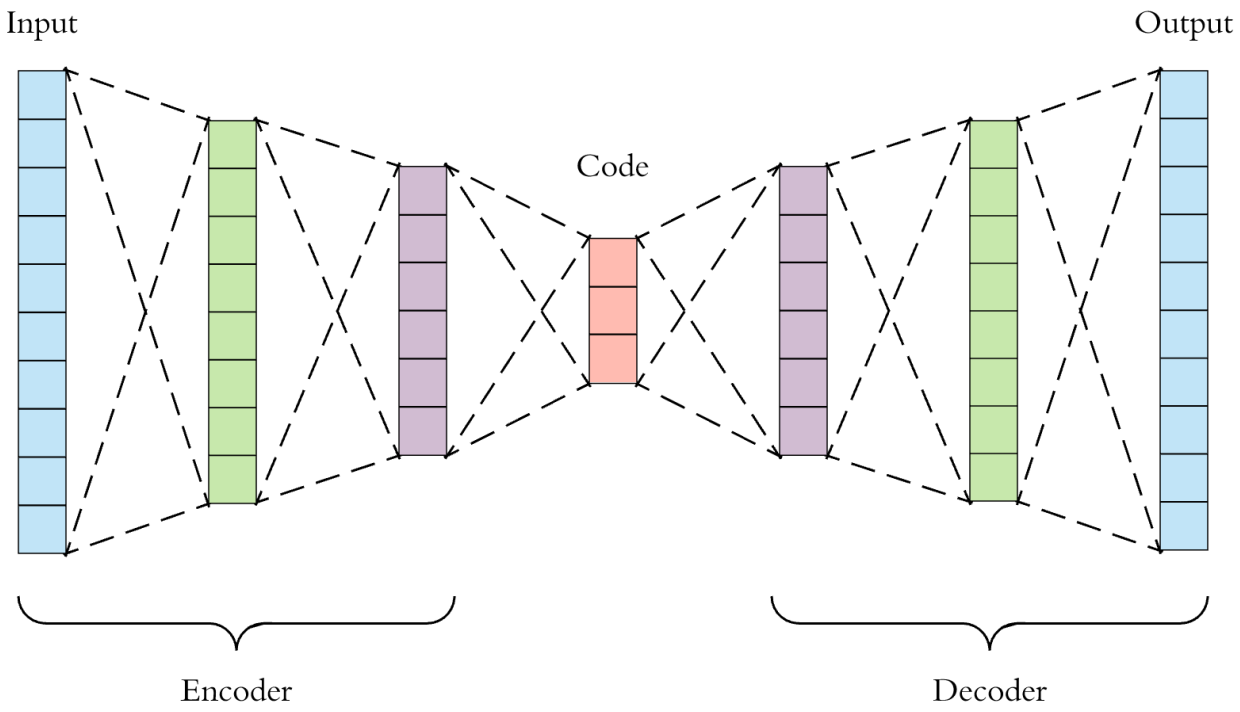
❤️

VAE (Variational Autoencoder)



VAE

AUTOENCODER(NN)

Basic Concepts

- Unsupervised Learning(without labels)
- Can be used as a subclass of supervised learning
- Expensive training time
- Learning objective:
  - Create the most similar output as the input
  - Find useful patterns in input datasets
  - Automatically learn to compress efficiently through NN

    - Finding the encoder/decoder pair that produces the minimum loss when decompressing the data
- Categories Autoencoder falls into:
    - Efficient data encoding
    - Feature learning
    - Representation Learning
    - Dimensionality Reduction

Architecture of AutoEncoder

$X \rightarrow z \rightarrow X'$(output)

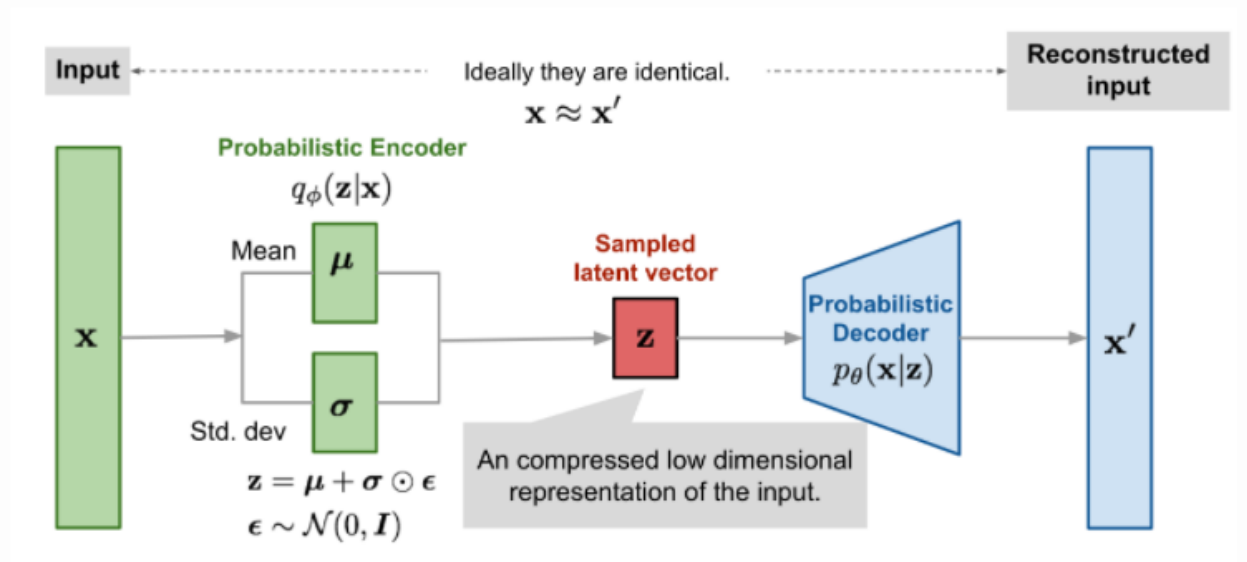
- Encoder and Decoder
    - The output shape of decoder is the same as the input shape of encoder.
    - The output content of decoder is the similarly represented as the input content of encoder.
- Usually uses MSE as the loss function.
- Code (z; aka. latent variable, latent vector, latent expression): Internal product first processed by the encoder, which then is passed through the decoder to create the output.
    - code is normally set as a smaller size than the input.

```
#code for MNIST
class Autoencoder(nn.Module):
    def __init__(self):
        super(Autoencoder,self).__init__()
        self.encoder = nn.Linear(28*28,20)
        self.decoder = nn.Linear(20,28*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)#reset to original size
        return out
```

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

for i in range(num_epoch):
  for j,[image, label] in enumerate(train_loader):
            optimizer.zero_grad()
            output = model.forward(image)
            loss = loss_func(ouput,image)
            loss.backward()
            optimizer.step()
```

- Loss: $L(x, x') = \sum_{i=1}^{n} |x_i - x'_i|$

VAE



Concept

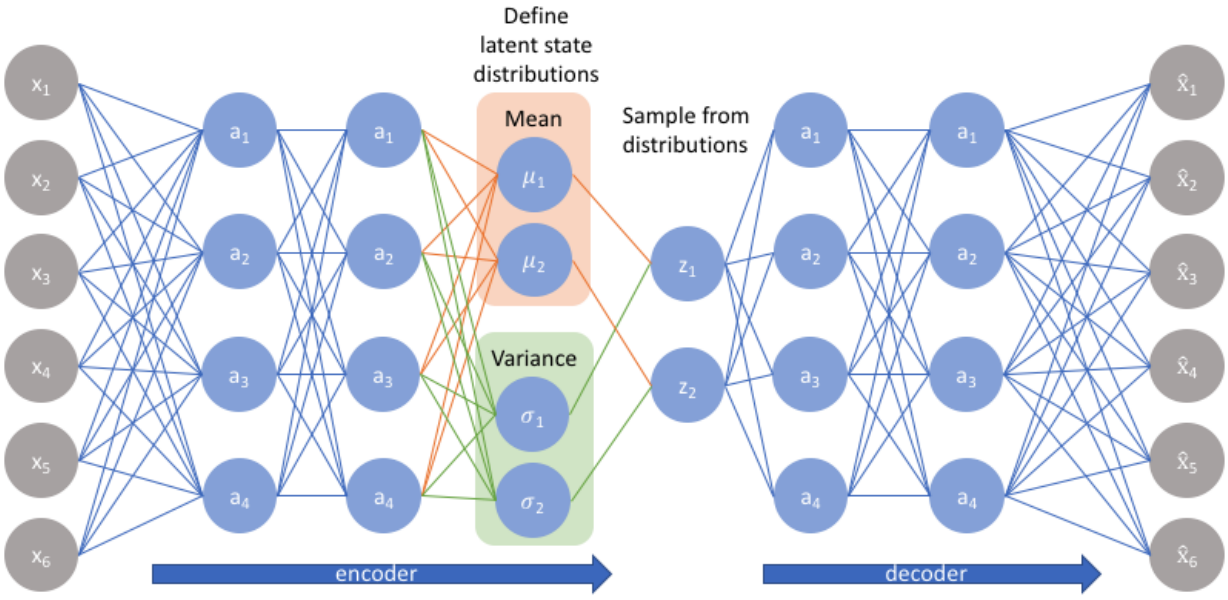
- Generative Model(along with GAN)
- Calculates $mean\ vector\ \mu$ and $variance\ vector\ \sigma$ in the input using encoder → samples latent variable z according to probability → use decoder to represent the original data

- By adjusting z, you can create continuously varying data
- latent variable follows probability distribution(continuous)
  - every time, same input → different output(different latent variable also)
  - existence of unknown input → still works

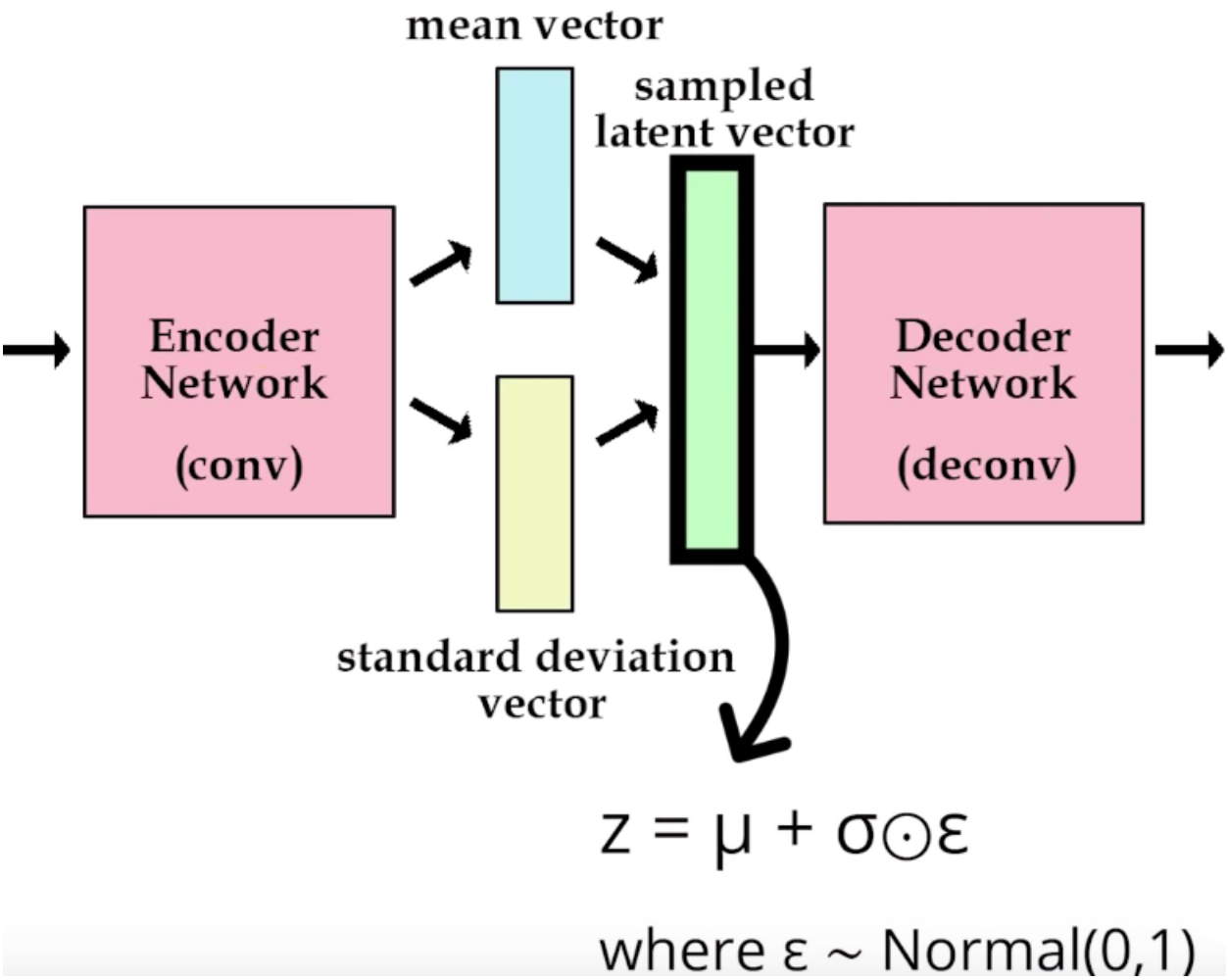
Application

- Nose reduction
- Finding problematic parts
- Clustering using latent variables

Architecture



Reparameterization Trick



- $z = \mu + \epsilon\sigma$
- ϵ : sampled value from normal distribution(mean = 0, std = 1)
  - → derivation (o), back propagation (o)

Loss

Considers two things:

- How different is the input-imitated-output from input
- How divergent is the latent variable

$E = E_{rec} + E_{reg}$

(E_{reg}: regularization term to regulate divergence of z; diverging from 0)

(E_{rec}: represents how different the output is compared to input)

Reconstruction Loss (E_{rec})

$E_{rec} = \frac{1}{h}\sum_{i=1}^{h}\sum_{j=1}^{m}(-x_{ij}logy_{ij} - (1-x_{ij})log(1-y_{ij}))$

- x_{ij} : input of VAE
- y_{ij} : output of VAE
- h : batch size
- m : number of neurons between input layer and output layer

(cross entropy loss; how far two values are, minimum when x == y)

$e_{rec} = -xlogy - (1-x)log(1-y)$

Regularization Term (E_{reg})

E_{reg} =$\frac{1}{h}\sum_{i=1}^{h}\sum_{k=1}^{n} -\frac{1}{2}(1 + log\sigma_{ik}^2 - \mu_{ik}^2 - \sigma_{ik}^2)$

- h: batch size
- n: number of latent vectors
- σ_{ik} : std
- μ_{ik} : mean

$e_{reg} = -\frac{1}{2}(1 + log\sigma^2 - \mu^2 - \sigma^2)$

- minimum(0) when σ_{ik} = 1, μ_{ik} = 0
- enlarges when σ_{ik} ≠ 1, μ_{ik} ≠ 0
- represents how latent variable is far from std =1, mean = 0

```
import numpy as np
#import cupy as np
import matplotlib.pyplot as plt
```

```
from sklearn import datasets

img_size = 8#height==width==8
n_in_out = img_size*img_size
n_mid = 16
n_z = 2#neurons in hidden layer

eta = 0.001
epochs = 201
batch_size = 32
interval =20

#create training dataset
digits_data = datasets.load_digits()
x_train = np.asarray(digits_data.data)
x_train /=15
t_train = digits_data.target

#정규분포의 파라미터를 계산하는 신경망층
class BaseLayer:
    def update(self, eta):
        self.w -= eta*self.grad_w
        self.b -= eta*self.grad_b


#은닉층
class MiddleLayer(BaseLayer):
    def __init__(self, n_upper, n):
        #He의 초깃값
        self.w = np.random.randn(n_upper, n)*np.sqrt(2/n_upper)
        self.b = np.zeros(n)

    def forward(self, x):
        self.x = x
        self.u = np.dot(x,self.w)+self.b
        self.y = np.where(self.u<=0,0,self.u)#relu

    def backward(self, grad_y):
        delta = grad_y*np.where(self.u<=0,0,1)
        self.grad_w = np.dot(self.x.T, delta)
        self.grad_b = np.sum(delta, axis = 0)
        self.grad_x = np.dot(delta, self.w.T)


#정규분포의 파라미터를 계산하는 신경망층
class ParamsLayer(BaseLayer):
    def __init__(self, n_upper, n):
        self.w = np.random.randn(n_upper, n)/np.sqrt(n_upper)
        self.b = np.zeros(n)

    def forward(self,x):
        self.x = x
        u= np.dot(x,self.w)+self.b
        self.y = u

    def backward(self, grad_y):
        delta = grad_y
        self.grad_w = np.dot(self.x.T, delta)
        self.grad_b = np.sum(delta, axis = 0)
```

```
        self.grad_x = np.dot(delta, self.w.T)

class OutputLayer(BaseLayer):
    def __init__(self, n_upper, n):
        self.w = np.random.randn(n_upper, n)/np.sqrt(n_upper)
        self.b = np.zeros(n)

    def forward(self, x):
        self.x = x
        u = np.dot(x, self.w)+self.b
        self.y = 1/(1+np.exp(-u))

    def backward(self,t):
        delta = self.y - t
        self.grad_w = np.dot(self.x.T, delta)
        self.grad_b = np.sum(delta, axis = 0)
        self.grad_x = np.dot(delta, self.w.T)

#잠재 변수를 샘플링하는 층
class LatentLayer:
    def forward(self, mu, log_var):
        self.mu = mu#평균
        self.log_var = log_var

        self.epsilon = np.random.randn(*log_var.shape)
        self.z = mu+self.epsilon*np.exp(log_var/2)

    def backward(self, grad_z):
        self.grad_mu = grad_z + self.mu

        self.grad_log_var = grad_z*self.epsilon/2*np.exp(self.log_var/2)-0.5*(1-np.exp(self.log_var))

#encoders
middle_layer_enc = MiddleLayer(n_in_out, n_mid)
mu_layer = ParamsLayer(n_mid, n_z)
log_var_layer = ParamsLayer(n_mid, n_z)
z_layer = LatentLayer()

#decoders
middle_layer_dec = MiddleLayer(n_z, n_mid)
output_layer = OutputLayer(n_mid, n_in_out)

def forward_propagation(x_mb):
    #encoder
    middle_layer_enc.forward(x_mb)
    mu_layer.forward(middle_layer_enc.y)
    log_var_layer.forward(middle_layer_enc.y)
    z_layer.forward(mu_layer.y, log_var_layer.y)

    #decoder
    middle_layer_dec.forward(z_layer.z)
    output_layer.forward(middle_layer_dec.y)


def backpropagation(t_mb):
    #decocder
    output_layer.backward(t_mb)
    middle_layer_dec.backward(output_layer.grad_x)
```

```
    #encoder
    z_layer.backward(middle_layer_dec.grad_x)
    log_var_layer.backward(z_layer.grad_log_var)
    mu_layer.backward(z_layer.grad_mu)
    middle_layer_enc.backward(mu_layer.grad_x + log_var_layer.grad_x)

#parameter update function
def update_params():
    middle_layer_enc.update(eta)
    mu_layer.update(eta)
    log_var_layer.update(eta)
    middle_layer_dec.update(eta)
    output_layer.update(eta)

#get loss function
def get_rec_error(y,t):
    eps = 1e-7
    return -np.sum(t*np.log(y+eps)+(1-t)*np.log(1-y+eps))/len(y)

def get_reg_error(mu, log_var):
    return -np.sum(1+log_var-mu**2 -np.exp(log_var))/len(mu)


rec_error_record = []
reg_error_record = []
total_error_record = []
n_batch = len(x_train)//batch_size

#train
for i in range(epochs):
    index_random = np.arange(len(x_train))
    np.random.shuffle(index_random)
    for j in range(n_batch):
        mb_index = index_random[j*batch_size:(j+1)*batch_size]
        x_mb = x_train[mb_index,:]

        forward_propagation(x_mb)
        backpropagation(x_mb)

        update_params()
    forward_propagation(x_train)

    rec_error = get_rec_error(output_layer.y, x_train)
    reg_error = get_reg_error(mu_layer.y, log_var_layer.y)
    total_error = rec_error+reg_error

    rec_error_record.append(rec_error)
    reg_error_record.append(reg_error)
    total_error_record.append(total_error)

    if i%interval == 0:
        print("epoch:",i,"rec_error:",rec_error, "reg_error:",reg_error, "total errror:",total_error)


plt.plot(range(1,len(rec_error_record)+1), rec_error_record, label = "rec error")
plt.plot(range(1,len(reg_error_record)+1), reg_error_record, label = "reg error")
plt.plot(range(1, len(total_error_record)+1), total_error_record, label = "total error")
plt.legend()
plt.xlabel("epochs")
```

```
plt.ylabel("error")
plt.show()
```

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/581b4a50-d078-4be4-ac1c-18db2a5e2e13/vae.ipynb