Variational AutoEncoder

VAE

Autoencoder vs VAE

Autoencoders primarily aim to reduce the dimensionality of data and extract significant features by compressing the input data into a lower-dimensional latent space representation and then reconstructing it back to a form similar to the original data. The focus is not on the exact probability distribution of the generated data.

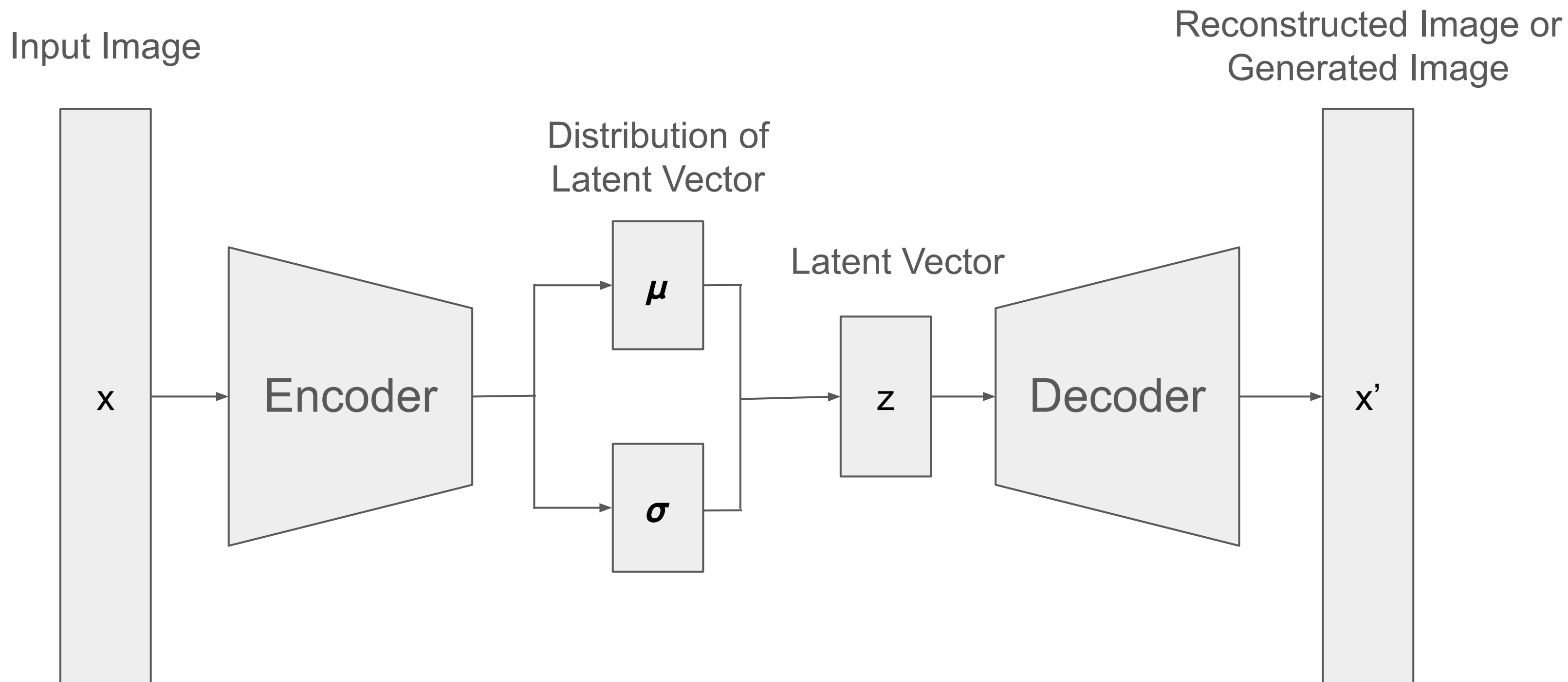
Autoencoder vs VAE

Variational Autoencoders (VAEs) extend autoencoders by learning a probabilistic representation of the input data.

VAEs model each point in the latent space with a probability distribution, typically Gaussian, learning the mean and variance for each latent variable z.

This allows for the generation of new data points by randomly sampling from these distributions.

VAE

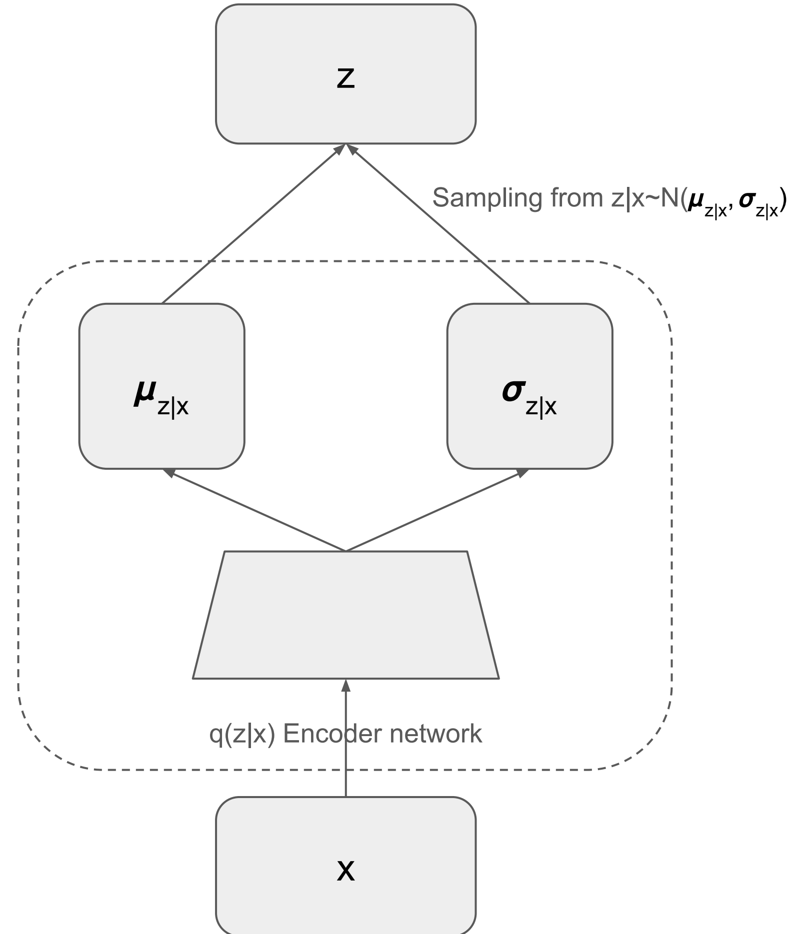


Encoder Network

q(z|x) plays a crucial role and is also referred to as the encoder.

This network takes input data x and outputs statistical parameters (**mean** and **variance**) associated with the latent vector z, which represents the position of the input data in the latent space.

This process can be understood as mapping the input data to a specific distribution in the latent space, from which the latent vector z is obtained through **sampling**.



Loss function for VAE

The loss function of a Variational Autoencoder (VAE) is composed of two main parts: Reconstruction Loss and KL Divergence.

Reconstruction Loss: This loss measures the difference between the original data and the data reconstructed through the VAE. **The goal is to make the reconstructed data as similar as possible to the original data.** Reconstruction loss is typically calculated using loss functions like Mean Squared Error (MSE) or Cross-Entropy.

KL Divergence: This term measures the difference between the distribution of the latent variables generated by the encoder and a predefined target distribution (usually a standard normal distribution). KL Divergence ensures that the model does more than just accurately reconstruct data; **it organizes the latent space in a way that facilitates the generation of new data.** In other words, it acts as a regularizer for the latent space, ensuring that similar data points are located close to each other in the latent space.

Reconstruction Loss

This component of the VAE loss function focuses on the accuracy of the output compared to the input. It essentially measures **how well the decoder is able to reconstruct** the input data after it has been encoded into the latent space and then decoded back into the original data space. Common choices for the reconstruction loss include:

Mean Squared Error (MSE): Used especially when the input data are continuous. It measures the average of the squares of the errors—that is, the average squared difference between the estimated values (reconstructed data) and the actual value (original data).

Binary Cross-Entropy: Often used when dealing with binary or categorical data, such as pixels in a black-and-white image. It measures the dissimilarity between two probability distributions, typically the predicted probabilities and the actual labels.

In our implementation

```
reproduction_loss = nn.functional.binary_cross_entropy(x_hat, x, reduction='sum')

# x: the original input data to the VAE.

# x_hat: the reconstructed data output by the VAE's decoder.
```

KL Divergence

Kullback-Leibler divergence is a measure of how one probability distribution diverges from a second, expected probability distribution.

In VAEs, KL divergence is used to measure how much the learned latent variable distribution $q(z|x)$ (approximated by the encoder) diverges from the prior distribution $p(z)$, which is often assumed to be the standard normal distribution $N(0,1)$.

KL Divergence

$q(z|x)$ is assumed to be a normal distribution with mean μ and variance σ^2 , and p(z) is the standard normal distribution with mean 0 and variance 1.

The KL divergence measures the information loss when using $q(z|x)$ to approximate p(z). Mathematically, it is calculated as:

$$D_{KL}(q(z|x)\|p(z)) = -\int q(z|x)\log\frac{p(z)}{q(z|x)}dz$$

Solving this gives:

$$D_{KL}(q(z|x)\|p(z)) = \frac{1}{2}\left(\sigma^2 + \mu^2 - 1 - \log\sigma^2\right)$$

Reference: https://statproofbook.github.io/P/norm-kl.html

Implementation

Encoder

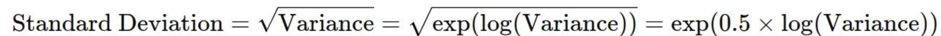
```
class Encoder(nn.Module):
    def __init__(self, input_dim, hidden_dim, latent_dim):
        super(Encoder, self).__init__()
        self.input1 = nn.Linear(input_dim, hidden_dim)
        self.input2 = nn.Linear(hidden_dim, hidden_dim)
        self.mean = nn.Linear(hidden_dim, latent_dim)
        self.var = nn.Linear (hidden_dim, latent_dim)
        self.LeakyReLU = nn.LeakyReLU(0.2)
        self.training = True
    def forward(self, x):
        h_ = self.LeakyReLU(self.input1(x))
        h_ = self.LeakyReLU(self.input2(h_))
        mean = self.mean(h_)
        log_var = self.var(h_)
        return mean, log_var
```

Decoder

```
class Decoder(nn.Module):
    def __init__(self, latent_dim, hidden_dim, output_dim):
        super(Decoder, self).__init__()
        self.hidden1 = nn.Linear(latent_dim, hidden_dim)
        self.hidden2 = nn.Linear(hidden_dim, hidden_dim)
        self.output = nn.Linear(hidden_dim, output_dim)
        self.LeakyReLU = nn.LeakyReLU(0.2)
    def forward(self, x):
        h = self.LeakyReLU(self.hidden1(x))
        h = self.LeakyReLU(self.hidden2(h))
        x_hat = torch.sigmoid(self.output(h))
        return x_hat
```

Encoder + Decoder

```
class Model(nn.Module):
    def __init__(self, Encoder, Decoder):
        super(Model, self).__init__()
        self.Encoder = Encoder
        self.Decoder = Decoder
    def reparameterization(self, mean, var):
        epsilon = torch.randn_like(var).to(device) # used var's shape
        z = mean + var*epsilon # added noise in sampling
        return z
    def forward(self, x):
        mean, log_var = self.Encoder(x)
        z = self.reparameterization(mean, torch.exp(0.5 * log_var))
        x_hat = self.Decoder(z)
        return x_hat, mean, log_var
```



Parameters and Model

```
x_dim  = 784
hidden_dim = 400
latent_dim = 200
epochs = 30
batch_size = 100
encoder = Encoder(input_dim=x_dim, hidden_dim=hidden_dim, latent_dim=latent_dim)
decoder = Decoder(latent_dim=latent_dim, hidden_dim = hidden_dim, output_dim = x_dim)
model = Model(Encoder=encoder, Decoder=decoder).to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
```

Loss function

```
def loss_function(x, x_hat, mean, log_var):
    reproduction_loss = nn.functional.binary_cross_entropy(x_hat, x, reduction='sum')
    KLD = - 0.5 * torch.sum(1+ log_var - mean.pow(2) - log_var.exp())
    return reproduction_loss, KLD
```

Loss function

```
def loss_function(x, x_hat, mean, log_var):
    reproduction_loss = nn.functional.binary_cross_entropy(x_hat, x, reduction='sum')
    KLD = - 0.5 * torch.sum(1+ log_var - mean.pow(2) - log_var.exp())
    return reproduction_loss, KLD
```

`log_var` represents the logarithm of the variance, $\log\sigma^2$. Thus, the formula can be rewritten as:

$$\sigma^2 = \exp(\log\sigma^2) = \exp(\text{log_var})$$

$$\log\sigma^2 = \text{log_var}$$

$$D_{KL}(q(z|x)\|p(z)) = \frac{1}{2}\left(\sigma^2 + \mu^2 - 1 - \log\sigma^2\right)$$

$$D_{KL} = \frac{1}{2}\sum\left(\exp(\text{log_var}) + \mu^2 - 1 - \text{log_var}\right)$$

```
KLD = - 0.5 * torch.sum(1+ log_var - mean.pow(2) - log_var.exp())
```

train()

```
model.train()
def train(epoch, model, train_loader, optimizer):
    for batch_idx, (x, _) in enumerate(train_loader):
        x = x.view(batch_size, x_dim)
        x = x.to(device)
        optimizer.zero_grad()
        x_hat, mean, log_var = model(x)
        BCE, KLD = loss_function(x, x_hat, mean, log_var)
        loss = BCE + KLD
        loss.backward()
        optimizer.step()
```