

Deep Unsupervised Learning: Auto encoders

Deep Unsupervised Learning

- Deep unsupervised learning leverages deep neural networks to find hidden structures, patterns, or representations in data without labeled outputs.
- Unlike supervised learning, which requires labeled data for training, unsupervised learning works with unlabeled data to discover inherent relationships.
- Key Techniques in Deep Unsupervised Learning
 - **Autoencoders (AE)**
 - **Purpose:** Learn efficient representations by reconstructing input data.
 - **Structure:** Encoder reduces data to a lower-dimensional latent space, and decoder reconstructs the original data.
 - ☒ *Variants:*
 - Denoising Autoencoder (DAE)
 - Variational Autoencoder (VAE)
 - Sparse Autoencoder
 - Contractive Autoencoder

- **Variational Autoencoder (VAE)**

- **Purpose:** Learn a probabilistic mapping of the data to a latent space and generate new data.
- **Key Feature:** Introduces a probabilistic layer in the latent space to ensure smooth latent representation.
- **Applications:** Image generation, anomaly detection.

- **Generative Adversarial Networks (GANs)**

- **Purpose:** Generate realistic synthetic data by pitting a **generator** against a **discriminator**.
- ☒ *Variants:*
 - Deep Convolutional GAN (DCGAN)
 - Conditional GAN (cGAN)
 - StyleGAN, CycleGAN

- **Self-Organizing Maps (SOMs)**

- **Purpose:** Reduce dimensionality and visualize high-dimensional data.
- **How it Works:** Maps high-dimensional data to a 2D grid, preserving topological relationships.

- **Restricted Boltzmann Machines (RBMs)**

- **Purpose:** Learn a probability distribution over the input data.
- **Structure:** Bipartite graph with visible and hidden units.

- **Deep Belief Networks (DBNs)**

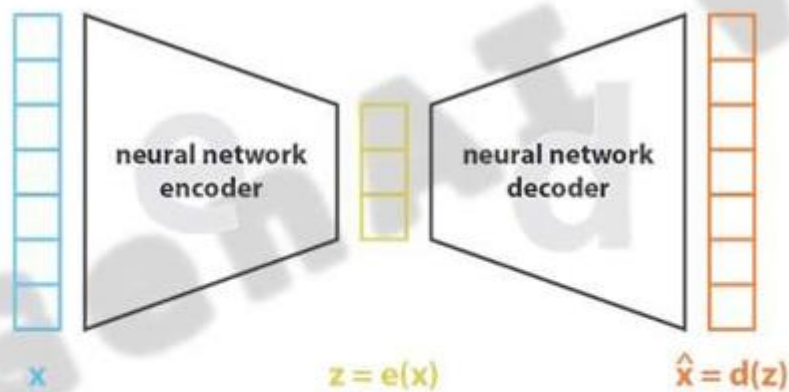
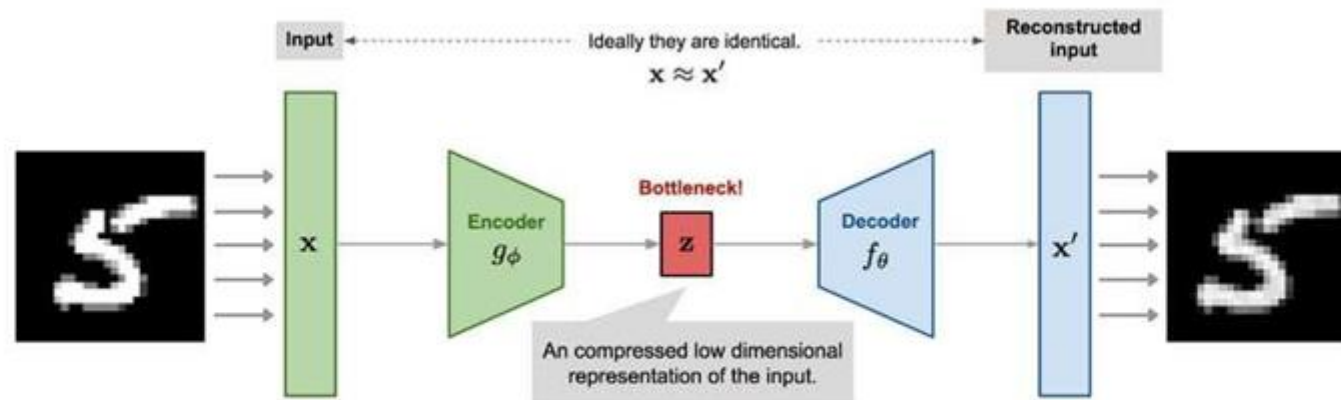
- **Purpose:** Unsupervised pre-training followed by fine-tuning with supervised learning.
- **Architecture:** Stack of RBMs with greedy layer-wise training.

Applications of Deep Unsupervised Learning

- 1. Anomaly Detection:** Identifying outliers in data.
- 2. Data Generation:** GANs for realistic image, text, and audio generation.
- 3. Dimensionality Reduction:** Autoencoders for feature extraction and PCA alternatives.
- 4. Recommendation Systems:** Learning latent user preferences.
- 5. Medical Imaging:** Identifying hidden patterns in large datasets.

1. Unsupervised Learning with Autoencoders (AE)

- Autoencoders are types of neural network architecture used for unsupervised learning
- It is a type of neural network architecture designed to efficiently compress (encode) input data down to its essential features, then reconstruct (decode) the original input from this compressed representation.
- Auto-encoders consists of three components:
 - **Encoder**: A module that compresses the input into a compact representation and capture the most relevant features (latent representation).
 - **Bottleneck/ latent space**: A module that contains the compressed knowledge representations and is most important part of the network.
 - **Decoder**: A module that reconstructs the input data from this compressed form to make it as similar as possible to the original input.
 - For example if the input is a noisy image of handwritten digits the autoencoder can learn to remove noise by compressing the image into a smaller feature set and reconstructing a cleaner version of the original image.
- Auto-encoders aim to minimize **reconstruction error** which is the difference between the input and the reconstructed output.
- They use loss functions such as **Mean Squared Error (MSE)** or **Binary Cross-Entropy (BCE)** and optimize through **backpropagation and gradient descent**.
- They are used in applications like image processing, anomaly detection, noise removal and feature extraction

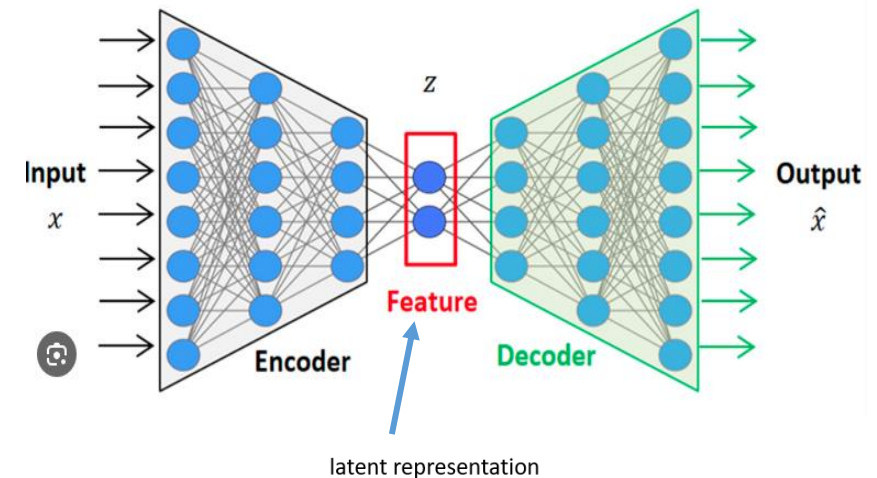
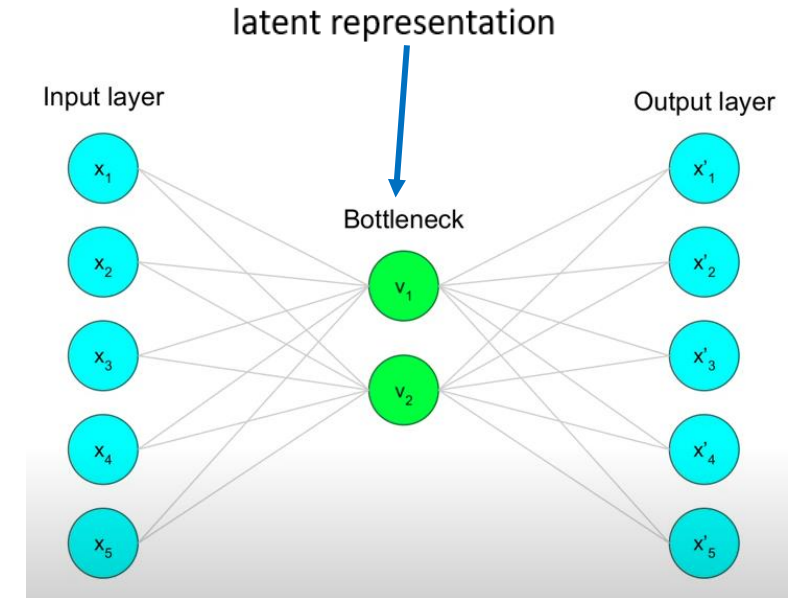


- Bottleneck/Code
- knowledge-representation of the input.
- Eg: maximum information possessed by an image is captured

$$\text{loss} = \|x - \hat{x}\|^2 = \|x - d(z)\|^2 = \|x - d(e(x))\|^2$$

Architecture of Auto-encoder

- The architecture of an autoencoder consists of three main components: the Encoder, the Bottleneck (Latent Space) and the Decoder.
- **1. Encoder:**
 - The encoder is a part of the network that compresses the input data into a lower-dimensional representation (latent space or code). It takes the input, typically a high-dimensional data point (e.g., an image or a vector), and compresses it into a compact vector that captures the essential features of the input.
 - **Input Layer:** This is where the original data enters the network e.g., an image or a set of features.
 - **Hidden Layers:** These layers apply transformations to the input data. The encoder's goal is to extract essential features and reduce the data's dimensionality.
 - **Output of Encoder (Latent Space):** The encoder outputs a compressed version of the data often called the latent representation or encoding. This is a condensed version of the input retaining only the important features.



- **2. Latent Space (bottleneck layer):**

- The encoder maps the input x to a smaller hidden representation h , which is generally called the latent space.
- This is the compressed, lower-dimensional representation of the input. The latent space is often a vector of reduced dimensionality, capturing the most important features of the input data.
- The latent space is a crucial feature because it forces the network to learn an efficient, dense representation of the data.

$$h = f(x) = \sigma(W_e x + b_e)$$

Where:

- x is the input vector (e.g., an image or feature vector).
- W_e and b_e are the weight matrix and bias of the encoder.
- σ is an activation function (e.g., sigmoid, ReLU).
- h is the latent space representation of the input x .

- **3. Decoder:**

- The decoder is a part of the network that reconstructs the original input data from the compressed latent representation.
- It takes the encoded, lower-dimensional representation h and tries to reconstruct the original data point x' .

$$x' = g(h) = \sigma(W_d h + b_d)$$

Where:

- h is the compressed representation from the encoder.
 - W_d and b_d are the weight matrix and bias of the decoder.
 - σ is an activation function (could be different from the encoder, depending on the task).
 - x' is the reconstructed output, which should approximate the original input x .
- The goal of the decoder is to output a reconstruction of the input that is as close as possible to the original input.

- **Training an Autoencoder:**

- Training an autoencoder is done by minimizing the **reconstruction error** (the difference between the input data and the reconstructed data).

- **2. Loss Function:**

- The goal of the autoencoder is to minimize **reconstruction error**, the difference between the input x and the reconstructed output x' . This difference is often measured using a **reconstruction loss**. The most commonly used loss functions are:

- **Mean Squared Error (MSE):**

- For continuous-valued data (e.g., images), the **Mean Squared Error (MSE)** is often used as the loss function:

$$\mathcal{L} = \frac{1}{N} \sum_{i=1}^N \|x_i - x'_i\|^2$$

Where:

- x_i is the original data point (input) for the i -th sample.
- x'_i is the reconstructed data point (output) for the i -th sample.
- N is the number of samples in the dataset.
- $\| \cdot \|$ denotes the Euclidean distance (L2 norm).

- **Binary Cross-Entropy (for Binary Data):**

- For binary data (e.g., binary images or one-hot encoded vectors), **Binary Cross-Entropy** is more appropriate:

$$\mathcal{L} = - \sum_{i=1}^N (x_i \log(x'_i) + (1 - x_i) \log(1 - x'_i))$$

Where:

- x_i is the original data point for the i -th sample (binary).
- x'_i is the reconstructed data point for the i -th sample.
- This loss function measures the difference between the true data and the reconstructed data in terms of likelihood.

- **Optimization Objective:**

- The goal of training the auto-encoder is to minimize the reconstruction loss L . This is done by adjusting the parameters W_e, b_e, W_d, b_d (weights and biases) of the encoder and decoder using an optimization algorithm, typically **Gradient Descent**.

$$\theta^* = \arg \min_{\theta} \mathcal{L}(x, x')$$

Where:

- θ represents the parameters of the network, i.e., $\theta = \{W_e, b_e, W_d, b_d\}$.
- $\mathcal{L}(x, x')$ is the loss function that measures the reconstruction error.

Sample code

```
input = layers.Input(shape=(784,))
x = layers.Dense(500,activation = 'relu')(input)
x = layers.Dense(300,activation = 'relu')(x)
x = layers.Dense(150,activation = 'relu')(x)
x = layers.Dense(100,activation = 'relu')(x)
latent_code = layers.Dense(64,activation = 'relu')(x)
x = layers.Dense(100,activation = 'relu')(latent_code)
x = layers.Dense(150,activation = 'relu')(x)
x = layers.Dense(300,activation = 'relu')(x)
x = layers.Dense(500,activation = 'relu')(x)
output = layers.Dense(784,activation = 'relu')(x)

autoencoder = Model(input,output)
```

```
autoencoder.compile(optimizer='adam', loss=losses.MeanSquaredError())
x_train = x_train.reshape(60000,784)
x_test = x_test.reshape(10000,784)
```

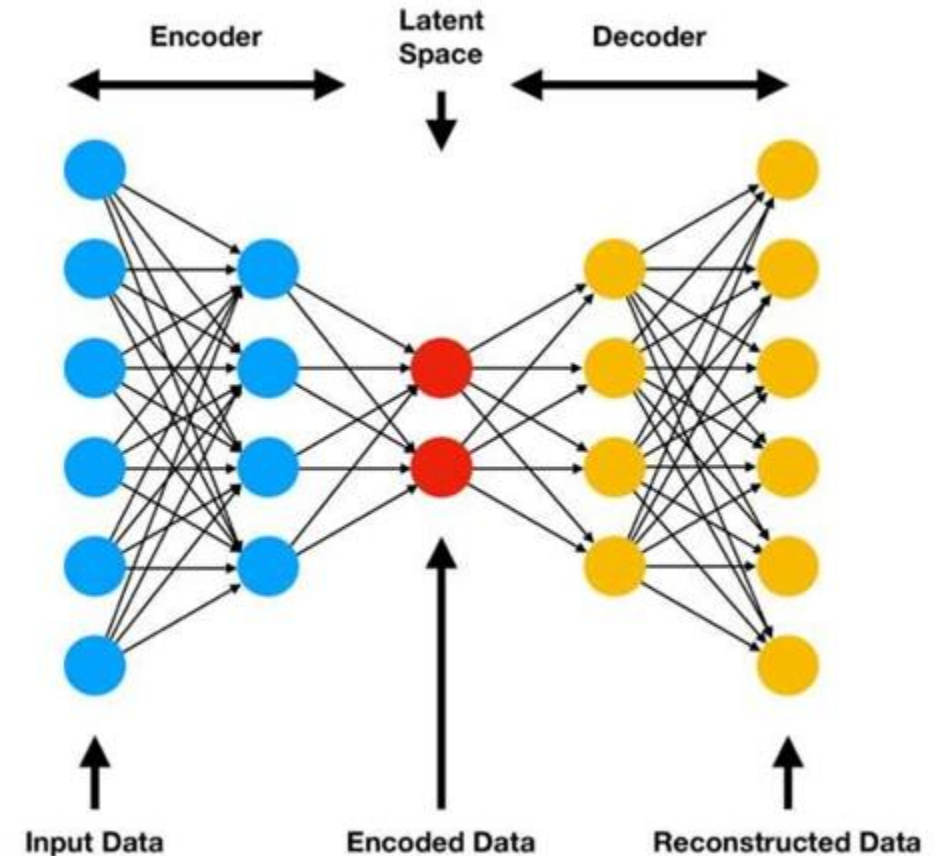
```
autoencoder.fit(x_train, x_train,
               epochs=10,
               shuffle=True,
               validation_data=(x_test, x_test))
```

TYPES OF AUTOENCODERS

- Depending upon the actual dimensions of the encoded layer with respect to the input, the loss function, and constraints, there are various types of autoencoders:
 - Undercomplete Autoencoders
 - Overcomplete Autoencoders
 - Denoising autoencoders
 - Stacked autoencoders
 - Sparse autoencoders
 - Variational Autoencoders
 - Contrastive Autoencoders

1. UNDER COMPLETE AUTOENCODERS

- Under complete autoencoders is an unsupervised neural network that you can use to *generate a compressed version of the input data.*
- It is done by taking in an image and trying to predict the same image as output, thus *reconstructing the image from its compressed bottleneck region*
 - Compression
 - Dimensionality reduction.



2. OVER COMPLETE AUTOENCODERS

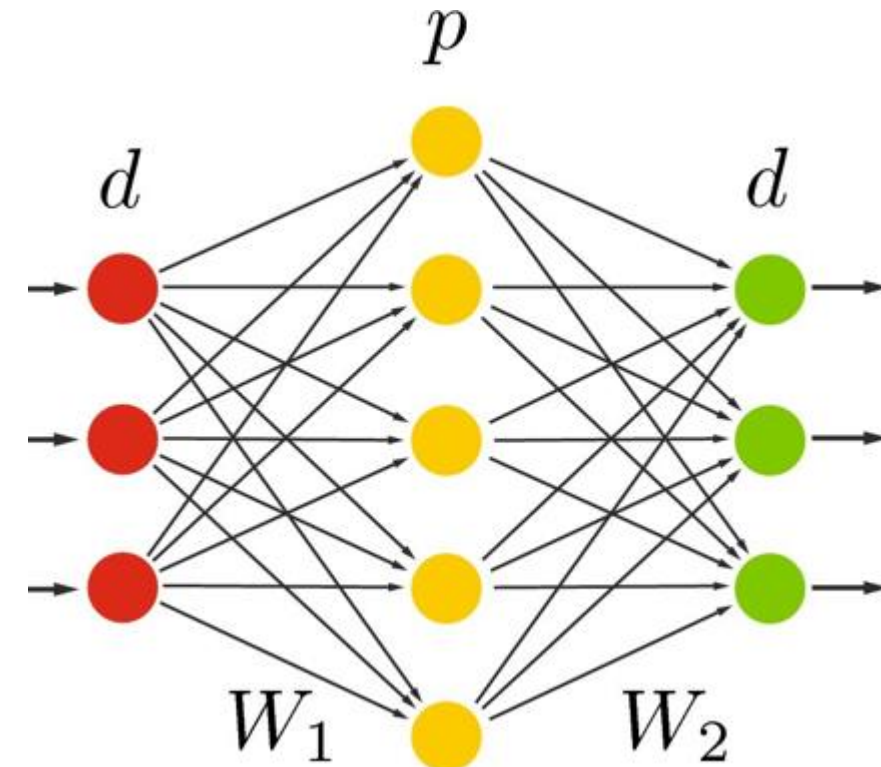
- Overcomplete Autoencoders are a type of autoencoders that use more hidden units than the number of input units. This means that the encoder and decoder layers have more units than the input layer.
- The idea behind using more hidden units is to learn a more complex, non-linear function that can better capture the structure of the data.

Advantages

- ability to **learn more complex representations** of the data, which can be useful for tasks such as feature extraction and denoising.
- **more robust to noise** and can handle missing data better than traditional autoencoders

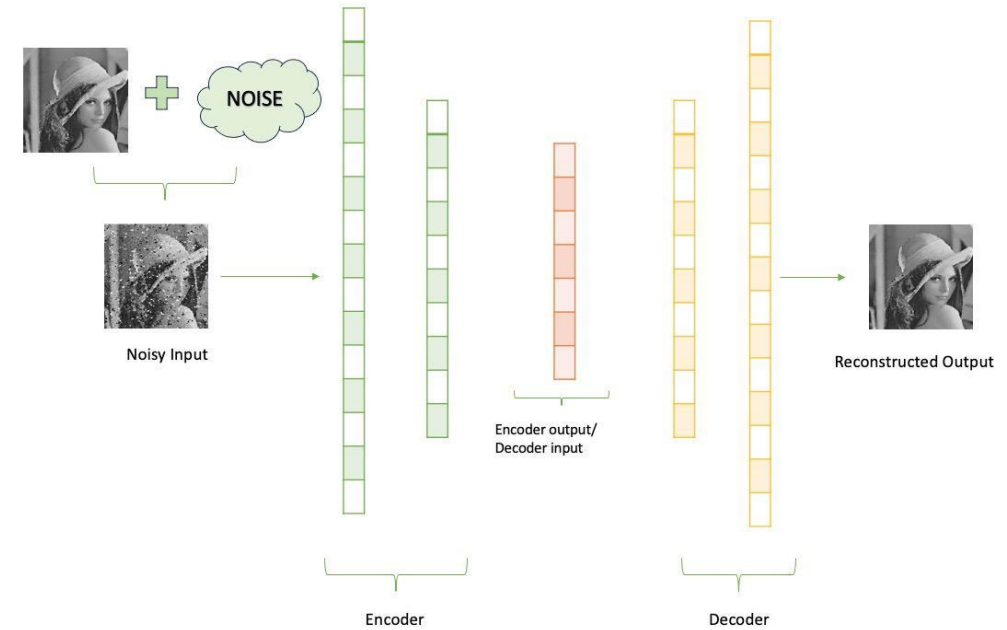
Disadvantages

- One of the main disadvantages is that they can be **more difficult to train** than traditional autoencoders.
- the extra hidden units can cause **overfitting**, which can lead to poor performance on new data.



3. Denoising Autoencoder

- Denoising autoencoder is a modification of the original autoencoder in which instead of giving the original input we give a corrupted or noisy version of input to the encoder
- Decoder loss is calculated concerning original input only.
 - Encoder:
 - The encoder of DAE is a neural network with one or more hidden layers similar to ordinary AE.
 - It receives noisy input data instead of the original input and generates an encoding in a low-dimensional space.
 - There are several ways to generate a corrupted input. The most common being adding a Gaussian noise or randomly masking some of the inputs.
 - Decoder:
 - Similar to encoders, decoders are implemented as neural networks with one or more hidden layers.
 - It takes the encoding generated by the encoder as input and reconstructs the original data.
 - **When calculating the Loss function, it compares the output values with the original input, not with the corrupted input.**



- **Objective Function of DAE**

- The objective of DAE is to minimize the difference between the original input (clean input without the noise) and the reconstructed output. This is quantified using a reconstruction loss function. Two types of loss function are generally used depending on the type of input data.
- **Mean Squared Error (MSE):**
 - If we have input image data in the form of floating pixel values i.e. values between (0 to 1) or (0 to 255) we use MSE

$$MSE(x, y) = \frac{1}{m} \sum_{i=1}^m (x_i - y_i)^2$$

Here,

- each of x_i is the pixel value of input data
- y_i is the pixel value of reconstructed data
 - $y_i = D(E(x_i * \text{noise}))$
 - Where E represents encoder and D represents decoder
- this is summed over all the training set

- Binary Cross-Entropy (log-loss):
 - If we have input image data in the form of bits pixel values i.e. values will be either 0 or 1 only then we can use binary cross entropy loss for each pixel value

$$LL(x, y) = -\frac{1}{m} \sum_{i=1}^m (x_i \ln(y_i) + (1 - x_i) \ln(1 - y_i))$$

Here

- each of x_i is the pixel value of input data with value being only 0 or 1
- y_i is the pixel value of reconstructed data.
 - $y_i = D(E(x_i * \text{noise}))$
- Where E represents encoder and D represents decoder
- this is summed over all the training set

- **Training Process of DAE**

- The training of DAE consists of below steps:
 - Initialize encoder and decoder with random weights
 - Noise is intentionally added to the input data.
 - Feedforward the input data through encoder and decoder to get the reconstructed image
 - Calculate the reconstruction loss as defined in our objective function w
 - Do backpropagation and update weights. The goal during training is to minimize the reconstruction loss.
- The training is typically done through optimization algorithms like stochastic gradient descent (SGD) or its variants.

```
# Load MNIST dataset
```

```
(x_train, _), (x_test, _) = mnist.load_data()
```

```
# Add noise to images
```

```
noise_factor = 0.5
```

```
x_train_noisy = x_train + noise_factor * np.random.normal(loc=0.0, scale=1.0, size=x_train.shape)
```

```
x_test_noisy = x_test + noise_factor * np.random.normal(loc=0.0, scale=1.0, size=x_test.shape)
```

```
# Clip values to be between 0 and 1
```

```
x_train_noisy = np.clip(x_train_noisy, 0., 1.)
```

```
x_test_noisy = np.clip(x_test_noisy, 0., 1.)
```

```
# Define input shape
```

```
input_shape = (28, 28)
```

```
input_img = Input(shape=input_shape)
```

```
# Flatten the input
```

```
x = Flatten()(input_img)
```

```
# Encoder: compressing input
```

```
encoded = Dense(128, activation='relu')(x)
```

```
encoded = Dense(64, activation='relu')(encoded)
```

```
encoded = Dense(32, activation='relu')(encoded)
```

```
# Decoder: reconstructing the image
```

```
decoded = Dense(64, activation='relu')(encoded)
```

```
decoded = Dense(128, activation='relu')(decoded)
```

```
decoded = Dense(28 * 28, activation='sigmoid')(decoded)
```

```
decoded = Reshape((28, 28))(decoded)
```

```
# Autoencoder Model
```

```
autoencoder = Model(input_img, decoded)
```

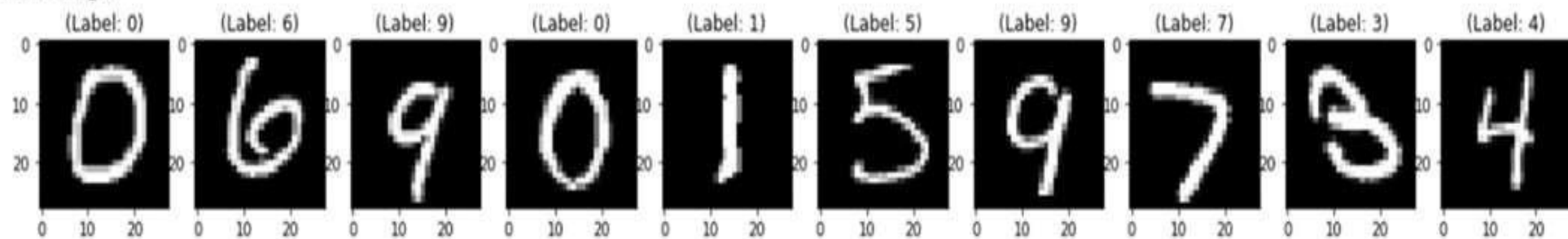
```
# Train the autoencoder
```

```
autoencoder.fit(x_train_noisy, x_train,  
                epochs=10,  
                batch_size=256,  
                shuffle=True,  
                validation_data=(x_test_noisy, x_test))
```

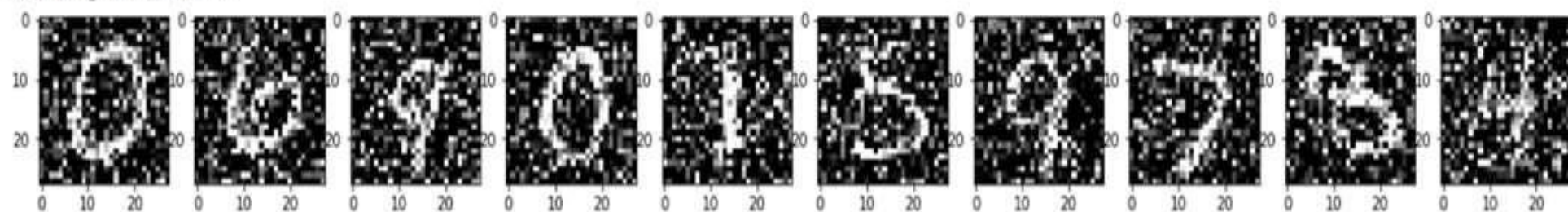
```
# Denoise test images
```

```
denoised_images = autoencoder.predict(x_test_noisy)
```

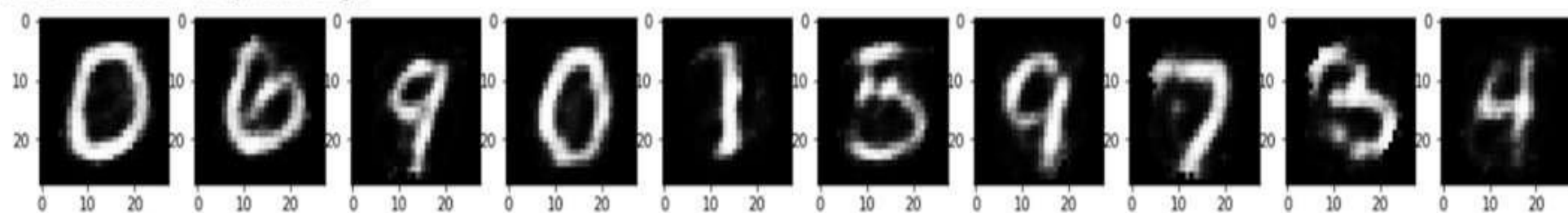

Test Images



Test Images with Noise

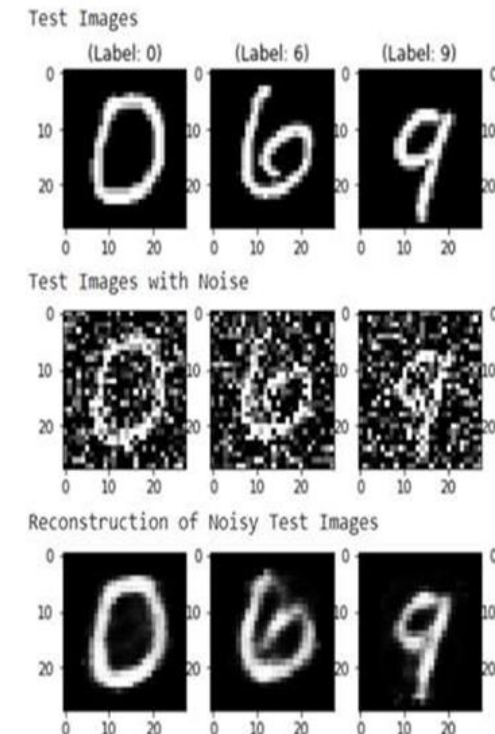


Reconstruction of Noisy Test Images



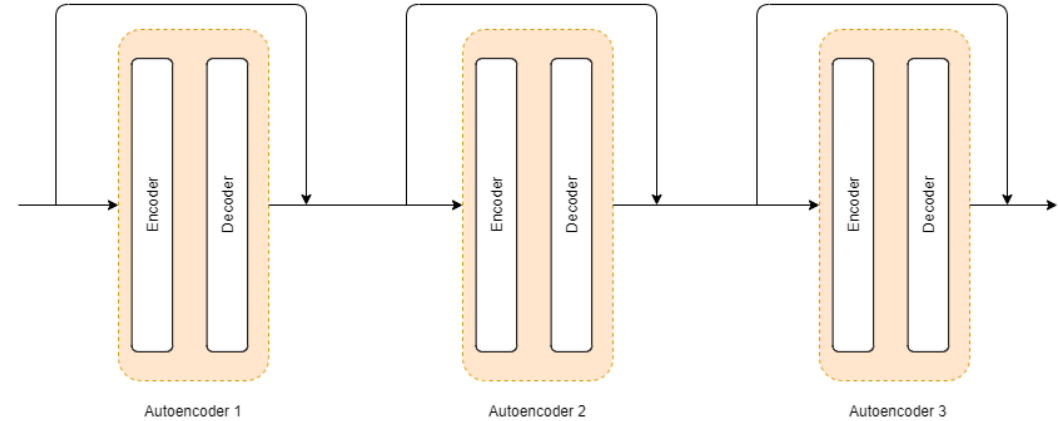
What DAE Learns?

- The above architecture of using a corrupted input helps decrease the risk of overfitting and prevents the DAE from becoming an identity function.
 - **Imputation of Missing Information:**
 - When DAEs are trained with **partially corrupted inputs** (e.g., masking some values), they learn to fill in or impute missing information.
 - Useful for tasks involving **incomplete datasets** (missing values).
 - **Generalization with Noisy Inputs:**
 - When DAEs are trained with **partially noisy inputs** (e.g., Gaussian noise), they generalize well to **unseen, real-world data** with varying noise levels.
 - Beneficial in applications where data quality is compromised, such as **image denoising** or **signal processing**.



4. Stacked Autoencoder

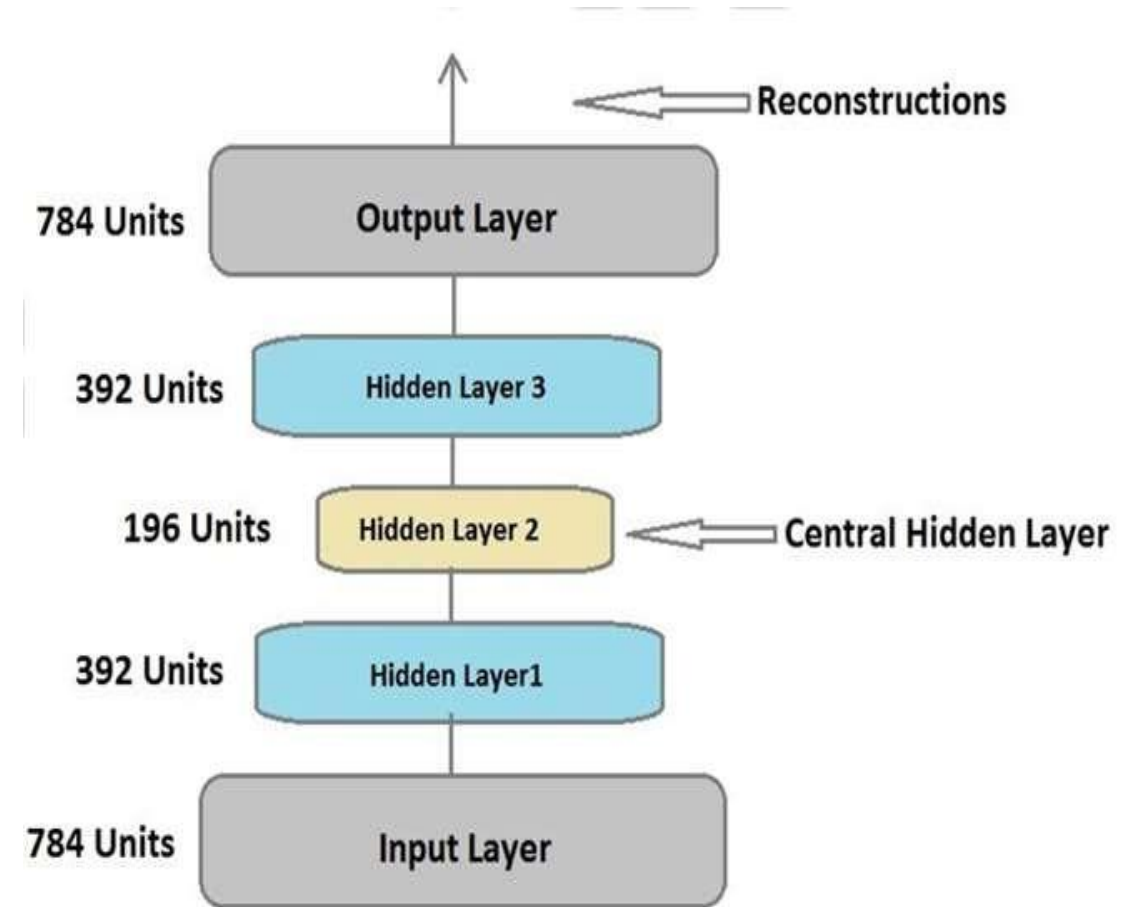
- Some datasets have a complex relationship within the features. Thus, using only one Autoencoder is not sufficient.
- A single Autoencoder might be unable to reduce the dimensionality of the input features. Therefore for such use cases, we use stacked autoencoders.
- The stacked autoencoders are, as the name suggests, multiple encoders stacked on top of one another.



• Implementation1

- The input data is first given to autoencoder 1.
- The output of the autoencoder 1 and the input of the autoencoder 1 is then given as an input to autoencoder 2.
- Similarly, the output of autoencoder 2 and the input of autoencoder 2 are given as input to autoencoder 3.
- Thus, the length of the input vector for autoencoder 3 is double than the input to the input of autoencoder 2.
- This technique also helps to solve the problem of insufficient data to some extent.

- Implementation 2
- Deep autoencoders with many layers of both encoder and decoder.
- The stacked autoencoder can be trained as a whole network with an aim to minimize the reconstruction error



```

from keras.models import Model, Sequential
from keras.layers import Input, Dense

# Layer-wise training for stacked autoencoder
def build_autoencoder(input_dim, encoding_dim):
    input_layer = Input(shape=(input_dim,))
    encoded = Dense(encoding_dim, activation='relu')(input_layer)
    decoded = Dense(input_dim, activation='sigmoid')(encoded)
    autoencoder = Model(inputs=input_layer, outputs=decoded)
    return autoencoder

# Define dimensions
input_dim = 784
encoding_dims = [256, 128, 64]

# Layer-wise pre-training
prev_input_dim = input_dim
for encoding_dim in encoding_dims:
    autoencoder = build_autoencoder(prev_input_dim, encoding_dim)
    autoencoder.compile(optimizer='adam', loss='binary_crossentropy')
    prev_input_dim = encoding_dim

```

```

# Final stacked autoencoder
stacked_autoencoder = Sequential()
stacked_autoencoder.add(Dense(256, activation='relu', input_shape=(784,)))
stacked_autoencoder.add(Dense(128, activation='relu'))
stacked_autoencoder.add(Dense(64, activation='relu'))

# Decoder
stacked_autoencoder.add(Dense(128, activation='relu'))
stacked_autoencoder.add(Dense(256, activation='relu'))
stacked_autoencoder.add(Dense(784, activation='sigmoid'))

stacked_autoencoder.compile(optimizer='adam', loss='binary_crossentropy')

```

5. Sparse autoencoders

- In a standard autoencoder, the network learns to encode and decode data without any constraints on the hidden layer's activations.
- Sparse autoencoders modify this behavior by adding a sparsity constraint, which forces the hidden units to activate only a small number of neurons at a time. This encourages the network to discover more meaningful and interpretable features.
- The sparsity of the network can be controlled by either manually zeroing the required hidden units, tuning the activation functions or by adding a loss term to the cost function.
- The objective function for a sparse autoencoder can be expressed as:

The objective function for a sparse autoencoder can be expressed as:

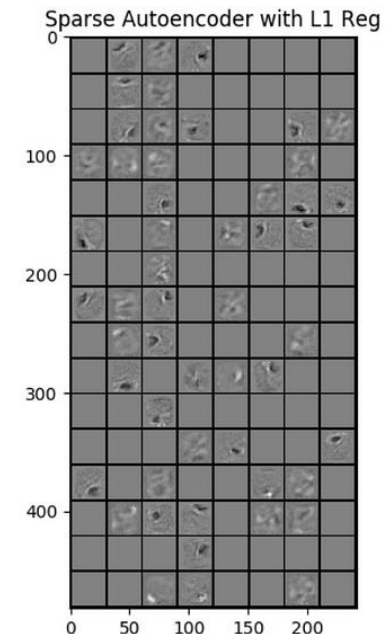
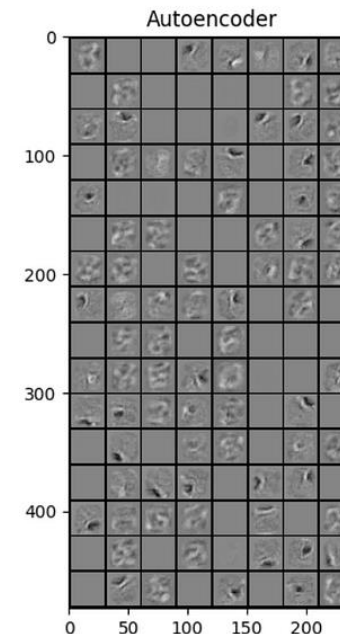
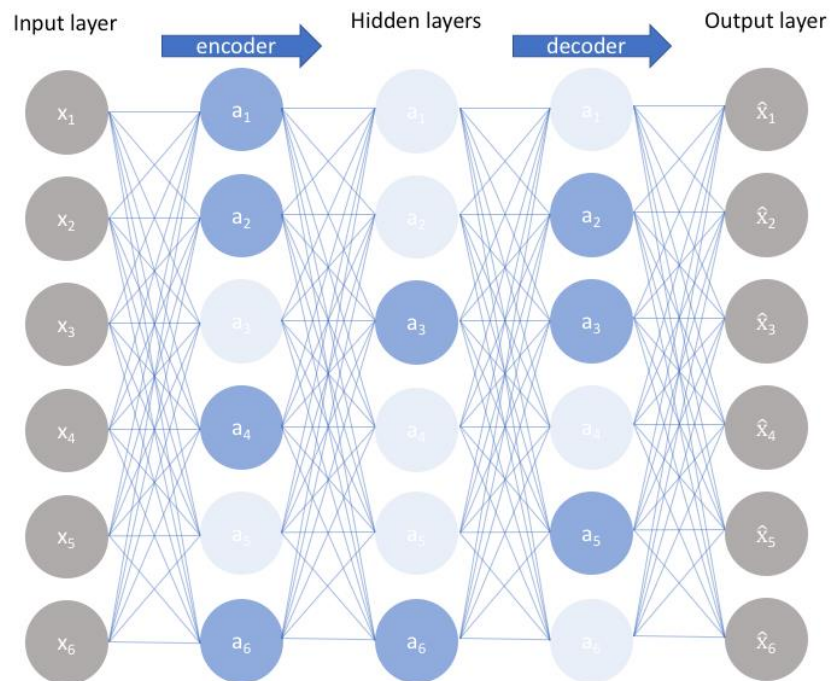
$$L = ||X - \hat{X}||^2 + \lambda \cdot \text{Penalty}(s)$$

- X : Input data.
- \hat{X} : Reconstructed output.
- λ : Regularization parameter.
- $\text{Penalty}(s)$: A function that penalizes deviations from sparsity, often implemented using KL-divergence.

The sparsity constraint can be enforced through various techniques:

1.L1 Regularization: Adds a penalty proportional to the absolute values of the weights.

2.KL Divergence: Measures the difference between the average activation of the hidden neurons and a target sparsity level.



Training Sparse Autoencoders

Training a sparse autoencoder typically involves:

- 1.Initialization:** Weights are initialized randomly or using pre-trained networks.
- 2.Forward Pass:** The input is fed through the encoder to obtain the latent representation, followed by the decoder to reconstruct the output.
- 3.Loss Calculation:** The loss function is computed, incorporating both the reconstruction error and the sparsity penalty.
- 4.Backpropagation:** The gradients are calculated and used to update the weights.

6. Convolutional autoencoder

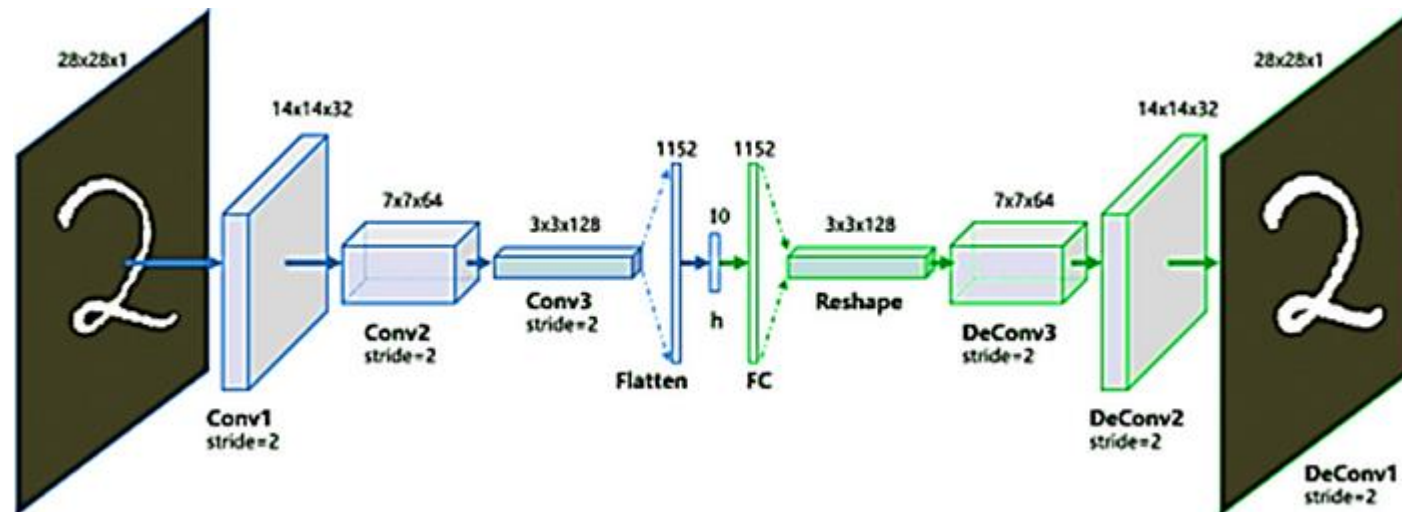
- A **Convolutional Autoencoder (CAE)** is a type of autoencoder that uses **convolutional layers** in its encoder and decoder architecture, making it well-suited for image data. Convolutional autoencoders excel at learning compressed representations of images by preserving spatial relationships.
- A convolutional autoencoder consists of two main components:

1. Encoder:

1. Compresses the input into a lower-dimensional latent representation.
2. Uses convolutional layers and pooling (e.g., MaxPooling) to reduce spatial dimensions.

2. Decoder:

1. Reconstructs the original input from the latent representation.
2. Uses transposed convolution (Conv2DTranspose) or upsampling to restore the original dimensions.




```

# Encoder
input_img = Input(shape=(28, 28, 1))
x = Conv2D(32, (3, 3), activation='relu', padding='same')(input_img)
x = MaxPooling2D((2, 2), padding='same')(x)
x = Conv2D(64, (3, 3), activation='relu', padding='same')(x)
x = MaxPooling2D((2, 2), padding='same')(x)

# Latent space representation
encoded = Conv2D(128, (3, 3), activation='relu', padding='same')(x)

# Decoder
x = Conv2D(64, (3, 3), activation='relu', padding='same')(encoded)
x = UpSampling2D((2, 2))(x)
x = Conv2D(32, (3, 3), activation='relu', padding='same')(x)
x = UpSampling2D((2, 2))(x)
decoded = Conv2D(1, (3, 3), activation='sigmoid', padding='same')(x)

```

```

# Define the autoencoder model
autoencoder = Model(input_img, decoded)
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')
autoencoder.summary()

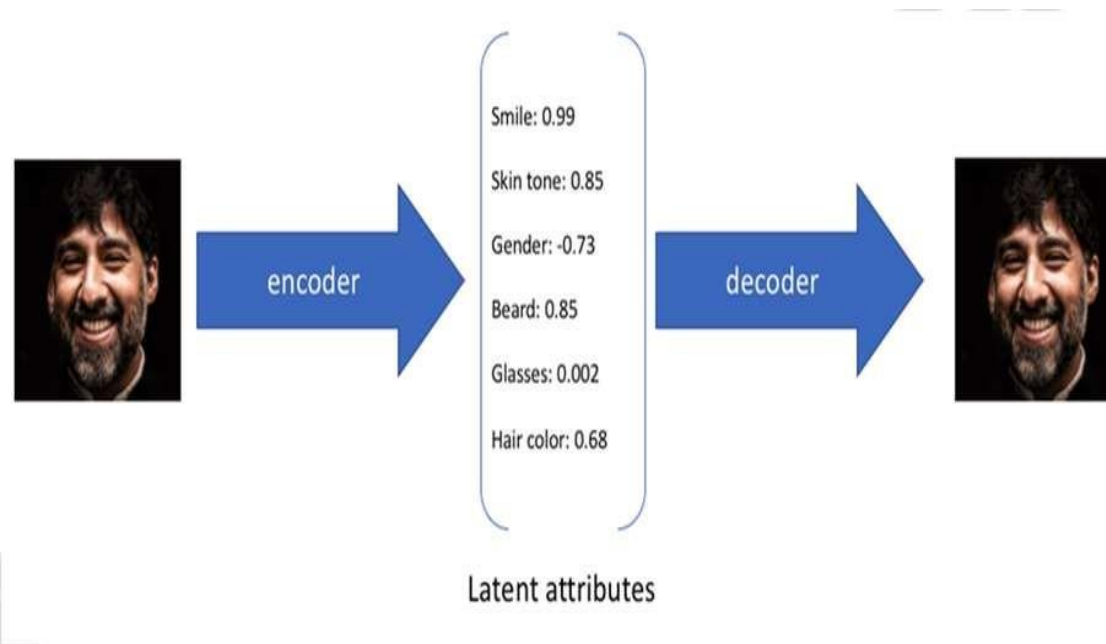
# Train the model
autoencoder.fit(x_train, x_train,
                epochs=20,
                batch_size=256,
                shuffle=True,
                validation_data=(x_test, x_test))

```

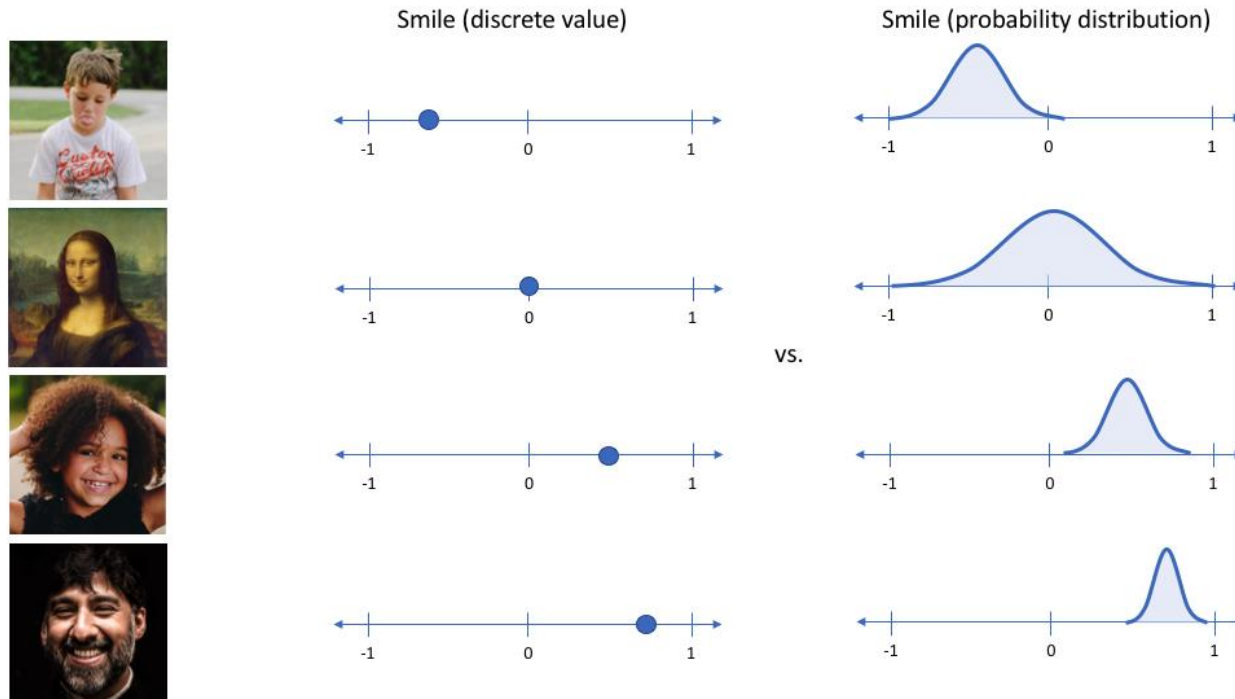

7. Variational Autoencoder

- Variational Autoencoders (VAEs) are generative models in machine learning (ML) that **create new data similar to the input they are trained on.**
- Unlike traditional autoencoders that encode a fixed representation (deterministic representation) **VAEs learn a continuous probabilistic representation of latent space.**
- This allows them to reconstruct input data accurately and generate new data samples that resemble the original input.

- Recap- Vanilla auto-encoder (ordinary AE):
 - Suppose we've trained an autoencoder model on a large dataset of faces with an encoding dimension of 6.
 - An ideal autoencoder will learn descriptive attributes of faces such as skin color, whether or not the person is wearing glasses, etc. in an attempt to describe an observation in some compressed representation (latent code).
 - In the example, we've described the input image in terms of its latent attributes using a single value to describe each attribute.

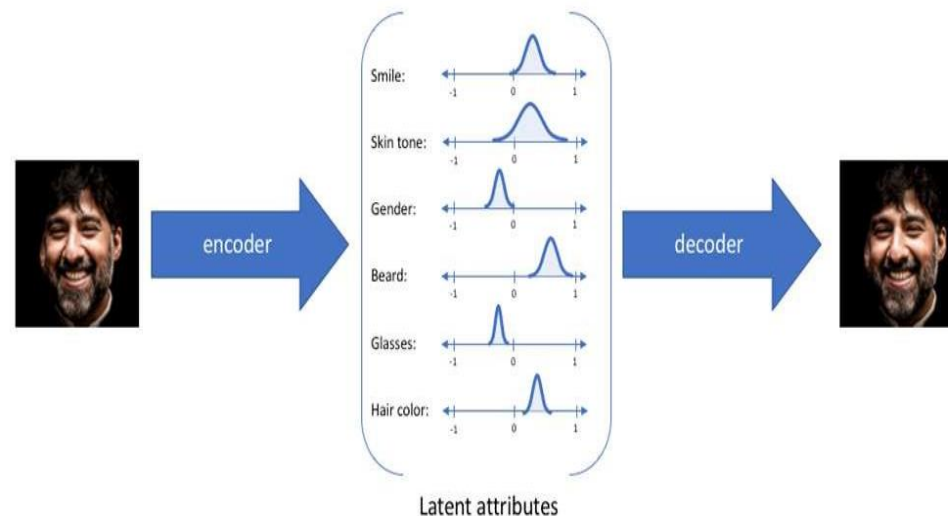


- What *single value* would you assign for the smile attribute if you feed in a photo of the Mona Lisa?



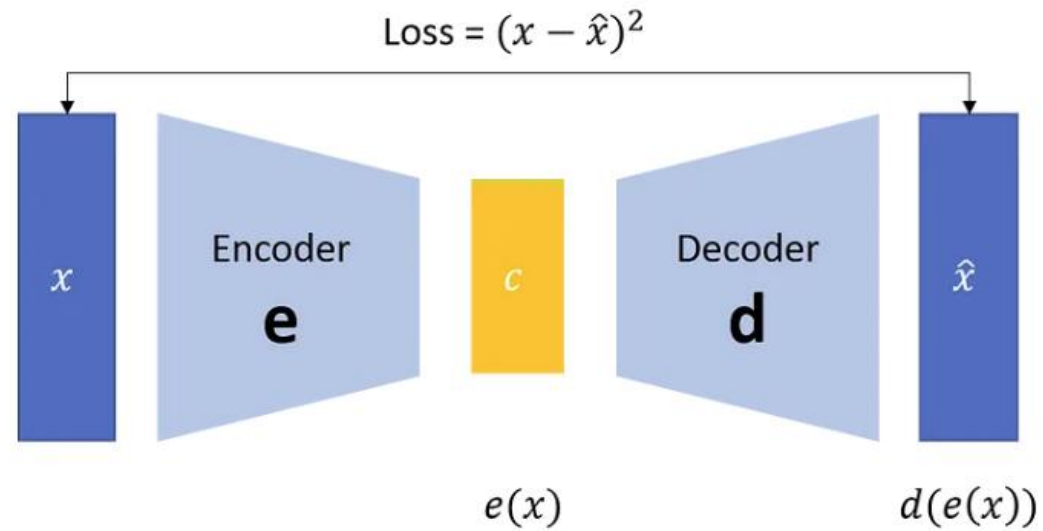
What VAE does:

- Variational Auto Encoder:
 - Represent *each latent attribute* for a given input as a probability distribution.
 - When decoding from the latent state, we randomly sample from each latent state distribution to generate a vector as input for our decoder model.

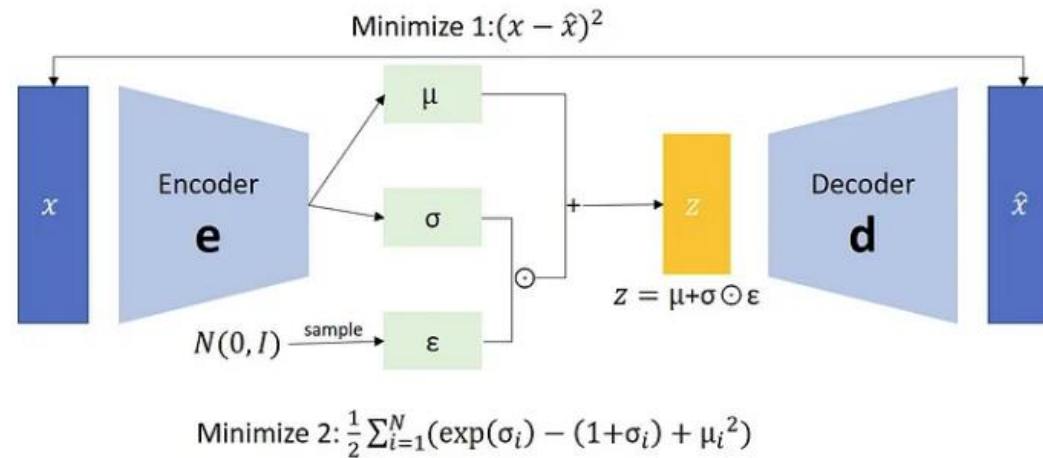


*Note: For variational autoencoders, the encoder model is sometimes referred to as the **recognition model** whereas the decoder model is sometimes referred to as the **generative model**.*

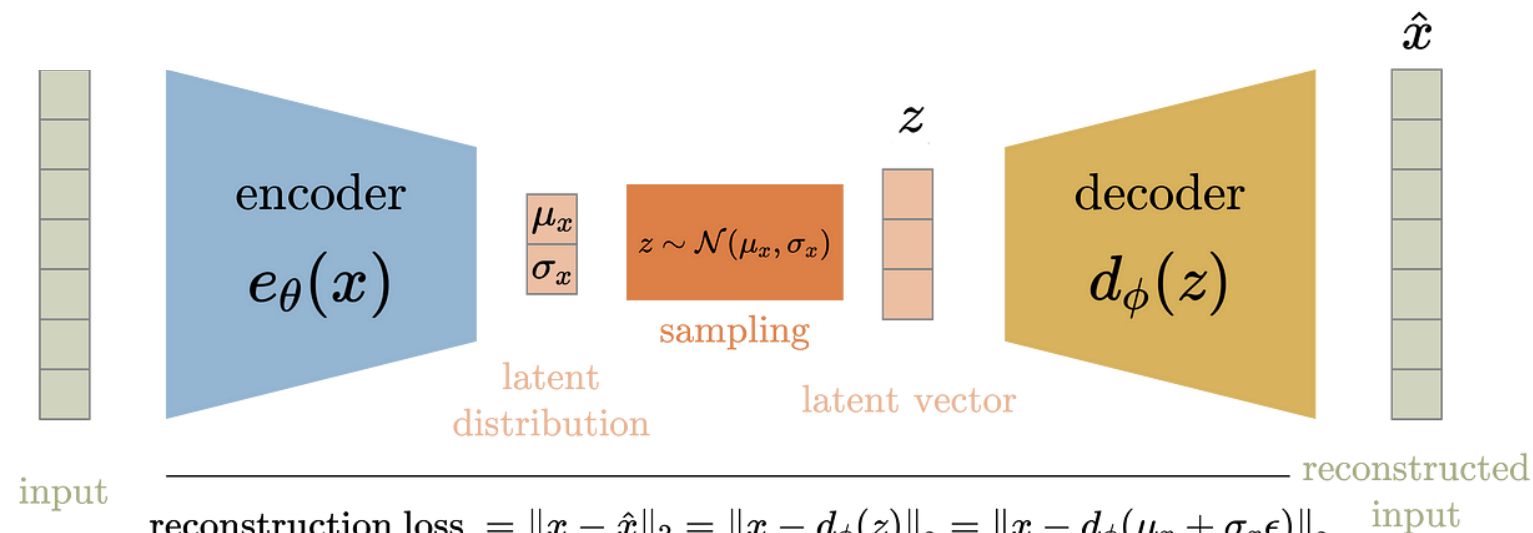
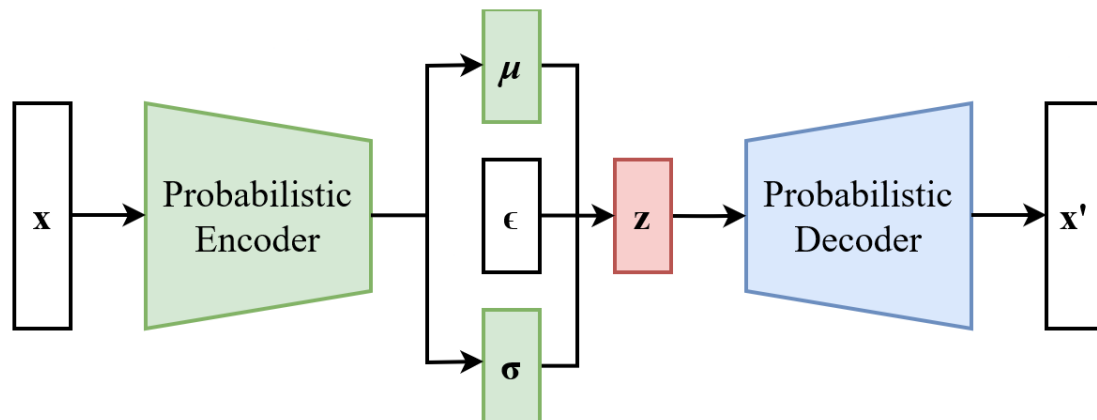
Autoencoder



Variational Autoencoder(VAE)



Architecture of VAE



$$\text{reconstruction loss} = \|x - \hat{x}\|_2 = \|x - d_\phi(z)\|_2 = \|x - d_\phi(\mu_x + \sigma_x \epsilon)\|_2$$

$$\mu_x, \sigma_x = e_\theta(x), \quad \epsilon \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$$

$$\text{similarity loss} = \text{KL Divergence} = D_{KL}(\mathcal{N}(\mu_x, \sigma_x) \parallel \mathcal{N}(\mathbf{0}, \mathbf{I}))$$

$$\text{loss} = \text{reconstruction loss} + \text{similarity loss}$$

1. Encoder:

- During training, the input data x is fed to the encoder.
- The encoder transforms the input data (like an image or a feature vector) into the **parameters of a probability distribution in a lower-dimensional latent space**.
 - Instead of compressing the input into a single point (as in a regular autoencoder), the VAE encoder maps it to a distribution, typically a Gaussian (Normal) distribution.
- The encoder produces two important outputs:
 1. **Mean vector (μ):**
This represents the *center* of the latent distribution for a given input.
 2. **Standard deviation vector (σ) (or log-variance $\log(\sigma^2)$):**
This determines the *spread* or *uncertainty* of the distribution

These define a **multivariate Gaussian distribution**:

$$q_{\phi}(z|x) = \mathcal{N}(z; \mu(x), \sigma^2(x))$$

Instead of outputting one latent vector, the encoder gives a **distribution** over latent variables z , which allows for **sampling** and **generative modeling**.

What is Encoder Network?

1 Input x is passed through a shared encoder network

This could be:

- A few convolutional layers (for images)
- Or fully connected layers (for tabular or vector data)

You get a **hidden representation**:

$$h = \text{Encoder}(x)$$

2 Split the final layer into two branches

- One branch predicts the **mean vector** μ
- One branch predicts the **log-variance** $\log(\sigma^2)$

So:

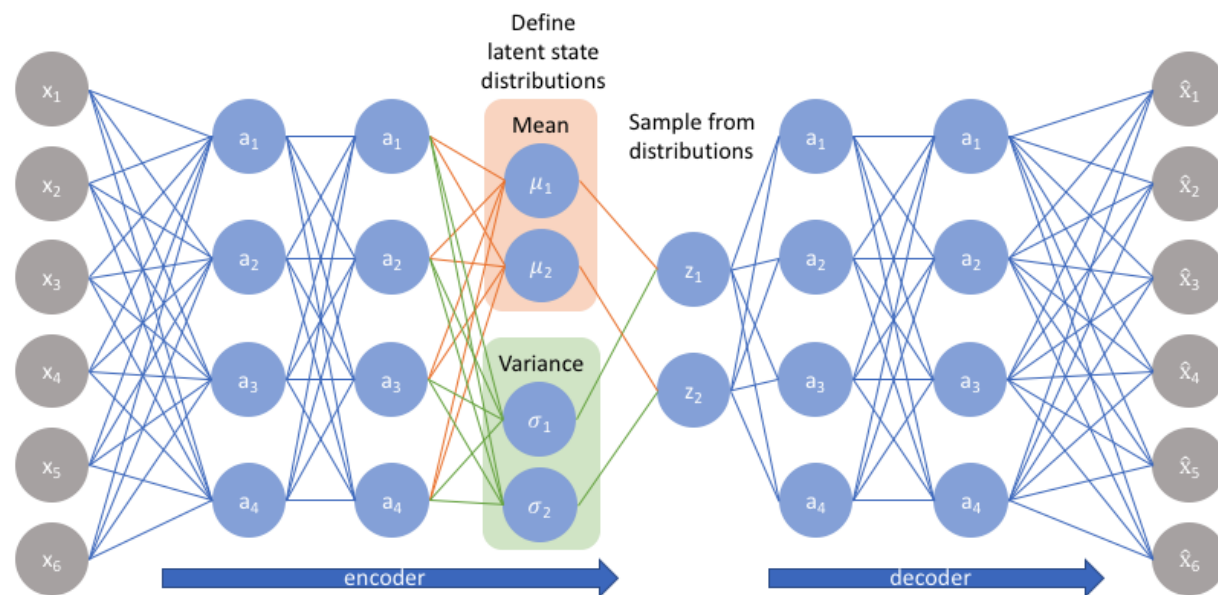
$$\begin{aligned}\mu &= W_{\mu}h + b_{\mu} \\ \log(\sigma^2) &= W_{\sigma}h + b_{\sigma}\end{aligned}$$

These are learned using separate linear layers (fully connected layers) from the shared hidden representation.

$$\sigma = \exp\left(\frac{1}{2} \cdot \log(\sigma^2)\right)$$

Note: We often predict **log-variance** instead of variance directly because:

- It helps with numerical stability
- Ensures the predicted variance is always positive when exponentiated

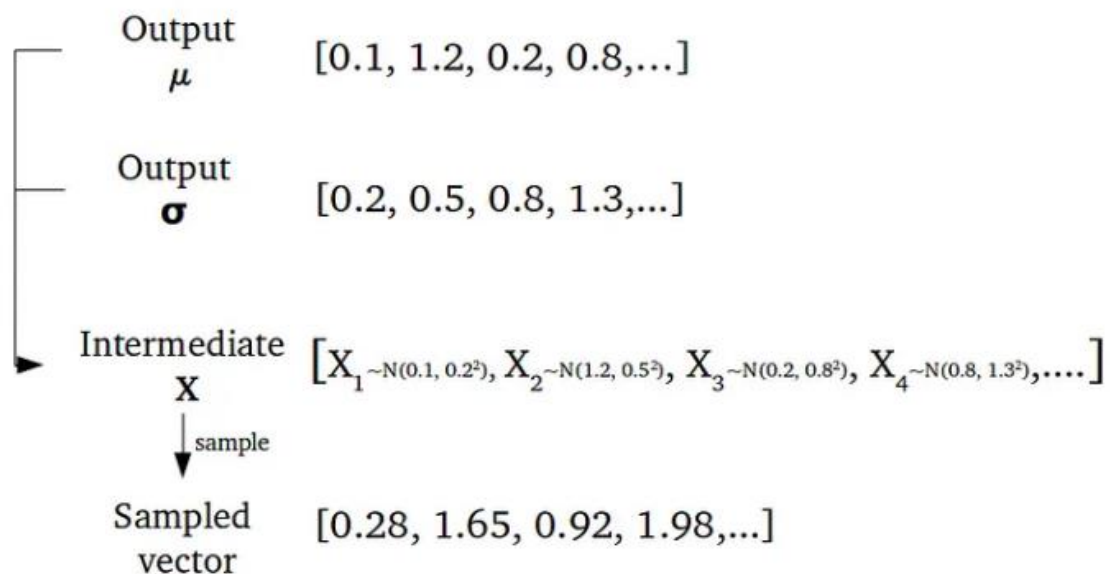


- Summary: What the encoder does?
 - Pass the sample x from the training data
 - Propagate this vector x through some NN (Feedforward MLP) and obtain some other vector x^{\sim}
 - Get the mean $\mu(x)$ and std $\sigma(x)$ from x^{\sim} from two more neural networks (maybe single layer)
 - The mean and std/variance vectors correspond to the **mean** and **variance** of a Gaussian distribution.
 - The dimensions of these two vectors ($\mu(x)$ and $\sigma(x)$) are a hyperparameter that can be specified by the user, known as the **latent dimension**.
- Note
 - It is important not to confuse this with the mean and variance of the original input image, X .
 - The encoder isn't calculating the mean or variance of the input; instead, it's defining a Gaussian space with these two vectors. For example, if the encoder produces a mean vector of $[1.5, 1.0, 2.0]$ and a variance vector of $[0.2, 0.4, 0.1]$, we can say the input has been transformed into a latent space with a mean of $[1.5, 1.0, 2.0]$ and variance of $[0.2, 0.4, 0.1]$.

• 2. Latent vector

- The latent vector is sampled using the mean and standard deviation.
- Generate random noise ϵ and get a point in the latent space $\mu(x) + \sigma(x)\epsilon$ (it is known as **reparametrization trick**)

$$z = \mu + \sigma \cdot \epsilon, \quad \epsilon \sim \mathcal{N}(0, 1) \quad \mathcal{N}(0, 1) \text{ means a Normal distribution with } \mu = 0 \text{ and } \sigma^2 = 1$$



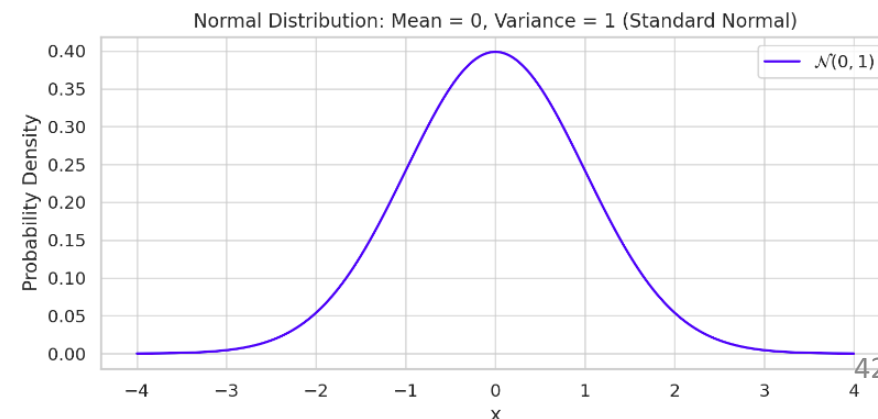
🔴 What is $\mathcal{N}(0, 1)$?

It's a **bell-shaped curve**:

- Mean: 0
- Standard deviation: 1
- Values can be **positive or negative**
- Most values fall in the range **-3 to +3**

So:

- ϵ could be **-0.5, 1.2, -2.3**, etc.
- It's **not limited to [0, 1]**

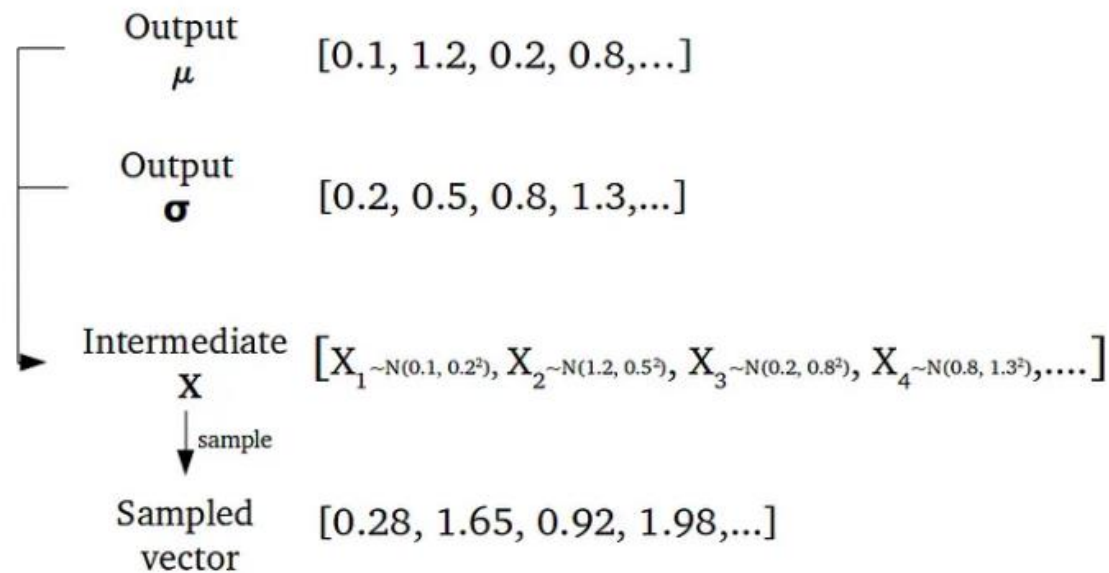


- By defining both the mean and variance, and then **sampling** from this latent Gaussian space, we introduce variability.
- Even with the same input image, while the mean and standard deviations remain the same, the encoder will produce different outputs (latent vector) for each sample, which, when fed into the decoder, will generate different images.
- This variability is what makes the model generative, as it introduces randomness in the output while keeping the mean and variance consistent.

Let us understand how this help:

- μ (mean): The average value expected for each dimension.
- σ (standard deviation): The spread or variability for each dimension.

These values (μ and σ) are used to define a set of normal distributions (one for each dimension). From each of these distributions, we can draw a random sample to get a final encoded vector. This process ensures that the latent space is continuous and smooth, allowing for easy sampling and interpolation.




How Mean & Variance is used to generate encoding vectors

As we can see, we have a “mean” encoding and a “standard deviation” encoding which is sampled to return final encoding i.e. [0.28,1.65,0.92,1.98...]. Now because we are sampling between μ and σ , at each run we generate a new final encoding which brings the “variation” into the picture. This means, that even for the same input, while the mean and standard deviations remain the same, the actual encoding will somewhat vary on every single pass simply due to sampling.

2.a.Reparameterization in VAEs

- Reparameterization is a **trick** that allows a Variational Autoencoder (VAE) to **sample random latent vectors z** from a distribution **in a way that is differentiable** — meaning it can be trained using backpropagation.

 Why is it needed?

During training, we want to:

- Sample $z \sim \mathcal{N}(\mu, \sigma^2)$
- Pass z to the decoder to reconstruct the input
- Minimize reconstruction loss and KL divergence

But sampling is **non-differentiable**, meaning gradients can't flow through it — and we can't train the network using gradient descent.

- Reparameterization

Instead of sampling z directly, we express the randomness separately:

$$z = \mu + \sigma \cdot \epsilon \quad \text{where} \quad \epsilon \sim \mathcal{N}(0, 1)$$

Here's what's happening:

- μ and σ are outputs from the encoder — they're deterministic
- ϵ is random noise sampled from a fixed standard normal distribution
- z now becomes a **deterministic function of μ, σ** and the random ϵ

💡 This clever trick allows the entire computation (including sampling) to be **differentiable**, so gradients can flow through μ and σ during training.

The **reparameterization trick** lets us **backpropagate through stochastic variables** by rewriting sampling as a **deterministic transformation** of random noise.

- 3. Decoder

- The decoder network takes z and tries to reconstruct the original input x .

$$\hat{x} = \text{Decoder}(z)$$

Where:

- $z = \mu + \sigma \cdot \epsilon$ (from the encoder using reparameterization)
- \hat{x} is the **reconstructed version** of the input

- **4. Loss computation (evidence lower bound (ELBO))**

- The model is trained using two main loss components:

- **(a) Reconstruction Loss**

- Measures how close \hat{x} is to the original input x (e.g., Mean Squared Error or Binary Cross-Entropy)

$$\mathcal{L}_{\text{recon}} = \text{Loss}(x, \hat{x})$$

- **(b) KL Divergence Loss**

Measures how close the latent distribution $q(z|x) = \mathcal{N}(\mu, \sigma^2)$ is to a standard normal $\mathcal{N}(0, 1)$

$$\mathcal{L}_{\text{KL}} = D_{\text{KL}}(q(z|x) || \mathcal{N}(0, 1))$$

- **Total VAE Loss**

$$\mathcal{L}_{\text{VAE}} = \mathcal{L}_{\text{recon}} + \beta \cdot \mathcal{L}_{\text{KL}}$$

- Where β is a weight (sometimes set to 1 or tuned for control)

- **5. Backpropagation & Training**

- The loss is minimized using gradient descent.
 - Gradients flow **through both encoder and decoder**, because of the reparameterization trick.
 - The encoder learns to map inputs to meaningful latent distributions
 - The decoder learns to generate or reconstruct data from sampled latent vectors
 - The KL loss keeps the latent space organized and smooth — great for sampling and generative tasks
- One of the key innovations of VAE is the use of **variational inference** to approximate the true posterior distribution of the latent variables. This is done by minimizing the **evidence lower bound (ELBO)**, which balances between reconstruction accuracy and the divergence between the approximate posterior and the prior distribution.

Use Cases of Auto-encoders:

- **Dimensionality Reduction:**
 - Autoencoders can be used as a non-linear alternative to techniques like PCA (Principal Component Analysis) to reduce the dimensionality of data. The encoder part maps the high-dimensional data into a lower-dimensional latent space.
- **Anomaly Detection:**
 - In anomaly detection, autoencoders can learn to reconstruct normal data patterns. When given new, anomalous data, the autoencoder will fail to reconstruct it well (i.e., the reconstruction error will be high), indicating an anomaly.
- **Image Denoising:**
 - Autoencoders can be used to remove noise from images. The denoising autoencoder learns to reconstruct clean images from noisy versions, making it useful in fields like medical imaging, surveillance, and more.
- **Data Generation:**
 - Variational autoencoders (VAEs) can be used to generate new samples similar to a given dataset. This is used in tasks like generating new images, text, or other types of data.
- **Image Compression:**
 - Autoencoders can be used to compress image data by encoding it into a smaller, latent space representation, which is useful in applications like image compression.

Appendix

- Difference between z in auto encoder and VAE

- 📦 **Autoencoder (AE)**

- Deterministic encoding
 - The encoder maps the input x **directly** to a single point in the latent space:

$$z = f_{\text{encoder}}(x)$$

- z is a **fixed** compressed representation of x
 - Used primarily for **dimensionality reduction** or **reconstruction**
 - No explicit notion of uncertainty or variability

- 🎲 **Variational Autoencoder (VAE)**

- Probabilistic encoding
 - The encoder maps input x to a **distribution** over the latent space — specifically, a Gaussian with mean μ and standard deviation σ :

$$q(z|x) = \mathcal{N}(z; \mu, \sigma^2)$$

- Then, z is **sampled** from this distribution using:

$$z = \mu + \sigma \cdot \epsilon, \quad \epsilon \sim \mathcal{N}(0, 1)$$

- z is **stochastic**, representing multiple plausible encodings
 - Enables **generative modeling**: new data can be created by sampling from the latent space

- [Variational autoencoders.](#)
- [Fundamental Generative AI Part 1: Variational Autoencoders \(VAE\) | by Inkyu Kim | Medium](#)

Why L1 Regularization Sparse

[L1 regularization and L2 regularization](#) are widely used in machine learning and deep learning. L1 regularization adds “absolute value of magnitude” of coefficients as penalty term while L2 regularization adds “squared magnitude” of coefficient as a penalty term.

- Although L1 and L2 can both be used as regularization term, the key difference between them is that L1 regularization tends to **shrink the penalty coefficient to zero** while L2 regularization would move coefficients **towards zero** but they will never reach. Thus L1 regularization is often used as a method of **feature extraction**. But why L1 regularization leads to sparsity?
- <https://medium.com/@syoya/what-happens-in-sparse-autencoder-b9a5a69da5c6>