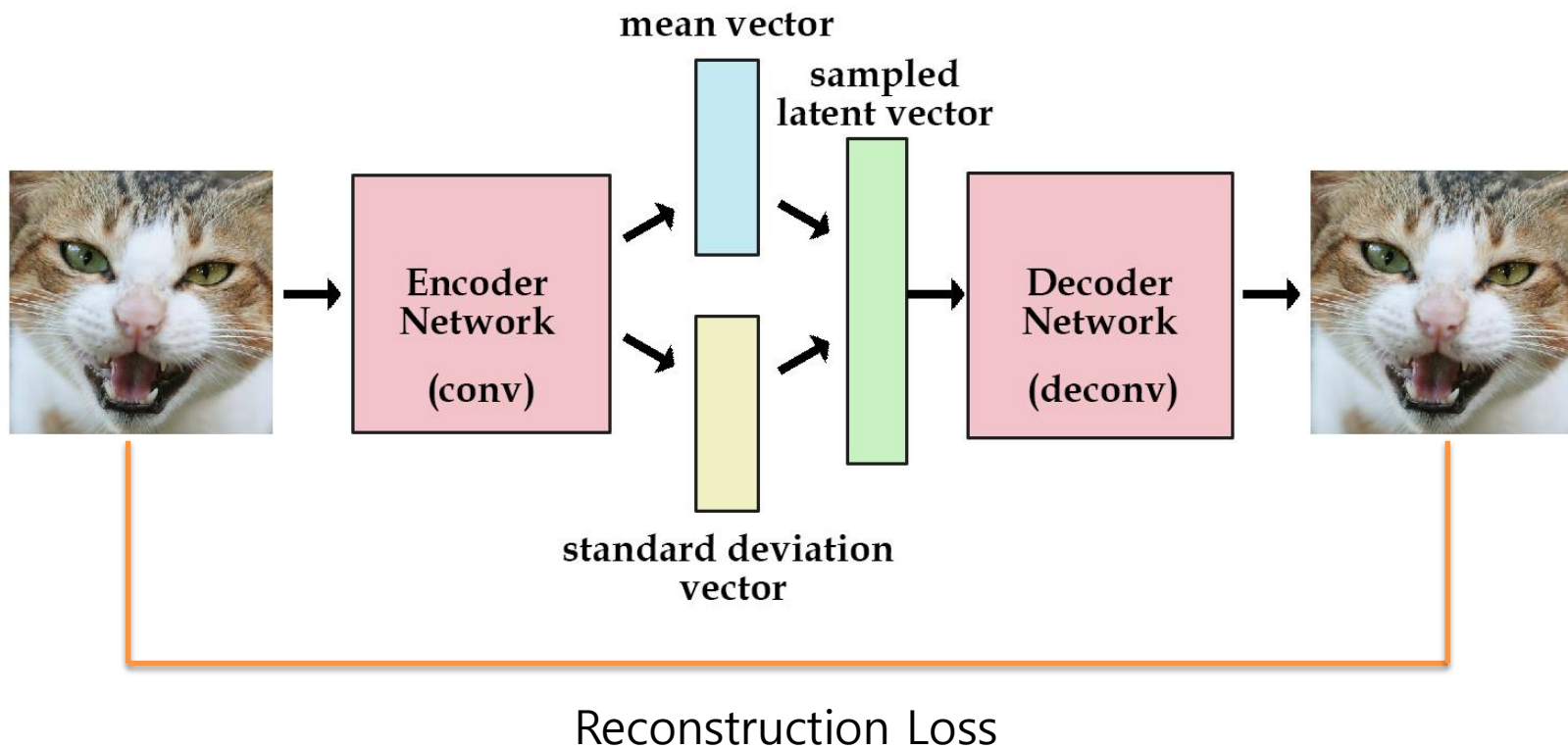
그렇다면 latent vector가 우리가 알고 있는 특정 분포의 모양을 가지고 있고 거기서부터 sampling 하게 하는 건 어떨까?

Variational Autoencoder

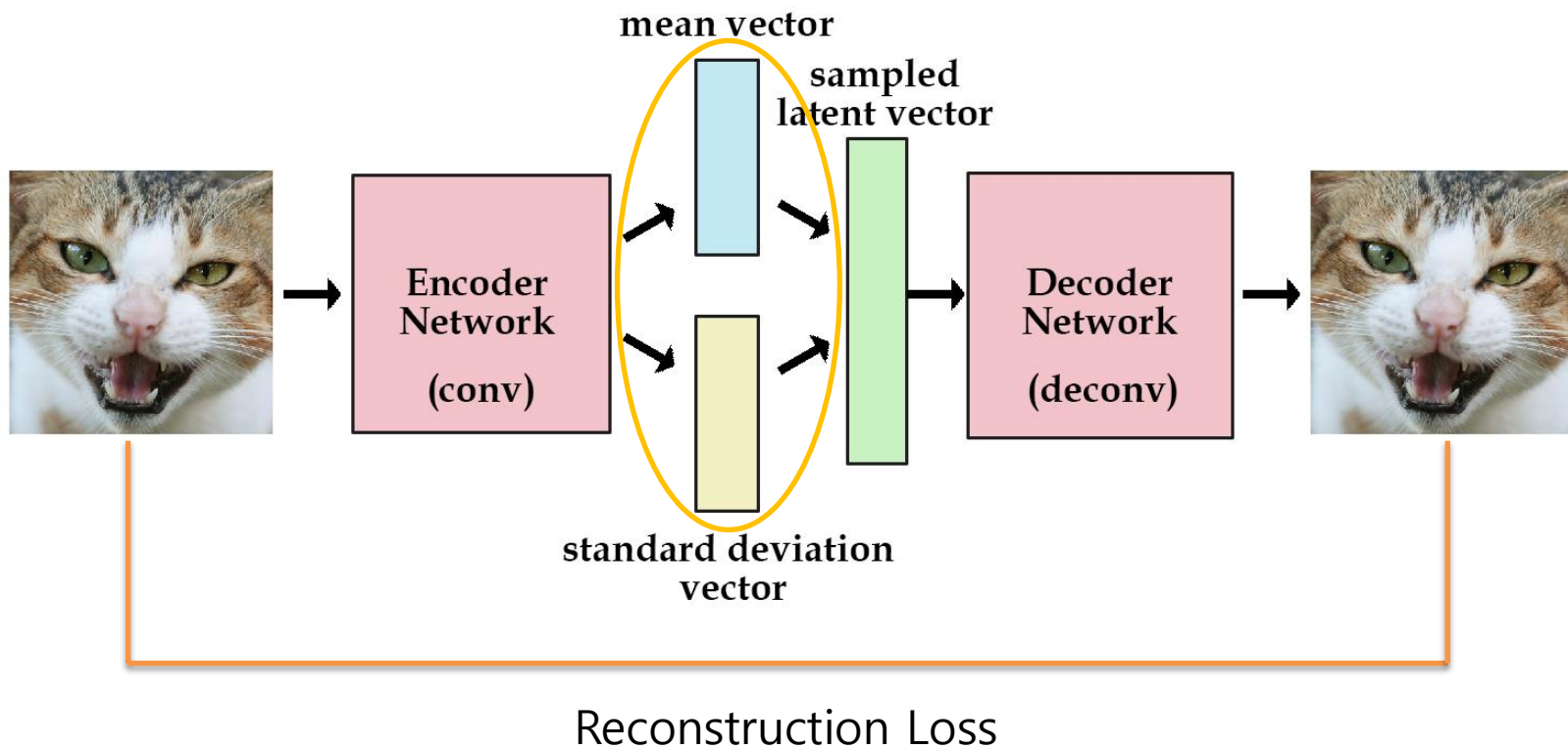
Variational Autoencoder



Variational Autoencoder

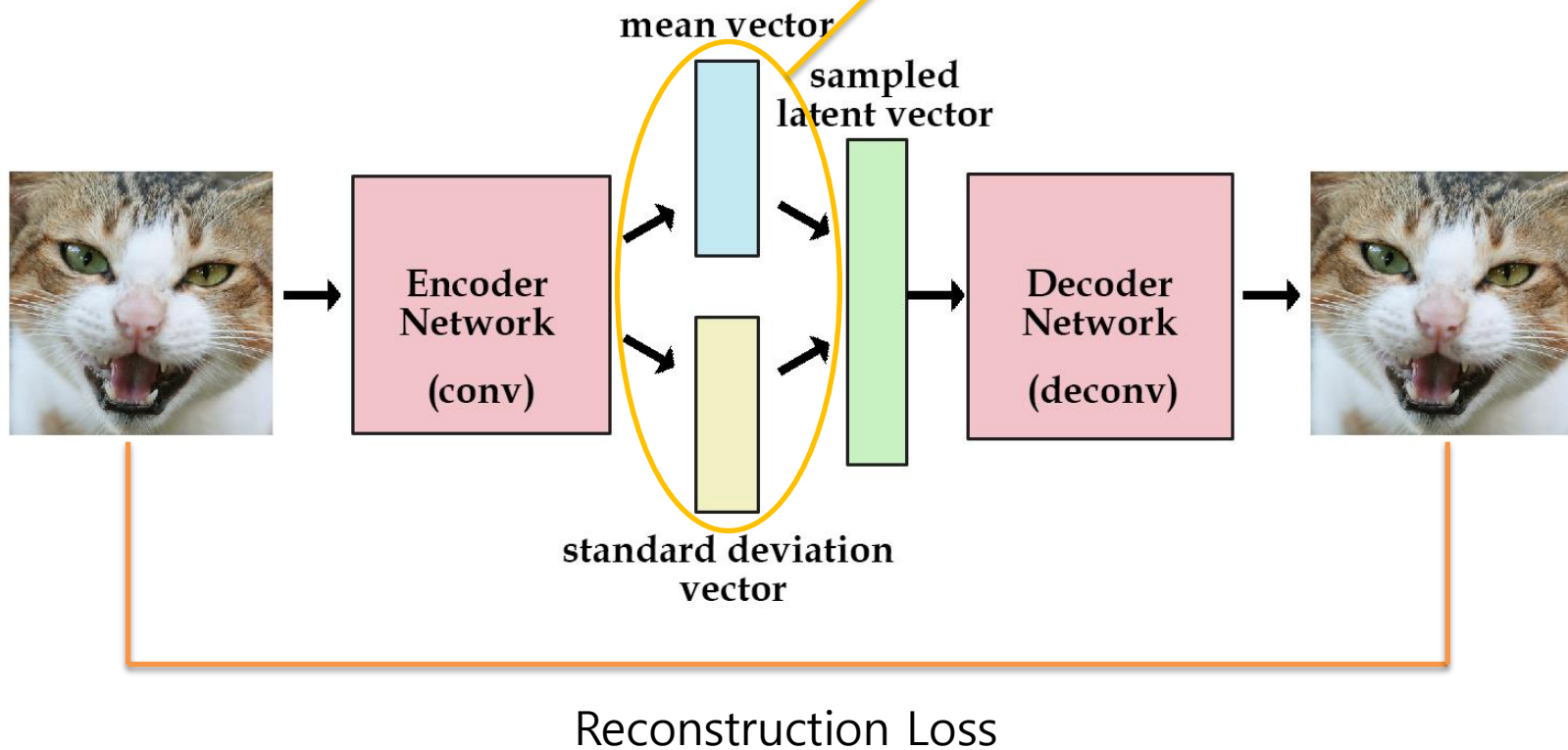


Variational Autoencoder

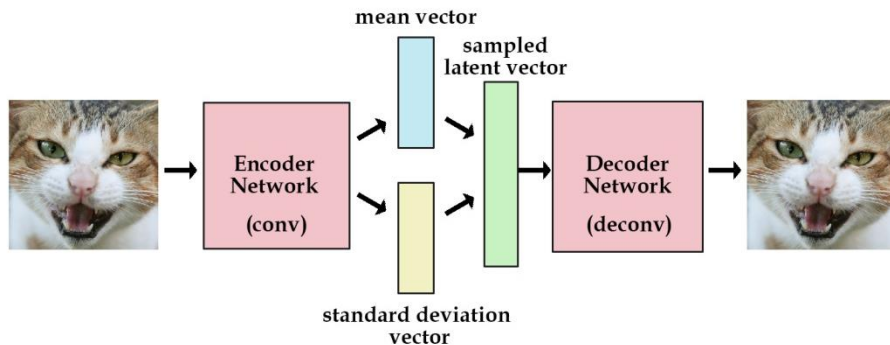


Variational Autoencoder

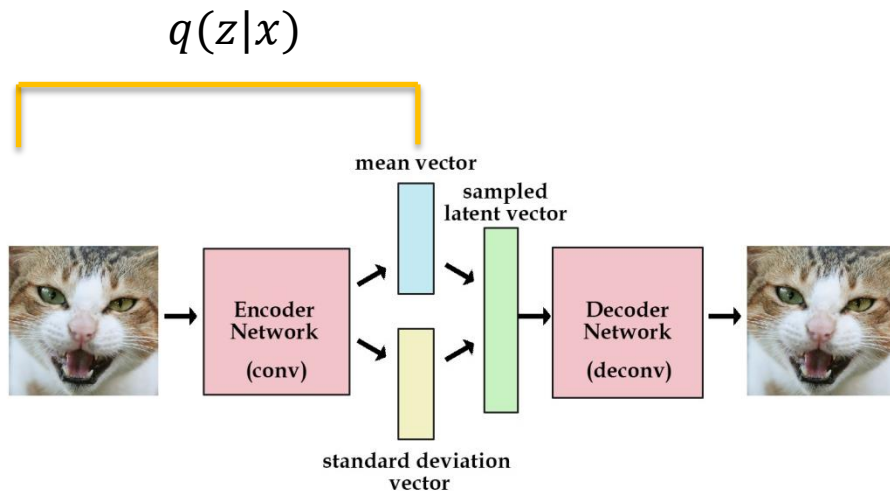
μ 와 σ 가 *unit gaussian*을 따르도록 학습
두 개의 KL divergence를 최소화



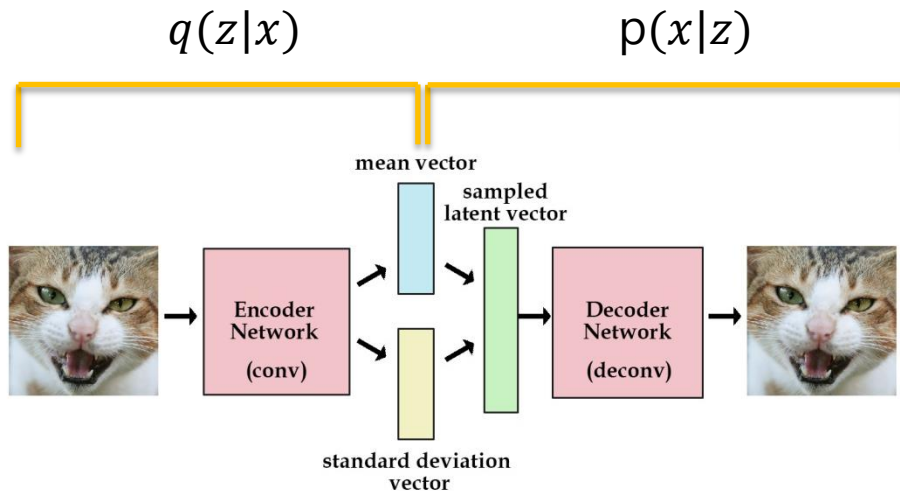
Variational Autoencoder



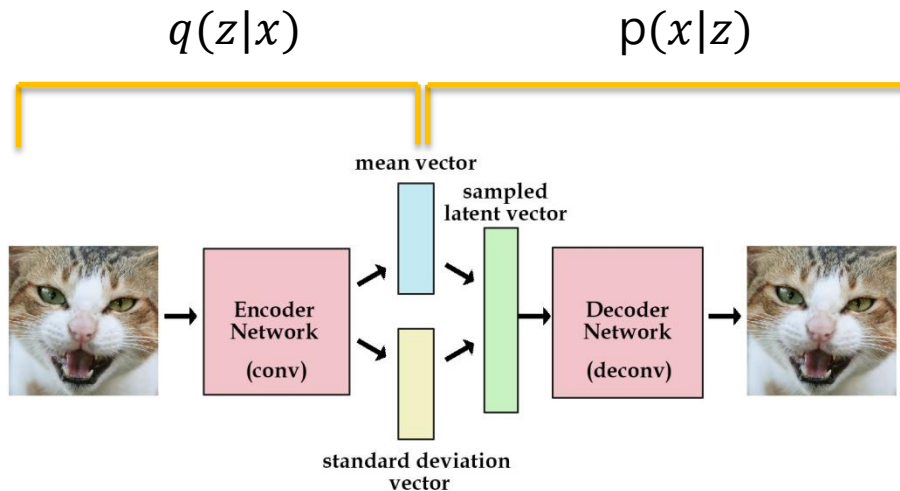
Variational Autoencoder



Variational Autoencoder

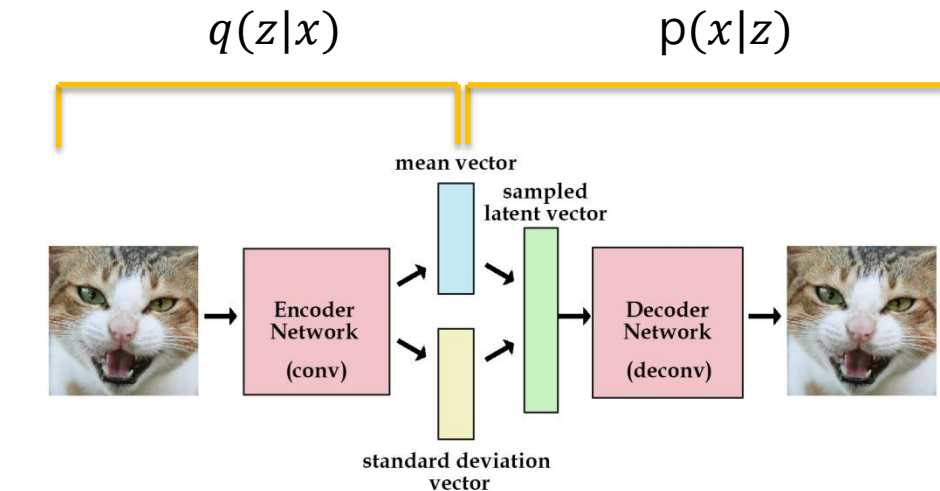


Variational Autoencoder



$$l_i(\theta, \phi) = -E_{z \sim q_\theta(z|x_i)}[\log p_\phi(x_i|z)] + KL(q_\theta(z|x_i)||p(z))$$

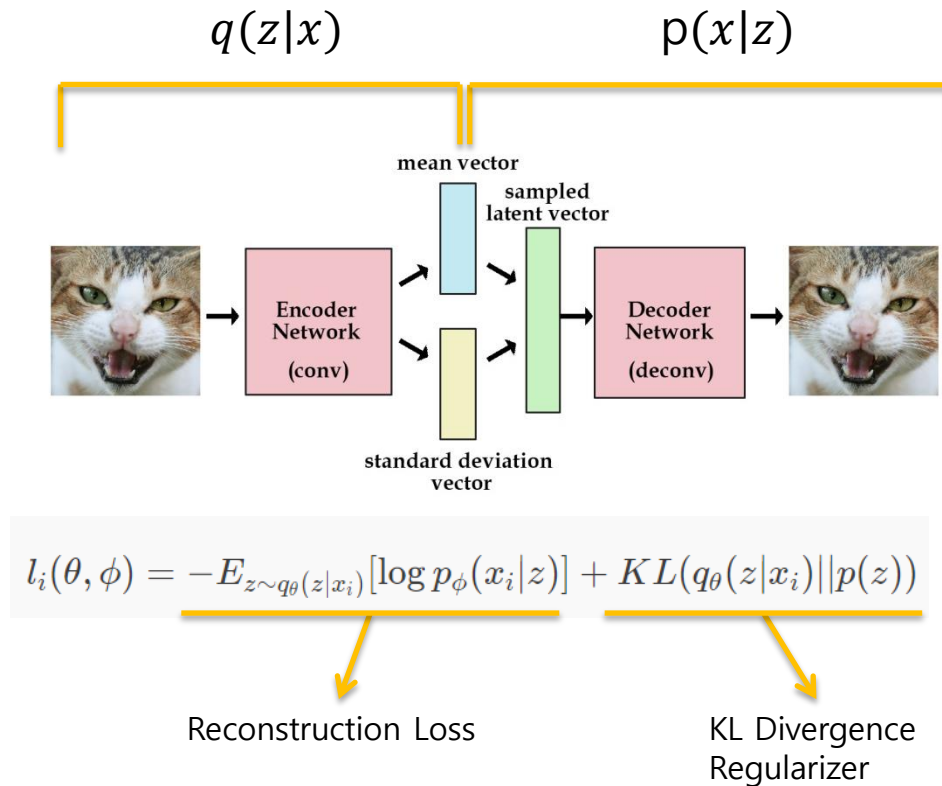
Variational Autoencoder



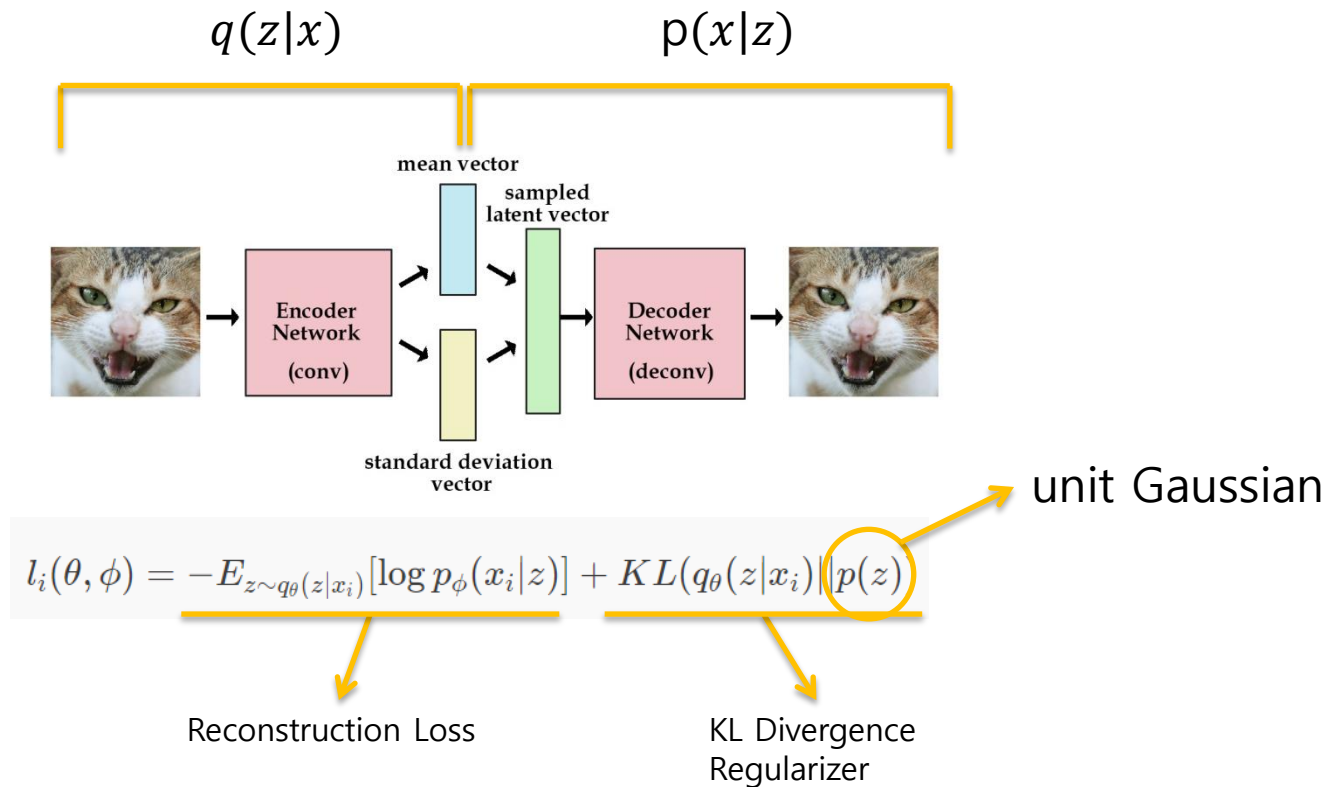
$$l_i(\theta, \phi) = -E_{z \sim q_\theta(z|x_i)} [\log p_\phi(x_i|z)] + KL(q_\theta(z|x_i) || p(z))$$

Reconstruction Loss

Variational Autoencoder



Variational Autoencoder



Variational Autoencoder

$$l_i(\theta, \phi) = -E_{z \sim q_\theta(z|x_i)}[\log p_\phi(x_i|z)] + KL(q_\theta(z|x_i)||p(z))$$

Variational Autoencoder

$$l_i(\theta, \phi) = -E_{z \sim q_\theta(z|x_i)}[\log p_\phi(x_i|z)] + \textcolor{blue}{KL(q_\theta(z|x_i)||p(z))}$$

Variational Autoencoder

tr: trace (정사각형 행렬의 대각합)
det: determinant (행렬식)

$$l_i(\theta, \phi) = -E_{z \sim q_\theta(z|x_i)}[\log p_\phi(x_i|z)] + \textcolor{brown}{KL(q_\theta(z|x_i)||p(z))}$$

$$D_{KL}[N(\mu(X), \Sigma(X))||N(0, 1)] = \frac{1}{2} (\text{tr}(\Sigma(X)) + \mu(X)^T \mu(X) - k - \log \det(\Sigma(X)))$$

Variational Autoencoder

tr: trace (정사각형 행렬의 대각합)
det: determinant (행렬식)

$$l_i(\theta, \phi) = -E_{z \sim q_\theta(z|x_i)}[\log p_\phi(x_i|z)] + KL(q_\theta(z|x_i) || p(z))$$

$$D_{KL}[N(\mu(X), \Sigma(X)) || N(0, 1)] = \frac{1}{2} (\text{tr}(\Sigma(X)) + \mu(X)^T \mu(X) - k - \log \det(\Sigma(X)))$$

$$D_{KL}[N(\mu(X), \Sigma(X)) || N(0, 1)] = \frac{1}{2} \left(\sum_k \Sigma(X) + \sum_k \mu^2(X) - \sum_k 1 - \log \prod_k \Sigma(X) \right)$$

대각행렬의
determinant는
각 대각요소의
곱과 같음



Variational Autoencoder

tr: trace (정사각형 행렬의 대각합)
det: determinant (행렬식)

$$l_i(\theta, \phi) = -E_{z \sim q_\theta(z|x_i)}[\log p_\phi(x_i|z)] + KL(q_\theta(z|x_i)||p(z))$$

$$D_{KL}[N(\mu(X), \Sigma(X))||N(0, 1)] = \frac{1}{2} (\text{tr}(\Sigma(X)) + \mu(X)^T \mu(X) - k - \log \det(\Sigma(X)))$$

$$D_{KL}[N(\mu(X), \Sigma(X))||N(0, 1)] = \frac{1}{2} \left(\sum_k \Sigma(X) + \sum_k \mu^2(X) - \sum_k 1 - \log \prod_k \Sigma(X) \right)$$

대각행렬의
determinant는
각 대각요소의
곱과 같음

7. If A is a **triangular matrix**, i.e. $a_{ij} = 0$ whenever $i > j$ or, alternatively, whenever $i < j$, then its determinant equals the product of the diagonal entries:

$$\det(A) = a_{1,1} a_{2,2} \cdots a_{n,n} = \prod_{i=1}^n a_{i,i}.$$

출처: 위키피디아 determinant

Variational Autoencoder

tr: trace (정사각형 행렬의 대각합)
det: determinant (행렬식)

$$l_i(\theta, \phi) = -E_{z \sim q_\theta(z|x_i)}[\log p_\phi(x_i|z)] + KL(q_\theta(z|x_i) || p(z))$$

$$D_{KL}[N(\mu(X), \Sigma(X)) || N(0, 1)] = \frac{1}{2} (\text{tr}(\Sigma(X)) + \mu(X)^T \mu(X) - k - \log \det(\Sigma(X)))$$

$$\begin{aligned} D_{KL}[N(\mu(X), \Sigma(X)) || N(0, 1)] &= \frac{1}{2} \left(\sum_k \Sigma(X) + \sum_k \mu^2(X) - \sum_k 1 - \log \prod_k \Sigma(X) \right) \\ &= \frac{1}{2} \left(\sum_k \Sigma(X) + \sum_k \mu^2(X) - \sum_k 1 - \sum_k \log \Sigma(X) \right) \end{aligned}$$

Variational Autoencoder

tr: trace (정사각형 행렬의 대각합)
det: determinant (행렬식)

$$l_i(\theta, \phi) = -E_{z \sim q_\theta(z|x_i)}[\log p_\phi(x_i|z)] + KL(q_\theta(z|x_i) || p(z))$$

$$D_{KL}[N(\mu(X), \Sigma(X)) || N(0, 1)] = \frac{1}{2} (\text{tr}(\Sigma(X)) + \mu(X)^T \mu(X) - k - \log \det(\Sigma(X)))$$

$$\begin{aligned} D_{KL}[N(\mu(X), \Sigma(X)) || N(0, 1)] &= \frac{1}{2} \left(\sum_k \Sigma(X) + \sum_k \mu^2(X) - \sum_k 1 - \log \prod_k \Sigma(X) \right) \\ &= \frac{1}{2} \left(\sum_k \Sigma(X) + \sum_k \mu^2(X) - \sum_k 1 - \sum_k \log \Sigma(X) \right) \\ &= \frac{1}{2} \sum_k (\Sigma(X) + \mu^2(X) - 1 - \log \Sigma(X)) \end{aligned}$$

Variational Autoencoder

tr: trace (정사각형 행렬의 대각합)
det: determinant (행렬식)

$$l_i(\theta, \phi) = -E_{z \sim q_\theta(z|x_i)}[\log p_\phi(x_i|z)] + KL(q_\theta(z|x_i) || p(z))$$

$$D_{KL}[N(\mu(X), \Sigma(X)) || N(0, 1)] = \frac{1}{2} (\text{tr}(\Sigma(X)) + \mu(X)^T \mu(X) - k - \log \det(\Sigma(X)))$$

$$D_{KL}[N(\mu(X), \Sigma(X)) || N(0, 1)] = \frac{1}{2} \left(\sum_k \Sigma(X) + \sum_k \mu^2(X) - \sum_k 1 - \log \prod_k \Sigma(X) \right)$$

$$= \frac{1}{2} \left(\sum_k \Sigma(X) + \sum_k \mu^2(X) - \sum_k 1 - \sum_k \log \Sigma(X) \right)$$

$$= \frac{1}{2} \sum_k (\Sigma(X) + \mu^2(X) - 1 - \log \Sigma(X))$$

$$D_{KL}[N(\mu(X), \Sigma(X)) || N(0, 1)] = \frac{1}{2} \sum_k (\exp(\Sigma(X)) + \mu^2(X) - 1 - \Sigma(X))$$

Variational Autoencoder

$$l_i(\theta, \phi) = -E_{z \sim q_\theta(z|x_i)}[\log p_\phi(x_i|z)] + \textcolor{blue}{KL(q_\theta(z|x_i)||p(z))}$$

$$D_{KL}[N(\mu(X), \Sigma(X))||N(0, 1)] = \frac{1}{2} (\text{tr}(\Sigma(X)) + \mu(X)^T \mu(X) - k - \log \det(\Sigma(X)))$$

$$D_{KL}[N(\mu(X), \Sigma(X))||N(0, 1)] = \frac{1}{2} \left(\sum_k \Sigma(X) + \sum_k \mu^2(X) - \sum_k 1 - \log \prod_k \Sigma(X) \right)$$

$$= \frac{1}{2} \left(\sum_k \Sigma(X) + \sum_k \mu^2(X) - \sum_k 1 - \sum_k \log \Sigma(X) \right)$$

$$= \frac{1}{2} \sum_k (\textcolor{blue}{\Sigma(X)} + \mu^2(X) - 1 - \textcolor{blue}{\log \Sigma(X)})$$

$$D_{KL}[N(\mu(X), \Sigma(X))||N(0, 1)] = \frac{1}{2} \sum_k (\textcolor{blue}{\exp(\Sigma(X))} + \mu^2(X) - 1 - \textcolor{blue}{\Sigma(X)})$$

Variational Autoencoder

$$l_i(\theta, \phi) = -E_{z \sim q_\theta(z|x_i)}[\log p_\phi(x_i|z)] + KL(q_\theta(z|x_i)||p(z))$$

$$D_{KL}[N(\mu(X), \Sigma(X))||N(0, 1)] = \frac{1}{2} (\text{tr}(\Sigma(X)) + \mu(X)^T \mu(X) - k - \log \det(\Sigma(X)))$$

$$D_{KL}[N(\mu(X), \Sigma(X))||N(0, 1)] = \frac{1}{2} \left(\sum_k \Sigma(X) + \sum_k \mu^2(X) - \sum_k 1 - \log \prod_k \Sigma(X) \right)$$

$$= \frac{1}{2} \left(\sum_k \Sigma(X) + \sum_k \mu^2(X) - \sum_k 1 - \sum_k \log \Sigma(X) \right)$$

$$= \frac{1}{2} \sum_k (\underline{\Sigma(X)} + \mu^2(X) - 1 - \underline{\log \Sigma(X)})$$

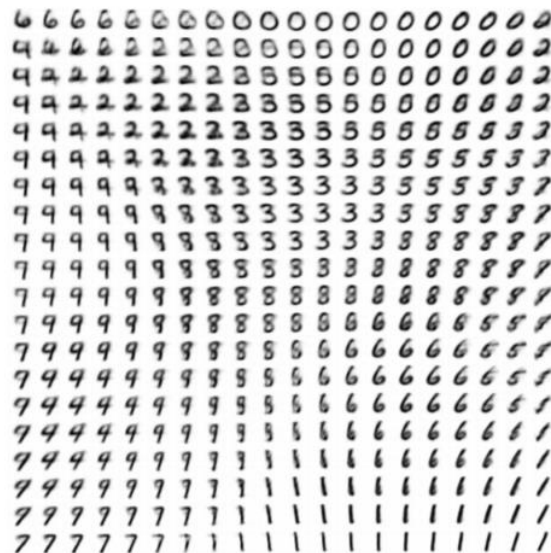
numerical stability

$$D_{KL}[N(\mu(X), \Sigma(X))||N(0, 1)] = \frac{1}{2} \sum_k (\underline{\exp(\Sigma(X))} + \mu^2(X) - 1 - \underline{\Sigma(X)})$$

Variational Autoencoder



(a) Learned Frey Face manifold

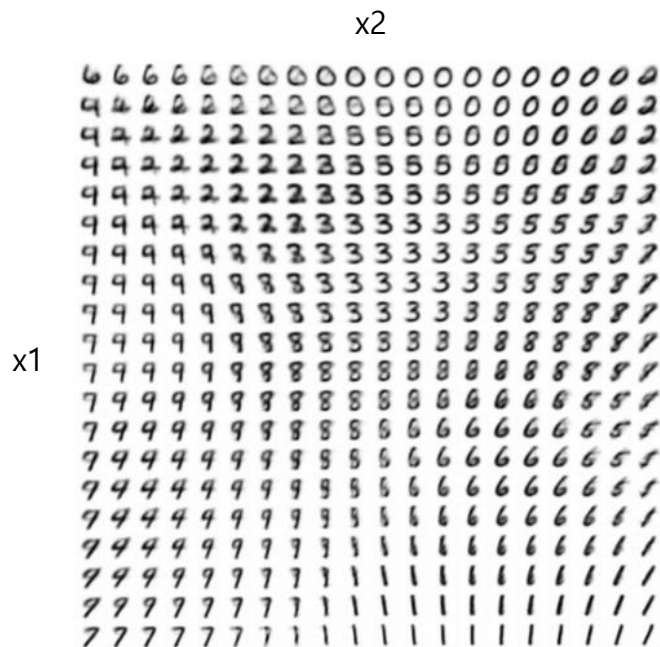


(b) Learned MNIST manifold

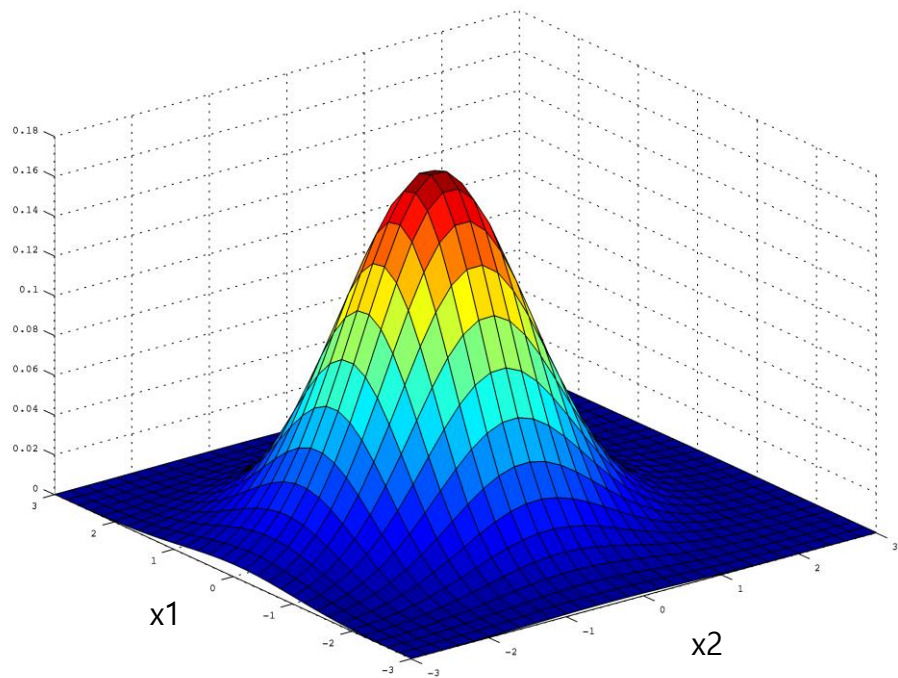
2D Latent Space

Figure 4: Visualisations of learned data manifold for generative models with two-dimensional latent space, learned with AEVB. Since the prior of the latent space is Gaussian, linearly spaced coordinates on the unit square were transformed through the inverse CDF of the Gaussian to produce values of the latent variables \mathbf{z} . For each of these values \mathbf{z} , we plotted the corresponding generative $p_{\theta}(\mathbf{x}|\mathbf{z})$ with the learned parameters θ .

Variational Autoencoder



(b) Learned MNIST manifold



Variational Autoencoder

```
class Encoder(nn.Module):
    def __init__(self):
        super(Encoder, self).__init__()
        self.fc1 = nn.Sequential(
            nn.Conv2d(1, 8, 3, padding=1),
            nn.ReLU(),
            nn.BatchNorm2d(8),
            nn.MaxPool2d(2, 2),
            nn.Conv2d(8, 16, 3, padding=1),
            nn.ReLU(),
            nn.BatchNorm2d(16),
            nn.MaxPool2d(2, 2),
            nn.Conv2d(16, 32, 3, padding=1),
            nn.ReLU(),
        )
        self.fc2_1 = nn.Sequential(
            nn.Linear(32*7*7, 800),
            nn.Linear(800, hidden_size),
        )
        self.fc2_2 = nn.Sequential(
            nn.Linear(32*7*7, 800),
            nn.Linear(800, hidden_size),
        )
        self.relu = nn.ReLU()
```

Variational Autoencoder

Encoder는 우선 일반적인
방법으로 feature 를 뽑고

```
class Encoder(nn.Module):
    def __init__(self):
        super(Encoder, self).__init__()
        self.fc1 = nn.Sequential(
            nn.Conv2d(1, 8, 3, padding=1),
            nn.ReLU(),
            nn.BatchNorm2d(8),
            nn.MaxPool2d(2, 2),
            nn.Conv2d(8, 16, 3, padding=1),
            nn.ReLU(),
            nn.BatchNorm2d(16),
            nn.MaxPool2d(2, 2),
            nn.Conv2d(16, 32, 3, padding=1),
            nn.ReLU(),
        )
        self.fc2_1 = nn.Sequential(
            nn.Linear(32*7*7, 800),
            nn.Linear(800, hidden_size),
        )
        self.fc2_2 = nn.Sequential(
            nn.Linear(32*7*7, 800),
            nn.Linear(800, hidden_size),
        )
        self.relu = nn.ReLU()
```

Variational Autoencoder

Encoder는 우선 일반적인
방법으로 feature 를 뽑고

지정한 hidden size로 μ 뽑고

```
class Encoder(nn.Module):
    def __init__(self):
        super(Encoder, self).__init__()
        self.fc1 = nn.Sequential(
            nn.Conv2d(1, 8, 3, padding=1),
            nn.ReLU(),
            nn.BatchNorm2d(8),
            nn.MaxPool2d(2, 2),
            nn.Conv2d(8, 16, 3, padding=1),
            nn.ReLU(),
            nn.BatchNorm2d(16),
            nn.MaxPool2d(2, 2),
            nn.Conv2d(16, 32, 3, padding=1),
            nn.ReLU(),
        )
        self.fc2_1 = nn.Sequential(
            nn.Linear(32*7*7, 800),
            nn.Linear(800, hidden_size),
        )
        self.fc2_2 = nn.Sequential(
            nn.Linear(32*7*7, 800),
            nn.Linear(800, hidden_size),
        )
        self.relu = nn.ReLU()
```

Variational Autoencoder

Encoder는 우선 일반적인
방법으로 feature 를 뽑고

지정한 hidden size로 μ 뽑고

지정한 hidden size로 σ 뽑고

```
class Encoder(nn.Module):
    def __init__(self):
        super(Encoder, self).__init__()
        self.fc1 = nn.Sequential(
            nn.Conv2d(1, 8, 3, padding=1),
            nn.ReLU(),
            nn.BatchNorm2d(8),
            nn.MaxPool2d(2, 2),
            nn.Conv2d(8, 16, 3, padding=1),
            nn.ReLU(),
            nn.BatchNorm2d(16),
            nn.MaxPool2d(2, 2),
            nn.Conv2d(16, 32, 3, padding=1),
            nn.ReLU(),
        )
        self.fc2_1 = nn.Sequential(
            nn.Linear(32*7*7, 800),
            nn.Linear(800, hidden_size),
        )
        self.fc2_2 = nn.Sequential(
            nn.Linear(32*7*7, 800),
            nn.Linear(800, hidden_size),
        )
        self.relu = nn.ReLU()
```

Variational Autoencoder

Encoder class 내부

```
def encode(self, x):
    out = self.fc1(x)
    out = out.view(batch_size, -1)
    out = self.relu(out)
    mu = self.fc2_1(out)
    log_var = self.fc2_2(out)

    return mu, log_var

def reparametrize(self, mu, logvar):
    std = logvar.mul(0.5).exp_()

    eps = torch.FloatTensor(std.size()).normal_()
    eps = Variable(eps).cuda()

    return eps.mul(std).add_(mu)

def forward(self, x):
    mu, logvar = self.encode(x)
    reparam = self.reparametrize(mu, logvar)

    return mu, logvar, reparam
```

Variational Autoencoder

입력이 들어오면
mu와 log_var 리턴



Encoder class 내부

```
def encode(self, x):
    out = self.fc1(x)
    out = out.view(batch_size, -1)
    out = self.relu(out)
    mu = self.fc2_1(out)
    log_var = self.fc2_2(out)

    return mu, log_var

def reparametrize(self, mu, logvar):
    std = logvar.mul(0.5).exp_()

    eps = torch.FloatTensor(std.size()).normal_()
    eps = Variable(eps).cuda()

    return eps.mul(std).add_(mu)

def forward(self, x):
    mu, logvar = self.encode(x)
    reparam = self.reparametrize(mu, logvar)

    return mu, logvar, reparam
```

Variational Autoencoder

Encoder class 내부

입력이 들어오면
mu와 log_var 리턴

```
def encode(self, x):  
    out = self.fc1(x)  
    out = out.view(batch_size, -1)  
    out = self.relu(out)  
    mu = self.fc2_1(out)  
    log_var = self.fc2_2(out)  
  
    return mu, log_var
```

sampling은 미분할 수 없기
때문에 back propagation을
위해 reparameterize

```
def reparametrize(self, mu, logvar):  
    std = logvar.mul(0.5).exp_  
  
    eps = torch.FloatTensor(std.size()).normal_  
    eps = Variable(eps).cuda()  
  
    return eps.mul(std).add_(mu)
```

```
def forward(self, x):  
    mu, logvar = self.encode(x)  
    reparam = self.reparametrize(mu, logvar)  
  
    return mu, logvar, reparam
```


Variational Autoencoder

Encoder class 내부

입력이 들어오면
mu와 log_var 리턴

```
def encode(self, x):  
    out = self.fc1(x)  
    out = out.view(batch_size, -1)  
    out = self.relu(out)  
    mu = self.fc2_1(out)  
    log_var = self.fc2_2(out)  
  
    return mu, log_var
```

sampling은 미분할 수 없기
때문에 back propagation을
위해 reparameterize

```
def reparametrize(self, mu, logvar):  
    std = logvar.mul(0.5).exp_  
  
    eps = torch.FloatTensor(std.size()).normal_  
    eps = Variable(eps).cuda()  
  
    return eps.mul(std).add_(mu)
```

모든 과정을 forward로 지정

```
def forward(self, x):  
    mu, logvar = self.encode(x)  
    reparam = self.reparametrize(mu, logvar)  
  
    return mu, logvar, reparam
```

Variational Autoencoder

```
class Decoder(nn.Module):
    def __init__(self):
        super(Decoder, self).__init__()
        self.fc1 = nn.Sequential(
            nn.Linear(hidden_size, 800),
            nn.ReLU(),
            nn.BatchNorm1d(800),
            nn.Linear(800, 1568),
            nn.ReLU(),
        )
        self.fc2 = nn.Sequential(
            nn.ConvTranspose2d(32, 16, 3, 2, 1, 1),
            nn.ReLU(),
            nn.BatchNorm2d(16),
            nn.ConvTranspose2d(16, 8, 3, 2, 1, 1),
            nn.ReLU(),
            nn.BatchNorm2d(8),
            nn.ConvTranspose2d(8, 1, 3, 1, 1, 1),
            nn.BatchNorm2d(1),
        )
        self.sigmoid = nn.Sigmoid()
        self.relu = nn.ReLU()

    def forward(self, x):
        out = self.fc1(x)
        out = self.relu(out)
        out = out.view(batch_size, 32, 7, 7)
        out = self.fc2(out)
        out = self.sigmoid(out)
        out = out.view(batch_size, 28, 28, 1)

        return out
```

Variational Autoencoder

원래 이미지 모양대로
decoding 해주는 부분



```
class Decoder(nn.Module):
    def __init__(self):
        super(Decoder, self).__init__()
        self.fc1 = nn.Sequential(
            nn.Linear(hidden_size, 800),
            nn.ReLU(),
            nn.BatchNorm1d(800),
            nn.Linear(800, 1568),
            nn.ReLU(),
        )
        self.fc2 = nn.Sequential(
            nn.ConvTranspose2d(32, 16, 3, 2, 1, 1),
            nn.ReLU(),
            nn.BatchNorm2d(16),
            nn.ConvTranspose2d(16, 8, 3, 2, 1, 1),
            nn.ReLU(),
            nn.BatchNorm2d(8),
            nn.ConvTranspose2d(8, 1, 3, 1, 1, 1),
            nn.BatchNorm2d(1),
        )
        self.sigmoid = nn.Sigmoid()
        self.relu = nn.ReLU()

    def forward(self, x):
        out = self.fc1(x)
        out = self.relu(out)
        out = out.view(batch_size, 32, 7, 7)
        out = self.fc2(out)
        out = self.sigmoid(out)
        out = out.view(batch_size, 28, 28, 1)

        return out
```

Variational Autoencoder

```
reconstruction_function = nn.BCELoss(size_average=True)

def loss_function(recon_x, x, mu, logvar):
    BCE = reconstruction_function(recon_x, x)

    # see Appendix B from VAE paper:
    # Kingma and Welling. Auto-Encoding Variational Bayes. ICLR, 2014
    # https://arxiv.org/abs/1312.6114
    # 0.5 * sum(1 + log(sigma^2) - mu^2 - sigma^2)
    KLD_element = mu.pow(2).add_(logvar.exp()).mul_(-1).add_(1).add_(logvar)
    KLD = torch.sum(KLD_element).mul_(-0.5)

    return BCE + KLD

parameters = list(encoder.parameters()) + list(decoder.parameters())
optimizer = torch.optim.Adam(parameters, lr=learning_rate)

try:
    encoder, decoder = torch.load('./model/conv_variational_autoencoder.pkl')
    print("\n-----model restored-----\n")
except:
    print("\n-----model not restored-----\n")
    pass
```

Variational Autoencoder

decoder에서 마지막 단에
sigmoid를 통과함으로써
0~1의 값을 가지고 이를
확률로 간주하여 원본과
binary cross entropy를 계산

```
reconstruction_function = nn.BCELoss(size_average=True)

def loss_function(recon_x, x, mu, logvar):
    BCE = reconstruction_function(recon_x, x)

    # see Appendix B from VAE paper:
    # Kingma and Welling. Auto-Encoding Variational Bayes. ICLR, 2014
    # https://arxiv.org/abs/1312.6114
    # 0.5 * sum(1 + log(sigma^2) - mu^2 - sigma^2)
    KLD_element = mu.pow(2).add_(logvar.exp()).mul_(-1).add_(1).add_(logvar)
    KLD = torch.sum(KLD_element).mul_(-0.5)

    return BCE + KLD

parameters = list(encoder.parameters()) + list(decoder.parameters())
optimizer = torch.optim.Adam(parameters, lr=learning_rate)

try:
    encoder, decoder = torch.load('./model/conv_variational_autoencoder.pkl')
    print("\n-----model restored-----\n")
except:
    print("\n-----model not restored-----\n")
    pass
```

Variational Autoencoder

decoder에서 마지막 단계
sigmoid를 통과함으로써
0~1의 값을 가지고 이를
확률로 간주하여 원본과
binary cross entropy를 계산

Gaussian 분포와의 KL diver-
gence 와 reconstruction loss
둘 다 리턴

```
reconstruction_function = nn.BCELoss(size_average=True)
```

```
def loss_function(recon_x, x, mu, logvar):  
    BCE = reconstruction_function(recon_x, x)  
  
    # see Appendix B from VAE paper:  
    # Kingma and Welling. Auto-Encoding Variational Bayes. ICLR, 2014  
    # https://arxiv.org/abs/1312.6114  
    # 0.5 * sum(1 + log(sigma^2) - mu^2 - sigma^2)  
    KLD_element = mu.pow(2).add_(logvar.exp()).mul_(-1).add_(1).add_(logvar)  
    KLD = torch.sum(KLD_element).mul_(-0.5)  
  
    return BCE + KLD
```

```
parameters = list(encoder.parameters()) + list(decoder.parameters())  
optimizer = torch.optim.Adam(parameters, lr=learning_rate)  
  
try:  
    encoder, decoder = torch.load('./model/conv_variational_autoencoder.pkl')  
    print("\n-----model restored-----\n")  
except:  
    print("\n-----model not restored-----\n")  
    pass
```

Variational Autoencoder

decoder에서 마지막 단계에 sigmoid를 통과함으로써 0~1의 값을 가지고 이를 확률로 간주하여 원본과 binary cross entropy를 계산

Gaussian 분포와의 KL divergence 와 reconstruction loss 둘 다 리턴

encoder와 decoder의 parameter를 묶어서 학습

```
reconstruction_function = nn.BCELoss(size_average=True)
```

```
def loss_function(recon_x, x, mu, logvar):  
    BCE = reconstruction_function(recon_x, x)  
  
    # see Appendix B from VAE paper:  
    # Kingma and Welling. Auto-Encoding Variational Bayes. ICLR, 2014  
    # https://arxiv.org/abs/1312.6114  
    # 0.5 * sum(1 + log(sigma^2) - mu^2 - sigma^2)  
    KLD_element = mu.pow(2).add_(logvar.exp()).mul_(-1).add_(1).add_(logvar)  
    KLD = torch.sum(KLD_element).mul_(-0.5)  
  
    return BCE + KLD
```

```
parameters = list(encoder.parameters()) + list(decoder.parameters())  
optimizer = torch.optim.Adam(parameters, lr=learning_rate)
```

```
try:  
    encoder, decoder = torch.load('./model/conv_variational_autoencoder.pkl')  
    print("\n-----model restored-----\n")  
except:  
    print("\n-----model not restored-----\n")  
pass
```

Variational Autoencoder

decoder에서 마지막 단계에 sigmoid를 통과함으로써 0~1의 값을 가지고 이를 확률로 간주하여 원본과 binary cross entropy를 계산

Gaussian 분포와의 KL divergence 와 reconstruction loss 둘 다 리턴

encoder와 decoder의 parameter를 묶어서 학습

저장된 encoder, decoder 모델을 불러오는 부분

```
reconstruction_function = nn.BCELoss(size_average=True)
```

```
def loss_function(recon_x, x, mu, logvar):  
    BCE = reconstruction_function(recon_x, x)  
  
    # see Appendix B from VAE paper:  
    # Kingma and Welling. Auto-Encoding Variational Bayes. ICLR, 2014  
    # https://arxiv.org/abs/1312.6114  
    # 0.5 * sum(1 + log(sigma^2) - mu^2 - sigma^2)  
    KLD_element = mu.pow(2).add_(logvar.exp()).mul_(-1).add_(1).add_(logvar)  
    KLD = torch.sum(KLD_element).mul_(-0.5)  
  
    return BCE + KLD
```

```
parameters = list(encoder.parameters()) + list(decoder.parameters())  
optimizer = torch.optim.Adam(parameters, lr=learning_rate)
```

```
try:  
    encoder, decoder = torch.load('./model/conv_variational_autoencoder.pkl')  
    print("\n-----model restored-----\n")  
except:  
    print("\n-----model not restored-----\n")  
    pass
```


방법 1

```
torch.save(the_model.state_dict(), PATH)
```

```
the_model = TheModelClass(*args, **kwargs)  
the_model.load_state_dict(torch.load(PATH))
```

방법 2

```
torch.save(the_model, PATH)  
the_model = torch.load(PATH)
```

모델 파라미터만 저장
(파이토치에서 추천하는 방법)

방법 1

```
torch.save(the_model.state_dict(), PATH)
```

```
the_model = TheModelClass(*args, **kwargs)  
the_model.load_state_dict(torch.load(PATH))
```

방법 2

```
torch.save(the_model, PATH)  
the_model = torch.load(PATH)
```

Variational Autoencoder

모델 저장하는 방법

모델 파라미터만 저장
(파이토치에서 추천하는 방법)

```
# 방법 1

torch.save(the_model.state_dict(), PATH)

the_model = TheModelClass(*args, **kwargs)
the_model.load_state_dict(torch.load(PATH))
```

모델 전체를 저장
(클래스 구조가 바뀌거나 하면
제대로 작동하지 않음)

```
# 방법 2

torch.save(the_model, PATH)
the_model = torch.load(PATH)
```

Variational Autoencoder

```
for i in range(num_epoch):
    for j,[image,label] in enumerate(train_loader):
        optimizer.zero_grad()

        image = Variable(image).cuda()
        reparam,mu,log_var = encoder(image)
        output = decoder(reparam)

        loss = loss_function(output, image, mu, log_var)

        loss.backward()
        optimizer.step()

    if j % 10 == 0:
        torch.save([encoder,decoder], './model/conv_variational_autoencoder.pkl')
        print(loss)
```

학습 및 모델 저장

Q&A

