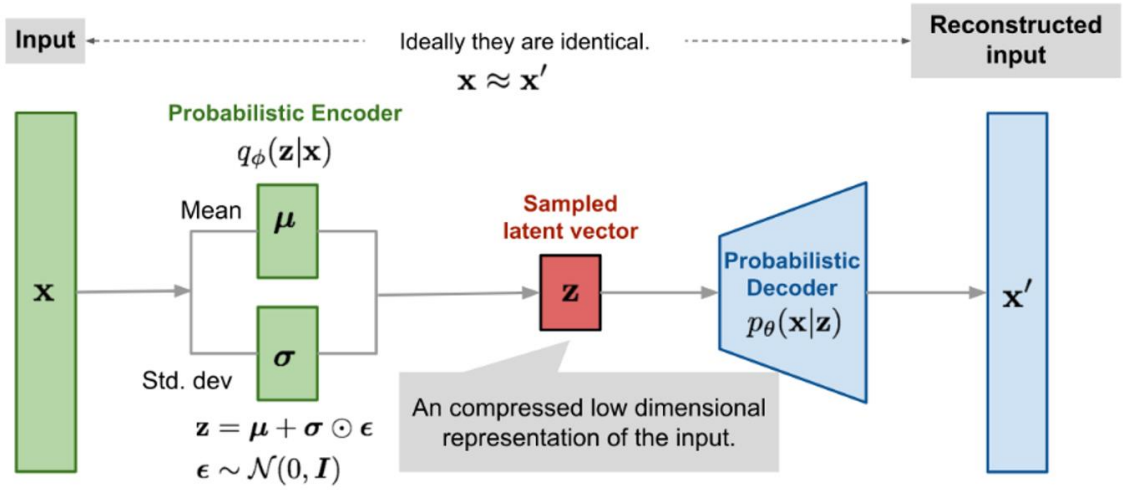
Last lecture:



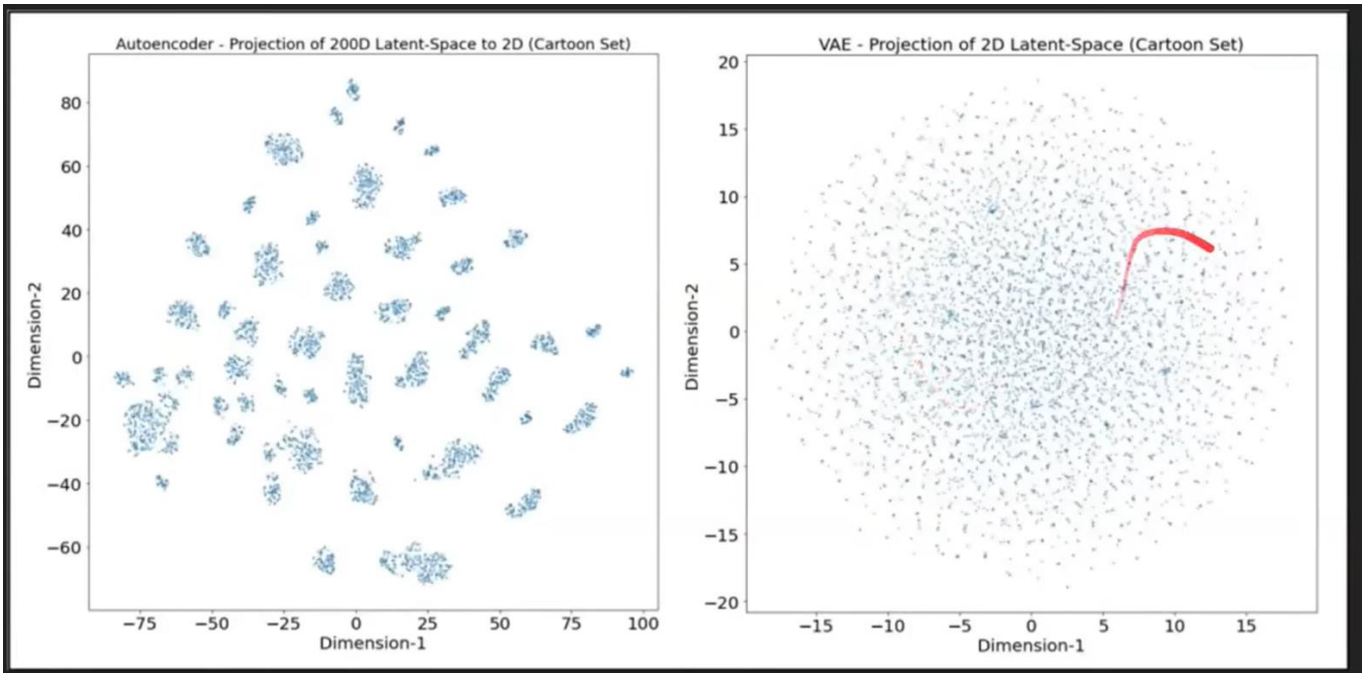
$$L_{\text{VAE}}(\theta, \phi) = -\mathbb{E}_{\mathbf{z} \sim q_\phi(\mathbf{z}|\mathbf{x})} \log p_\theta(\mathbf{x}|\mathbf{z}) + D_{\text{KL}}(q_\phi(\mathbf{z}|\mathbf{x}) \| p_\theta(\mathbf{z}))$$

Thus, the loss becomes:

$$\mathcal{L}_{\text{VAE}} = \frac{1}{2}\|x - \hat{x}\|^2 + \frac{1}{2}\sum_{i=1}^{k}\left(-\log \sigma_i^2 - 1 + \sigma_i^2 + \mu_i^2\right).$$

Interpretation

1. **First term:** Ensures $\hat{x}$ (reconstructed input) is close to x (original input).
2. **Second term:** Regularizes the encoder by forcing $q_\phi(z|x)$ to align with the prior $p(z)$.

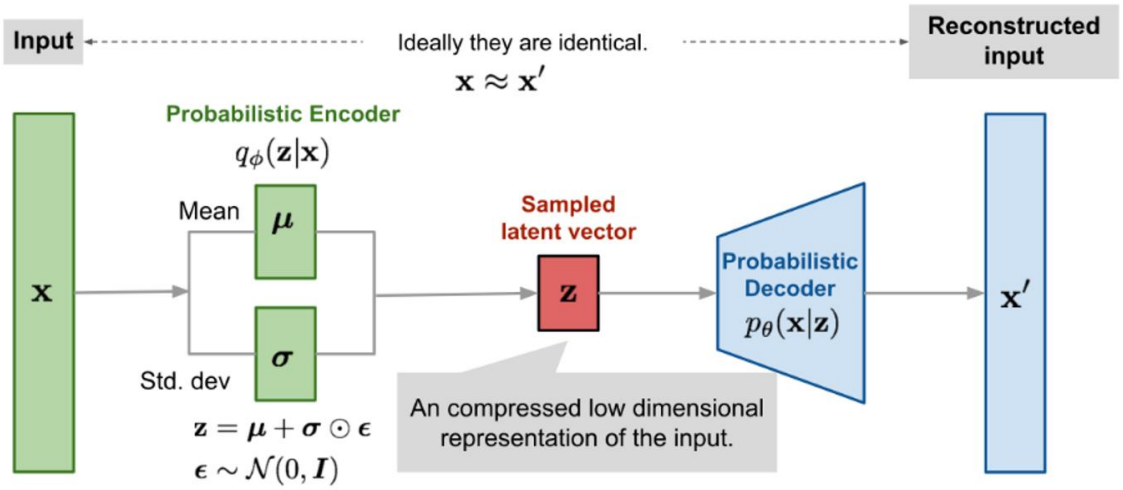

Autoencoder - Projection of 200D Latent-Space to 2D (Cartoon Set)
Dimension-2
Dimension-1
80
60
40
20
0
−20
−40
−60
−75
−50
−25
0
25
50
75
100
VAE - Projection of 2D Latent-Space (Cartoon Set)
Dimension-2
Dimension-1
20
15
10
5
0
−5
−10
−15
−20
−15
−10
−5
0
5
10
15

Today lecture:

- Reparameterization trick
- Implementation of VAE in pytorch for MNIST dataset.

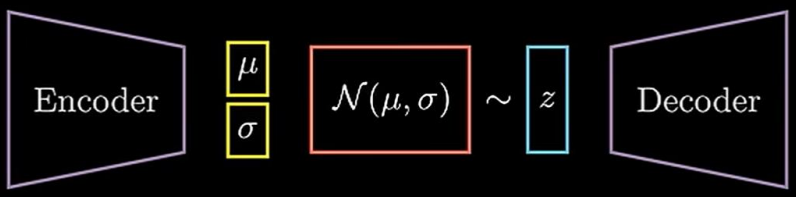
Reparameterization trick:

- How to train VAE

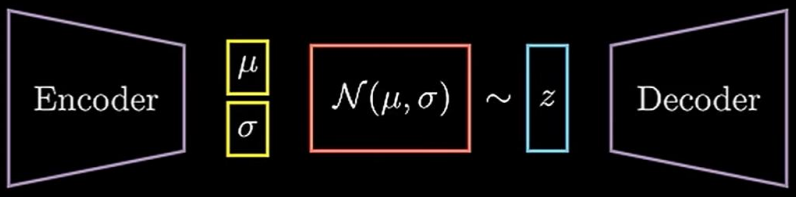


$$L_{\text{VAE}}(\theta, \phi) = -\mathbb{E}_{\mathbf{z} \sim q_\phi(\mathbf{z}|\mathbf{x})} \log p_\theta(\mathbf{x}|\mathbf{z}) + D_{\text{KL}}(q_\phi(\mathbf{z}|\mathbf{x}) \| p_\theta(\mathbf{z}))$$

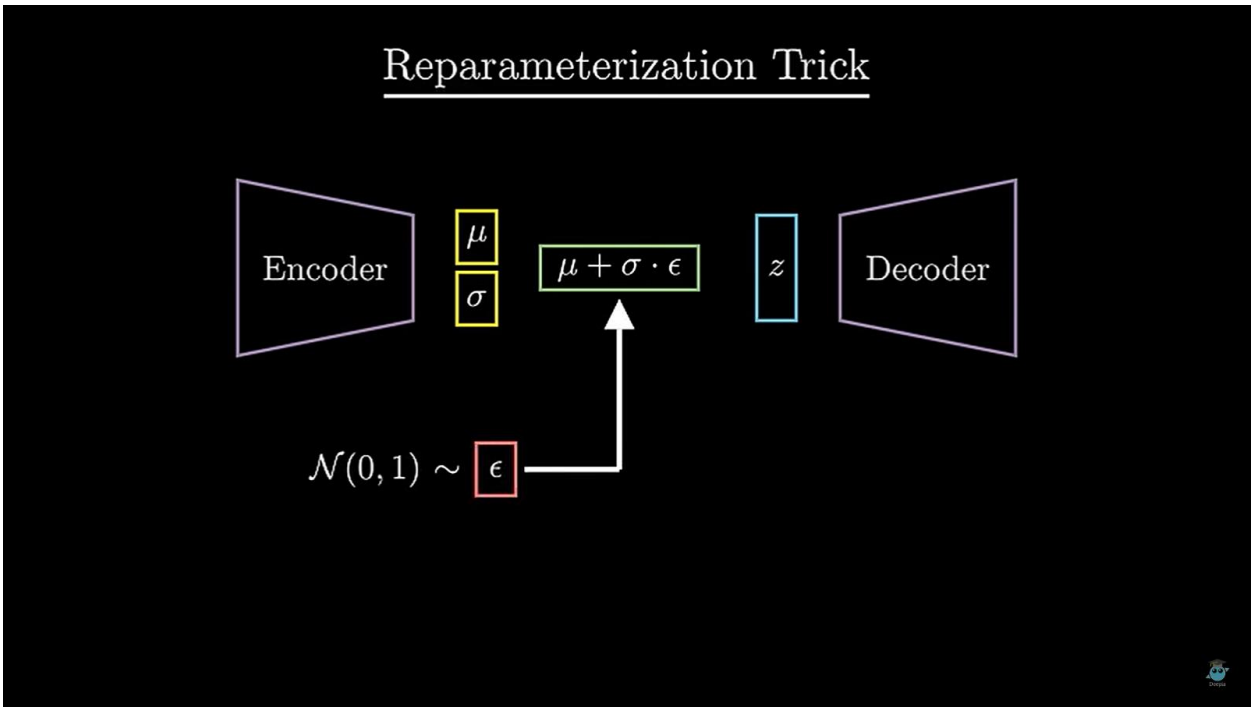
$$\mathcal{L}(x, x') = L_2(x, x') + D_{KL}(\mathcal{N}(\mu, \sigma) \mid \mathcal{N}(0, 1))$$



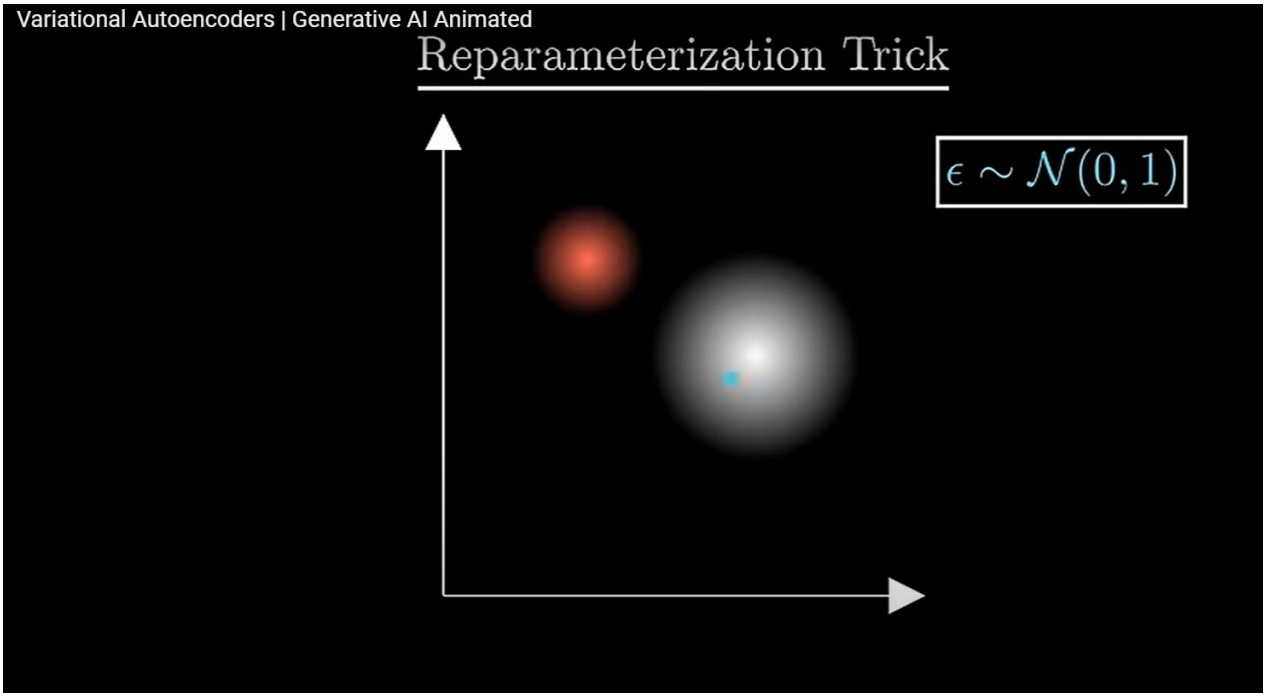
How to backpropagate through the sampling process?

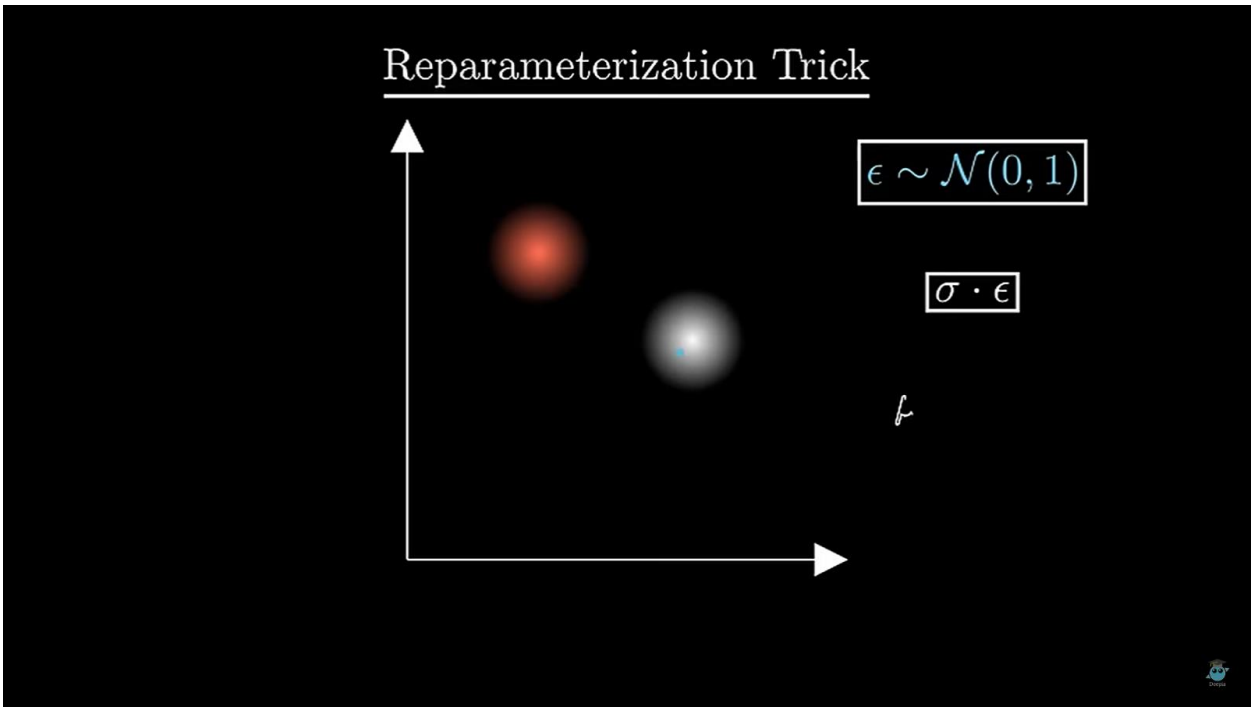


? ?

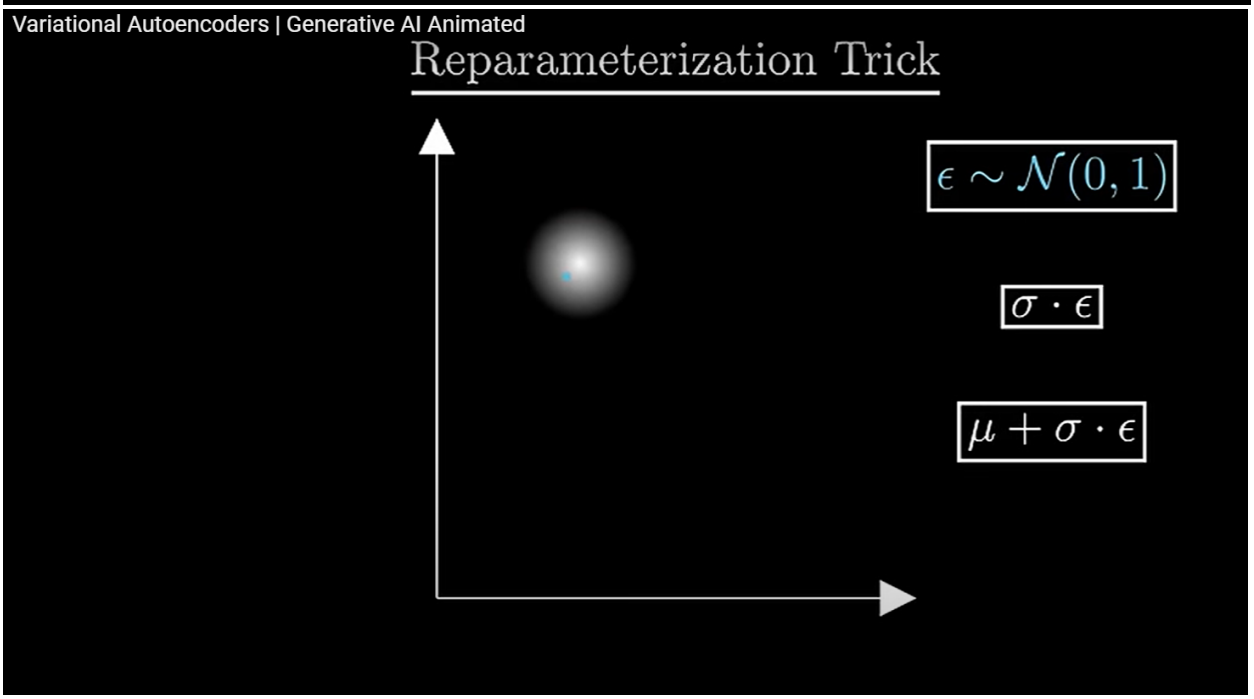


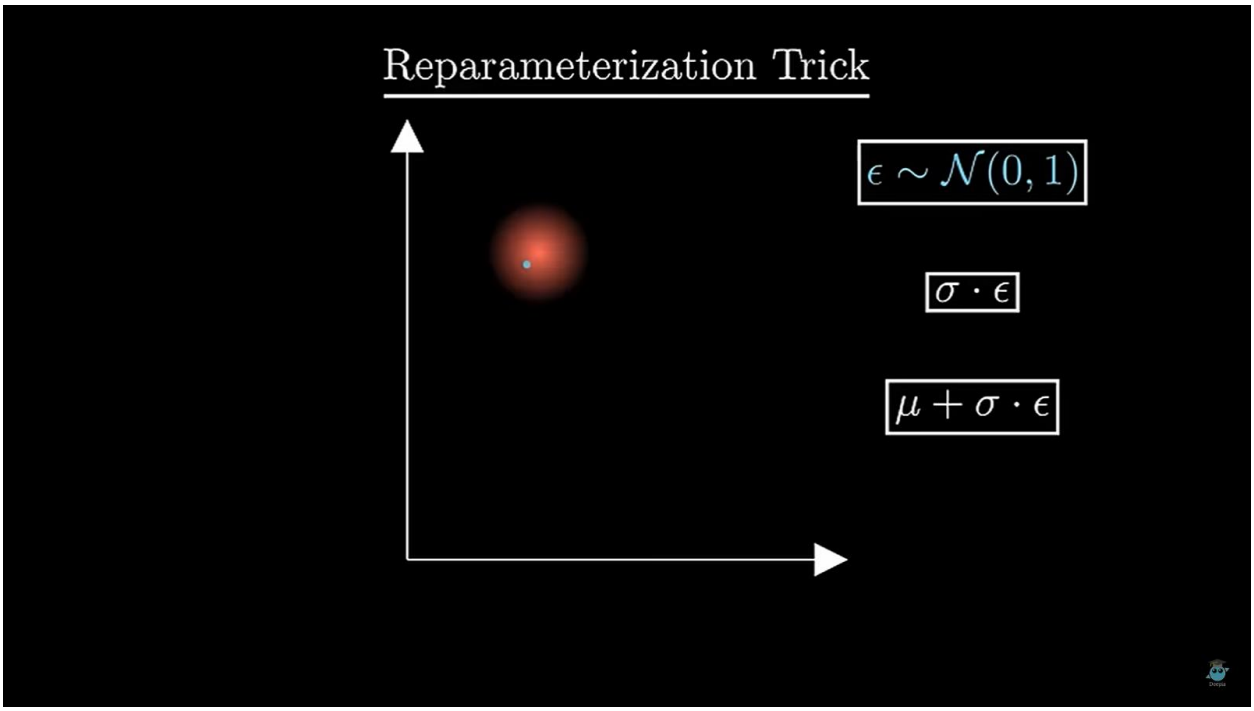
Variational Autoencoders | Generative AI Animated





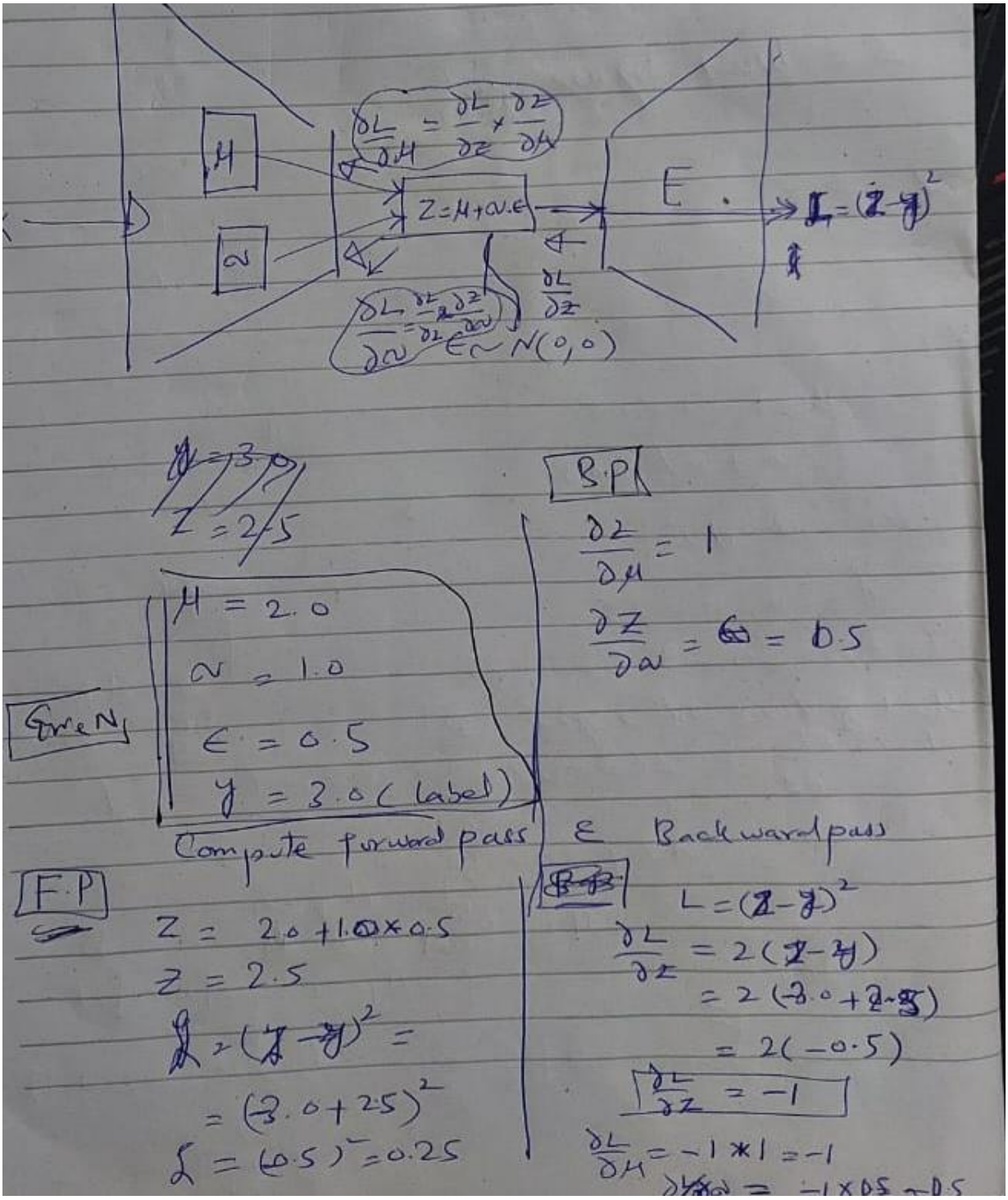
Variational Autoencoders | Generative AI Animated





It is just as we sampling from the posterior distribution ensures that the backpropagation process remains differentiable with respect to the mean and standard deviation. This enables gradients to propagate effectively throughout the entire network.

SUMMARY OF REPARAMETERIZATION TRICK:



The Reparameterization Trick

The reparameterization trick allows us to express the sampling process in a way that separates the deterministic parameters from the stochastic noise. Instead of sampling directly from $N(\mu, \sigma^2)$, we can rewrite the sampling operation as:

$$z = \mu + \sigma \cdot \epsilon$$

where $\epsilon \sim N(0, 1)$. Here, ϵ is sampled from a standard normal distribution, introducing randomness while keeping the operation deterministic with respect to μ and σ.

Toy Example

Step 1: Define Parameters

Let's assume we have:

- Mean: $\mu = 2.0$
- Standard deviation: $\sigma = 1.0$

Step 2: Sample from Standard Normal Distribution

We sample ϵ:

$$\epsilon = 0.5 \quad (\text{sampled from } N(0, 1))$$

Step 3: Compute Latent Variable Using Reparameterization

Now we compute z:

$$z = 2.0 + 1.0 \cdot 0.5 = 2.0 + 0.5 = 2.5$$

Step 4: Compute Loss Function

Suppose we have a loss function:

$$L(z) = (z - y)^2$$

where $y = 3.0$. For our computed value of $z = 2.5$:

$$L(z) = (2.5 - 3)^2 = (-0.5)^2 = 0.25$$

Step 5: Calculate Gradients

Now compute gradients with respect to the loss:

1. **Gradient of Loss with respect to z**:

   Using the chain rule:

   $$\frac{\partial L}{\partial z} = 2(z - y) = 2(2.5 - 3) = -1$$

2. **Chain Rule for Backpropagation**:
   - From our reparameterized equation:
   - For parameters μ and σ:
   - Since:
   -

   $$z = mu + sigma * epsilon$$

   We can compute:

   $$dL/dmu = dL/dz * dz/dmu$$

   $$dL/dsigma = dL/dz * dz/dsigma$$

Step 6: Computing Gradients with Respect to Parameters

Now we can compute how changes in mu and $sigma$ affect the loss:

- **Gradient with respect to mu:**

$$dz/dmu = 1$$

So,

$$dL/dmu = dL/dz * dz/dmu = -1 * 1 = -1$$

- **Gradient with respect to $sigma$:**

$$dz/dsigma = epsilon$$

So,

$$dL/dsigma = dL/dz * dz/dsigma = -1 * epsilon = -1 * 0.5 = -0.5$$

Reparameterization Trick

The expectation term in the loss function invokes generating samples from $\mathbf{z} \sim q_\phi(\mathbf{z}|\mathbf{x})$. Sampling is a stochastic process and therefore we cannot backpropagate the gradient. To make it trainable, the reparameterization trick is introduced: It is often possible to express the random variable $\mathbf{z}$ as a deterministic variable $\mathbf{z} = \mathcal{T}_\phi(\mathbf{x}, \boldsymbol{\epsilon})$, where $\boldsymbol{\epsilon}$ is an auxiliary independent random variable, and the transformation function $\mathcal{T}_\phi$ parameterized by ϕ converts $\boldsymbol{\epsilon}$ to $\mathbf{z}$.

For example, a common choice of the form of $q_\phi(\mathbf{z}|\mathbf{x})$ is a multivariate Gaussian with a diagonal covariance structure:

$$\mathbf{z} \sim q_\phi(\mathbf{z}|\mathbf{x}^{(i)}) = \mathcal{N}(\mathbf{z}; \boldsymbol{\mu}^{(i)}, \boldsymbol{\sigma}^{2(i)}\boldsymbol{I})$$
$$\mathbf{z} = \boldsymbol{\mu} + \boldsymbol{\sigma} \odot \boldsymbol{\epsilon}, \text{where } \boldsymbol{\epsilon} \sim \mathcal{N}(0, \boldsymbol{I}) \quad ; \text{Reparameterization trick.}$$

where $\odot$ refers to element-wise product.

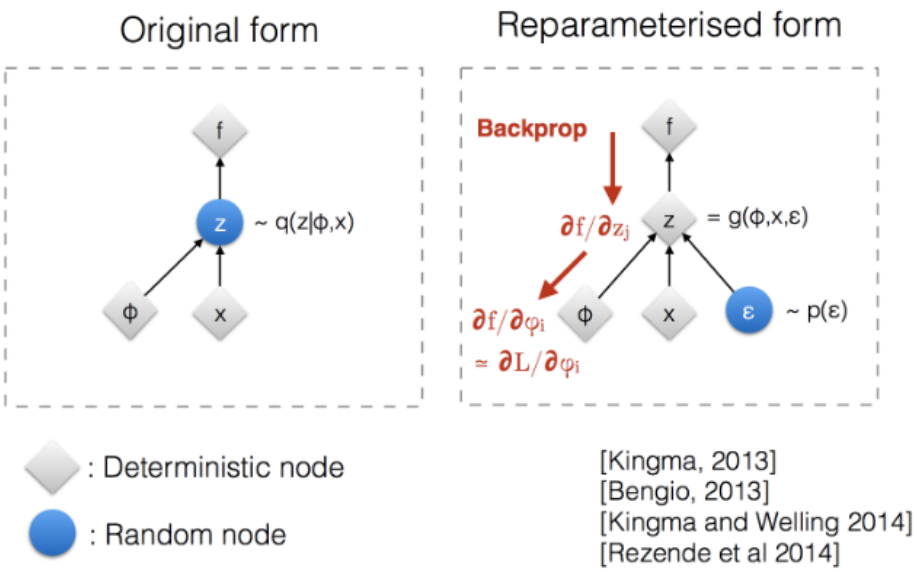
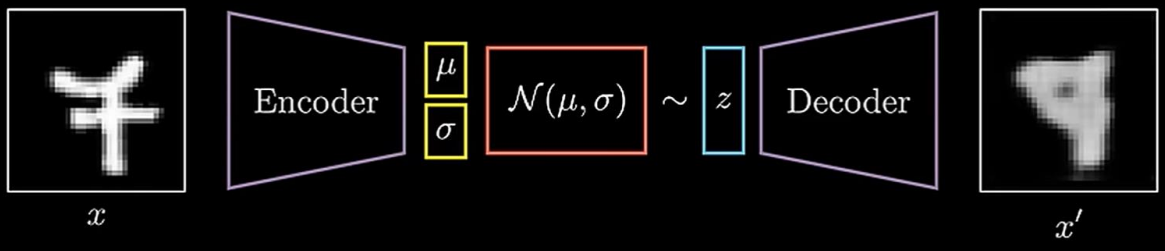


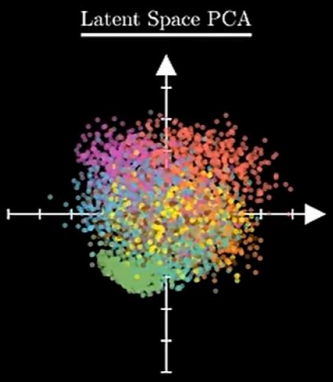
Fig. 8. Illustration of how the reparameterization trick makes the $\mathbf{z}$ sampling process trainable.(Image source: Slide 12 in Kingma's NIPS 2015 workshop talk)

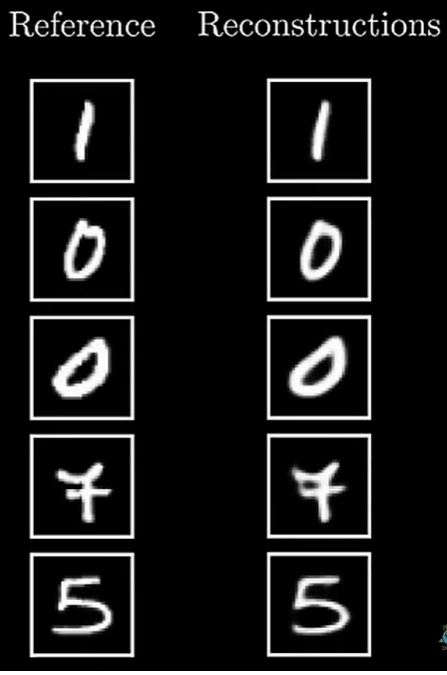
The reparameterization trick works for other types of distributions too, not only Gaussian. In the multivariate Gaussian case, we make the model trainable by learning the mean and variance of the distribution, μ and σ, explicitly using the reparameterization trick, while the stochasticity remains in the random variable ϵ∼N(0,I).

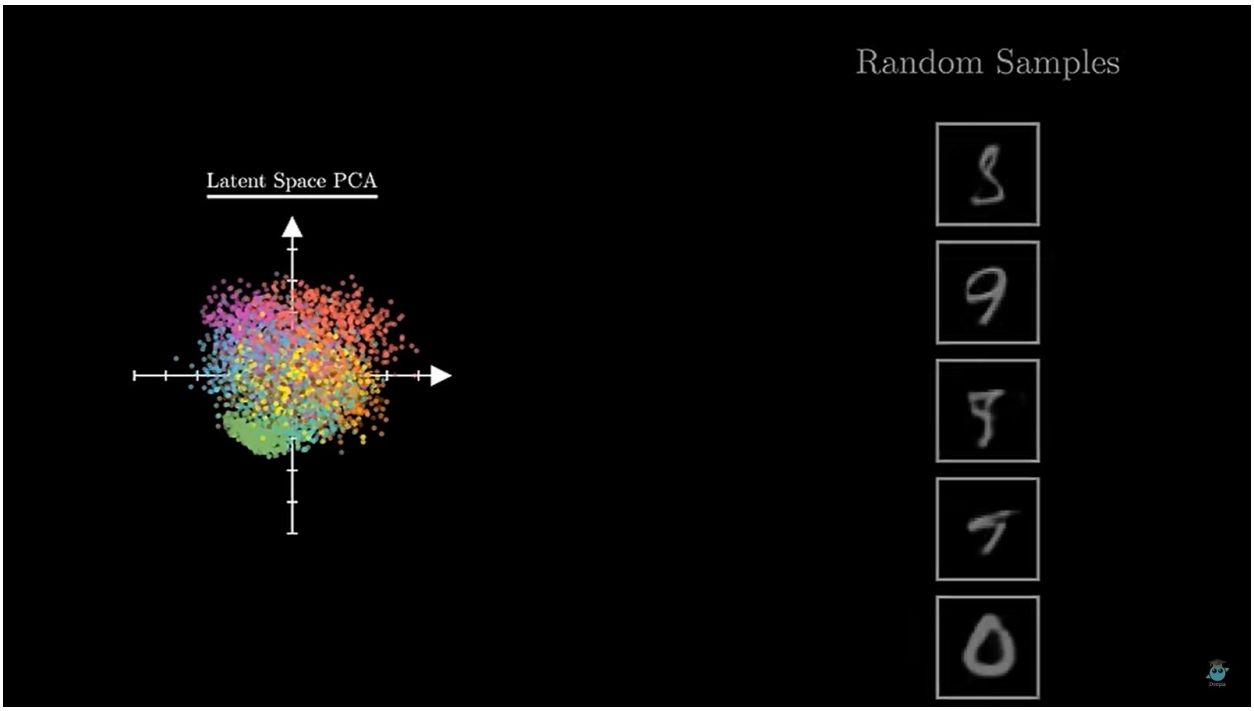


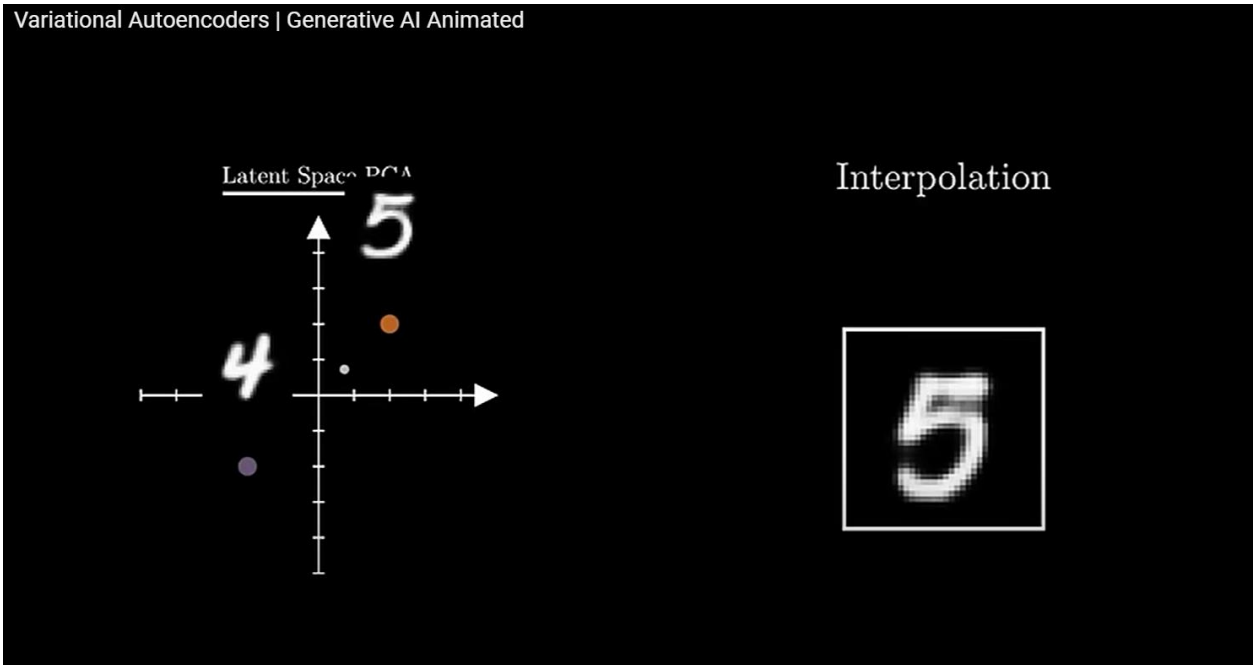
$$\mathcal{L} = \boxed{\mathcal{L}_{KL}(\mathcal{N}(\mu, \sigma) \mid \mathcal{N}(0, 1))} + \mathcal{L}_2(x, x')$$

$$\mathcal{L}_{KL} = -\frac{1}{2}(1 + \log(\sigma^2) - \mu^2 - \sigma^2)$$









Key Explanation:

1. **Latent Space Representation**:

- The graph on the left represents the latent space, which is typically a low-dimensional representation learned by the VAE.
- Each point in this latent space corresponds to a specific encoded feature of an input image. For instance:
  - The orange dot might represent an image of the digit "5."
  - The purple dot might represent an image of the digit "4."

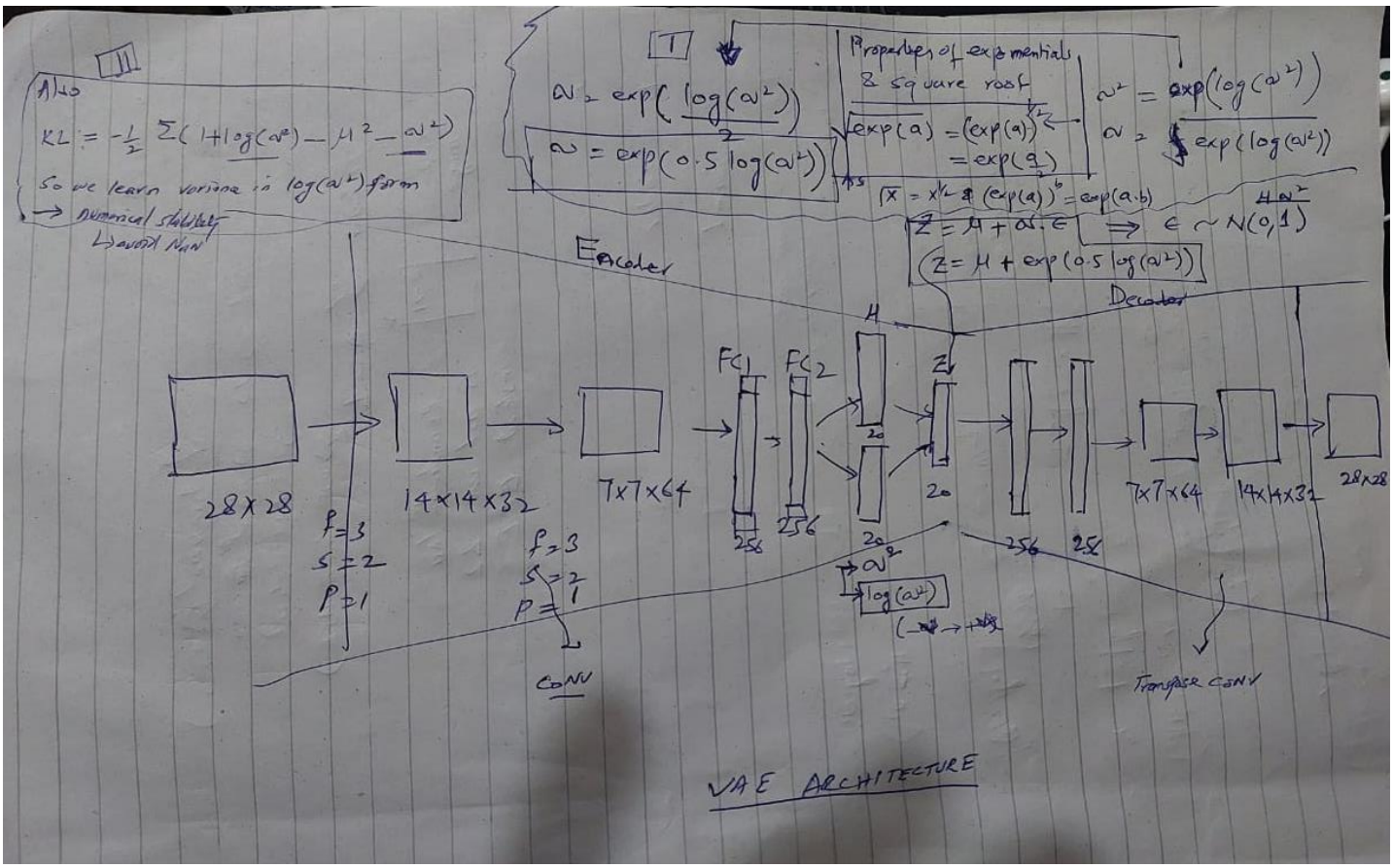
2. **Interpolation in Latent Space**:
   - Interpolation involves moving smoothly between two points in the latent space.
   - In this case, interpolation is shown for a point in the latent space that corresponds to the digit "5."
   - By decoding the intermediate points between two locations (e.g., moving from a latent representation near "4" to the one near "5"), the VAE generates intermediate images that transition smoothly from one digit to the other.
3. **Output Reconstruction (Right Panel)**:
   - As the interpolated points are decoded through the VAE's decoder network, the output visually transitions into recognizable features of the number "5."
   - The reconstructed images help showcase how the latent space is structured. Points that are close in the latent space correspond to inputs that have similar high-level features.

Importance in VAEs:

- **Smooth Representation**: The ability of VAEs to interpolate smoothly between data points reflects the continuity and meaningful organization of the learned latent space.
- **Applications**:
  - Interpolation demonstrates the VAE's ability to generalize and synthesize data (e.g., generating variations of an image).
  - It can be used for creating realistic variations of an image or blending different images in generative tasks.

In this example, the image shows how the latent space captures meaningful high-level features, allowing smooth transitions between different data representations

Implementation:



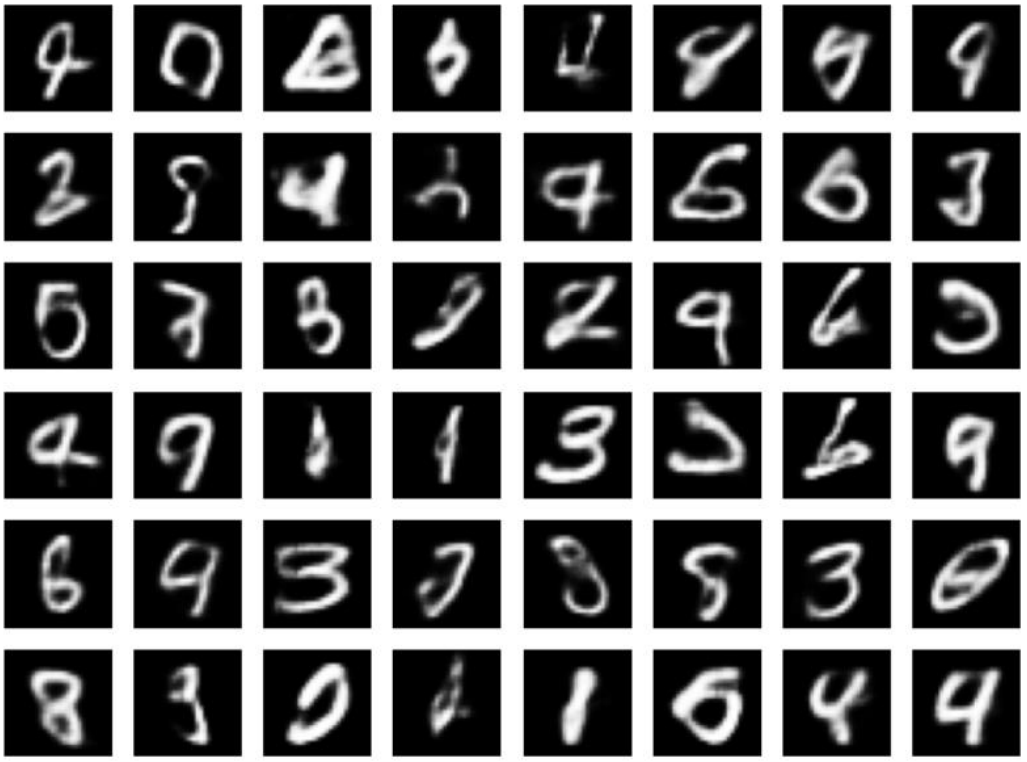

```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision.datasets as datasets
import torchvision.transforms as transforms
from torch.utils.data import DataLoader
from torch.autograd import Variable
import numpy as np
import matplotlib.pyplot as plt

# Define hyperparameters
batch_size = 128
learning_rate = 1e-3
num_epochs = 10
latent_dim = 20

# Load MNIST dataset
train_dataset = datasets.MNIST(root='./data', train=True, transform=transforms.ToTensor(),
download=True)
```

```
test_dataset = datasets.MNIST(root='./data', train=False, transform=transforms.ToTensor())

#note : transforms.Tensor ()convert the data from 0 to 1 range
#       becasue of that we use the cross entropy loss

train_loader = DataLoader(dataset=train_dataset, batch_size=batch_size, shuffle=True)
test_loader = DataLoader(dataset=test_dataset, batch_size=batch_size, shuffle=False)

# Define the encoder network
class Encoder(nn.Module):
    def __init__(self):
        super(Encoder, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3, stride=2, padding=1) # ouput size 14x14x
32
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, stride=2, padding=1) # output size
7x7x64

        self.fc1 = nn.Linear(7  * 7 * 64, 256) #
        self.fc2_mean = nn.Linear(256, latent_dim)
        self.fc2_logvar = nn.Linear(256, latent_dim)

    def forward(self, x):
        x = torch.relu(self.conv1(x))
        x = torch.relu(self.conv2(x))
        x = x.view(x.size(0), -1)  # view the tensor as single dimensional like flatten it
        x = torch.relu(self.fc1(x))
        z_mean = self.fc2_mean(x)
        z_logvar = self.fc2_logvar(x)
        return z_mean, z_logvar

# Define the decoder network
class Decoder(nn.Module):
    def __init__(self):
        super(Decoder, self).__init__()
        self.fc1 = nn.Linear(latent_dim, 256)
        self.fc2 = nn.Linear(256, 7 * 7 * 64)
        self.conv1 = nn.ConvTranspose2d(64, 32, kernel_size=3, stride=2, padding=1,
output_padding=1)
        self.conv2 = nn.ConvTranspose2d(32, 1, kernel_size=3, stride=2, padding=1,
output_padding=1)

    def forward(self, z):
        x = torch.relu(self.fc1(z))
        x = torch.relu(self.fc2(x))
        x = x.view(x.size(0), 64, 7, 7) # view the tensor as a single dimensional vector
        x = torch.relu(self.conv1(x))
        x = torch.sigmoid(self.conv2(x)) # sigmoid output 786 value of range [0 to 1] for
reconstruction image
        return x

# Define the VAE model
class VAE(nn.Module):
    def __init__(self):
        super(VAE, self).__init__()
        self.encoder = Encoder()
        self.decoder = Decoder()
```

```
    def reparameterize(self, mu, logvar):
        std = torch.exp(0.5 * logvar) #=> logvar = log(std^2)=> output of variance vector
                                      #=> std = exp(0.5 * logvar) => inorder to make it
positve and negative values for a stable training.
        eps = torch.randn_like(std) # Returns a tensor with the same size as input that is
filled with random numbers from a normal distribution with mean 0 and variance 1
        return mu + eps * std

    def forward(self, x):
        z_mean, z_logvar = self.encoder(x)
        z = self.reparameterize(z_mean, z_logvar)
        x_recon = self.decoder(z)
        return x_recon, z_mean, z_logvar

# Instantiate the VAE model and define the loss function and optimizer
model = VAE()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)

#

# Define the loss function
def loss_fn(x_recon, x, z_mean, z_logvar):
    recon_loss = nn.functional.binary_cross_entropy(x_recon, x, reduction='sum') # reduction =
sum means output will be summed
    kl_div = -0.5 * torch.sum(1 + z_logvar - z_mean.pow(2) - z_logvar.exp())
    return recon_loss + kl_div

# Train the model
train_loss = []
for epoch in range(num_epochs):
    epoch_loss = 0.0
    for i, (images, _) in enumerate(train_loader):  # as we dont use label so put _ over there
        # Forward pass
        x = Variable(images)  # The code you are seeing might be written for an older version
of PyTorch (pre-0.4).
                              # Autograd automatically supports
                              #Tensors with requires_grad set to True.
                              #The original purpose of Variables was to be able to use
automatic differentiation (Source):

        x_recon, z_mean, z_logvar = model(x)
        # Compute loss
        loss = loss_fn(x_recon, x, z_mean, z_logvar)
        epoch_loss += loss.item()
        # Backward pass and optimize
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
    epoch_loss /= len(train_loader.dataset)
    train_loss.append(epoch_loss)
    print('Epoch [{}/{}], Loss: {:.4f}'.format(epoch+1, num_epochs, epoch_loss))

# Test the model
model.eval()  #Layers like Dropout and BatchNorm change their behavior.
#Dropout: During training, dropout randomly zeros out some activations to prevent overfitting.
In evaluation mode, dropout is disabled (i.e., it doesn't drop activations).
```

```
#BatchNorm: During training, BatchNorm calculates running statistics (mean and variance) based
on the current batch. In evaluation mode, it uses the running (global) statistics instead of
the current batch statistics to ensure consistency during inference.
test_loss = 0.0
with torch.no_grad():
    for images, _ in test_loader:
        x = Variable(images)
        x_recon, z_mean, z_logvar = model(x)
        loss = loss_fn(x_recon, x, z_mean, z_logvar)
        test_loss += loss.item()
test_loss /= len(test_loader.dataset)
print('Test Loss: {:.4f}'.format(test_loss))

# Generate new images
sample = Variable(torch.randn(64, latent_dim))
sample = model.decoder(sample)
sample = sample.view(64, 1, 28, 28).data
plt.figure(figsize=(8, 8))
for i in range(64):
    plt.subplot(8, 8, i+1)
    plt.imshow(sample[i][0], cmap='gray')
    plt.axis('off')
plt.show()
```

Results:

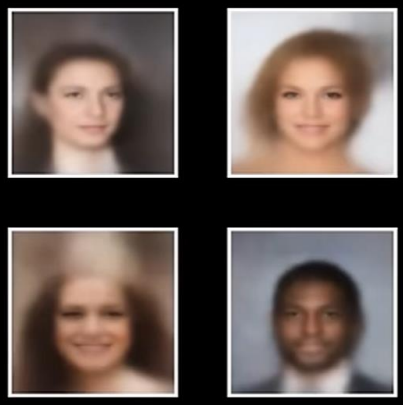


VAEs produce blurry images

CelebA dataset

CelebA dataset

Reconstructions



VAE Samples

VAE Samples

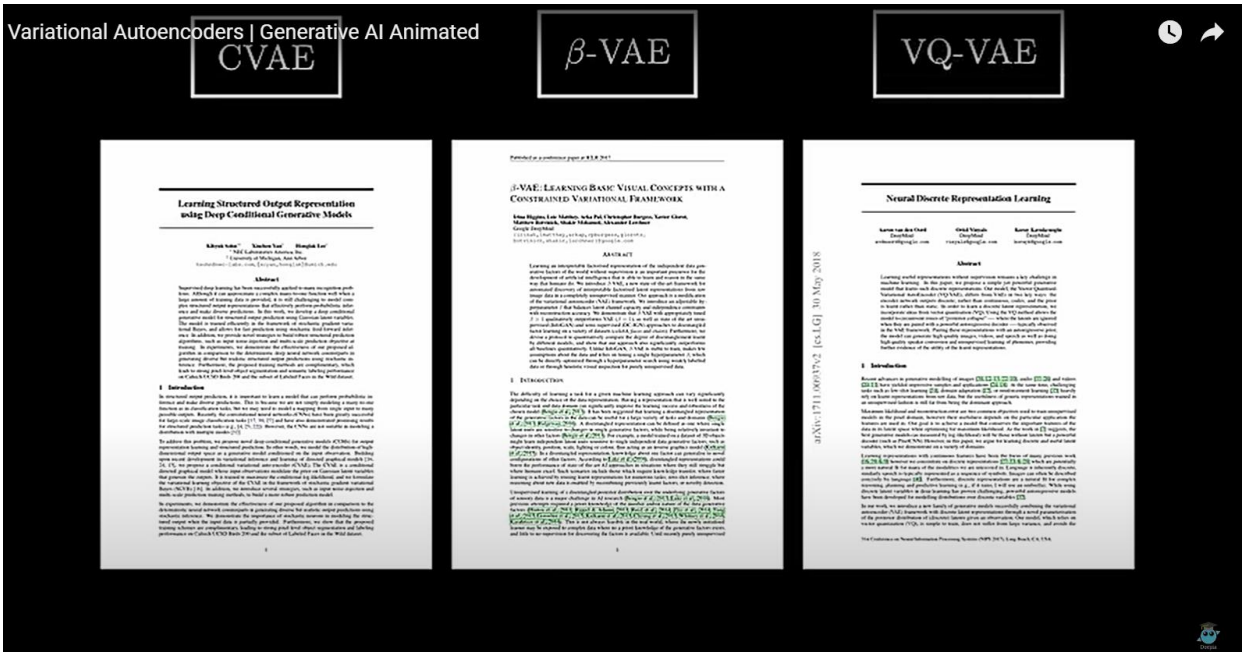
GAN Samples

VAE Samples

Diffusion samples

GAN Samples

How do we even choose which image to generate?



Assignment 4:

Perform the following tasks for VAE:

1. Train the model for 100 epochs.
2. Replace the Binary Cross-Entropy (BCE) loss with Mean Squared Error (MSE) loss and evaluate the results.
3. Experiment by training the model with BCE, MSE, and then a combination of both loss functions, and compare the outcomes.
4. Use the CIFAR-10 dataset for training.