Variational Autoencoder (VAE)

- A **Variational Autoencoder (VAE)** is a type of **generative model** that learns how to encode data into a compressed representation and then decode it back to the original form.
- Unlike traditional autoencoders, VAEs focus on probabilistic modeling, learning how to generate new data samples similar to the training data by sampling from a distribution (usually Gaussian).

Variational Autoencoder (VAE)

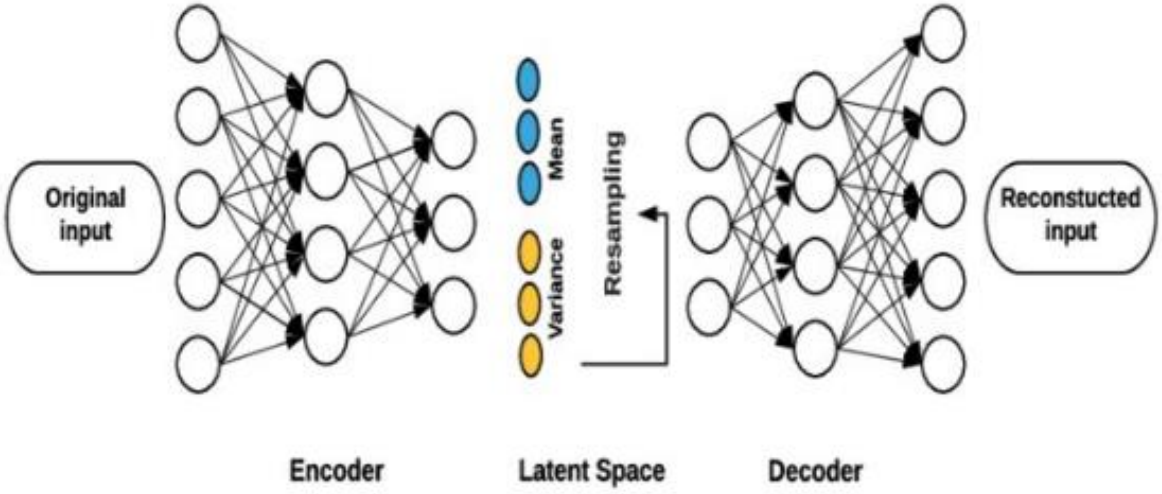
- **Why Learn VAEs?**

  1. Generative Model: VAEs can generate new samples similar to the training data (e.g., new images of handwritten digits).

  2. Dimensionality Reduction: VAEs can reduce the dimensionality of complex data while preserving important information.

  3. Smooth Latent Space: The latent space in VAEs is continuous and structured, allowing interpolation between data points.

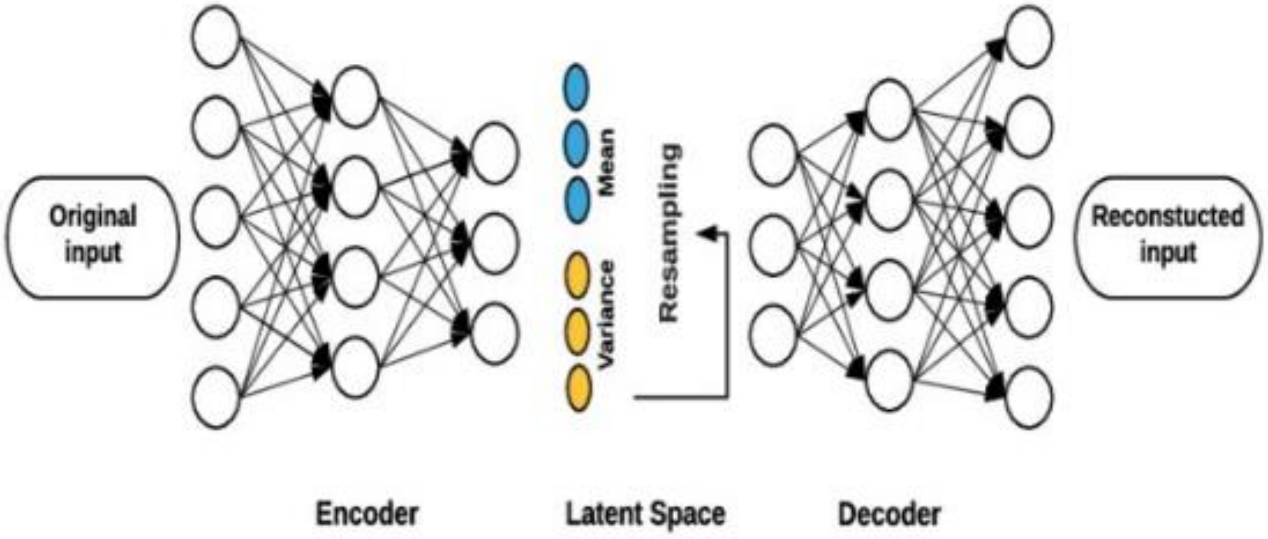
Components of a VAE



1. Encoder

- The encoder is a neural network that maps the input data x to a latent space (a lower-dimensional representation).
- Instead of encoding to a single point, VAEs encode the input into a **distribution** (mean μ and variance σ^2) in the latent space. This distribution is typically assumed to be Gaussian.
- The encoder outputs two vectors:
  - **Mean vector** $\mu(x)$
  - **Variance vector** $\sigma^2(x)$
- Mathematically:
- $z \sim N(\mu(x), \sigma^2(x))$ where z is a latent variable sampled from a Gaussian distribution.

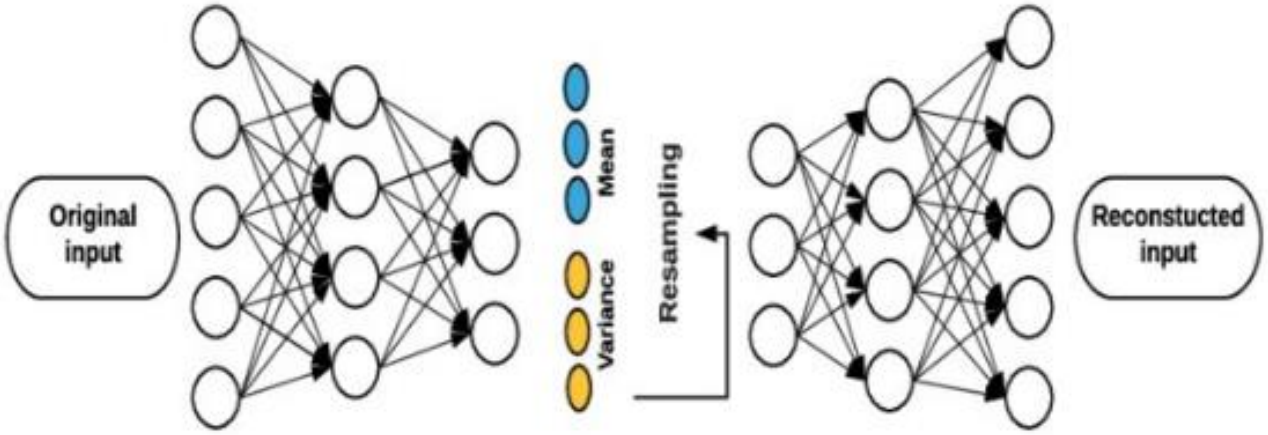
Components of a VAE



2. Latent Space

- The latent space is a lower-dimensional space where the encoded data resides. VAEs are unique in that they model this latent space as a **probability distribution**.
- Instead of encoding data as a point, VAE encodes data as a **distribution**, usually a Gaussian (normal) distribution.
- This allows the VAE to generate new data by sampling from this latent distribution.

Components of a VAE



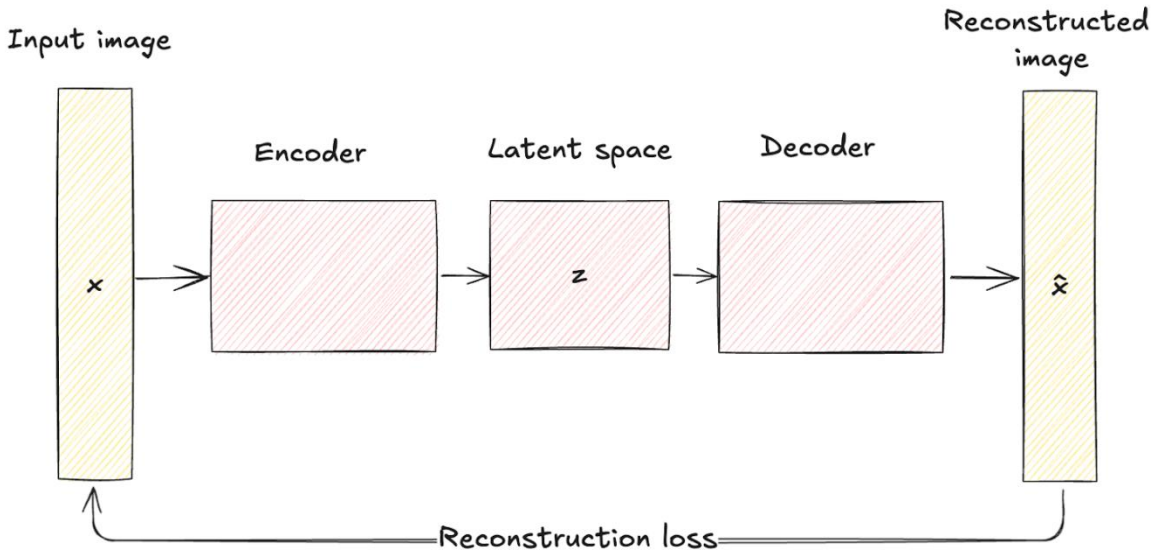
3. **Decoder**

- The decoder is another neural network that reconstructs the input data x from the latent variable z. It tries to generate the original data by "decoding" the sampled latent vector.
- The decoder outputs reconstructed data $\hat{x}$ from the sampled latent variable z.

Variational Autoencoder vs Traditional Autoencoder

Architecture comparison

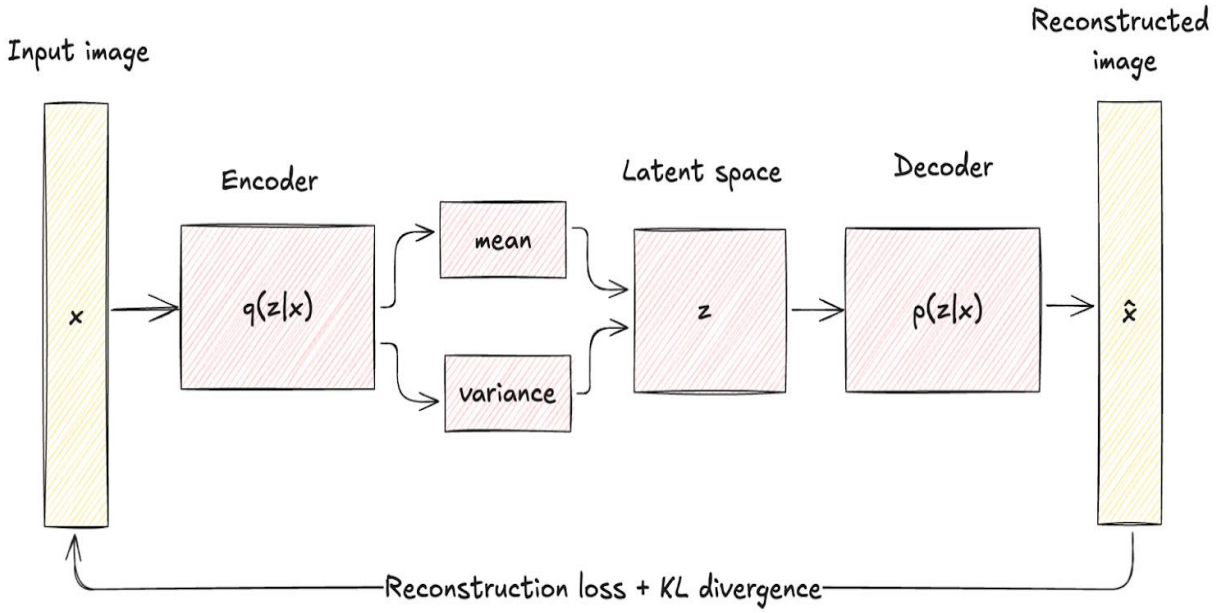
- As seen before, traditional autoencoders consist of an encoder network that maps the input data *x* to a fixed, lower-dimensional latent space representation *z*.
- This process is deterministic, meaning each input is encoded into a specific point in the latent space.
- The decoder network then reconstructs the original data from this fixed latent representation, aiming to minimize the difference between the input and its reconstruction.
- Traditional autoencoders' latent space is a compressed representation of the input data without any probabilistic modeling, which limits their ability to generate new, diverse data since they lack a mechanism to handle uncertainty.



Variational Autoencoder vs Traditional Autoencoder

- VAEs introduce a probabilistic element into the encoding process. Namely, the encoder in a VAE maps the input data to a probability distribution over the latent variables, typically modeled as a Gaussian distribution with mean μ and variance σ^2.
- This approach encodes each input into a distribution rather than a single point, adding a layer of variability and uncertainty.
- Architectural differences are visually represented by the deterministic mapping in traditional autoencoders versus the probabilistic encoding and sampling in VAEs.

Variational Autoencoder vs Traditional Autoencoder



- However, the goal of a variational autoencoder is not to reconstruct the original input; it's to generate *new* samples that *resemble* the original input. For that reason, an additional optimization term is needed, which is Kullback-Leibler divergence.

Variational Autoencoder

- Here is how the process flow looks:
- The input data x is fed into the encoder, which outputs the parameters of the latent space distribution $q(z|x)$ (mean μ and variance σ^2).
- Latent variables z are sampled from the distribution $q(z|x)$ using techniques like the reparameterization trick.
- The sampled z is passed through the decoder to produce the reconstructed data $\hat{x}$, which should be similar to the original input x.

Variational Autoencoder

- To generate the new samples from the latent space, we need to regularize the latent space, ensuring that it conforms to a specified distribution.
- The process involves a delicate balance between two essential components: the reconstruction loss and the regularization term, often represented by the Kullback-Leibler divergence.
- The reconstruction loss compels the model to accurately reconstruct the input, while the regularization term encourages the latent space to follow a chosen distribution.
- This helps prevent overfitting and makes the generalized model.

VAE Loss Function

- The total loss function combines the **reconstruction loss** and **KL divergence** (regularization) term.
- The objective is to minimize the total loss (i.e., reconstruction loss plus regularization loss):

$$L = L_{reconstruction} + L_{KL}$$

Where, $L_{reconstruction}$ ensures accurate reconstruction of the data.

- L_{KL} ensures that the learned latent representation is close to a standard Gaussian distribution.

Steps to Compute the Loss

1. **Encode** the input data to obtain the mean μ and variance σ^2 using the encoder network.
2. **Sample** from the latent distribution using the reparameterization trick:

$$z = \mu + \sigma \cdot \epsilon, \quad \epsilon \sim \mathcal{N}(0, I)$$

3. **Decode** z to reconstruct the input data $\hat{x}$ using the decoder network.
4. Compute the **reconstruction loss** based on the actual input and the reconstructed output.
5. Compute the **KL divergence** between the learned latent distribution and the prior distribution.
6. Combine both losses to obtain the final loss value.

VAE Loss Function

1. Reconstruction Loss

• The reconstruction loss measures how well the VAE can reconstruct the input data from the latent representation. This is typically computed using:

- **Binary Cross-Entropy (BCE)** if the data is binary:

$$\text{BCE} = -\frac{1}{N}\sum_{i=1}^{N}\left[x_i \log(\hat{x}_i) + (1 - x_i)\log(1 - \hat{x}_i)\right]$$

- **Mean Squared Error (MSE)** if the data is continuous:

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(x_i - \hat{x}_i)^2$$

where N is the number of data points, x_i is the original input, and $\hat{x}_i$ is the reconstructed output.

VAE Loss Function

2. Kullback-Leibler Divergence

- The KL divergence measures how much the learned latent distribution diverges from the prior distribution (usually a standard normal distribution). It can be computed as:

$$\mathrm{KL}(q(z|x)||p(z)) = -\frac{1}{2}\sum_{j=1}^{d}\left(1 + \log(\sigma_j^2) - \mu_j^2 - \sigma_j^2\right)$$

where:

- d is the dimensionality of the latent space,
- μ_j and σ_j^2 are the mean and variance of the learned latent distribution $q(z|x)$.

VAE Loss Function

- The Kullback-Leibler (KL) divergence in a VAE loss function quantifies the difference between the approximate posterior distribution $q(z|x)$ (the distribution that our encoder outputs for each data point x) and a prior distribution $p(z)$ (usually chosen as a standard Gaussian, $N(0,I)$.
- Let's go through how we derive the KL divergence expression used in the VAE.

KL divergence

1. KL Divergence Definition

The KL divergence between two distributions $q(z|x)$ and $p(z)$ is defined as:

$$\mathrm{KL}(q(z|x)||p(z)) = \mathbb{E}_{q(z|x)}\left[\log \frac{q(z|x)}{p(z)}\right]$$

Since $q(z|x)$ is often modeled as a Gaussian with mean μ and variance σ^2, this is equivalent to:

$$\mathrm{KL}(q(z|x)||p(z)) = \mathbb{E}_{q(z|x)}\left[\log q(z|x) - \log p(z)\right]$$

KL divergence

2. Substitute the Distributions

Assume:

- The prior distribution $p(z) = \mathcal{N}(0, I)$, a standard normal with mean zero and unit variance.
- The approximate posterior $q(z|x) = \mathcal{N}(z; \mu, \sigma^2)$, a Gaussian with mean μ and variance σ^2 parameterized by the encoder network.

KL divergence

3. Expand the KL Divergence Expression

Expanding using these Gaussian distributions:

$$\mathrm{KL}(\mathcal{N}(\mu, \sigma^2)||\mathcal{N}(0, I)) = \int q(z|x) \log \frac{q(z|x)}{p(z)} \, dz$$

a. Plug in the Gaussian Probability Density Functions

For Gaussian distributions, the probability density function is:

$$q(z|x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(z-\mu)^2}{2\sigma^2}\right)$$

$$p(z) = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{z^2}{2}\right)$$

KL divergence

Thus,

$$\mathrm{KL}(q(z|x)||p(z)) = \int \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(z-\mu)^2}{2\sigma^2}\right) \left[\log \frac{\frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(z-\mu)^2}{2\sigma^2}\right)}{\frac{1}{\sqrt{2\pi}} \exp\left(-\frac{z^2}{2}\right)}\right] dz$$

b. Simplify the Logarithmic Term

Simplifying the log term inside the integral:

$$\log \frac{\frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(z-\mu)^2}{2\sigma^2}\right)}{\frac{1}{\sqrt{2\pi}} \exp\left(-\frac{z^2}{2}\right)} = \log \frac{1}{\sqrt{\sigma^2}} - \frac{(z-\mu)^2}{2\sigma^2} + \frac{z^2}{2}$$

KL divergence

- To compute KL divergence, we need the following:
- If ϵ is a random variable following a standard normal distribution, denoted $\epsilon \sim N(0,1)$, we want to find the expected values $E(\epsilon)$ and $E(\epsilon^2)$.

1. **Expectation $E(\epsilon) = 0$**

- Since $\epsilon \sim N(0,1)$, the mean of a standard normal distribution is 0. Therefore:
- $E(\epsilon)=0$

2. **Expectation $E(\epsilon^2) =1$**

- For a standard normal distribution, the variance σ^2 is 1, and we know that:
- $E(\epsilon^2)=Var(\epsilon)+(E(\epsilon))^2$
- Since $Var(\epsilon)=1$ and $E(\epsilon)=0$, this simplifies to:
- $E(\epsilon^2)=1$

KL divergence

- The expectation $E_{q(z/x)}$ over this simplified term is:

$E(\log \frac{1}{\sqrt{\sigma^2}} - \frac{(z-\mu)^2}{2\sigma^2} + \frac{z^2}{2})$, where $z=\mu+\sigma.\epsilon$ and $\epsilon \sim N(0,1)$

$$= \sum_{j=1}^{d} E(log\frac{1}{\sigma_j} - \frac{(z_j-\mu_j)^2}{2{\sigma_j}^2} + \frac{{z_j}^2}{2}), where\ z_j = \mu_j + \sigma_j.\epsilon$$

$$= \sum_{j=1}^{d} [E(-log\sigma_j) - \frac{{z_j}^2 - 2z_j\mu_j + {\mu_j}^2}{2{\sigma_j}^2} + \frac{{z_j}^2}{2}]$$

$$= -\frac{1}{2}\sum_{j=1}^{d} [E(log{\sigma_j}^2) + E\left(\frac{(\mu_j+\sigma_j.\epsilon)^2}{{\sigma_j}^2}\right) - E\left(\frac{2(\mu_j+\sigma_j.\epsilon)\mu_j}{{\sigma_j}^2}\right) + E\left(\frac{{\mu_j}^2}{{\sigma_j}^2}\right) - E((\mu_j+\sigma_j.\epsilon)^2)]$$

KL divergence

$$= -\frac{1}{2}\sum_{j=1}^{d}[log{\sigma_j}^2 + \frac{1}{{\sigma_j}^2}\left(E\left({\mu_j}^2 + 2\mu_j\sigma_j\epsilon + {\sigma_j}^2\epsilon^2\right)\right) - \frac{1}{{\sigma_j}^2}\left(E\left(2\mu_j\mu_j + 2\mu_j\sigma_j.\epsilon\right)\right) + E\left(\frac{{\mu_j}^2}{{\sigma_j}^2}\right) - E({\mu_j}^2 + 2\mu_j\sigma_j\epsilon + {\sigma_j}^2\epsilon^2)\,]$$

$$= -\frac{1}{2}\sum_{j=1}^{d}[log{\sigma_j}^2 + \frac{{\mu_j}^2 + 0 + {\sigma_j}^2.1}{{\sigma_j}^2} - \frac{2{\mu_j}^2 + 0}{{\sigma_j}^2} + \frac{{\mu_j}^2}{{\sigma_j}^2} - ({\mu_j}^2 + 0 + {\sigma_j}^2.1)]$$

$$= -\frac{1}{2}\sum_{j=1}^{d}\left(log{\sigma_j}^2 + \frac{{\mu_j}^2}{{\sigma_j}^2} + 1 - \frac{{\mu_j}^2}{{\sigma_j}^2} - {\mu_j}^2 - {\sigma_j}^2\right) = -\frac{1}{2}\sum_{j=1}^{d}(1 + log{\sigma_j}^2 - {\mu_j}^2 - {\sigma_j}^2)$$

Therefore, $$KL = -\frac{1}{2}\sum_{j=1}^{d}(1 + log{\sigma_j}^2 - {\mu_j}^2 - {\sigma_j}^2)$$

VAE Loss Function

3. **Total Loss Function**

- The total loss function for the VAE is a combination of the reconstruction loss and the KL divergence:

$$L = L_{reconstruction} + \beta L_{KL}$$

- The parameter β can be adjusted to control the trade-off between reconstruction fidelity and the regularization enforced by the KL divergence.

Reparameterization Trick

- One of the challenges in training VAEs is that we need to backpropagate through a stochastic process (sampling from a distribution), which is not possible directly. To solve this, we use the **reparameterization trick**.

- Instead of directly sampling $z \sim \mathcal{N}(\mu, \sigma^2)$, we sample an auxiliary variable $\epsilon \sim \mathcal{N}(0, 1)$ and then compute:

$$z = \mu(x) + \sigma(x) \cdot \epsilon$$

This trick allows us to have a deterministic path for backpropagation.

Reparameterization Trick

- The **reparameterization trick** is a key technique in training **Variational Autoencoders (VAEs)**.
- It allows us to handle the stochastic (random) nature of the latent variables during backpropagation, which is needed for optimizing the VAE's parameters using gradient-based methods like **Stochastic Gradient Descent (SGD)**.
- To understand how it works, let's break it down step by step.

Problem: Stochastic Latent Variables

- In a VAE, after encoding an input x through the encoder, instead of encoding it into a **single point**, we encode it into a **distribution**. Specifically, the encoder outputs two vectors:
- **Mean vector** $\mu(x)$: Represents the center (mean) of the Gaussian distribution in the latent space for the input x.
- **Log-variance vector** $\log(\sigma^2(x))$: Represents the spread (variance) of this Gaussian distribution.
- The latent variable z is then **sampled** from this Gaussian distribution: $z \sim N(\mu(x),\sigma^2(x))$
- This means that the latent variable z is a **random** variable drawn from a Gaussian distribution, with mean $\mu(x)$ and variance $\sigma^2(x)$.

Problem: Stochastic Latent Variables

- **Challenge: Backpropagation and Random Sampling**
- **Backpropagation** is the technique used in neural networks to compute gradients and update parameters.
- During backpropagation, gradients need to flow back through the entire computational graph to update the parameters of the encoder and decoder.
- However, because **random sampling** from the latent variable z is non-differentiable (sampling introduces randomness), we can't directly backpropagate through this stochastic node. This makes training the VAE using gradient-based optimization challenging.

Solution: The Reparameterization Trick

- The **reparameterization trick** is a clever way to move the randomness outside the neural network in a way that allows gradients to still flow through the encoder and decoder.
- **How it Works:**

  1. Gaussian Sampling: Instead of directly sampling z from the distribution $N(\mu,\sigma^2)$, we express z in terms of a **deterministic transformation** of a random variable.

  2. Reparameterize the Latent Variable: We rewrite z as:

  $z=\mu(x)+\sigma(x)\cdot\epsilon$ Where:

  - $\mu(x)$: The mean output by the encoder.
  - $\sigma(x)$: The standard deviation (which is the square root of variance σ^2.
  - $\epsilon \sim N(0,1)$: A random noise sampled from a standard normal distribution with mean 0 and variance 1.

Solution: The Reparameterization Trick

3. **Deterministic Transformation**: The key insight is that we express z as a **deterministic transformation** of two parts:

- The **deterministic** outputs of the encoder: $\mu(x)$ and $\sigma(x)$.
- The **random noise** ϵ, which is independent of the encoder's parameters.
- In other words, instead of sampling z directly from $N(\mu(x),\sigma^2(x))$, we sample from N(0,1) (which is easy to handle) and then shift and scale it using the encoder outputs.
- This is known as **reparameterization** because it turns the random sampling process into a deterministic operation that still includes randomness through ϵ, but in a way that allows gradients to propagate.

Solution: The Reparameterization Trick

- **Reparameterization Trick Formula:**
- $z=\mu(x)+\sigma(x)\cdot\epsilon$, where $\epsilon\sim N(0,1)$
- This transformation is fully differentiable with respect to $\mu(x)$ and $\sigma(x)$, so the gradients can flow through them during backpropagation.
- **How the Reparameterization Trick Helps in Training:**
- **Gradient Flow**:
  - Now that z is expressed as a deterministic function of $\mu(x)$, $\sigma(x)$, and ϵ, we can compute the gradients of the loss function (which depends on z) with respect to the encoder parameters (which compute $\mu(x)$ and $\sigma(x)$.
  - This allows us to train the VAE using standard gradient-based optimization techniques like backpropagation and stochastic gradient descent (SGD).
- **Preserving Stochastic Nature**:
  - Even though the reparameterization trick converts the sampling into a deterministic process, it still preserves the stochastic nature of the latent variable z because ϵ is a random variable.
  - So, the VAE retains its ability to sample from the latent space, but now it can also be trained effectively.

Solution: The Reparameterization Trick

- **Visualizing the Reparameterization Trick:**
- Imagine the process of generating the latent variable z in the VAE as follows:
- **Without the reparameterization trick**:
  - The encoder computes $\mu(x)$ and $\sigma^2(x)$, and then we directly sample $z \sim N(\mu(x), \sigma^2(x))$.
  - This sampling is non-differentiable, so backpropagation can't flow through this sampling step, preventing effective training.
- **With the reparameterization trick**:
  - The encoder still computes $\mu(x)$ and $\sigma^2(x)$, but instead of sampling directly, we sample $\epsilon \sim N(0,1)$, which is differentiable.
  - Then, we construct $z = \mu(x) + \sigma(x) \cdot \epsilon$, allowing us to perform gradient descent on the encoder parameters because $\mu(x)$ and $\sigma(x)$ are part of the deterministic path.
- In this way, the **reparameterization trick** transforms the non-differentiable sampling operation into a differentiable one, allowing VAEs to be trained efficiently.

Example with MNIST dataset

- We demonstrate how a **2D latent space** captures the distribution of data and enables **interpolation** between points.
- We use a simple dataset like **MNIST (handwritten digits)** and map it to a 2D latent space using a Variational Autoencoder (VAE).
- After mapping the dataset to the latent space, we will visually show:
- **Latent Space Distribution**: How different digit classes (e.g., 0 to 9) are distributed in the 2D latent space.
- **Interpolation**: Generating new images by interpolating between two different points in the latent space.

Example with MNIST dataset

1. Visualizing the Latent Space Distribution

- In a 2D latent space, each point represents a compressed version of a data point (e.g., an image of a digit).
- The VAE encodes each image into a point in this 2D space, where similar digits should be placed near each other.
- We will:

  -- Train a VAE on the MNIST dataset (or a similar dataset).

  -- Encode each image in the dataset to a 2D point.

  -- Color each point based on the digit it represents (e.g., '0' in red, '1' in blue, etc.).

Example with MNIST dataset

2. Interpolation in Latent Space

- After visualizing the latent space, we will demonstrate **interpolation**.
- Interpolation in the latent space means moving smoothly between two points (latent vectors) and decoding these points into actual images.
- This will show how the VAE captures a continuous transformation between two data points.
- Steps:

  1. Select two points in the latent space, each representing different digits (e.g., a '2' and a '7').

  2. Linearly interpolate between these points in the latent space.

  3. Decode the interpolated points back into the image space to see how the digit morphs from '2' to '7'.

Example with MNIST dataset

- **Latent Space**: In a Variational Autoencoder (VAE), after compressing an input (like an image of a handwritten digit) through the **encoder**, you get a **latent representation** of the input in a lower-dimensional space (often 2D or more dimensions). This space is called the **latent space**.
- Every input image (like a handwritten "0" or "1") corresponds to a point in this latent space. This means each digit is represented as a unique point (or distribution) in the latent space.
- **Decoding from Latent Space**:
- The **decoder** is the part of the VAE that takes a point from this latent space and "decodes" it back to an image (like reconstructing a "0" or "1").
- So, if you pick any point in the latent space and feed it to the decoder, the decoder will generate an image that corresponds to that point.

Example with MNIST dataset

- **Linear Interpolation: Moving Between Two Points**
- **Interpolation** means creating a transition or progression between two values.
- **Linear interpolation** between two points means creating a straight-line path between them and picking points along that path.
- **Example: Interpolation Between Two Digits**
- Let’s say we want to interpolate between two digits, for example, **'0'** and **'1'**.
- **Find the Latent Space Representation**:
  - We take an image of a **'0'** and an image of a **'1'**, and we pass them through the **encoder**.
  - The encoder gives us two points in the latent space:
    - Point 1 (representing '0'): Let's call it z_0.
    - Point 2 (representing '1'): Let's call it z_1.

Example with MNIST dataset

- **Interpolate Between the Two Points**:
- Instead of jumping directly from z_0 to z_1, we can create several points in between.
- To do this, we use **linear interpolation**: we compute points that are partway between z_0 and z_1.
- Mathematically, we can compute intermediate points using the formula:

  $$z_{interpolated} = (1-t)\cdot z_0 + t\cdot z_1$$

  - t is a value between 0 and 1.
  - When t=0, the point is exactly z_0 (so it represents '0').
  - When t=1, the point is exactly z_1 (so it represents '1').
  - As we change t from 0 to 1, we get points in between z_0 and z_1.
- **Decode the Interpolated Points**:
- Each of these interpolated points is passed through the **decoder**, which converts them back into images.
- The images you get will start to look like a gradual transition from '0' to '1'.

Example with MNIST dataset

- **Visualization of Interpolation:**
- Imagine a straight line connecting the point for a **'0'** and the point for a **'1'** in the latent space.
- By picking points along this line and decoding them, you generate images that gradually transform from looking like a '0' into looking like a '1'.
- **Visual Example:**
- Think of this interpolation as a **morphing** or **transition** process. Here's what might happen as you move from a '0' to a '1':
- t=0: You see a clear image of a '0'.
- t=0.2: The digit starts changing slightly, losing some of the round shape of '0'.
- t=0.5: The digit is halfway between a '0' and '1', looking like a blurry or distorted digit.
- t=0.8: The digit starts looking more like a '1', with a straightening of the figure.
- t=1: You see a clear image of a '1'.
- In short, **linear interpolation** allows you to create a smooth **morphing** effect between any two digits in the latent space.

Example with MNIST dataset

- **Illustration**
- **Original points**:
  - z_0 (Latent representation of '0')
  - z_1 (Latent representation of '1')
- **Interpolation points**:
  - $Z_{interpolated}$ for t=0.25, t=0.5, t=0.75, etc.
- Each $z_{interpolated}$ is fed into the decoder, producing images that are steps between '0' and '1'.

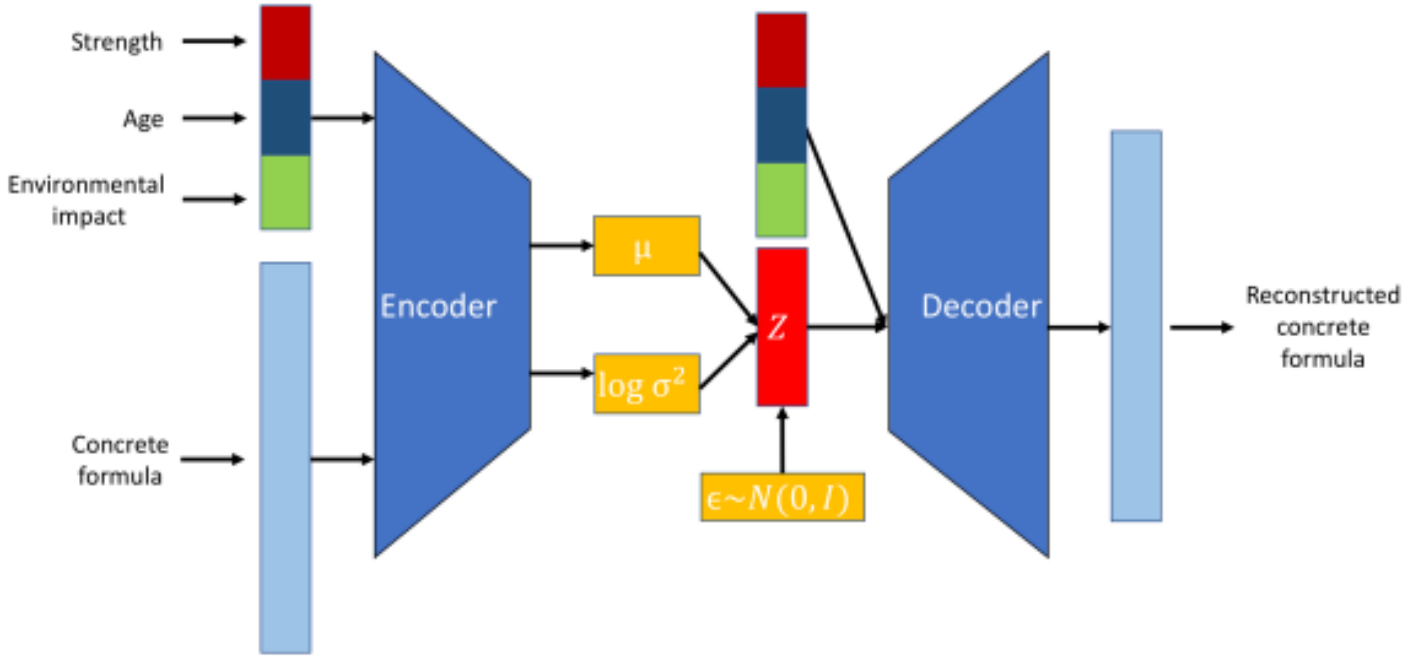
Types of Variational Autoencoders

- VAEs have evolved into various specialized forms to address different challenges and applications in machine learning. In this section, we'll examine the most prominent types, highlighting use cases, advantages, and limitations.

1. Conditional variational autoencoder

- Conditional Variational Autoencoders (CVAEs) are a specialized form of VAEs that enhance the generative process by conditioning on additional information.
- A VAE becomes conditional by incorporating additional information, denoted as *c*, into both the encoder and decoder networks.
- This conditioning information can be any relevant data, such as class labels, attributes, or other contextual data.

CVAE



- Use cases of CVAEs include:
- **Controlled data generation.** For example, in image generation, a CVAE can create images of specific objects or scenes based on given labels or descriptions.

CVAE

- **Image-to-image translation:** CVAEs can transform images from one domain to another while maintaining specific attributes. For instance, they can be used to translate black-and-white images to color images or to convert sketches into realistic photos.
- **Text generation.** CVAEs can generate text conditioned on specific prompts or topics, making them useful for tasks like story generation, chatbot responses, and personalized content creation.
- The pros and cons are:

  i) Finer control over generated data

  ii) Improved representation learning

  iii) Increased risk of overfitting

Other variants of VAE

- Disentangled Variational Autoencoders, often called Beta-VAEs
- Adversarial Autoencoders (AAEs)
- Variational Recurrent Autoencoders (VRAEs)
- Hierarchical Variational Autoencoders (HVAEs)

Thank You