Politecnico di Milano

Project Report

055697 - NUMERICAL ANALYSIS FOR MACHINE LEARNING

# From Autoencoders to GANs, and Diffusion Models in Machine Learning

*Author:*
Filip Fabris
filip.fabris@mail.polimi.it

*Supervisor:*
Dr. Edie Miglio
filip.fabris@mail.polimi.it

March 4, 2024

# 1 Introduction

Today deep learning is one of most active research fields in domain of machine learning. In this project we will examine three main concepts: Autoencoders, generative adversarial networks (GANs) and diffusion models. Goal of this project is to understand the mechanisms behind each concept, clarify their core principles and finally unravel the interconnections that exists between them. To achieve this goal, we will first explain what Autoencoders are, then we will explain how GAN was invented from Autoencoders, and finally we will explain the rise of Defussion models

# 2 Autoencoders

Firstly lets just briefly explain what Autoencoders are. Autoencoders are a class of neural networks that are used in unsupervised learning tasks to learn compressed and meaningful representations of input data. They are designed to learn compressed and meaningful representations of input data.

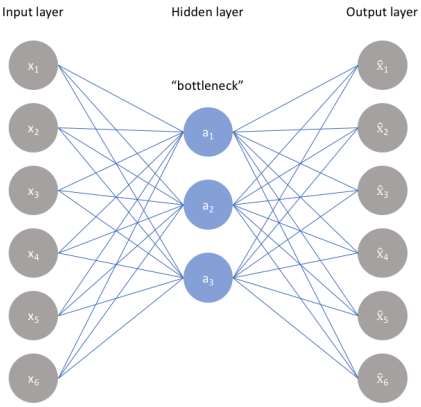


Figure 1: Autoencoder architecture with a bottleneck component

Traditionally, autoencoders were used for dimensionality reduction, where the high-dimensional data can be represented in the low dimensional space, something like PCA. But the main difference is that PCAs were limited by their linearity and they couldn‘t represent data with a high-dimensional non-linear manifold into a low dimensional space.

Autoencoders can do that thanks to neural networks. This is why the autoencoder and its variants are used in a lot of applications which we will introduce in following sections.

## 2.1 For what are Autoencoders used?

Autoencoders find diverse applications in various domains. In this section, we explore some prominent use cases.

### 2.1.1 Dimensionality Reduction

The reconstructed image is the same as our input but with reduced dimensions. It helps in providing the similar image with a reduced pixel value.


Figure 2: Dimensionality reduction using autoencoders

### 2.1.2 Data Denoising

Another application of auto encoders is data denoising.

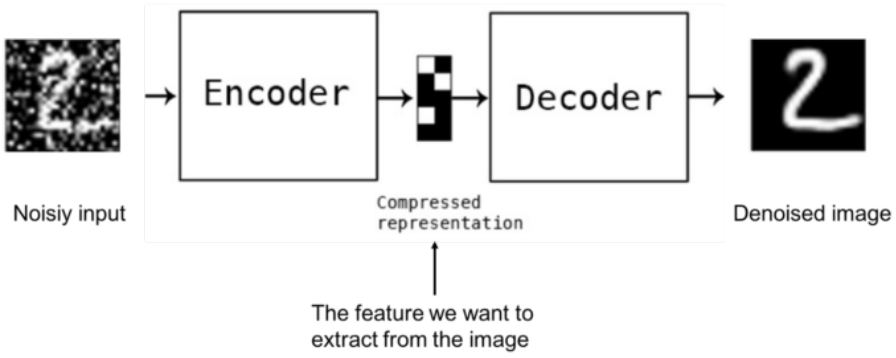


Figure 3: Data denoising using autoencoders

### 2.1.3 Feature variation

It extracts only the required features of an image and generates the output by removing any noise or unnecessary interruption.

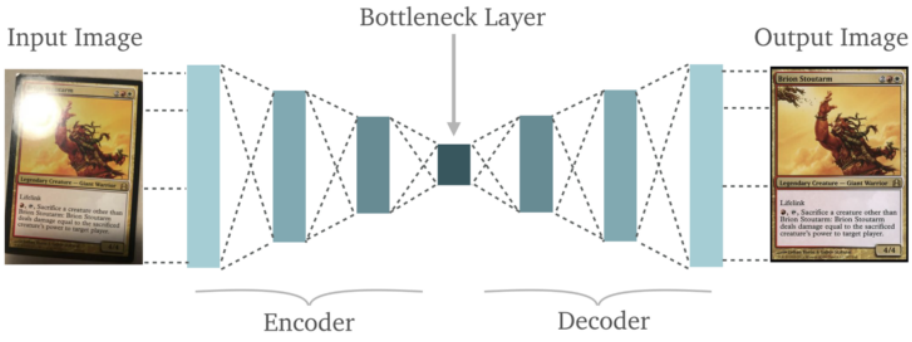


Figure 4: Feature variation using autoencoders

### 2.1.4 Other applications

Autoencoders are also used in other applications such as image compression, image generation, anomaly detection, image augmentation, and more.

## 2.2 Arhiteture of Autoencoders

Before we dive into the working of autoencoders, let us first introduce main parts that are used in autoencoders such as encoder, decoder, and bottleneck.

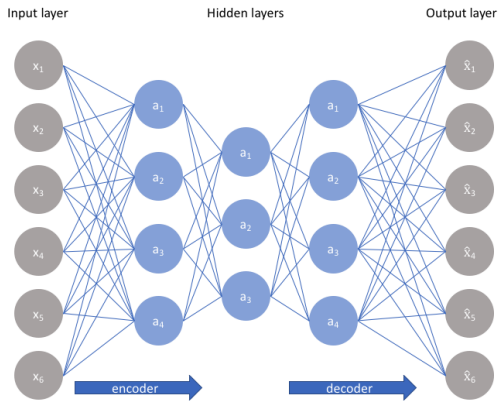


Figure 5: Autoencoder architecture with encoder and decoder components

### 2.2.1 Encoder

Encoder is a neural network that takes the input data and maps it to a lower dimensional representation, often referred to as the latent space. Mathematically, this can be represented as:

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

where $x$ is the input data, $h$ is the encoded representation, and $f_{\text{encoder}}$ is the encoding function.

### 2.2.2 Bottleneck

Bottleneck represent middle layer of the network. It is a layer that has a smaller number of neurons compared to both encoder and decoder. This forces the network to reduce the information such that the noise is reduced, and they could only approximate the original data rather than copy it end-to-end.

In essence bottleneck is a compression algorithm that compresses the input data into a lower dimensional representation.

It produces latent variables or so called latent representations introduced by Information Bottleneck method[1]. This random variables are extracted from the distribution and they give us abstract knowledge of the topology and the distribution of the data.

The latent variables in above formula are denoted as $h$

### 2.2.3 Decoder

Decoder is a neural network that reconstructs the input data from the encoded representation. It aims to generate an output that is as close as possible to the original input. Mathematically, the decoding function is represented as:

$$x' = f_{\text{decoder}}(h)$$

where $x'$ is the reconstructed data, $h$ is the encoded representation, and $f_{\text{decoder}}$ is the decoding function.

[1]introduced in 1999 by Naftali Tishby, Fernando C. Pereira, and William Bialek which hypothesis that it can extract vital information or representation by compressing the amount of information that can traverse through the network. This information is known as latent variables or latent representations.

## 2.3 Training Autoencoders

To train a basic autoencoder, the network is optimized to minimize the difference between the input data and the reconstructed output. This is often done by minimizing a loss function, such as mean squared error (MSE), between the input and output:

The loss function $\mathcal{L}$ is defined as follows:

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

where $N$ is the number of training samples,
$x_i$ is the input data,
$x'_i$ is the corresponding reconstructed output.

## 2.4 Types of Autoencoders

The basic autoencoder architecture we discussed is known as the vanilla autoencoder. However, various modifications and extensions have been proposed, including:

- **Denoising Autoencoder:** Trained to remove noise from input data.
- **Sparse Autoencoder:** Introduces sparsity constraints in the encoded representation.
- **Variational Autoencoder (VAE):** Models probability distributions in the latent space.

## 2.5 Sparse autoencoders

Sparse autoencoders offer us an alternative method for introducing an information bottleneck without requiring a reduction in the number of nodes at our hidden layers. Rather, we'll construct our loss function such that we penalize activations within a layer. For any given observation, we'll encourage our network to learn an encoding and decoding which only relies on activating a small number of neurons.

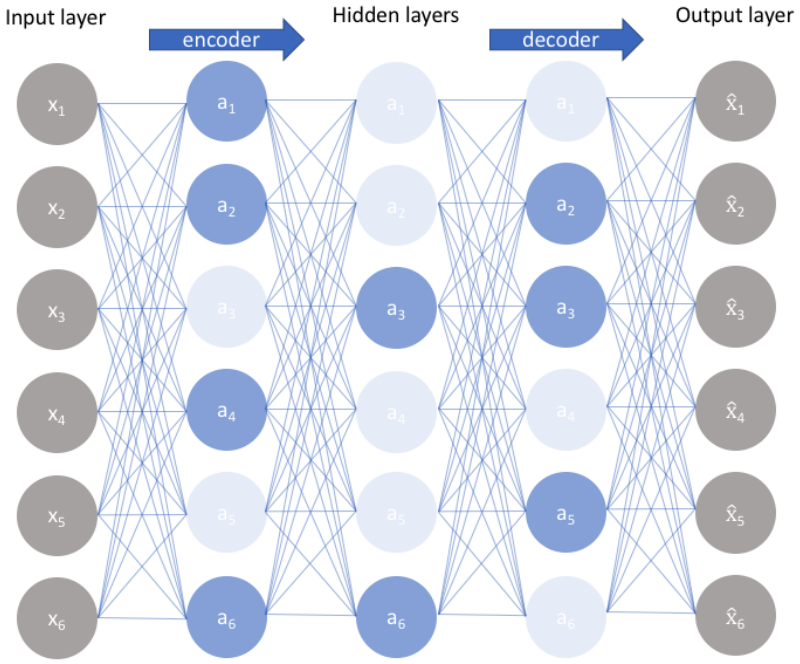


Figure 6: Sparse autoencoder architecture

We should note that this is a different approach towards regularization, as we normally regularize the weights of a network, not the activations.

There are two main ways by which we can impose this sparsity constraint; both involve measuring the hidden layer activations for each training batch and adding some term to the loss function in order to penalize excessive activations. These terms are:

### 2.5.1 Loss function

**L1 Regularization**: We can add a term to our loss function that penalizes the absolute value of the vector of activation $a$ in layer $h$ for observation $i$ scaled by a tunning parameter $\lambda$.

$$\mathcal{L}(x, x') = \frac{1}{N}\sum_{i=1}^{N}\|x_i - x'_i\|^2 + \lambda\sum_{j=1}^{m}|a_j^{(h)}|$$

**KL-Divergence Regularization**: We can add a term to our loss function that penalizes the average activation. KL-Divergence measure of how one probability distribution differs from another.

Average activation of each neuron in the hidden layer $h$ for a given training batch $x'$ is defined as follows:

$$\hat{\rho}_j = \frac{1}{m}\sum_{i=1}^{m} a_j^{(h)}(x'_i) \tag{2}$$

where $m$ represents the number of training samples (batch size),

$j$ represents the $j$-th neuron in the hidden layer,

$a_j^{(h)}(x'_i)$ represents the activation of the $j$-th neuron in the hidden layer for the $i$-th training sample.

In essence, by constraining the average activation of a neuron over a collection of samples we're encouraging neurons to only fire for a subset of the observations.

KL-Divergence is defined as follows:

$$\mathrm{KL}(\rho\|\hat{\rho}) = \sum_{j=1}^{l^{(h)}} \rho\log\left(\frac{\rho}{\hat{\rho}_j}\right) + (1-\rho)\log\left(\frac{1-\rho}{1-\hat{\rho}_j}\right) \tag{3}$$

where $l^{(h)}$ represents the number of neurons in the hidden layer,

$\rho$ is the sparsity hyper parameter,

$\hat{\rho}_j$ represents the average activation of the $j$-th neuron in the hidden layer.

So finally the loss function for KL-Divergence Regularization is defined as follows:

$$\mathcal{L}(x, x') = \frac{1}{N}\sum_{i=1}^{N}\|x_i - x'_i\|^2 + \lambda\sum_{j=1}^{m}\mathrm{KL}(\hat{\rho}||\rho_j) \tag{4}$$

## 2.6 Denoising

Autoencoders can be used to remove noise from data. This is done by training the network to reconstruct the original input from a input to which we artificialy added noise. The goal is that network learns to extract the essential features from the input data and discard the noise.

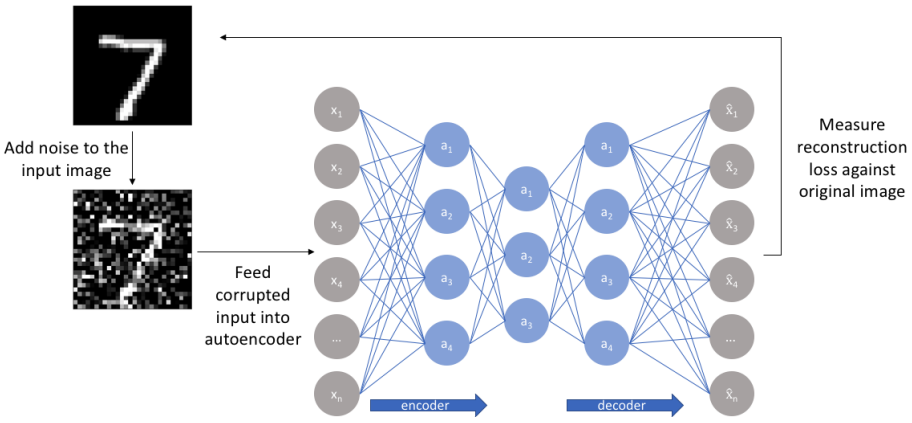


Figure 7: Denoising autoencoder architecture

With this approach, our model isn't able to simply develop a mapping which memorizes the training data because our input and target output are no longer the same. Rather, the model learns a vector field for mapping the input data towards a lower-dimensional manifold. If this manifold accurately describes the natural data, we've effectively "canceled out" the added noise.

### 2.6.1 Corruption process

There are various ways to corrupt the input data. Some common approaches include:

- **Gaussian Noise:** Add random Gaussian noise to the input data.
- **Dropout:** Randomly set some input units to zero.
- **Masking Noise:** Randomly set some features of the input data to zero.

### 2.6.2 Loss function

To train a denoising autoencoder, we define loss function as follows:

$$\mathcal{L}_{\text{DAE}}(x, \tilde{x}, x') = \frac{1}{N} \sum_{i=1}^{N} \|x_i - x'_i\|^2 \tag{5}$$

where $N$ : Number of training samples,
$x_i$ : Clean data for the $i$-th training sample,
$\tilde{x}_i$ : Corrupted input for the $i$-th training sample,
$x'_i$ : Reconstructed output for the $i$-th training sample.

In comparing the loss functions of a denoising autoencoder and a vanilla autoencoder, a key distinction arises in the input provided to the encoder. In the denoising autoencoder, the encoder receives a corrupted input as $\tilde{x}$. Then the decoder attempts to reconstruct the original input $x$ from the encoded representation $h$.

In contrast, the vanilla autoencoder receives the original input $x$ and attempts to reconstruct the same input $x$.

## 2.7 Variational Autoencoders (VAEs)

VAEs are a type of autoencoder that add probabilistic constraints to the latent space variables. This allows them to model probability distributions in the latent space. This models are widely used in applications such as image generation, image interpolation, and data augmentation.

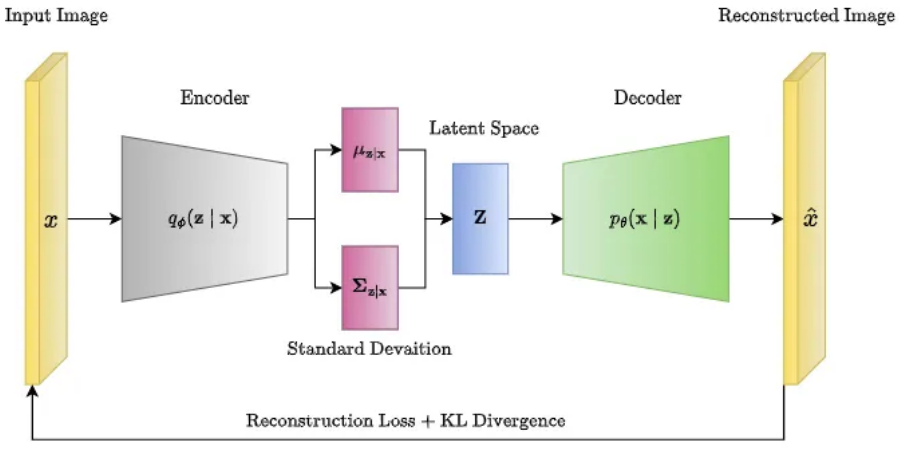


Figure 8: Variational autoencoder architecture

### 2.7.1 Latent space

In a vanilla autoencoder, encoder outputs a point in the latent space and decoder maps the point in the latent space to the reconstructed output.
But in Variational Autoencoders, the encoder outputs mean vector $\mu$ and a variance vector $\sigma^2$ - so it outputs a probability distribution in the latent space and decoder maps the sample from the latent space to the reconstructed output.
Latent space is defined as follows:

$$h = z \sim \mathcal{N}(\mu, \sigma^2)$$

where $z$ is the latent vector, $\mu$ is the mean vector, and $\sigma^2$ is the variance vector.

Instead of a single point in the latent space as in vanilla autoencoder, the VAE covers a certain "area" centered around the mean value with a size corresponding to the standard deviation. This gives the decoder a lot more to work with — a sample from anywhere in the area will be very similar to the original input

### 2.7.2 Latent space regularity

The regularity that is expected from the latent space in order to make the generative process possible requires:

- Continuity: Two close points in the latent space should not yield completely different contents once decoded.
- Completeness: For a chosen distribution, a point sampled from the latent space should result in "meaningful" content once decoded.

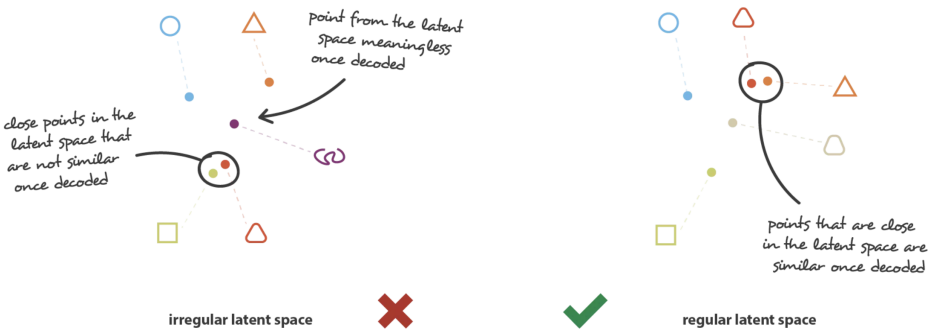


Figure 9: Latent space regularity

In practice regularisation is done by enforcing distributions to be close to a standard normal distribution (centred and reduced). So we can say that VAEs are regularized autoencoders whose encodings distribution is forced to be close to a standard normal distribution.

$$Z \sim \mathcal{N}(0, 1)$$

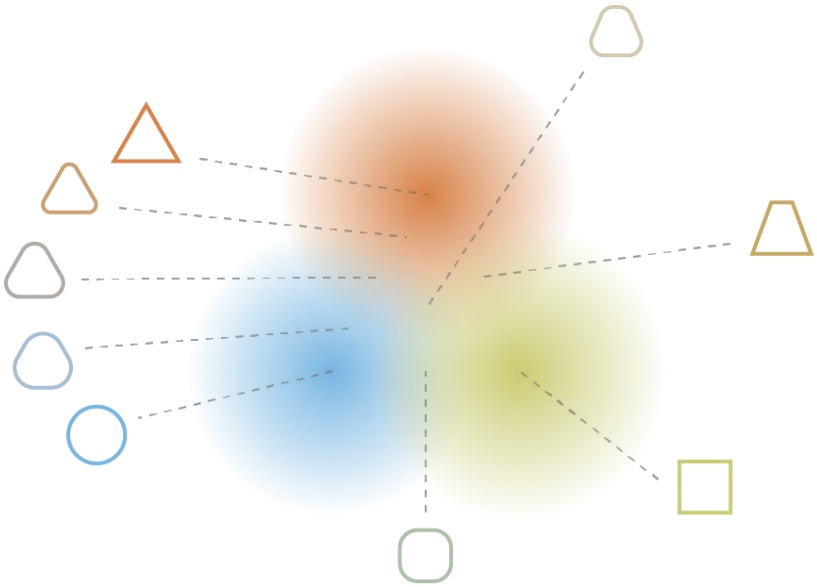

Figure 10: Latent space regularity distribution

### 2.7.3 Reparameterization trick

To train a VAE, we need to sample from the latent space. However, we cannot directly sample from the latent space because it is a probability distribution. Instead, we use the reparameterization trick to sample from the latent space. The reparameterization trick is defined as follows:

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

where $z$ is the latent vector, $\mu$ is the mean vector, $\sigma$ is the standard deviation vector, $\epsilon$ is a random vector sampled from a standard normal distribution, and $\odot$ represents element-wise multiplication.

### 2.7.4 Loss function

To train a VAE, we define the loss function as follows:

$$\mathcal{L}_{\text{VAE}}(x, x') = \frac{1}{N} \sum_{i=1}^{N} \|x_i - x_i'\|^2 + \text{KL}(q(z|x_i)||p(z)) \tag{6}$$

where $N$ is the number of training samples, $x_i$ is the input data, $x_i'$ is the reconstructed output, $q(z|x_i)$ is the encoder distribution, $p(z)$ is the prior distribution, and KL is the Kullback-Leibler divergence.

The KL divergence term ensures that the learned distribution over the latent space is close to the prior distribution, which helps regularize the model and ensures that the latent space has a meaningful structure.

### 2.7.5 Applications

In comparison to traditional Autoencoders (AEs), VAEs introduce a probabilistic twist to the encoding-decoding process, enabling them to excel in generative tasks such as image generation, text generation, and data augmentation.

Major applications encompass:

- **Image Generation and Interpolation:** VAEs can generate new images that closely resemble the training dataset, facilitating tasks such as synthetic image creation and artistic design. Moreover, they are adept at interpolating between images, offering smooth transitions and understanding of semantic image features.

- **Data Augmentation:** In scenarios where data is scarce, VAEs can augment existing datasets by generating new data points, thereby enhancing the performance of machine learning models without the need for additional real data.
- **Generative Tasks in Various Domains:** Beyond image generation, VAEs are instrumental in text generation, speech synthesis, and music generation. They can also be applied to other domains such as molecular design and drug discovery.
- **Anomaly Detection:** By learning to reconstruct normal data, VAEs can identify anomalies or outliers in a dataset. This is particularly valuable in areas like fraud detection, where deviations from the norm are subtle yet critical.

# 3 Generative adversarial networks

Generative adversarial networks, or GANs, revolutionized generative modeling. Introduced in 2014, GANs consist of a generator and a discriminator engaged in competitive learning. The generator creates realistic data samples from random noise, refining its output through training iterations. Simultaneously, the discriminator learns to differentiate between real and generated samples. This adversarial interplay results in the generation of high-quality, realistic data. GANs are widely applied in tasks such as image synthesis, style transfer, and data augmentation.

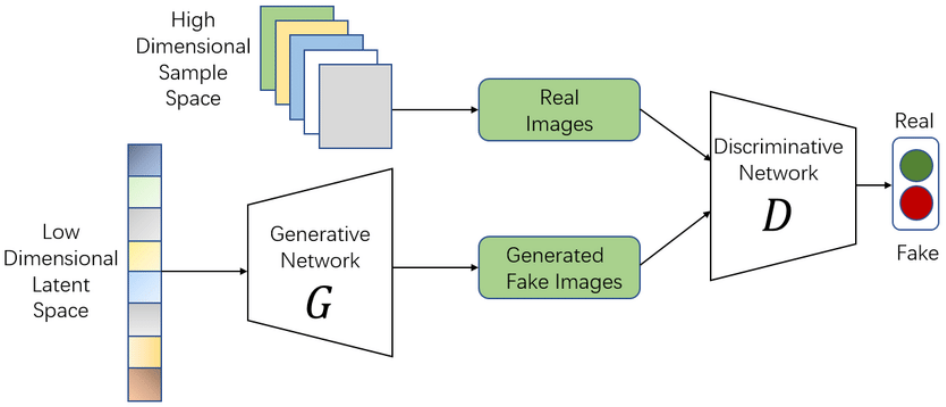


Figure 11: Generative adversarial network representation

## 3.1 Generator

The generator takes a random noise vector as input and generates a sample from the data distribution. The generator is trained to fool the discriminator into classifying the generated sample as real.

## 3.2 Discriminator

The discriminator takes a sample from the data distribution or a generated sample as input and classifies it as real or fake. The discriminator is trained to correctly classify the real samples as real and the generated samples as fake.

## 3.3 Training

The generator and discriminator are trained simultaneously in a competitive process. The generator is trained to fool the discriminator into classifying the generated sample as real. The discriminator is trained to correctly classify the real samples as real and the generated samples as fake. This adversarial interplay results in the generation of high-quality, realistic data.

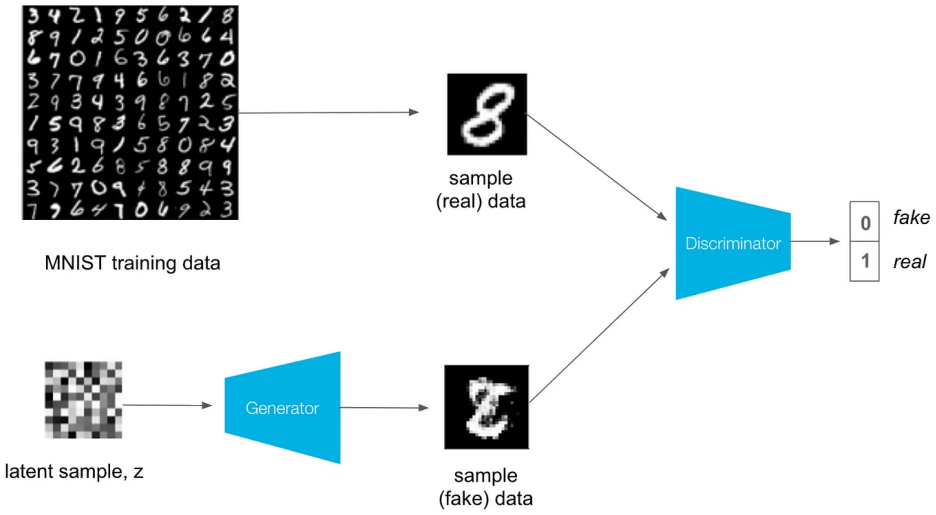


Figure 12: GAN training process

## 3.4 Loss function

The loss function for the generator is defined as follows:

$$\mathcal{L}_{\text{generator}}(z) = \log(1 - D(G(z))) \tag{7}$$

where $z$ is the random noise vector, $G$ is the generator, $D$ is the discriminator, and $\mathcal{L}_{\text{generator}}$ is the loss function for the generator.

The loss function for the discriminator is defined as follows:

$$\mathcal{L}_{\text{discriminator}}(x, z) = -\log(D(x)) - \log(1 - D(G(z))) \tag{8}$$

where $x$ is the input data, $z$ is the random noise vector, $G$ is the generator, $D$ is the discriminator, and $\mathcal{L}_{\text{discriminator}}$ is the loss function for the discriminator.

## 3.5 Applications

GANs are widely applied in tasks such as image synthesis, style transfer, and data augmentation.

### 3.5.1 Image Synthesis

GANs can generate realistic images that closely resemble the training dataset. This is particularly valuable in scenarios where data is scarce, such as in medical imaging. GANs can also be used to generate images from text descriptions, enabling applications such as image captioning.

### 3.5.2 Style Transfer

GANs can transfer the style of one image to another, enabling applications such as image colorization and image-to-image translation. This is achieved by training the generator to translate an image from one domain to another, while the discriminator learns to distinguish between real and generated images.

### 3.5.3 Data Augmentation

In scenarios where data is scarce, GANs can augment existing datasets by generating new data points, thereby enhancing the performance of machine learning models without the need for additional real data.

# 4 Diffusion Models

Inspired by physical processes like molecular diffusion, diffusion models simulate the gradual generation of data. They apply a series of transformations iteratively to a base distribution. This iterative process progressively refines the base distribution until it converges to the desired data distribution.

So this models operate by simulating a diffusion process, which transforms data into a Gaussian random noise through a predefined number of steps and the essence of diffusion models lies in learning the reverse process—starting from noise and gradually reconstructing the data.

Diffusion models have emerged as a powerful class of generative models, achieving state-of-the-art results in generating high-fidelity images, audio, and text.

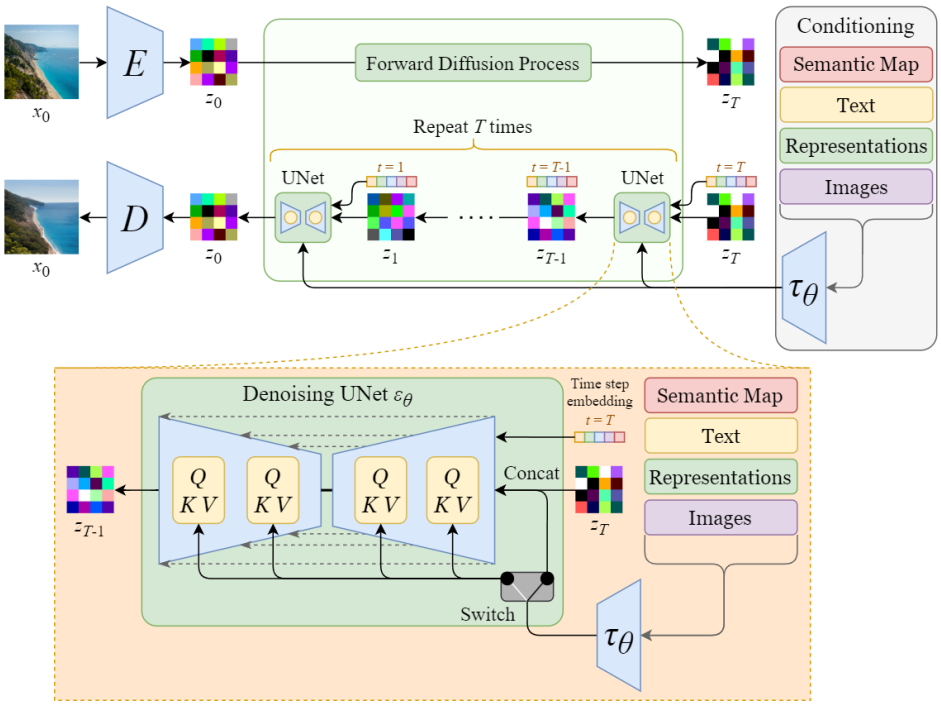


Figure 13: Diffusion model architecture

## 4.1 How Diffusion Models Work

The workflow of diffusion models can be divided into two phases: the forward diffusion phase and the reverse diffusion phase. The forward phase incrementally adds Gaussian noise to the data across a series of steps, until the data is completely indistinguishable from random noise. Mathematically, this process can be expressed as:

$$x_t = \sqrt{\alpha_t} x_{t-1} + \sqrt{1-\alpha_t}\epsilon, \quad \epsilon \sim \mathcal{N}(0, \mathbf{I}), \tag{9}$$

where $x_t$ represents the data at step $t$, $\alpha_t$ is a variance schedule, and $\epsilon$ is sampled from a standard Gaussian distribution.

The reverse phase, on the other hand, involves a neural network model learning to estimate the clean data from the noisy data, effectively reversing the diffusion process. This is achieved by training the model to predict the noise that was added at each step, gradually denoising the input until it recovers the original data. The reverse process can be expressed as:

$$x_{t-1} = \frac{x_t - \sqrt{1-\alpha_t}\epsilon}{\sqrt{\alpha_t}}, \tag{10}$$

where $x_{t-1}$ represents the data at step $t-1$, $x_t$ is the noisy data at step $t$, $\alpha_t$ is a variance schedule, and $\epsilon$ is the noise added at step $t$. The goal of the reverse process is to estimate the clean data from the noisy data, effectively denoising the input.

## 4.2 Mathematical Formulation of the Loss Function

The training of diffusion models is guided by a loss function that encourages the model to accurately reverse the diffusion process. A commonly used loss is the variational lower bound, which can be formulated as:

$$\mathcal{L}_{\text{total}} = \mathbb{E}_{x_0,\epsilon}\left[\log p_\theta(x_0|x_T) + \sum_{t=1}^{T} \log p_\theta(x_{t-1}|x_t)\right], \tag{11}$$

represents the total loss calculated across all diffusion steps. where $p_\theta$ denotes the model's distribution parameterized by $\theta$, and $T$ is the total number of diffusion steps. This loss function combines the log-likelihood of the original data given the final noisy data and the log-likelihood of each reverse step, effectively measuring the model's ability to reconstruct the original data from noise.

Another loss function that is used is the Kullback-Leibler divergence, which can be formulated as:

$$\mathcal{L}_{\text{total}} = \mathbb{E}_{x_0,\epsilon}\left[\log p_\theta(x_0|x_T) + \sum_{t=1}^{T} KL(q(x_{t-1}|x_t, x_0)||p_\theta(x_{t-1}|x_t))\right], \tag{12}$$

represents the total loss calculated across all diffusion steps. where $p_\theta$ denotes the model's distribution parameterized by $\theta$, and $T$ is the total number of diffusion steps. This loss function combines the log-likelihood of the original data given the final noisy data and the Kullback-Leibler divergence of each reverse step, effectively measuring the model's ability to reconstruct the original data from noise.

Both equations aim to minimize the difference between the original data and the data generated by reversing the diffusion process, thereby training the model to accurately reconstruct data from noise. The first equation focuses on maximizing the likelihood of each reverse step directly, while the second equation also incorporates the KL divergence to ensure the model's distribution closely matches the true generative process.

## 4.3 Breakthroughs and Training Process

Diffusion models have seen significant breakthroughs in recent years, including improvements in sampling efficiency, model architecture, and training techniques. These advancements have enabled diffusion models to generate highly realistic and diverse outputs across various domains.

The training process of diffusion models involves iteratively applying the forward diffusion to the training data, and then training the model to predict the reverse diffusion steps. This requires careful balancing of the noise levels and ensuring the model can effectively learn the data distribution. Advanced techniques, such as learning rate schedules and architecture designs, have been developed to optimize this process.

So the diffusion model is trained by applying a series of transformations iteratively to a base distribution. This iterative process progressively refines the base distribution until it converges to the desired data distribution, this we can see in the figure 14.

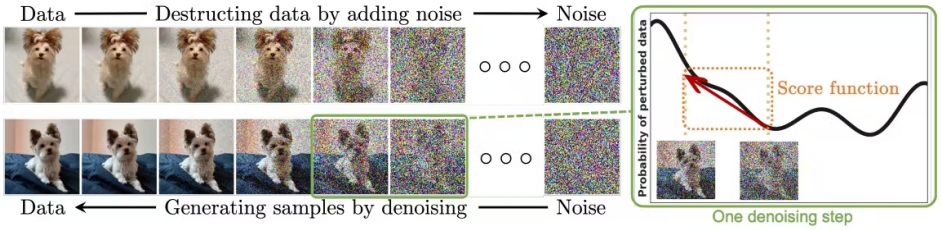


Figure 14: Diffusion model training

## 4.4 Applications and Future Prospects

The application spectrum of diffusion models is vast and continually expanding, covering areas from photorealistic image generation to the synthesis of human-like speech. The fidelity and versatility of the outputs have opened new frontiers in content creation, medical imaging, and beyond. Looking ahead, the exploration of diffusion models in unsupervised learning, domain adaptation, and their integration with other AI modalities heralds a promising trajectory for future research and application.

# 5 Conclusion

Throughout this report, we have introduced and examined three prominent generative models in machine learning: **Autoencoders**, **Generative Adversarial Networks (GANs)**, and **Diffusion Models**.

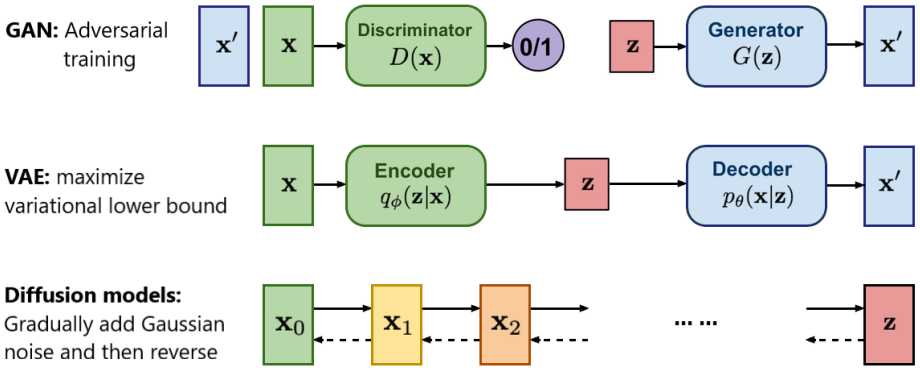


Figure 15: Conceptual summary of autoencoders, GANs, and diffusion models

Autoencoders are notable for their ability in data compression and reconstruction, making them a potent tool for dimensionality reduction, denoising, and feature extraction. Their architecture, comprising an encoder and a decoder, enables efficient data representation, critical for tasks that demand data compression with minimal information loss.

Generative Adversarial Networks (GANs) have transformed synthetic data generation. Utilizing a competitive mechanism between a generator and a discriminator, GANs can create highly realistic images, texts, and other data forms. This capability not only improves synthetic data quality but also opens new avenues in art creation, photo editing, and game development, where realistic content generation is crucial.

Diffusion Models, the forefront of generative model technology, are inspired by the natural diffusion process. These models convert data into a noise-like distribution and then learn to reverse this transformation to generate data. They excel in producing realistic and varied outputs, establishing new benchmarks for realism and detail in generated content. Demonstrating superior performance in text-to-image synthesis, audio generation, and beyond, diffusion models highlight their versatility and promise for future exploration and application.

In conclusion, autoencoders, GANs, and diffusion models each offer distinctive approaches to understanding and generating complex data. Autoencoders are adept at learning efficient data representations, GANs at generating realistic synthetic data, and diffusion models at creating high-quality content through a novel reverse diffusion process. Collectively, these technologies exemplify the swift advancements in machine learning, paving the way towards a future where generating and interpreting complex data is achieved with unparalleled precision and creativity.

# 6 References


1. K. Preechakul, N. Chatthee, S. Wizadwongsa, and S. Suwajanakorn, "Diffusion Autoencoders: Toward a Meaningful and Decodable Representation," *Vidyasirimedhi Institute of Science and Technology*, Rayong, Thailand, 2022. [Online]. Available: `https://diff-ae.github.io/`

2. J. Jordan, "Introduction to autoencoders", *Jeremy Jordan's Blog*, [Online]. Available: `https://www.jeremyjordan.me/autoencoders/`

3. H. Bandyopadhyay, "Autoencoders in Deep Learning: Tutorial and Use Cases [2023]", *V7 Labs Blog*, [Online]. Available: `https://www.v7labs.com/blog/autoencoders-guide`

4. "How to Work with Autoencoders [Case Study Guide]", *Neptune.ai Blog*, [Online]. Available: `https://neptune.ai/blog/autoencoders-case-study-guide`

5. "Autoencoders Tutorial", *Edureka Blog*, [Online]. Available: `https://www.edureka.co/blog/autoencoders-tutorial/`

6. R. Shende, "Autoencoders, Variational Autoencoders (VAE), and $\beta$-VAE", *Medium Blog*, [Online]. Available: `https://medium.com/@rushikesh.shende/autoencoders-variational-autoencoders-vae-and-%CE%B2-vae-ceba9998773d`

7. "Variational Autoencoder in TensorFlow", *LearnOpenCV Blog*, [Online]. Available: `https://learnopencv.com/variational-autoencoder-in-tensorflow/`

8. K. Preechakul, N. Chatthee, S. Wizadwongsa, and S. Suwajanakorn, "Diffusion Autoencoders: Toward a Meaningful and Decodable Representation," in *CVPR*, 2022. [Online]. Available: `https://openaccess.thecvf.com/content/CVPR2022/papers/Preechakul_Diffusion_Autoencoders_Toward_a_Meaningful_and_Decodable_Representation_CVPR_2022_paper.pdf`

9. N. Siddiqui, "Comparative Study f Generative Models for Text-to-Image o-Image Generation", *University of Windsor*, [Online]. Available: `https://scholar.uwindsor.ca/cgi/viewcontent.cgi?article=9960&context=etd`

10. R. O'Connor, "How DALL-E 2 Actually Works", *AssemblyAI Blog*, [Online]. Available: `https://www.assemblyai.com/blog/how-dall-e-2-actually-works/`

11. R. O'Connor, "Diffusion Models for Machine Learning: Introduction", *AssemblyAI Blog*, [Online]. Available: `https://www.assemblyai.com/blog/diffusion-models-for-machine-learning-introduction/`