

# EXPLORING VARIATIONAL AUTOENCODERS (VAES): A DEEP DIVE INTO LATENT REPRESENTATIONS AND DATA GENERATION

NAME: RUKAYAT ARAMIDE IBRAHIM

STUDENT NO: 2200481

Github Link: [https://github.com/Rukayat-spec/Variational\\_Autoencoder](https://github.com/Rukayat-spec/Variational_Autoencoder)

## INTRODUCTION

One of the most potent and adaptable tools in generative modelling are variational autoencoders (VAEs), which combine neural network designs with probabilistic techniques. VAEs are examined in this tutorial, which covers their applications, mathematical underpinnings, architecture, and implementation. You will get an understanding of VAEs and practical experience with their implementation at the conclusion, enabling you to apply them in your own work.

## 1. WHAT IS AN AUTOENCODER?

An artificial neural network type called an autoencoder is made to develop effective representations of input data, often known as latent representations. By teaching the network to reconstruct the input data from its compressed, encoded form, it accomplishes this. Since autoencoders learn directly from the data by minimising reconstruction error, they are regarded as unsupervised learning models because they do not require labelled input (Bergmann et al., 2024).

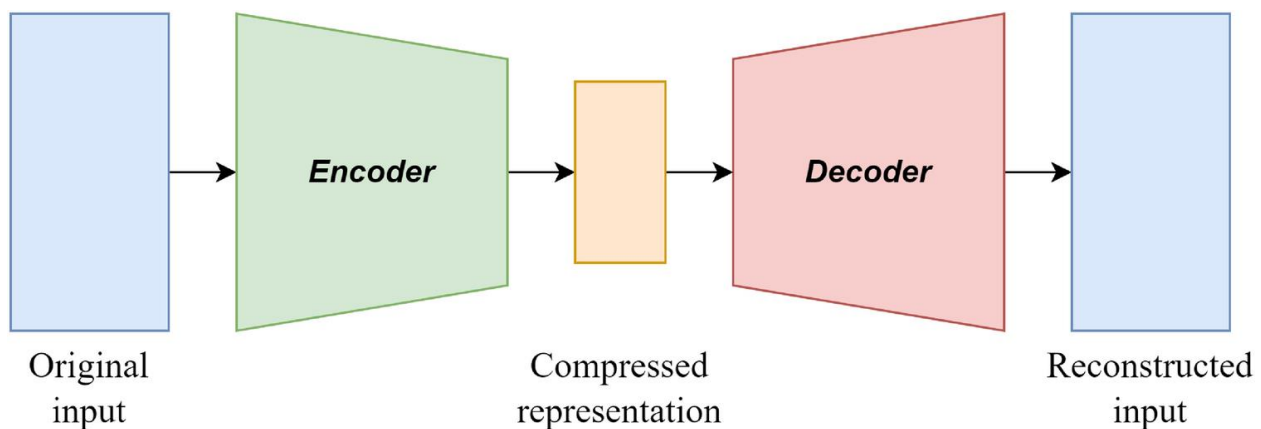


Fig 1: A pictorial representation of an autoencoder network model (Wang et al., 2023).

### 1.1 STRUCTURE OF AN AUTOENCODER

An autoencoder consists of two main parts:

### Encoder:

- The encoder compresses the input data into a compact, lower-dimensional latent representation.
- This process reduces the data dimensionality and captures its most important features, essentially learning a compressed summary of the input.
- Mathematically, the encoder maps the input  $x$  to the latent representation  $z$ ,  $z=f(x)$ , where  $f$  is a function defined by the encoder's neural network layers.

### Decoder:

- The decoder takes the latent representation  $z$  and reconstructs it back to the original data space.
- It attempts to regenerate an output  $x^\wedge$  that closely resembles the input  $x$ .
- Mathematically, the decoder maps  $z$  to  $x^\wedge$ ,  $x^\wedge=g(z)$ , where  $g$  is the function defined by the decoder's neural network layers.

Together, the encoder and decoder form a symmetric architecture that learns how to compress and reconstruct the data.

## 1.2 Applications of Autoencoders

- **Data Compression:** For effective storage and transmission, compress high-dimensional data like pictures or videos into lower-dimensional latent spaces.
- **Noise reduction:** By learning from both clean and noisy input versions, denoising autoencoders purify data.
- **Anomaly detection:** Use large reconstruction errors to find abnormalities, which is helpful for medical diagnostics and fraud detection.
- **Feature Extraction:** Capture key data characteristics in latent space for use in other machine learning models.

## 1.3 Limitations of Standard Autoencoders

- **Deterministic Nature:** For given inputs, standard autoencoders generate fixed, predictable latent representations. For identical inputs, the hidden representation remains constant after training. This restricts their capacity to produce a range of outputs or investigate data variations.
- **Generative Modeling:** Conventional autoencoders are not generative by nature; they are unable to produce novel, significant examples that closely resemble the training set.

Variational Autoencoders (VAEs), which incorporate a probabilistic framework into the architecture, were made possible by these constraints. Better generative modelling and seamless

data space exploration are made possible by VAEs, which learn a distribution in the latent space as opposed to fixed points (Choudhary, 2024).

## 2. INTRODUCTION TO VARIATIONAL AUTOENCODERS (VAES)

A probabilistic framework for latent space representation is incorporated into Variational Autoencoders (VAEs), which are an extension of ordinary autoencoders. VAEs translate inputs into probability distributions in the latent space as opposed to typical autoencoders, which encode inputs into fixed latent points. Multivariate Gaussian distributions with a mean vector ( $\mu$ ) and a standard deviation vector ( $\sigma$ ) are commonly used to parameterise these distributions. VAEs are effective tools for generative modelling because of their probabilistic methodology, which allows them to sample from these distributions.

### 2.1 Why Use VAEs?

- **Generative Modeling:** New data points that mirror the training data can be produced using VAEs. For instance, a VAE may produce realistic new digit images given a dataset of handwritten digits.
- **Probabilistic Interpretation:** VAEs provide insights into uncertainty and underlying patterns by capturing the inherent variability in the data through the encoding of data into distributions.
- **Smooth Latent Space:** Smooth transitions between generated samples are made possible by the continuous and structured latent space in VAEs. For uses like interpolating between two data points or visual morphing, this feature is essential.

Because VAEs' probabilistic nature makes both representation learning and data generation easier, they are frequently utilised in applications including picture synthesis, text generation, and anomaly detection.

## 3. KEY COMPONENTS OF VAES

A VAE comprises three primary components: an **encoder**, a **decoder**, and a **latent space**.

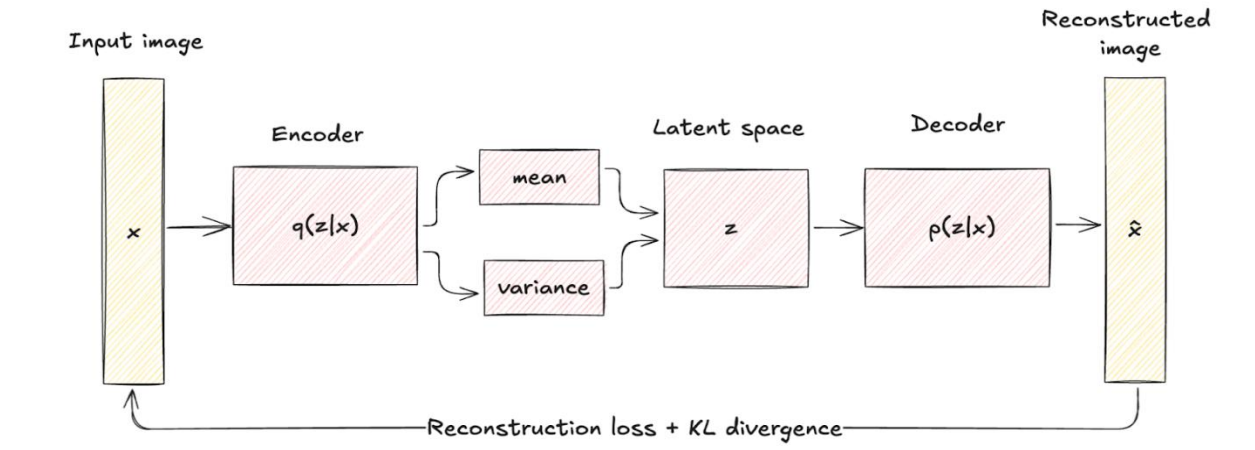


Fig 2: Image of a Variational Autoencoder architecture (Kurtis Pykes, 2024).

## Encoder

A probability distribution (such as Gaussian) defines the latent representation to which the encoder converts input data. Two parameters are output by it:

- **Mean ( $\mu$ ):** The center of the distribution.
- **Variance ( $\sigma^2$ ):** The spread of the distribution.

## Latent Space

A condensed, probabilistic representation of the input data is called the latent space. Each input is represented by a distribution rather than a fixed number, allowing for sampling and seamless data point transitions.

## Decoder

Data from the latent space is reconstructed by the decoder. It takes a latent variable that has been sampled and produces output that is comparable to the initial input.

# 4. MATHEMATICAL FRAMEWORK

## 4.1 Latent Variables and Probabilistic Modeling

The VAE assumes that data ( $x$ ) is generated from latent variables ( $z$ ) via a likelihood distribution  $p(x/z)$ . The joint probability distribution is:

$$p(x, z) = p(x/z) p(z)$$

The goal is to learn the posterior distribution  $p(z/x)$ , which is computationally intractable. Instead, VAEs approximate it with  $q(z/x)$ , a learned distribution.

## 4.2 Evidence Lower Bound (ELBO)

To train a VAE, the objective is to maximize the **Evidence Lower Bound (ELBO)**:

$$ELBO = E_{q(z/x)}[\log(p(x/z))] - KL(q(z/x) // p(z))$$

- The first term ensures accurate reconstruction.
- The second term (Kullback-Leibler divergence) regularizes the latent space by aligning  $q(z/x)$  with a prior distribution (usually a standard normal distribution).

## 4.3 KL Divergence

The KL divergence quantifies the difference between two distributions:

$$KL(q(z/x) // p(z)) = E_{q(z/x)}[\log(q(z/x)) - \log(p(z))]$$

This term enforces smoothness and continuity in the latent space.

## 5. APPLICATIONS OF VAES

Variational Autoencoders (VAEs) provide versatility in a variety of applications and are particularly good at generative modelling and representation learning.

- **Image Generation:** Create realistic images for game design, art, and style transfer.
- **Anomaly Detection:** Identify deviations in cybersecurity, quality control, and fraud detection.
- **Drug Discovery:** Design novel molecules for accelerated research.
- **Text Generation:** Generate coherent text for translation, content creation, and chatbots.

VAEs improve efficiency across a range of domains, spur creativity, and resolve challenging issues by learning data distributions.

## 6. IMPLEMENTATION OF VAES WITH PYTHON

Let's build a VAE using Python and PyTorch with Stanford Dogs Dataset.

### Step 1. Importing Necessary Libraries

PyTorch and torchvision, which are necessary libraries for creating models and processing images, are imported. Additionally, PIL is used for managing images, and libraries like NumPy,

Pandas, and Matplotlib are used for handling and visualising data. During training, tqdm is used to track progress.

```
[1] import numpy as np
import pandas as pd
import os
import matplotlib.pyplot as plt
import torch
from torch import nn, optim
import torch.nn.functional as F
from torchvision import datasets, transforms
from torch.utils.data import Dataset, DataLoader
from torch.autograd import Variable
from PIL import Image
from tqdm import tqdm_notebook as tqdm

# Set batch size and device
batch_size = 32
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

## Step 2. Creating Class to Handle Image Loading and Transformations

The photos are loaded and preprocessed using the Load Dataset class. An image directory and optional transformations are included in the class's initialisation. Two transformations are applied to the images when they are loaded from the directory: one for data augmentation and normalisation, and another for resizing and cropping.

- transform1 applies a centre crop and resizes the photos to 64x64 pixels.
- Normalisation, random rotations, and random horizontal flipping are all included in transform2.

This ensures the images are ready for feeding into the neural network.



```
# Dataset class to handle image loading and transformations
class Load_Dataset(Dataset):
    def __init__(self, img_dir, transform1=None, transform2=None):
        self.img_dir = img_dir
        self.img_names = os.listdir(img_dir)
        self.transform1 = transform1
        self.transform2 = transform2

        self.imgs = []
        for img_name in self.img_names:
            img = Image.open(os.path.join(img_dir, img_name))
            if self.transform1 is not None:
                img = self.transform1(img)
            self.imgs.append(img)

    def __getitem__(self, index):
        img = self.imgs[index]
        if self.transform2 is not None:
            img = self.transform2(img)
        return img

    def __len__(self):
        return len(self.imgs)
```

### Step 3. Resizing and Center Cropping the Images

After scaling the photos to a set 64x64 pixel size, centre cropping is the next step in the preparation process. This guarantees consistency in the dimensions of the input images, which is essential for deep learning models.



```
# First preprocessing of data: resize and center crop images
transform1 = transforms.Compose([transforms.Resize(64), transforms.CenterCrop(64)])

# Data augmentation and conversion to tensors
random_transforms = [transforms.RandomRotation(degrees=10)]
transform2 = transforms.Compose([
    transforms.RandomHorizontalFlip(p=0.5),
    transforms.RandomApply(random_transforms, p=0.3),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])
```

### Step 4. Creating Dataset and DataLoader

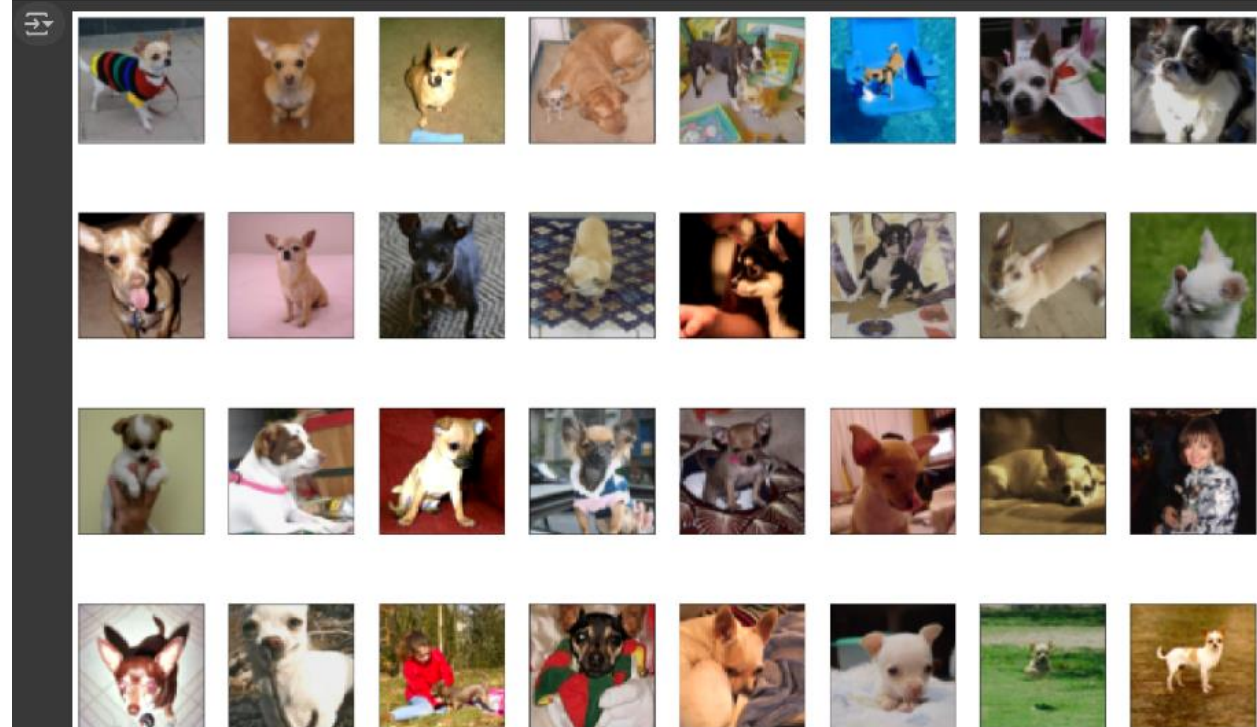
The `Load_Dataset` class is used to construct the dataset, and a `DataLoader` is instantiated to manage data parallel loading, batching, and shuffling. To ensure effective parallelisation during training, the batch size is set to 32 and the number of workers for data loading is set to 4.

```
[4] # Create dataset and DataLoader
train_dataset = Load_Dataset(
    img_dir='/content/drive/MyDrive/Dogs', # Modify with correct path to your data
    transform1=transform1,
    transform2=transform2
)
train_loader = DataLoader(dataset=train_dataset, batch_size=batch_size, shuffle=True, num_workers=4)
```

## Step 5. Visualizing Some Images

After preprocessing, a collection of the dataset's photos is visualised to assess their appearance and quality. After converting the photos to a NumPy array and normalising the pixel values, the photographs are shown.

```
# Visualize some images
x = next(iter(train_loader))
fig = plt.figure(figsize=(25, 16))
for ii, img in enumerate(x):
    ax = fig.add_subplot(4, 8, ii + 1, xticks=[], yticks=[])
    img = img.numpy().transpose(1, 2, 0)
    plt.imshow((img+1.)/2.)
```





## Step 6. Defining the Variational Autoencoder (VAE)

The Variational Autoencoder architecture is implemented by the VAE class, which is specified as follows:

- Encoder: LeakyReLU activation and batch normalisation are features of several convolutional layers that make up the encoder. For the latent space, the encoder produces two values: logvar (log variance) and mu (mean).
- Decoder: To rebuild the pictures from the latent space, the decoder is constructed using transposed convolutional layers and ReLU activation.
- Reparameterisation: To enable backpropagation, the reparameterization technique is used to sample the latent vector during training.
- Forward Pass: To produce reconstructed images, the forward pass entails encoding the input, sampling from the latent space, and decoding.
- Loss Function: The loss function uses Kullback-Leibler Divergence (KLD) to regularise the latent space and binary cross-entropy (BCE) to estimate the reconstruction error.

```
import torch
from torch import nn
import torch.nn.functional as F
import torchvision.models as models
from torchvision.models import VGG16_Weights

class ResBlock(nn.Module):
    def __init__(self, channels):
        super(ResBlock, self).__init__()
        self.conv1 = nn.Conv2d(channels, channels, 3, padding=1)
        self.bn1 = nn.BatchNorm2d(channels)
        self.conv2 = nn.Conv2d(channels, channels, 3, padding=1)
        self.bn2 = nn.BatchNorm2d(channels)

    def forward(self, x):
        residual = x
        x = F.relu(self.bn1(self.conv1(x)))
        x = self.bn2(self.conv2(x))
        x += residual
        x = F.relu(x)
        return x

class SharpVAE(nn.Module):
    def __init__(self, latent_dim=256, no_of_sample=10, batch_size=32, channels=3):
        super(SharpVAE, self).__init__()
        self.no_of_sample = no_of_sample
        self.batch_size = batch_size
        self.channels = channels
        self.latent_dim = latent_dim
        self.current_epoch = 0

        # Enhanced Encoder
        self.enc1 = nn.Sequential(
            nn.Conv2d(channels, 64, 3, padding=1)
```

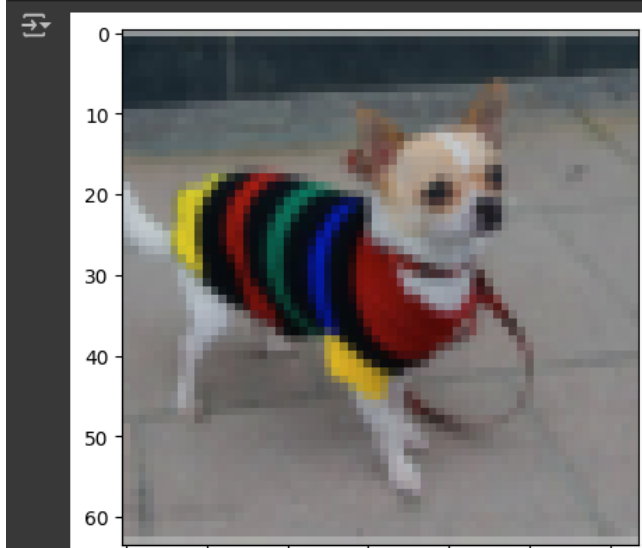
## Step 7. Setting Learning Rate, Epochs, and Latent Dimension

Thirty-two is the latent dimension, fifty is the number of epochs, and 0.001 is the learning rate. Adam, a popular optimiser for deep learning model training because of its effectiveness, is utilised.

```
[8] # Hyperparameters
    lr = 0.0005
    epochs = 10
    latent_dim = 256

    # Initialize model and optimizer
    model = SharpVAE(latent_dim=latent_dim, batch_size=32).to(device)
    optimizer = optim.AdamW(model.parameters(), lr=lr, weight_decay=1e-5)
    scheduler = optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=epochs, eta_min=1e-6)

    # Visualize a sample image
    plt.imshow((x[0].numpy().transpose(1, 2, 0) + 1) / 2.)
    plt.show()
```



## Step 8. Creating the Training Loop

For the designated number of epochs, the training loop iterates across the dataset. The data is sent through the network in each epoch while the model is in training mode. Based on the loss calculated by the loss function, the optimiser modifies the model's weights. Following each epoch, a sample image is used to assess the model, and the rebuilt image is shown.

```
import matplotlib.pyplot as plt

# List to store the training losses for plotting
train_losses = []

# Training loop
for epoch in range(epochs):
    model.current_epoch = epoch
    epoch_loss = 0.0 # Initialize epoch loss to zero for each epoch

    for batch_idx, data in enumerate(train_loader):
        data = data.to(device)
        optimizer.zero_grad()

        # Forward pass
        recon_batch, mu, logvar = model(data)

        # Compute loss
        loss = model.loss_function(recon_batch, data, mu, logvar)
        loss.backward()

        # Gradient clipping
        torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)

        # Optimizer step
        optimizer.step()

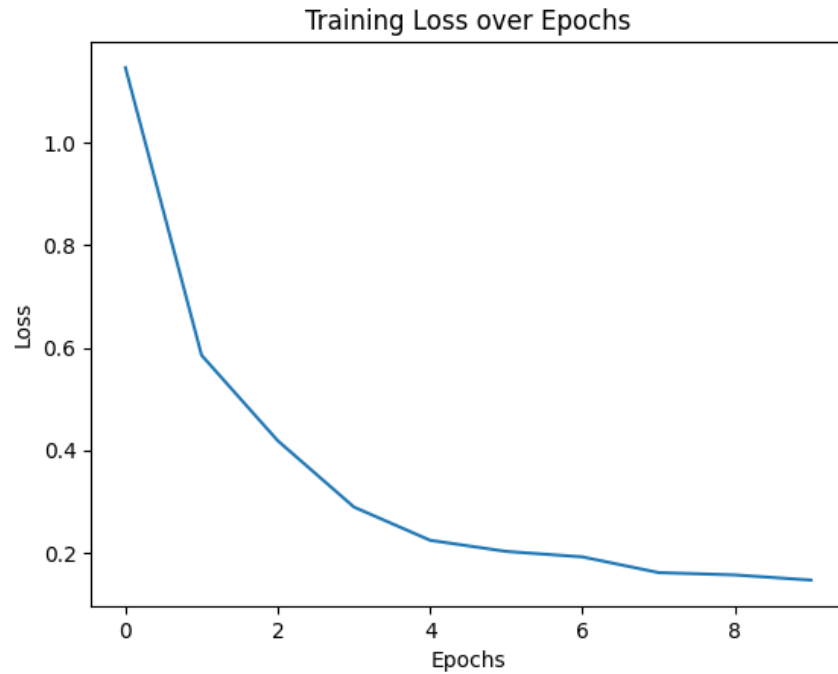
        # Accumulate loss for the epoch
        epoch_loss += loss.item()

    # Print loss and show visualization at specific intervals (e.g., every 100 batches)
    if batch_idx % 100 == 0:
        print(f'Epoch {epoch}, Batch {batch_idx}, Loss {loss.item():.4f}')

        # Visualization: Show original and reconstructed images
        with torch.no_grad():
            model.eval()
            plt.figure(figsize=(10, 5))
```

## Step 9. Loss Evaluation Report

This section focuses on visualizing the training loss over multiple epochs to assess the model's learning progress and convergence.

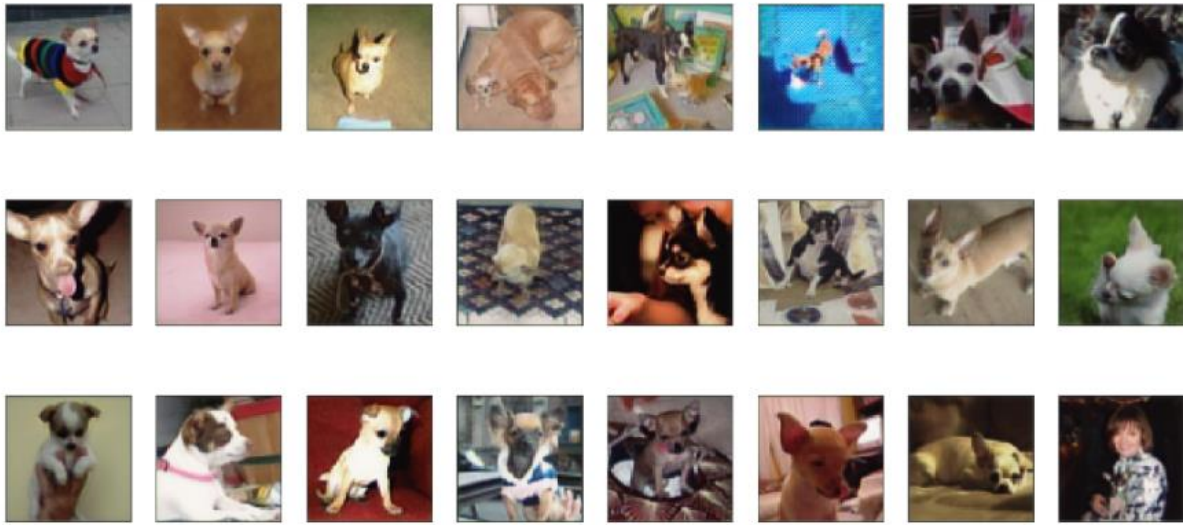


### Step 10. Generating and Visualizing Reconstructed Images

Following training, reconstructed images from the latent space are produced and shown by the model. To assess the quality of the reconstructions, the photos are displayed side by side with the originals. This procedure shows how effectively a compressed latent representation can be used to teach the model to produce realistic images.

```
# Generate and visualize reconstructed images
reconstructed, mu, _ = model(x.to(device))
reconstructed = reconstructed.view(-1, 3, 64, 64).detach().cpu().numpy().transpose(0, 2, 3, 1)

fig = plt.figure(figsize=(25, 16))
for ii, img in enumerate(reconstructed):
    ax = fig.add_subplot(4, 8, ii + 1, xticks=[], yticks=[])
    plt.imshow((img + 1.) / 2.)
```



## Step 11. Reconstructed images and Interpolation

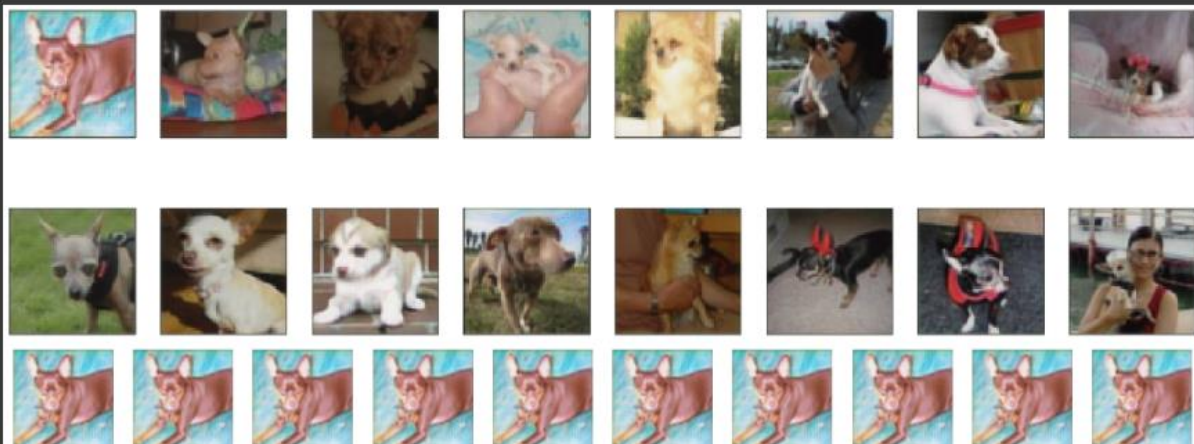
By running input data through the model, reconstructed images are produced, which are then displayed to evaluate the quality of the reconstruction. The latent representations of two images are combined linearly to produce latent space interpolation; the decoded results demonstrate smooth transitions. The model shows successful interpolation and reconstruction, indicating that it has picked up useful features.

```
z1 = mu[first_dog_idx].unsqueeze(0) # Latent representation of
z2 = mu[second_dog_idx].unsqueeze(0) # Latent representation of

# Interpolate between the latent representations
interpolation = []
for alpha in np.linspace(0, 1, num_interpolation_steps):
    z_interp = (1 - alpha) * z1 + alpha * z2
    # Decode the interpolated latent representation
    interp_img = model.decode(z_interp, model.encode(x_batch[0].unsqueeze(0))[2]).cpu().numpy().transpose(1, 2, 0)
    interpolation.append(interp_img)

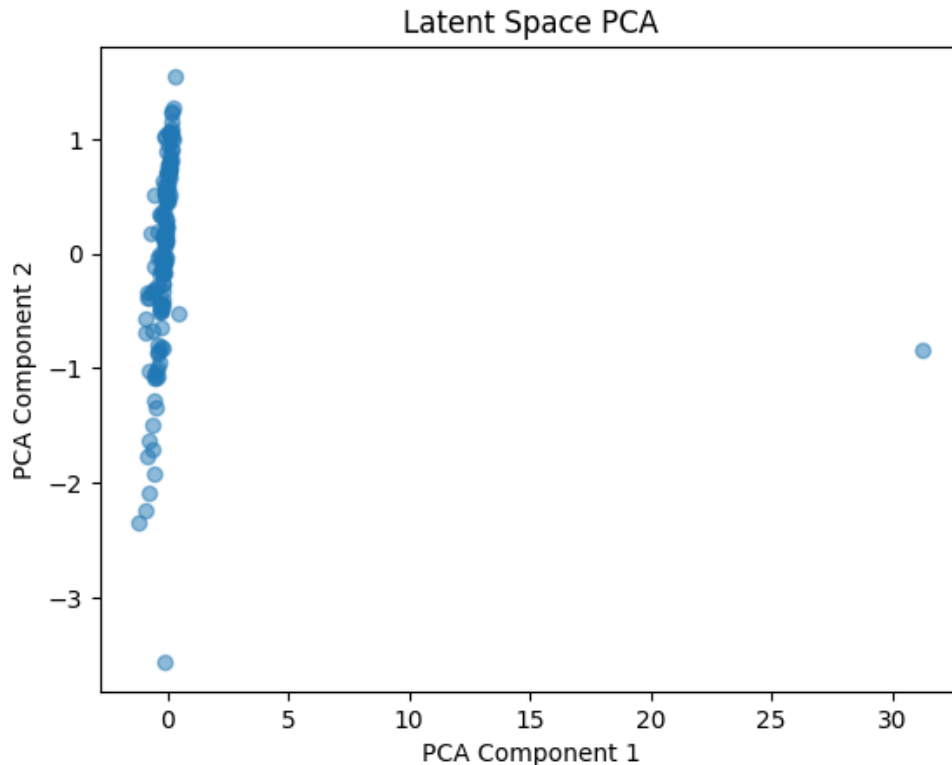
# Visualize interpolation
fig, axes = plt.subplots(1, num_interpolation_steps, figsize=(15, 5))
for i, ax in enumerate(axes):
    ax.imshow((interpolation[i][0] + 1.) / 2.)
    ax.axis('off')
plt.show()

evaluate_model()
```



## Step 12. Latent Space Visualization

This section explains how Principal Component Analysis (PCA) is used to visualize the latent space of a VAE. The visualization shows that the VAE effectively compresses input data into a structured, lower-dimensional representation. Tight clustering of data points indicates shared underlying features, suggesting a smooth latent space suitable for realistic reconstructions and interpolation. This structured organization highlights the VAE's ability to encode meaningful patterns and its potential for generalization and data synthesis.



## 7. CHALLENGES AND FUTURE DIRECTIONS

Variational Autoencoders (VAEs) must frequently be fine-tuned to balance reconstruction loss and KL divergence, and may can yield less lucid results than GANs. VAE-GANs and other hybrid techniques seek to improve output diversity and quality. Domain-specific designs have the potential to increase the applications of VAEs by enhancing 3D modelling, audio synthesis, virtual reality, and robotics.

## 8. CONCLUSION

This tutorial explored VAEs as a powerful tool for generative modeling and latent representation learning. By examining their architecture, mathematical foundations, and practical applications, we demonstrated how VAEs compress data, reconstruct images, and generate new samples. Practical examples, including latent space visualization and interpolation, highlighted their ability to capture meaningful patterns.

With their versatility in tasks like image synthesis, anomaly detection, and data augmentation, VAEs hold great potential despite challenges like balancing reconstruction and KL divergence. Ongoing advancements, such as VAE-GANs, promise to enhance their capabilities further. Equipped with these insights, readers can leverage VAEs to drive innovation across diverse fields.

## REFERENCES

- Bergmann, D., & Stryker, C. (2024). Autoencoder. IBM. Available at: <https://www.ibm.com/topics/autoencoder> (Accessed: 31, November 2024).
- Choudhary, A. S. (2024). An Overview of Variational Autoencoders (VAEs). Analytics Vidhya. Available at: <https://www.analyticsvidhya.com/blog/2023/07/an-overview-of-variational-autoencoders/> (Accessed: 28, November 2024).
- Jatasra, S. (2023). Unravelling the Mysteries of Variational Autoencoders: A Deep Dive into Intelligent Data Generation. Available at: <https://www.linkedin.com/pulse/unravelling-mysteries-variational-autoencoders-deep-dive-jatasra> (Accessed: 25, November 2024).
- Kurtis P. (2024). Variational Autoencoders: How They Work and Why They Matter. Available at: <https://www.datacamp.com/tutorial/variational-autoencoders> (Accessed: 4, December 2024).
- Olaoye, G., Luz, A. and Charles, E., (2024). "Variational autoencoders (VaEs) for synthetic data generation"  
[https://www.researchgate.net/publication/385592671\\_Variational\\_autoencoders\\_vaes\\_for\\_synthetic\\_data\\_generation](https://www.researchgate.net/publication/385592671_Variational_autoencoders_vaes_for_synthetic_data_generation).
- Wang, Q., Qin, K., Lu, B. *et al.*, (2023). "Time-feature attention-based convolutional auto-encoder for flight feature extraction". *Sci Rep* **13**, 14175. <https://doi.org/10.1038/s41598-023-41295-y>.