POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Exploring diffusion-based approaches for the generation of Implicit 3D representations

Tesi di Laurea Magistrale in
Physics Engineering - Ingegneria Fisica

Author: **Riccardo Tripodi**

Student ID: 10620589
Advisor: Prof. Matteo Matteucci
Co-advisors: Paolo Cudrano, Cristian Sbrolli
Academic Year: 2024-2025

Abstract

The creation of 3D models has long been a central focus in computer graphics, with decades of research dedicated to advancing the field. The introduction of sophisticated neural networks and generative models has accelerated progress, making it possible to generate increasingly realistic and diverse 3D content. Generating high-quality 3D content has a wide range of applications, such as computer graphics, gaming, and virtual reality. To achieve this, several generative approaches have been investigated, including generative adversarial networks, variational autoencoders, normalizing flows, and autoregressive models. Diffusion models have recently gained prominence, demonstrating exceptional performance in the 2D image domain and surpassing other generative techniques. Despite their remarkable success in 2D applications, achieving comparable advancements in 3D remains an ongoing challenge. In fact, dealing with 3D data requires careful consideration, especially when choosing the appropriate 3D representation, as it directly affects design choices and the training efficiency of a model.

In this work, we initially explore symmetry-aware generative architectures designed to leverage the inherent equivariances and permutation symmetries present in implicit 3D representations. While theoretically promising, these approaches presented practical difficulties during integration with diffusion pipelines and ultimately did not surpass simpler baselines in generation quality.

Motivated by these challenges, we developed a scalable and efficient two-stage generative pipeline based on latent diffusion. Our approach first employs a variational autoencoder to encode input neural networks representing 3D surfaces into a regularized latent space, allowing for effective sampling and decoding of novel shapes. A latent diffusion model then learns to map this latent space into a standard Gaussian distribution, enabling generation from pure noise in the compressed latent domain.

Our final model achieves results comparable to state-of-the-art methods in terms of generation quality and diversity, while using at least an order of magnitude fewer parameters. This demonstrates the potential of latent diffusion models for 3D data generation.

Keywords: 3D, implicit representations, diffusion models, latent diffusion

Abstract in lingua italiana

La creazione di modelli 3D è da tempo un tema centrale nella computer grafica. L'introduzione di reti neurali e di modelli generativi ne ha accelerato lo sviluppo, rendendo possibile la generazione di contenuti 3D sempre più realistici e diversificati. La generazione di contenuti 3D di ha numerose applicazioni, tra cui la computer grafica, il gaming e la realtà virtuale. Per raggiungere questo obiettivo, sono stati studiati diversi approcci generativi, come le generative adversarial networks, i variational autoencoders e i modelli autoregressivi. I modelli di diffusione hanno recentemente acquisito grande rilevanza, mostrando prestazioni eccezionali nel dominio delle immagini 2D e superando altre tecniche generative. Nonostante il loro successo nelle applicazioni 2D, ottenere risultati comparabili nel 3D rappresenta ancora una sfida aperta. Infatti, il trattamento dei dati 3D richiede un'attenta considerazione, in particolare nella scelta della rappresentazione più adatta, poiché questa influisce direttamente sulle scelte progettuali e sull'addestramento del modello.

In questo lavoro, esploriamo inizialmente architetture generative, progettate per sfruttare le simmetrie per permutazione intrinseche alle rappresentazioni implicite delle forme 3D. Sebbene promettenti dal punto di vista teorico, questi approcci hanno mostrato difficoltà pratiche nell'integrazione con pipeline basate sulla diffusione. Motivati da queste difficoltà, abbiamo sviluppato una pipeline generativa a due stadi, scalabile ed efficiente, basata sulla diffusione latente. Il nostro approccio impiega inizialmente un autoencoder variazionale per codificare le reti neurali che rappresentano le superfici 3D in uno spazio latente, permettendo il campionamento e la decodifica di nuove forme. In seguito, un modello di diffusione apprende a mappare questo spazio latente in una distribuzione gaussiana standard, consentendo la generazione a partire da puro rumore.

Il nostro modello finale raggiunge risultati paragonabili allo stato dell'arte in termini di qualità e diversità della generazione, utilizzando tuttavia almeno un ordine di grandezza in meno nel numero di parametri. Questo dimostra il potenziale dei modelli di diffusione latente per la generazione di dati 3D.

Parole chiave: 3D, rappresentazioni implicite, modelli di diffusione, diffusione latente

Contents

Introduction

The generation of 3D models is a fundamental task in computer graphics with applications ranging from digital content creation to robotics and augmented reality. Recent advances in generative modeling—particularly those based on neural networks—have enabled the synthesis of highly realistic and diverse 3D shapes.

This thesis explores a novel generative pipeline for 3D shape synthesis based on implicit representations and diffusion models. Unlike traditional 3D representations (e.g., voxels, point clouds, meshes), implicit representations encode a shape as a continuous function—typically implemented as a neural network—that predicts whether a point in space lies inside or outside the surface of the object. This compact and resolution-independent format allows for smooth, high-fidelity modeling.

A first line of exploration focuses on symmetry-aware generative models, designed to incorporate the permutation symmetries inherent in multilayer perceptrons used for representing implicit fields. Drawing inspiration from geometric deep learning, these models aim to embed structural priors directly into the generative architecture. While conceptually appealing, integrating such symmetry-aware methods within diffusion-based pipelines revealed practical challenges that limited their performance in our setting.

As an alternative to symmetry-based models and existing baselines from the literature, we propose a two-stage generative pipeline based on latent diffusion models, specifically designed to reduce computational complexity. In the first stage, a variational autoencoder is trained to compress neural implicit functions into a structured latent space. This enables reconstruction of functional representations from compact codes. In the second stage, a diffusion model is trained within this latent space to model its distribution and enable generation from Gaussian noise. This setup provides a framework to assess the capacity of a variational autoencoder to encode and reconstruct implicit functions, and to evaluate the ability of a diffusion model to learn the distribution of latent shape representations. The approach also allows us to examine its scalability and adaptability across object categories with varying geometric complexity

Experiments are conducted on representative shape classes commonly used in the liter-

ature, such as cars, chairs, and airplanes. The results, presented in the final chapters, provide insights into the performance and limitations of the proposed pipeline, and serve as a foundation for future research directions in generative modeling of implicit 3D representations.

Outline

The thesis is structured to provide a comprehensive understanding of the methodologies and techniques employed:

- **Background:** This section lays out the conceptual tools that underpin the rest of the thesis, including 3D data representations, Transformer architecture, Graph Neural Networks, and generative models such as GANs, VAEs, and Diffusion Models.
- **Implicit Representations Dataset:** This chapter presents the design and training strategy for implicit shape representations, which serve as the input data to the generative model. It explains how each 3D object in the dataset is individually encoded using a neural network that learns an occupancy function.
- **Symmetry-aware Diffusion Model:** This section introduces a generative model with a neural architecture that is equivariant to symmetry transformations, addressing the limitations of previous models by incorporating permutation symmetries of MLPs.
- **Latent Diffusion:** This chapter describes a two-stage generative pipeline based on the latent diffusion paradigm, which improves scalability and efficiency by performing generative modeling in a compressed latent space.
- **Results:** This final chapter presents the experimental setup, dataset, evaluation metrics, and results obtained, providing a comprehensive analysis of the effectiveness of the proposed methods.

By exploring these methodologies, the thesis aims to bridge recent progress in generative modeling with structural insights offered by representation theory and geometric deep learning, ultimately contributing to the field of 3D shape generation.

1 | Background

The goal of this thesis is to explore novel methods for **3D shape generation** using modern deep learning techniques. At a high level, 3D shape generation refers to the process of producing new, plausible 3D objects—such as chairs, airplanes, or any physical object—using computational models. This task has broad applications, from computer graphics and virtual reality to robotics and biomedical imaging. The ability to automatically generate 3D shapes enables faster design pipelines, data augmentation, and even generative creativity.

To address this task, a variety of **neural architectures** have been developed in recent years, including convolutional networks, transformers, and graph neural networks. These architectures differ in how they process data, capture dependencies, and handle structural inductive biases. Their effectiveness also depends on the **representations** used for 3D data—such as voxels, meshes, point clouds, or continuous implicit functions—which each come with their own advantages and challenges. On top of this, different **generative modeling paradigms**, such as generative adversarial networks, variational autoencoders, and diffusion models, have been introduced to learn and sample from complex data distributions.

In this chapter, we lay out the conceptual tools that will underpin the rest of the thesis:

- We begin by presenting the main ways 3D shapes can be represented for use in machine learning models.
- We then introduce the neural architectures that will be employed throughout the thesis, focusing on those particularly suited to 3D data or to weight-space processing.
- Finally, we review the core generative modeling approaches that form the foundation of our proposed methods.

This background is meant to provide the reader with the necessary context to understand and situate the contributions developed in the following chapters.

1.1. 3D data representations

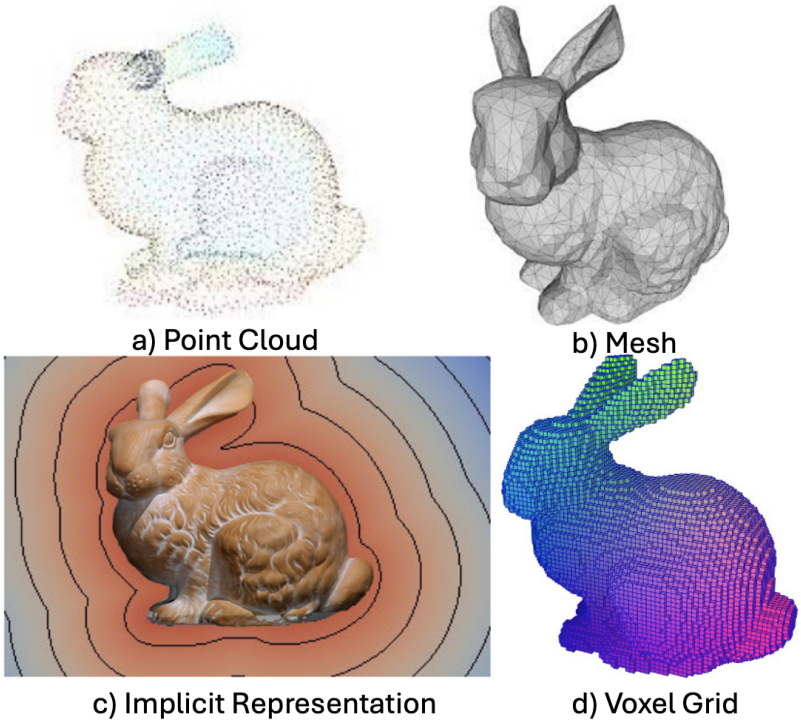


Figure 1.1: Different 3D shape representations commonly used in deep learning, illustrated on the Stanford Bunny: **(a)** Point cloud representation captures the geometry using a sparse set of sampled surface points; **(b)** Mesh representation describes the surface with interconnected vertices, edges, and faces, providing explicit topological information; **(c)** Implicit representation models the shape as a continuous function (e.g., signed distance or occupancy function), offering a compact and differentiable way to represent geometry; **(d)** Voxel grid representation discretizes the 3D space into a regular grid of binary or scalar values. Each representation offers different trade-offs in terms of expressiveness, memory efficiency, and compatibility with neural architectures.

Capturing and representing three-dimensional (3D) data in a form suitable for computation is a fundamental problem in computer vision, graphics, and machine learning. In practice, this involves transforming real-world objects—continuous and often geometrically complex—into discrete digital representations that can be stored, manipulated, and processed algorithmically. Unlike 2D data, where the regular grid of pixels offers a natural and consistent structure, 3D data admits multiple representations, each designed to balance competing demands such as geometric fidelity, memory efficiency, smoothness,

resolution, and compatibility with learning algorithms. For instance, some representations are highly compact but poorly suited for capturing fine surface details, while others provide high-resolution accuracy at the cost of excessive memory usage. Additionally, the structure of the representation affects how easily it can be integrated into deep learning pipelines: regular formats are more amenable to convolutional processing, whereas irregular formats require specialized architectures. The choice of 3D representation is therefore a crucial design decision, deeply influencing both the computational efficiency and the expressiveness of models trained on 3D data.

Several common formats are used to represent 3D data in practice, each with its own strengths and limitations:

Voxel grids divide 3D space into a regular grid of cubes (voxels). Each voxel stores information such as occupancy, intensity, or other relevant features, effectively discretizing the 3D environment into a structured format. This regularity makes voxel grids particularly well-suited for 3D convolutional neural networks (CNNs), which can naturally operate on grid-like inputs and are effective for processing volumetric data. However, voxel grids come with a significant trade-off between resolution and memory consumption. Achieving high-resolution representations requires a fine-grained grid, but the number of voxels grows cubically with resolution, leading to an exponential increase in memory usage. This makes high-resolution voxel representations computationally expensive to store and process. Moreover, voxel grids are typically dense structures, meaning that memory is allocated uniformly across the volume regardless of whether regions are occupied or empty, resulting in further inefficiencies when representing sparse geometries.

Point clouds are collections of 3D points, each with coordinates (x, y, z) and potentially other attributes such as color or intensity. Unlike voxel grids, point clouds do not discretize space into a regular structure; instead, they represent surfaces directly using a set of sampled points. This makes them highly memory-efficient for sparse data, as memory is only used for regions where geometry exists. Point clouds are a natural output of sensors like LiDAR, which emit laser pulses and record the 3D positions of the reflected signals to generate a set of surface samples from real-world environments. Due to this, point clouds are widely used in autonomous driving, robotics, and 3D scanning. However, their lack of regular structure and inherent unordered nature pose challenges for standard convolutional neural networks, which expect grid-like inputs. To address this, specialized neural architectures such as PointNet [51] have been developed to process point sets directly and extract meaningful features from them.

Mesh representation approximates the surface of a 3D object using a collection of small polygonal faces—typically triangles—joined together to form a continuous surface. These polygons define the surface geometry through their vertices and edges, which describe how the mesh is connected and oriented in space. Meshes are especially effective for modeling complex geometries such as human figures or terrain surfaces, offering a compact and efficient representation with fewer elements compared to dense voxel grids. They can capture fine surface details and are widely used in computer graphics, animation, and CAD applications. However, their irregular and non-uniform structure poses challenges for deep learning methods, which often rely on regular input formats; as a result, additional preprocessing steps or specialized architectures are typically required to accommodate varying mesh resolutions and topologies.

Implicit representations describe a shape through a function that determines the relationship between any point in space and the shape's surface. This function can represent various properties, such as distance, occupancy, or signed distance, allowing for the encoding of highly detailed and smooth surfaces. Implicit representations offer the benefit of being resolution-independent and capable of modeling complex shapes with high precision. Implicit representations are typically parameterized by neural networks that learn to approximate geometry and appearance from data. Unlike explicit representations such as voxels, point clouds, or meshes, implicit models store information compactly as neural network's weights, making them highly scalable for high-resolution reconstructions. Their resolution independence allows for fine-grained detail without the constraints of predefined grid sizes, and they naturally represent complex topologies, including objects with holes and smooth surfaces, without requiring intricate data structures. However, training implicit functions demands high-quality supervision and is computationally more intensive than direct methods on point clouds or voxel grids. Additionally, since many 3D applications rely on explicit meshes, converting implicit representations into usable formats often requires post-processing techniques like marching cubes [38], complicating integration with standard graphics pipelines. Despite these challenges, implicit models are particularly well-suited for 3D generation, as they offer high-resolution, smooth geometry without voxel artifacts, efficient storage and transmission, and flexible shape interpolation. Given these advantages, we adopt implicit representations as the primary 3D data format in this work.

1.2. Transformer architecture

The Transformer architecture, introduced in the seminal paper "Attention Is All You Need" [61], revolutionized the field of natural language processing. It has since become the foundation for models in numerous fields outside NLP being employed as a general sequence processing architecture for any input modality that could be represented as a sequence of tokens. Also in this work, the transformer architecture has been widely used so below is an exhaustive description of its origins, components, and applications.

Before Transformers, Recurrent Neural Networks (RNNs) and Long Short-Term Memory (LSTMs) dominated sequence modeling tasks (e.g., machine translation, speech recognition). However, these models had major drawbacks. Their sequential processing nature restricted parallelism, making training inefficient. Additionally, they suffered from the vanishing gradient problem, which hindered their ability to learn long-range dependencies due to exponential gradient decay. Even with mechanisms like attention, LSTMs and Gated Recurrent Units (GRUs) struggled to handle very long sequences effectively. The introduction of the Transformer architecture addressed these challenges by eliminating recurrence, enabling full parallelization, and leveraging self-attention to capture long-range dependencies. Moreover, it significantly reduced computational complexity by replacing O(T) recurrent dependencies with O(1) matrix multiplications, leading to more efficient training and inference.

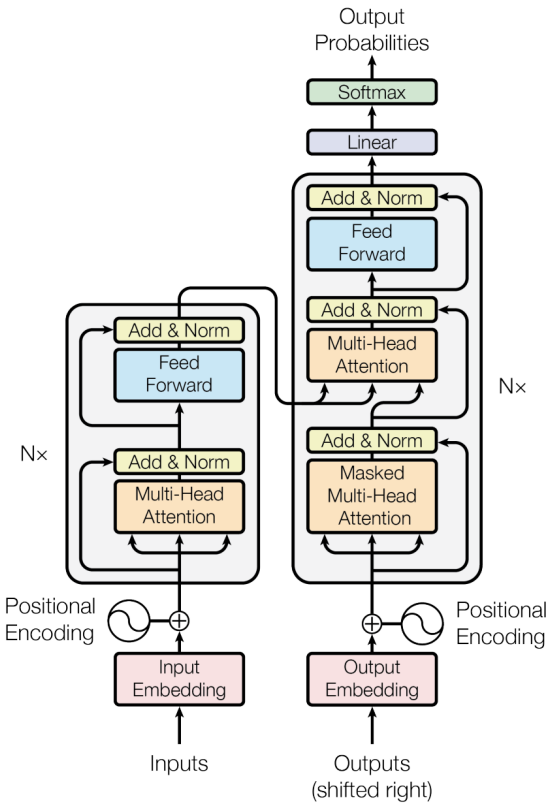


Figure 1.2: The Transformer model architecture from original paper [61].

The original Transformer consists of two main components: the **encoder** and the **decoder** (Figure 1.2). The encoder processes an input sequence into a context-aware representation, and the decoder generates an output sequence based on that representation. This makes it particularly effective for sequence-to-sequence tasks.

The fundamental building block of the Transformer is self-attention, which allows tokens to attend to all other tokens in a sequence. Given an input sequence represented as a matrix $X \in \mathbb{R}^{n \times d}$, where n is the sequence length and d is the embedding dimension, three matrices are computed:

$$Q = XW_Q, \tag{1.1a}$$

$$K = XW_K, \tag{1.1b}$$

$$V = XW_V, \tag{1.1c}$$

where $W_Q, W_K, W_V \in \mathbb{R}^{d \times d_k}$ are learned projection matrices that transform the input into **query (Q)**, **key (K)**, and **value (V)** representations.

The attention scores are computed as the scaled dot product between queries and keys:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V. \tag{1.2}$$

The softmax function ensures that the attention scores sum to 1 across all tokens, determining how much each token contributes to the final representation. The scaling factor d_k prevents large values in the dot product from leading to small gradient updates, stabilizing training.

Rather than using a single attention mechanism, the Transformer employs **multi-head attention**. Each head independently computes:

$$Q_i = XW_{Q_i}, \tag{1.3a}$$
$$K_i = XW_{K_i}, \tag{1.3b}$$
$$V_i = XW_{V_i}, \quad \text{for } i = 1, \ldots, h. \tag{1.3c}$$

The results are concatenated and projected back:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W_O. \tag{1.4}$$

The Transformer architecture itself does not have any inherent mechanism to understand the order of tokens in a sequence. Unlike RNNs and CNNs, which process input sequentially or with local receptive fields, the Transformer processes the entire sequence at once using self-attention. Each token attends to every other token based on content similarity, but this operation is permutation invariant and does not encode any information about whether one token appears before another. This means that if we shuffled the input tokens, the attention scores would remain the same, losing the sequence structure, which is essential for sequential tasks. To overcome this, the Transformer explicitly adds position-dependent information to each token's embedding.

This ensures that different positions in the sequence result in different learned representations, preserving the order while still leveraging self-attention. One common approach is to add fixed sinusoidal positional embeddings defined as follows:

$$PE(pos, 2i) = \sin\left(\frac{pos}{10000^{2i/d}}\right), \tag{1.5a}$$
$$PE(pos, 2i+1) = \cos\left(\frac{pos}{10000^{2i/d}}\right), \tag{1.5b}$$

where pos is the position and i indexes the embedding dimension. Alternatively, positional

embeddings can be learned directly through backpropagation as trainable parameters, allowing the model to optimize position encodings tailored to the specific data and task. Both fixed and learned positional embeddings have been shown to be effective, with the choice often depending on the application and dataset characteristics.

Each Transformer layer includes a **feed-forward network**:

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2, \tag{1.6}$$

where $W_1 \in \mathbb{R}^{d \times d_{ff}}$, $W_2 \in \mathbb{R}^{d_{ff} \times d}$.

Additionally, each layer incorporates **residual connections** and **normalization**:

$$z = \text{LayerNorm}(x + \text{SubLayer}(x)). \tag{1.7}$$

In the decoder, **masked self-attention** is introduced to prevent future tokens from influencing past tokens. This is achieved by modifying the attention computation so that the softmax is only applied over positions up to the current token index.

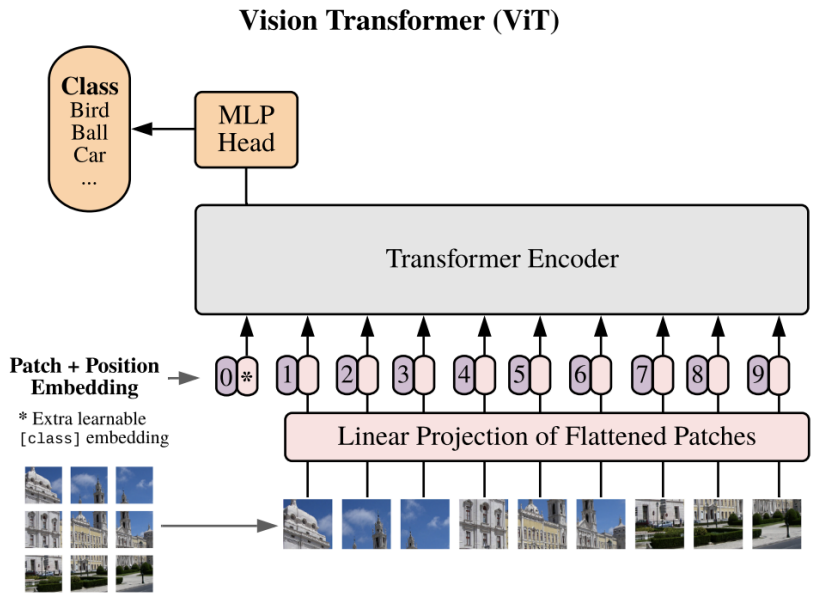


Figure 1.3: ViT model from original paper [19].

Variations of this architecture use either only the encoder or only the decoder depending on the task. Encoder-only models, such as BERT [34], apply self-attention bidirectionally

over the entire input sequence, allowing each token to attend to tokens both before and after it. This *non-causal* or *bidirectional* attention enables the model to capture rich context from the full input, making encoder-only architectures well suited for tasks like masked language modeling, classification, and question answering where understanding the entire input simultaneously is crucial. Decoder-only models, exemplified by GPT [7], employ *causal* or *autoregressive* self-attention, which masks future tokens so that each token can only attend to previous tokens in the sequence. This enforces a strict left-to-right dependency and prevents the model from "seeing" future tokens during training or generation, allowing decoder-only architectures to excel at generative tasks such as language modeling and text completion.

The Transformer architecture, initially developed for natural language processing, is fundamentally modality-agnostic and can be applied to any domain where the input data can be represented as a sequence of tokens. Rapidly, the Transformer framework spread beyond text to fields such as computer vision. Vision Transformer (ViT) [19] exemplifies this by adapting the Transformer architecture to image data. Unlike traditional convolutional neural networks, which operate on images through local receptive fields, ViT divides an image into fixed-size patches and treats these patches as a sequence of discrete tokens (see Figure 1.3), analogous to words in a sentence. Each patch is embedded into a high-dimensional vector space and then processed by a standard Transformer encoder, enabling the model to capture global context and relationships across the entire image via self-attention mechanisms. This tokenization approach allows ViT to leverage the powerful long-range dependency modeling capabilities of Transformers, often outperforming CNNs on vision benchmarks, particularly when trained on large-scale datasets. More broadly, this demonstrates the versatility of the Transformer architecture as a universal backbone that can be applied to diverse data modalities through appropriate tokenization strategies.

1.3. Graph Neural Network

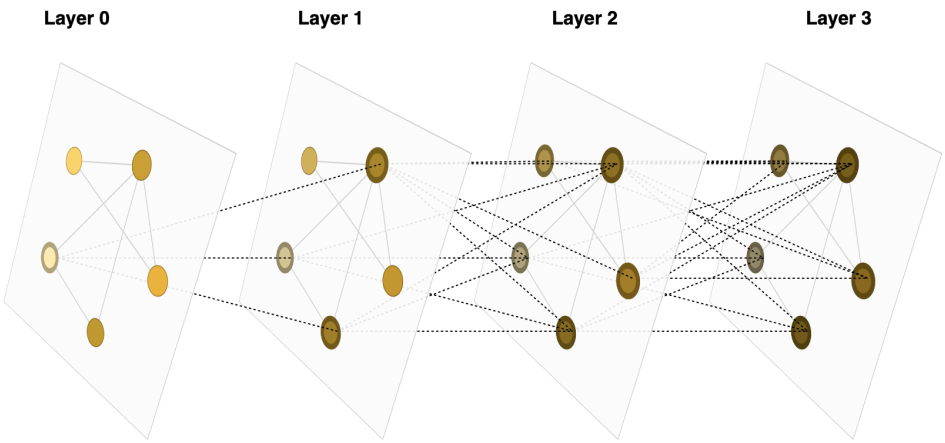


Figure 1.4: Information propagation from a node to the connected nodes through the layers of a graph neural network.

Graph Neural Networks (GNNs) [67] are a class of deep learning models designed to process data structured as graphs, where nodes represent entities and edges define relationships. They are widely used in domains such as social network analysis, molecular modeling, recommendation systems, and knowledge graphs, where relational dependencies are crucial. Unlike traditional neural networks that operate on fixed-dimensional inputs, GNNs learn node representations by iteratively aggregating information from neighboring nodes. A general message-passing framework for GNNs can be expressed as:

$$h_v^{(t)} = \sigma\left(W_t \cdot \mathrm{AGG}\left(\left\{h_u^{(t-1)} : u \in N(v)\right\}\right) + B_t h_v^{(t-1)}\right), \tag{1.8}$$

where $h_v^{(t)}$ is the node feature at layer t, $N(v)$ represents the set of neighboring nodes, and $\mathrm{AGG}(\cdot)$ is an aggregation function such as mean, sum, or attention-based pooling. The transformation matrix W_t and bias term B_t update the node embeddings, and σ is a non-linear activation function.

This iterative aggregation enables GNNs to capture hierarchical structures and long-range dependencies in graphs, making them highly effective for tasks like node classification, link prediction, and graph generation. Additionally, GNNs variants include edge representations in Equation (1.8) allowing the computation of edge-level tasks or leverge edge-level information for node-level tasks.

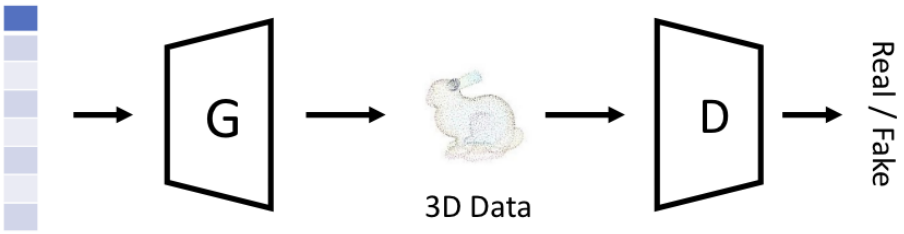


Figure 1.5: Simplified working scheme of a Generative Adversarial Network, illustrating the interaction between the generator and the discriminator during training. The GAN workflow is indipendent of the modality of data employed.

1.4. Generative Models

Generative modeling is a fundamental area of machine learning that focuses on learning to synthesize new data samples that resemble a given dataset. Originally developed and extensively studied in the context of 2D images, generative methods have demonstrated remarkable capabilities in producing high-quality, diverse samples across various domains. In this chapter, we explore three prominent families of generative models: Generative Adversarial Networks (GANs), Variational Autoencoders (VAEs), and Diffusion Models. Each of these approaches offers a distinct theoretical framework and set of practical advantages for data generation. While initially designed for image synthesis, all three have been successfully adapted and extended to handle the more complex domain of 3D data, enabling the generation of 3D shapes, scenes, and other volumetric information. The following sections provide an overview of these methods, highlighting their core principles, key differences, and the ways in which they have been applied to 3D generative tasks.

1.4.1. Generative Adversarial Networks

Generative Adversarial Networks (GANs), introduced by Goodfellow et al. [25], are a class of generative models that learn to synthesize data from a target distribution using a game-theoretic framework. A GAN consists of two neural networks—the *generator* G and the *discriminator* D—which are trained simultaneously in an adversarial manner. The generator attempts to produce samples that are indistinguishable from real data, while the discriminator seeks to accurately differentiate between real and generated samples.

Let $p_{\text{data}}(\mathbf{x})$ denote the true data distribution over instances $\mathbf{x}$, and let $p_{\mathbf{z}}(\mathbf{z})$ represent a prior distribution over latent variables $\mathbf{z}$, typically chosen as a standard normal or uniform distribution. The generator $G : \mathcal{Z} \to \mathcal{X}$ maps latent vectors to the data space, while the

discriminator $D : \mathcal{X} \to [0, 1]$ outputs the probability that a given sample is real.

The training process is formulated as a minimax game with the following value function:

$$\min_G \max_D V(D, G) = \mathbb{E}_{\mathbf{x} \sim p_{\text{data}}(\mathbf{x})}[\log D(\mathbf{x})] + \mathbb{E}_{\mathbf{z} \sim p_{\mathbf{z}}(\mathbf{z})}[\log(1 - D(G(\mathbf{z})))]. \tag{1.9}$$

In this formulation, the discriminator is optimized to maximize the likelihood of assigning the correct label to both real and generated samples, while the generator is optimized to minimize the probability that its outputs are correctly identified as fake by the discriminator. Through this adversarial training, the generator implicitly learns the data distribution by improving its ability to fool the discriminator.

At convergence, the generator's distribution p_g approximates the true data distribution p_{data}, and the discriminator becomes unable to distinguish between the two, yielding $D(\mathbf{x}) \approx 0.5$ for all $\mathbf{x}$. The optimal discriminator for a fixed generator can be shown [25] to be:

$$D^*(\mathbf{x}) = \frac{p_{\text{data}}(\mathbf{x})}{p_{\text{data}}(\mathbf{x}) + p_g(\mathbf{x})}. \tag{1.10}$$

Substituting D^* into the value function (Equation 1.9) gives the generator's cost function as:

$$C(G) = \max_D V(D, G) = -\log 4 + 2 \cdot \text{JS}(p_{\text{data}} \| p_g), \tag{1.11}$$

where $\text{JS}(\cdot\|\cdot)$ denotes the Jensen–Shannon divergence. Thus, training a GAN corresponds to minimizing the JS divergence between the real and generated data distributions.

Despite their theoretical appeal, GANs are notoriously difficult to train in practice. Common issues include *mode collapse* [3], where the generator produces a limited variety of samples, and *training instability* [63], stemming from the adversarial nature of the optimization. To mitigate these challenges, several variants have been proposed. For instance, Wasserstein GANs (WGAN) [2] replace the JS divergence with the Earth Mover (Wasserstein-1) distance to provide smoother gradients, while Least Squares GANs (LSGAN) [42] employ a least-squares loss to stabilize the discriminator's training.

Recent advances in deep generative modeling have prompted a surge of interest in extending GANs to 3D content synthesis. While GANs were originally designed for 2D image generation [25], their underlying framework has proven flexible enough to accommodate various forms of 3D data. A key challenge in this transition lies in the choice of 3D representation, as different formats impose distinct inductive biases and computational constraints.

In most of these approaches, the adversarial training loop remains consistent with the original GAN framework: a generator maps latent codes to 3D objects, and a discriminator attempts to distinguish generated samples from real ones. The choice of representation dictates the architecture of both networks and the form of supervision. For example, voxel-based models use 3D convolutional discriminators [64], while point cloud methods often rely on permutation-invariant networks [1].

To this end, researchers have proposed GAN-based models tailored to specific 3D representations. Point-based approaches, such as l-GAN [1] and tree-GAN [56], model unordered sets of 3D coordinates, enabling the generator to output sparse surface geometry. In contrast, voxel-based methods like 3D-GAN [64] operate on dense volumetric grids, offering a natural extension of 2D convolutions to 3D but at the cost of increased memory usage. Surface-based models such as MeshGAN [11] generate explicit polygonal meshes, preserving topological information but requiring specialized architectures to handle mesh connectivity. More recent efforts have turned to implicit representations, where the generator learns to model a continuous signed distance function (SDF) over 3D space. Methods like SurfGen and SDFStyleGAN [39, 66] fall into this category, leveraging implicit fields for high-resolution shape synthesis and extracting surfaces via isosurface extraction algorithms.

An alternative to fully 3D-supervised generative modeling involves leveraging 2D image data as a supervisory signal to guide the synthesis of 3D structures [46]. This approach, commonly known as 3D-aware adversarial generation, capitalizes on the widespread availability of 2D images to overcome the data limitations often associated with 3D datasets. Rather than directly training a generator to output 3D objects and comparing them to real 3D samples, these models incorporate a rendering stage into the generation pipeline.

In practice, the generator produces a latent 3D representation—such as an occupancy grid, a mesh, or an implicit function like a signed distance field. This representation is then passed through a differentiable renderer that projects it into a 2D image from a randomly sampled viewpoint. The resulting image is evaluated by a discriminator trained to classify whether the image is a realistic photograph or a rendered projection. The generator is optimized not only to generate plausible 3D structures but also to ensure that their rendered appearances conform to natural image distributions.

This rendering-based supervision framework allows the model to learn 3D priors implicitly by grounding them in the statistics of 2D images. It shifts the burden of learning from scarce 3D annotations to the far more abundant and diverse corpus of unlabeled 2D data. The use of differentiable rendering is key, as it enables the backpropagation

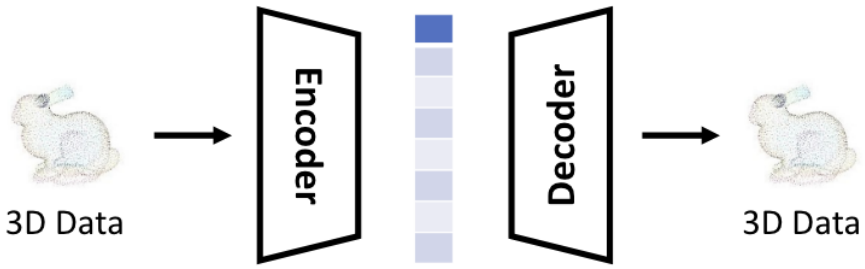


Figure 1.6: Simplified working scheme of a Variational Autoencoder, illustrating the interaction between the encoder and the decoder during training. The VAE workflow is indipendent of the modality of data employed.

of adversarial gradients from the 2D discriminator to the 3D generator, closing the loop between geometry and appearance in an end-to-end trainable manner. One notable example of this paradigm is HoloGAN [46], which introduces an implicit 3D inductive bias by learning volumetric feature representations within a canonical 3D space. These features are subsequently transformed according to a sampled camera pose, yielding 2D feature maps aligned with the desired viewpoint. A dedicated rendering network then decodes these 2D projections into photorealistic images, effectively bridging the gap between 3D structure and 2D appearance without requiring explicit 3D supervision.

1.4.2. Variational Autoencoders

A traditional autoencoder [27] is a neural network architecture composed of two parts: an encoder and a decoder. The encoder compresses the input data x into a typically lower-dimensional latent representation z, often referred to as the bottleneck. The decoder then attempts to reconstruct the original input x from this compressed latent vector z. The model is trained to minimize the reconstruction error, encouraging the network to capture the most salient features of the data in the latent space. However, this deterministic encoding does not impose any explicit structure or distribution on the latent space. As a result, the latent representations can be discontinuous or arbitrarily distributed, making it difficult to generate new, meaningful data points by sampling from the latent space.

A Variational Autoencoder (VAE) [33] is a type of generative model that extends the standard autoencoder framework by incorporating principles from probabilistic inference. It is designed to learn a continuous, structured latent space where similar data points are mapped to nearby regions, making it useful for tasks like data generation, denoising, and representation learning.

Instead of learning a deterministic mapping from inputs to a latent representation, VAEs model the latent space as a probability distribution. Given an input x, the encoder outputs a distribution over latent variables z, typically assumed to be Gaussian:

$$q_\phi(z \mid x) = \mathcal{N}\big(z \mid \mu_\phi(x), \Sigma_\phi(x)\big), \tag{1.12}$$

where $\mu_\phi(x)$ and $\Sigma_\phi(x)$ are functions of x parameterized by the encoder network with parameters ϕ. The decoder, in turn, reconstructs x from samples drawn from this distribution:

$$p_\theta(x \mid z). \tag{1.13}$$

Here, $p_\theta(x \mid z)$ represents the likelihood of reconstructing x given z, parameterized by the decoder with parameters θ.

Training a VAE involves maximizing the likelihood of the data while regularizing the latent space to follow a prior distribution $p(z)$, typically a standard normal $\mathcal{N}(0, I)$. Direct optimization of the marginal likelihood $p_\theta(x)$ is intractable, so instead, VAEs maximize the Evidence Lower Bound (ELBO):

$$L(\phi, \theta) = \mathbb{E}_{q_\phi(z|x)}\big[\log p_\theta(x \mid z)\big] - D_{KL}\big(q_\phi(z \mid x) \parallel p(z)\big). \tag{1.14}$$

The first term is the reconstruction loss, which ensures that the generated data resembles the original input. The second term is the Kullback-Leibler (KL) divergence, which regularizes $q_\phi(z \mid x)$ to be close to the prior $p(z)$, encouraging smooth latent representations.

Since sampling from $q_\phi(z \mid x)$ is non-differentiable, VAEs use the reparameterization trick to enable backpropagation. Instead of sampling z directly, they rewrite:

$$z = \mu_\phi(x) + \epsilon \cdot \sigma_\phi(x), \quad \epsilon \sim \mathcal{N}(0, I). \tag{1.15}$$

This allows gradients to flow through μ_ϕ and σ_ϕ during training.

VAEs have become a popular choice for 3D generation tasks due to their reconstruction-based training objective, which leads to stable and predictable convergence. One of the earliest approaches in this direction, proposed by Brock et al. [5], applies 3D convolutional VAEs to learn probabilistic distributions over voxelized object shapes. This direct

modeling of volumetric data enables the generation of coherent and diverse 3D forms.

To extend beyond simple voxel grids, SDM-Net [23] introduces a part-based mesh generation framework. Their method decomposes objects into deformable components and employs a dual-VAE architecture: one network learns a latent representation for individual parts, while the other captures the overall object structure by composing these parts. Building on this part-aware philosophy, TMNet [24] further enhances generative capabilities by introducing texture synthesis. It predicts texture maps for each mesh component, enabling the creation of richly detailed 3D models.

The flexibility of VAEs also allows their adaptation to other 3D representations. For example, point cloud generation has been explored using VAEs that preserve permutation invariance and spatial consistency [31]. More recently, neural radiance fields (NeRFs) have been integrated into the VAE framework [35], enabling probabilistic modeling of view-dependent appearance and geometry.

Overall, the inherent stability and expressiveness of VAEs make them a compelling choice for learning structured, high-fidelity 3D representations across a wide range of data formats.

1.4.3. Diffusion Models

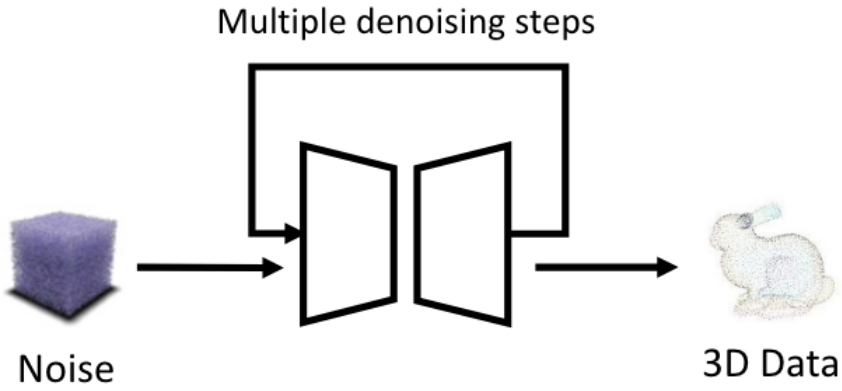


Figure 1.7: Simplified working scheme of a diffusion model, illustrating the denoising loop from noise to the final output. The diffusion model workflow is indipendent of the modality of data employed.

A diffusion model [16, 28, 59] is a generative model that learns to create data by gradually transforming noise into structured outputs through a sequence of denoising steps.

Inspired by non-equilibrium thermodynamics [58], diffusion models systematically reverse a stochastic process that progressively corrupts data into noise. Diffusion models rely on the idea of gradual refinement. Instead of learning to generate an entire sample at once (as in GANs [25] or VAEs), a diffusion model learns a sequence of transformations that reconstruct data from a noisy state. The process consists of two main phases: **forward diffusion** (noise addition) and **reverse diffusion** (denoising process).

In the forward diffusion, the model progressively adds small amounts of noise to a real sample over many steps, slowly destroying its structure. This mimics a diffusion process in physics, where particles spread out and become more disordered over time. After many steps, the original data distribution is transformed into pure Gaussian noise.

While in the reverse diffusion, the model learns to reverse the noise addition process step by step, gradually refining the sample. Starting from pure noise, it applies a learned denoising function iteratively, reducing randomness and restoring meaningful structure at each step. By the final step, the noise has been completely removed, leaving a generated sample that resembles real data.

Forward Diffusion Process

The forward process in DDPMs is a Markovian process that progressively adds Gaussian noise to a data sample over T time steps. This process ensures that the sample is transformed into a nearly Gaussian distribution by the final step.

Let $x_0 \sim q(x_0)$ be a real data sample from the data distribution $q(x)$. The forward diffusion process generates a sequence $x_1, x_2, \ldots, x_T$ by adding noise:

$$q(x_t \mid x_{t-1}) = \mathcal{N}(x_t; (1-\beta_t)x_{t-1}, \beta_t I) \tag{1.16}$$

where:

- β_t is a variance schedule with small values, controlling the noise added at each step.
- I is the identity matrix.

Through recursion, we can derive a direct formulation:

$$q(x_t \mid x_0) = \mathcal{N}(x_t; \bar{\alpha}_t x_0, (1-\bar{\alpha}_t)I) \tag{1.17}$$

where:

- $\alpha_t = 1 - \beta_t$,

- $\bar{\alpha}_t = \prod_{s=1}^{t} \alpha_s$ (cumulative product of noise factors).

A key property is that we can sample x_t directly given x_0 by:

$$x_t = \bar{\alpha}_t x_0 + \sqrt{1 - \bar{\alpha}_t}\epsilon, \quad \epsilon \sim \mathcal{N}(0, I) \tag{1.18}$$

This means that after enough steps, x_T is nearly an isotropic Gaussian.

Reverse Diffusion Process

The goal of DDPMs is to learn a parameterized reverse process that gradually denoises x_T back to x_0.

Since the forward process is a Markov chain with known Gaussian transitions, we assume the reverse process follows a Gaussian form:

$$p_\theta(x_{t-1} \mid x_t) = \mathcal{N}(x_{t-1}; \mu_\theta(x_t, t), \Sigma_\theta(x_t, t)) \tag{1.19}$$

where $\mu_\theta(x_t, t)$ and $\Sigma_\theta(x_t, t)$ are parameterized by a neural network.

In most implementations, the variance $\Sigma_\theta(x_t, t)$ is fixed or parameterized minimally, while $\mu_\theta(x_t, t)$ is learned.

A key insight is that the posterior $q(x_{t-1} \mid x_t, x_0)$ is also Gaussian:

$$q(x_{t-1} \mid x_t, x_0) = \mathcal{N}(x_{t-1}; \tilde{\mu}_t(x_t, x_0), \tilde{\beta}_t I) \tag{1.20}$$

where:

$$\tilde{\mu}_t(x_t, x_0) = \frac{\bar{\alpha}_t - 1}{\bar{\alpha}_t}\beta_t x_0 + \frac{1 - \bar{\alpha}_t}{\bar{\alpha}_t}\left(1 - \bar{\alpha}_{t-1}\right) x_t \tag{1.21}$$

and $\tilde{\beta}_t$ is a function of β_t.

Instead of predicting x_0 explicitly, we reparameterize μ_θ in terms of the noise ϵ that was added during the forward process:

$$\mu_\theta(x_t, t) = \frac{1}{\alpha_t}\left(x_t - \frac{1 - \bar{\alpha}_t}{\alpha_t}\beta_t \epsilon_\theta(x_t, t)\right) \tag{1.22}$$

where $\epsilon_\theta(x_t, t)$ is a neural network that predicts the added noise.

Training DDPMs via Variational Lower Bound

DDPMs aim to learn the underlying data distribution by maximizing the likelihood of the observed data. However, computing this likelihood directly is extremely difficult because the data lives in a very high-dimensional space with complex structure. To handle this, DDPMs instead optimize a related objective called the Evidence Lower Bound (ELBO), which is a mathematically derived lower bound on the true likelihood. By maximizing the ELBO, the model effectively learns to reverse a gradual noising process—starting from pure noise and step-by-step recovering the original data. This process makes training feasible and allows the model to generate high-quality samples by learning how to denoise data at different levels of corruption.

The evidence lower bound (ELBO) is:

$$L_{\text{vlb}} = \mathbb{E}_q \left[\log p_\theta(x_0 \mid x_1) p_\theta(x_1 \mid x_2) \dots p_\theta(x_{T-1} \mid x_T) p(x_T)\, q(x_1, \dots, x_T \mid x_0) \right] \tag{1.23}$$

which can be rewritten as:

$$L_{\text{vlb}} = \mathbb{E}_q \sum_{t=1}^{T} D_{\text{KL}} \left(q(x_t \mid x_0) \parallel p_\theta(x_t \mid x_{t+1}) \right) \tag{1.24}$$

Ho et al. in [28] find out that minimizing KL divergence simplifies as a denoising score-matching loss leads to improved sample quality:

$$L_{\text{simple}} = \mathbb{E}_{x_0, \epsilon, t} \left[\|\epsilon - \epsilon_\theta(x_t, t)\|^2 \right] \tag{1.25}$$

where $\epsilon \sim \mathcal{N}(0, I)$, and x_t is generated using the forward process.

This loss encourages $\epsilon_\theta(x_t, t)$ to correctly predict the noise component in x_t, effectively learning the reverse process.

Sampling from a DDPM

Once trained, we can generate samples by:

- Sampling $x_T \sim \mathcal{N}(0, I)$.
- Iteratively applying the learned reverse process, repeating until x_0 is obtained:

$$x_{t-1} = \mu_\theta(x_t, t) + \sigma_t z, \quad z \sim \mathcal{N}(0, I) \tag{1.26}$$

where σ_t is defined as:

$$\sigma_t = \sqrt{\frac{\beta_t}{1 - \bar{\alpha}_t}} \tag{1.27}$$

Following the success of diffusion models in 2D image synthesis, the generative modeling community has increasingly turned its attention toward 3D shape generation. Diffusion models have been adapted to handle both explicit representations (such as point clouds, meshes, and voxel grids) and implicit ones (such as signed distance fields and radiance fields), enabling a flexible framework for 3D generation.

In the case of point cloud data, several works extend denoising diffusion processes directly to the 3D domain. For example, Cai et al. [8] formulate point cloud generation within a score-based generative modeling framework, learning the data distribution through score matching. PVD [68] addresses both the spatial sparsity of point clouds and the dense nature of voxel grids by coupling the two representations in a hybrid diffusion process, facilitating mutual structural guidance. DPM [40] iteratively denoises corrupted point cloud samples, learning a reverse diffusion process tailored to this modality. Building on these ideas, LION [65] shifts the denoising process into the latent space of point clouds, analogous to latent diffusion for images, yielding improved sample quality and training efficiency.

MeshDiffusion [37] operates on signed distance fields derived from the DMTet [55] representation and interprets mesh optimization as a denoising task. Tetrahedral Diffusion Models extend this principle to tetrahedral meshes by learning over displacements and signed distances within a volumetric grid. SLIDE [41] proposes a sparse latent point diffusion framework for mesh reconstruction, prioritizing computational efficiency while preserving shape fidelity.

Diffusion-based 3D generation has also been explored through multimodal pipelines. For instance, Point·E [48] converts text prompts into 3D point clouds by first synthesizing 2D views using GLIDE [47], then using a conditioned diffusion model to infer corresponding point clouds, achieving robust text-to-3D generation.

Implicit representations have become a compelling alternative for their continuity and compactness. Several models apply diffusion to signed distance fieldsand radiance fields. SSDNeRF [10], DiffRF [44], and Shap·E [30] explore denoising over 3D radiance field parameters, while methods like SDF-Diffusion [36] and SDFusion [12] focus on voxel-based or neural SDF representations.

Additionally, approaches like Rodin [62] and the work by Shue et al. [57] utilize tri-plane features to encode 3D geometry. These features are refined through diffusion and decoded

via either occupancy networks or volumetric rendering to produce final 3D shapes.

Collectively, these advances demonstrate the versatility of diffusion models across a wide spectrum of 3D data formats. By tailoring the denoising process to suit each representation, these models successfully capture complex spatial structures, offering a promising path forward for high-quality and diverse 3D content generation.

2 | Implicit Representations Dataset

To train a generative model for 3D shape synthesis, we first need to define an appropriate representation of 3D data. The choice of representation not only influences the quality and fidelity of the generated outputs, but also affects how the data is encoded, processed, and learned by the model. In this work, we focus on implicit representations—a class of continuous functions that describe 3D shapes in a compact and flexible manner.

This chapter presents the design and training strategy for these implicit shape representations, which will serve as the input data to the generative model introduced in the following chapters. We explain how each 3D object in the dataset is individually encoded using a neural network that learns an occupancy function. This function determines, for any point in 3D space, whether that point lies inside or outside the object. The resulting network is a continuous, differentiable model of the shape that can be queried at arbitrary resolutions.

We begin by describing the concept of implicit shape representations and their training process, including the formulation of the learning objective, data sampling, and inference procedure. We then detail how one such network is trained for each 3D shape in the dataset, enabling the generation of a diverse collection of neural representations. These trained networks will form the basis of the dataset used to train our generative model, which learns to model the distribution over shape representations rather than over explicit 3D geometry directly.

By the end of this chapter, the reader will understand how raw 3D meshes are converted into functional neural representations and how these serve as structured inputs for generative modeling.

2.1. Overview

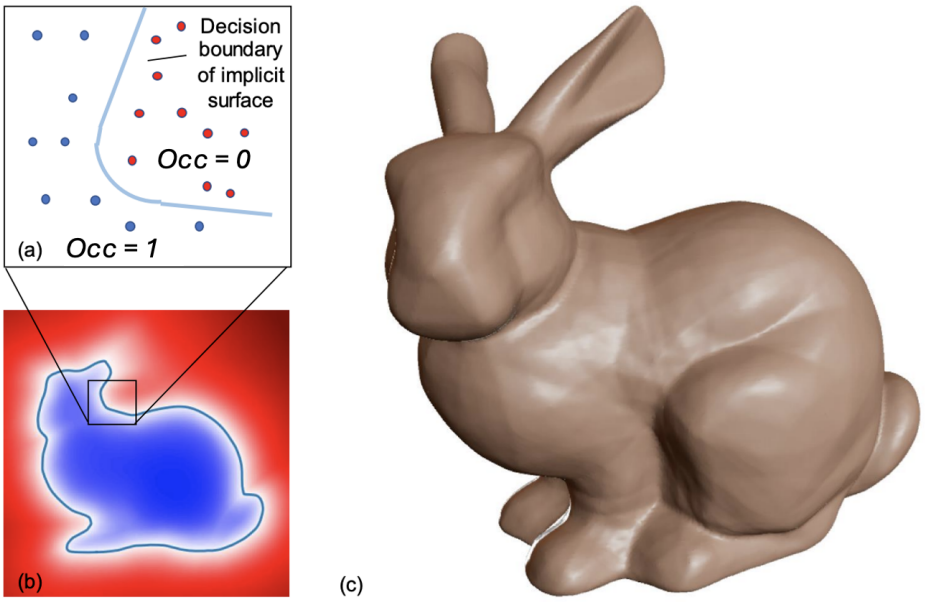


Figure 2.1: Implicit representation applied to the Stanford Bunny: (a) depiction of the underlying implicit surface trained on sampled points inside Occ = 1 and outside Occ = 0 the surface, (b) 2D cross-section of the implicit field, (c) 3D rendering retrivied from Marching Cubes algorithm. Adapted from [49].

A neural network used as an implicit representation of a 3D shape encodes the shape not as an explicit list of vertices and faces, but as a continuous function defined over the 3D space. In this formulation, the network—typically a multi-layer perceptron (MLP) due to its ability to approximate complex, continuous functions while remaining compact in terms of memory and parameters—is trained to map any input point (x, y, z) within a normalized domain to an occupancy probability. This probability indicates whether the point lies inside or outside the object. Values close to 1 suggest that the point is within the object (occupied), and values near 0 indicate it is outside.

During training, the network is provided with examples of 3D shapes along with ground-truth occupancy values at sampled points. The learning objective is to minimize the discrepancy between the predicted occupancy probabilities and the actual occupancy labels. Through this process, the network learns a smooth, continuous decision boundary that implicitly defines the surface of the shape. The surface is often conceptualized as the set of points where the occupancy probability equals a threshold signifying the transition from inside to outside.

Once the networks are trained, they can be used to generate high-resolution 3D models. To do this, one evaluates a network over a dense grid of N points spanning the entire domain. Each point on the grid receives an occupancy value from the network. With those values in hand, the marching cubes algorithm [38] is then applied. Marching cubes is a standard computer graphics technique that processes the grid of occupancy probabilities to extract an isosurface, the set of points where the occupancy equals the chosen threshold.

2.2. Training and Inference

In this section, we describe how the implicit representation of a 3D shape is learned from data and how it can be used to reconstruct the underlying geometry. The goal is to train a neural network to approximate an occupancy function that classifies points in space as either inside or outside the object. This approach allows for a continuous and compact representation of 3D geometry, enabling high-resolution reconstructions without relying on discrete mesh structures. We now formalize the learning objective and outline the training and inference procedures.

Formally, let $g : \Omega^3 \to \{0, 1\}$ be the target ground truth occupancy function representing a 3D shape, where:

$$g(x, y, z) = \begin{cases} 1, & \text{if } (x, y, z) \text{ is inside the object,} \\ 0, & \text{if } (x, y, z) \text{ is outside the object.} \end{cases} \tag{2.1}$$

Consider a training dataset $\mathbf{D}$ comprising:

- Input points $\mathbf{X} = \{(x_1, y_1, z_1), (x_2, y_2, z_2), \ldots, (x_n, y_n, z_n)\} \subset \Omega^3$
- Occupancy labels $\mathbf{o} = \{o_1, o_2, \ldots, o_n\}$, where $\forall i \in \{1, \ldots, n\}$:

$$o_i = g(x_i, y_i, z_i) \tag{2.2}$$

The neural network $f_\theta : \Omega^3 \to [0, 1]$ with parameters θ learns to approximate the ground truth occupancy function such that

$$\forall (x_i, y_i, z_i) \in \mathbf{X}, \quad f_\theta(x_i, y_i, z_i) \approx o_i \tag{2.3}$$

The network minimizes the binary cross-entropy loss:

$$\mathcal{L}(\theta) = \frac{1}{n} \sum_{i=1}^{n} \left[\text{BCE} \left(f_\theta(x_i, y_i, z_i), o_i \right) \right] \tag{2.4}$$

where BCE is defined as:

$$\text{BCE} = - \left(y \log \hat{y} + (1 - y) \log(1 - \hat{y}) \right), \tag{2.5}$$

where:

- y_i is the true label (0 or 1) for sample i,
- $\hat{y}_i$ is the predicted probability for sample i,
- N is the total number of samples in the dataset.

The binary cross-entropy loss (also known as log loss) measures the dissimilarity between predicted probabilities and true binary labels.

The training is performed using stochastic gradient descent or a variant such as Adam [32], and gradients are computed using backpropagation [54]. The training data is constructed from 3D mesh models, which provide a watertight surface description of the object geometry. From each mesh, a set of points $\{\mathbf{x}_i\}_{i=1}^{N}$ is sampled within a bounding volume that contains the object. To evaluate the generalization ability of the model and prevent overfitting to the training data, the sampled points are split into a training set and a validation set. The training set is used to optimize the parameters of the MLP, while the validation set is used to periodically assess performance on unseen points during training. This validation step is crucial for monitoring whether the model is merely memorizing the training data or is instead learning a robust approximation of the underlying occupancy function. Metrics such as validation loss or accuracy can be used to implement early stopping or hyperparameter tuning, ensuring that the final model generalizes well to new, unobserved inputs rather than simply performing well on the training set.

Differently from other works [49] where one single network is trained on several 3D shapes, we train one network for each shape. Infact, considering a collection of datasets $\mathcal{D} = \{D_1, D_2, \ldots, D_M\}$, for each dataset D_i, we independently learn a specialized neural network $f_{\theta_i} : \Omega^3 \to [0, 1]$. In order to leverage the parallel computation capabilities of modern GPUs, multiple implicit representations are trained on the same GPU to speed up the whole process.

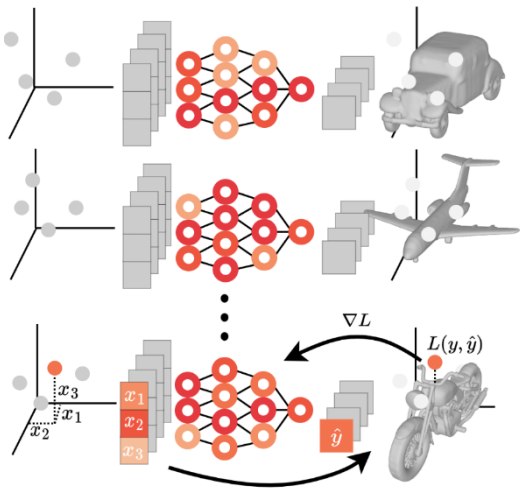


Figure 2.2: Each 3D shape in the dataset is paired with a neural network that serves as its implicit representation. These networks are trained to approximate a continuous occupancy function, allowing the reconstruction of the original geometry from arbitrary 3D query points.

For inference, in order to reconstruct a 3D object from a trained implicit representation, sample N points uniformly:

$$\mathbf{S} = \{(x_1, y_1, z_1), \ldots, (x_N, y_N, z_N)\} \subset \Omega^3 \tag{2.6}$$

Apply the learned neural network f_θ to predict occupancy for sampled points:

$$\hat{o}_i = f_\theta(x_i, y_i, z_i), \quad \forall (x_i, y_i, z_i) \in \mathbf{S} \tag{2.7}$$

Define a binary occupancy threshold $\tau \in [0, 1]$:

$$\text{Occupancy}(x_i, y_i, z_i) = \begin{cases} 1, & \text{if } \hat{o}_i \geq \tau \\ 0, & \text{otherwise} \end{cases} \tag{2.8}$$

Apply the Marching Cubes algorithm [38] to the occupancy points to reconstruct the 3D mesh.

2.3. Implicit Representations Architecture

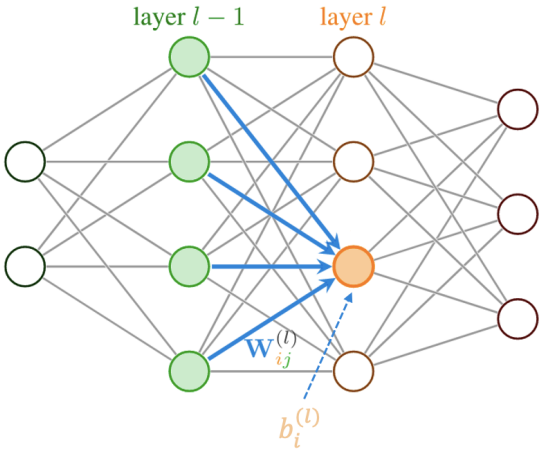


Figure 2.3: Multilayer perceptron and which learnable parameters are related to the output value of a neuron.

In this work, the architecture of the implicit representations is always a multilayer perceptron. A multilayer perceptron (MLP) is a type of artificial neural network that forms the foundation for many modern machine learning models. The MLP is particularly known for its ability to approximate complex nonlinear functions, making it a universal function approximator under certain conditions.

The MLP is composed of multiple layers of neurons, organized into an **input layer**, one or more **hidden layers**, and an **output layer**. Each neuron in a given layer is connected to all neurons in the subsequent layer, forming a fully connected architecture.

Formally, let $\mathbf{x} \in \mathbb{R}^d$ denote the input vector. An MLP with L layers (excluding the input layer) processes the input through a sequence of transformations. For each layer $l = 1, \ldots, L$, the output $\mathbf{h}^{(l)}$ is defined as:

$$\mathbf{h}^{(l)} = \phi^{(l)}\left(\mathbf{W}^{(l)}\mathbf{h}^{(l-1)} + \mathbf{b}^{(l)}\right) \tag{2.9}$$

where $\mathbf{h}^{(0)} = \mathbf{x}$, $\mathbf{W}^{(l)} \in \mathbb{R}^{n_l \times n_{l-1}}$ is the weight matrix, $\mathbf{b}^{(l)} \in \mathbb{R}^{n_l}$ is the bias vector, and $\phi^{(l)}$ is a nonlinear activation function applied element-wise. The final layer output $\mathbf{h}^{(L)}$ serves as the network's prediction.

The nonlinearity introduced by the activation function ϕ is essential. Without it, the network would be equivalent to a single linear transformation regardless of its depth.

Common choices for ϕ include the Rectified Linear Unit (ReLU):

$$\phi(z) = \max(0, z) \tag{2.10}$$

the sigmoid function:

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{2.11}$$

and the hyperbolic tangent:

$$\tanh(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}} \tag{2.12}$$

Training an MLP involves minimizing a loss function $\mathcal{L}$ over a dataset, adjusting weights and biases using gradient descent and the backpropagation algorithm. Backpropagation efficiently computes gradients using the chain rule of calculus, enabling the network to learn from data through successive updates.

A key theoretical result related to MLPs is the *Universal Approximation Theorem* [29], which asserts that a feedforward neural network with a single hidden layer containing a finite number of neurons can approximate any continuous function on a compact subset of $\mathbb{R}^n$, provided the activation function is nonconstant, bounded, and continuous. While this does not imply that such networks can always learn such functions in practice, it underscores the representational power of MLPs.

Differently to the standard MLP network, in order to enhance the ability of the MLP to model high-frequency spatial variations, we apply a positional encoding to the spatial input coordinates, inspired by prior work in neural scene representations [43]. Although multilayer perceptrons are theoretically capable of approximating any continuous function, it has been observed in practice that they exhibit a spectral bias towards learning lower-frequency components of a signal [52]. This bias can hinder the network's capacity to accurately model functions with rapid spatial changes. To address this, we adopt a technique wherein the raw input coordinates are first mapped to a higher-dimensional space using a fixed, non-learned transformation composed of periodic functions. This mapping, denoted $\gamma(\cdot)$, transforms each scalar input $p \in [-1, 1]$ into a vector of sine and cosine functions with increasing frequencies:

$$\gamma(p) = \big(\sin(2^0 \pi p), \cos(2^0 \pi p), \ldots, \sin(2^{L-1} \pi p), \cos(2^{L-1} \pi p)\big) \tag{2.13}$$

The encoded representation $\gamma(\mathbf{x})$ is then passed to the MLP in place of the original spatial coordinates $\mathbf{x}$. This transformation enables the network to more effectively model complex structures and fine details by introducing high-frequency components at the input level.

The primary parameters of a multilayer perceptron (MLP) are the weight matrices and bias vectors associated with each of its layers. For a fully connected layer l, the learnable parameters consist of a weight matrix $\mathbf{W}^{(l)} \in \mathbb{R}^{n_l \times n_{l-1}}$ and a bias vector $\mathbf{b}^{(l)} \in \mathbb{R}^{n_l}$, where n_{l-1} and n_l denote the number of input and output neurons for that layer, respectively. The total number of parameters in the layer is thus $n_l \cdot n_{l-1} + n_l$. Across the entire network, the total number of parameters grows as:

$$\sum_{l=1}^{L} (n_l \cdot n_{l-1} + n_l) \tag{2.14}$$

where L is the number of layers. The computational cost of a forward pass through the network is dominated by the matrix-vector multiplications and nonlinear activations in each layer. For a single input, the cost per layer is approximately $\mathcal{O}(n_l \cdot n_{l-1})$, and thus the total cost scales with the sum of these terms across all layers.

The depth (number of layers) and width (number of neurons per layer) of the MLP have a direct and multiplicative effect on both the model's capacity and its computational burden. Increasing depth or width results in more parameters, which can lead to higher expressiveness and better generalization given sufficient data, but also increases memory usage and computational time during training and inference. Additionally, larger models tend to require more training epochs to converge and may be more prone to overfitting without appropriate regularization. Therefore, the choice of architecture involves a trade-off between model complexity, computational efficiency, and generalization performance.

2.4. Symmetries in the Weight Space of an MLP

In general terms, a symmetry of a system refers to a transformation that preserves some intrinsic property of that system. These transformations can be continuous (like rotations or translations) or discrete (such as permutations). Symmetries play a fundamental role in various machine learning contexts. For instance, in image classification tasks, shifting an image does not change the object it depicts, making translations a natural symmetry of the problem. Similarly, in molecular property prediction, the orientation of a molecule in space should not affect its predicted properties, highlighting the importance of rotational invariance. Discrete symmetries often appear in systems involving multiple indistinguish-

able entities, such as particles with no inherent ordering, where permutations leave the system unchanged. They also arise in time-symmetric physical laws, such as Newtonian mechanics or systems in thermodynamic equilibrium, where reversing the direction of time preserves the underlying dynamics.

In the following, we explore the symmetry structure inherent in multilayer perceptrons, focusing on how permutations of hidden units give rise to function-preserving transformations in the parameter space. We also review prior work that leverages these symmetries to improve model understanding, optimization, and generalization.

We define the weight space V of an M-layer multilayer perceptron as the direct sum of the weight matrices and bias vectors from each layer:

$$V := \bigoplus_{m=1}^{M} (W_m \oplus B_m)\,, \tag{2.15}$$

where $W_m \in \mathbb{R}^{d_m \times d_{m-1}}$ is the weight matrix and $B_m \in \mathbb{R}^{d_m}$ is the bias vector associated with the m-th layer of the MLP. This definition simply means that we treat all the parameters of the network as components of a larger vector space, combining the parameters from all layers together.

Importantly, due to the structure of MLPs and the way standard nonlinearities (like ReLU or sigmoid) operate pointwise on each neuron, the order of hidden units within each layer does not affect the function the network computes. In other words, we can permute the neurons of a hidden layer without changing the output of the network, provided we consistently apply the same permutation to the relevant connections in the adjacent layers.

This property leads to a natural symmetry in the parameter space. To formalize this, we introduce a symmetry group G, defined as:

$$G := S_{d_1} \times S_{d_2} \times \cdots \times S_{d_{M-1}}, \tag{2.16}$$

where each S_{d_m} is the symmetric group acting on the d_m hidden units of the m-th layer. Each element $g \in G$ is a tuple of permutations, one for each hidden layer. These permutations are applied to the network's parameters in a specific way that preserves the network's functional behavior.

More concretely, the action of a group element $g = (\tau_1, \ldots, \tau_{M-1})$ on a point $v \in V$ is

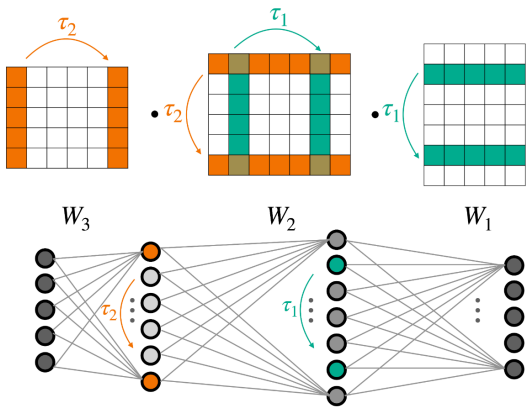


Figure 2.4: Symmetries of weight spaces, illustrated here for a 3-layer MLP. For any pointwise nonlinearity σ, the permutations τ_1, τ_2 can be applied to rows and columns of successive weight matrices of the MLP without altering the function it represents. From [45].

defined by transforming each layer's weights and biases as follows:

$$W_1' = P_{\tau_1}^\top W_1, \quad b_1' = P_{\tau_1}^\top b_1, \tag{2.17}$$

$$W_m' = P_{\tau_m}^\top W_m P_{\tau_{m-1}}, \quad b_m' = P_{\tau_m}^\top b_m \quad \text{for } m = 2, \ldots, M-1, \tag{2.18}$$

$$W_M' = W_M P_{\tau_{M-1}}, \quad b_M' = b_M. \tag{2.19}$$

Here, P_{τ_m} denotes the permutation matrix corresponding to the permutation τ_m. This transformation effectively permutes the output neurons of one layer and the input neurons of the next layer, maintaining consistency across layers so that the function computed by the network remains unchanged.

An important consequence of this setup is that each component space W_m and B_m is mapped back into itself under the action of the group. That is, the group transformations never take parameters out of their respective subspaces. This structure allows us to say that the full weight space V is a direct sum of invariant subspaces, each of which carries a representation of the group G.

The practical implications of this symmetry are both theoretical and applied. Theoretically, it implies that many different points in parameter space represent the same function — they are connected by symmetry transformations. As a result, the function space realized by an MLP is highly redundant: a single function can be represented by multiple, symmetrically equivalent parameter configurations.

From an optimization perspective, this redundancy means that gradient-based methods are navigating a weight space filled with symmetric replicas of equivalent solutions. While this redundancy does not inherently harm training, it can lead to challenges in analyzing the geometry of the loss landscape [13, 18]. Understanding these symmetries can guide the design of more effective learning algorithms and regularization strategies. For instance, symmetry-aware methods [4] may avoid wasting computational effort exploring symmetric equivalents, or could enforce canonical forms to reduce redundancy.

Building on these ideas, the field of Geometric Deep Learning [6] has emerged to provide a unified theoretical and algorithmic framework for designing models that incorporate and exploit such symmetries. Geometric Deep Learning generalizes classical deep learning to non-Euclidean domains—such as graphs, manifolds, and other structured spaces—where symmetries are often more complex than simple translations or permutations. By explicitly encoding invariance or equivariance to symmetry groups, these models can achieve greater generalization from limited data, improved sample efficiency, and increased robustness to transformations.

Significant milestones in this field include graph neural networks (Section 1.3), which respect the permutation symmetry of nodes; equivariant convolutional networks on manifolds, such as spherical CNNs [14] for processing signals on the sphere; and SE(3)-equivariant architectures [22] for modeling 3D molecular and physical systems. These architectures leverage group representations and geometric priors to align the inductive biases of the model with the underlying structure of the data, often leading to state-of-the-art performance in tasks ranging from drug discovery to 3D vision.

In the context of MLPs, these principles motivate the design of permutation-equivariant models that respect neuron-level symmetries or impose constraints on the parameter space to reduce redundancy. As such, the geometric perspective not only provides theoretical clarity but also guides the development of practical algorithms tailored to the symmetry structure inherent in many machine learning problems.

3 | Symmetry-aware Diffusion Model

Generative modeling in the weight space of neural networks potentially could be a powerful paradigm for producing high-fidelity and compact representations of complex signals, through the use of implicit neural fields. Recent work, most notably HyperDiffusion [20], has demonstrated that diffusion models can be effectively trained to generate neural network parameters directly—bypassing the need for explicit surface representations or encoder-decoder structures.

However, a key limitation of HyperDiffusion and similar models lies in their disregard for inherent symmetries in the weight space of MLPs. Specifically, fully connected networks exhibit permutation symmetries: permuting the hidden units of one or more layers does not alter the function computed by the network, provided that the permutations are applied consistently to the relevant input and output dimensions. These symmetries induce large equivalence classes in weight space, where multiple distinct parameter vectors represent the same function. Ignoring this structure leads to redundant modeling, inefficient use of data, and a misalignment between the geometry of the learned generative model and the true underlying function space.

In this chapter, we introduce a generative diffusion model that explicitly accounts for the permutation symmetries of MLP weight spaces. Before presenting the proposed symmetry-aware architecture, we begin by discussing HyperDiffusion in greater detail, treating it as a compelling but symmetry-agnostic baseline. This will serve both to motivate the need for symmetry-aware approaches and to establish a reference point for comparison. Through this perspective, we aim to bridge recent progress in generative modeling with the structural insights offered by representation theory and geometric deep learning.

3.1. Baseline: Hyperdiffusion

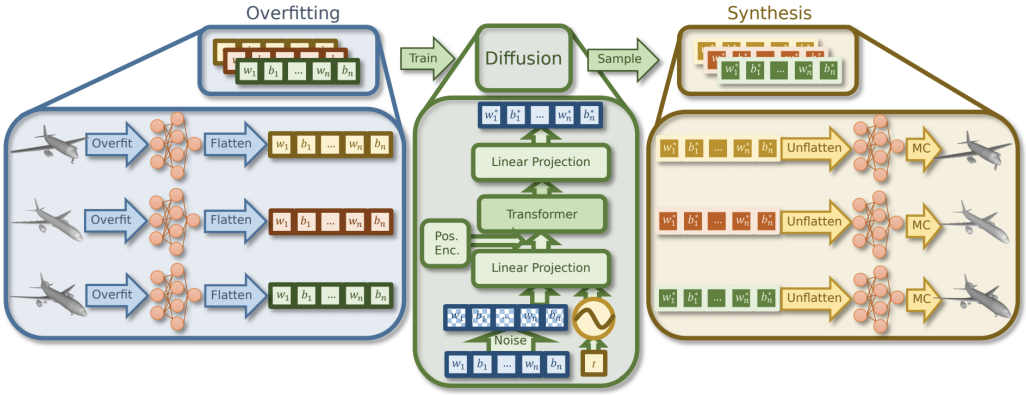


Figure 3.1: Overview of HyperDiffusion from original paper [20].

The central idea behind *HyperDiffusion* is to generate implicit neural representations—specifically, neural fields parameterized by multilayer perceptrons (MLPs)—through a denoising diffusion process applied directly in the weight space of these networks. This approach represents a significant departure from conventional generative modeling techniques used in the 3D and 4D domains, which typically rely on explicit signal representations such as voxel grids, point clouds, or surface meshes, or operate in latent spaces learned through encoder-decoder architectures. In contrast, HyperDiffusion operates in the space of neural function parameters themselves, modeling the distribution over MLP weights that define continuous fields.

The methodology is organized into two primary phases. In the first phase, each training sample is independently represented by overfitting a compact, fully connected MLP. These MLPs are trained independently, without weight sharing across the dataset, which allows the network to specialize for each instance and to achieve a high-fidelity approximation of fine geometric details. Techniques such as sinusoidal positional encoding (Eq: 1.5) and ReLU activations (Eq: 2.10) are employed to improve the MLPs' ability to model high-frequency components. Once trained, each network is flattened into a single vector $\theta_i \in \mathbb{R}^D$, where D is the total number of parameters, thus transforming the collection of MLPs into a dataset in weight space.

The second phase of the pipeline constructs a generative model over this dataset of weight vectors by learning a diffusion process. The model defines a forward process in which Gaussian noise is incrementally added to the clean weights θ_0 and θ_t is the noisy version at timestep t. The reverse process is parameterized by a Transformer-based neural network

that predicts either the original parameters θ_0 or the added noise, depending on the choice of training objective.

During generation, the model begins from a random Gaussian vector $\theta_T \sim \mathcal{N}(0, \mathbf{I})$ and iteratively applies the learned denoising process to sample from the modeled distribution. The resulting MLP, parameterized by $\hat{\theta}_0$, defines a novel neural field. To extract a surface from this field, the function is evaluated over a dense spatial grid, and a mesh is reconstructed using the Marching Cubes algorithm [38]. This enables the synthesis of continuous, high-resolution 3D or 4D representations directly from learned weight-space dynamics.

3.1.1. Transformer Model for Weight-Space Denoising

The core denoising model used in *HyperDiffusion* is a Transformer-based neural network that operates directly in the space of MLP weights. Its primary function is to learn a reverse diffusion process that maps noised neural network parameters back to clean, functionally meaningful weights. Rather than relying on explicit signal representations or encoder-decoder autoencoders, this model treats the parameter vectors themselves as the object of generation.

Let $\mathbf{x} \in \mathbb{R}^D$ denote a noised MLP parameter vector obtained at a given timestep of the diffusion process. The first step in the forward pass is to partition $\mathbf{x}$ into T non-overlapping chunks,

$$\mathbf{x} = \text{concat}(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_T), \quad \mathbf{x}_t \in \mathbb{R}^{d_t}, \quad \sum_{t=1}^{T} d_t = D, \tag{3.1}$$

where each chunk $\mathbf{x}_t$ corresponds to a subset of the MLP's weights and/or biases, determined by a user-defined split policy. This policy may group parameters by layer, or evenly segment the vector into fixed-size tokens.

Each chunk $\mathbf{x}_t$ is then independently projected into a shared embedding space of dimension d_{emb} using a dedicated encoder $f_t^{\text{enc}} : \mathbb{R}^{d_t} \to \mathbb{R}^{d_{\text{emb}}}$. This yields a sequence of embedded tokens,

$$\mathbf{z}_t = f_t^{\text{enc}}(\mathbf{x}_t), \quad \text{for } t = 1, \ldots, T, \tag{3.2}$$

which are stacked to form a tensor $\mathbf{Z} \in \mathbb{R}^{T \times d_{\text{emb}}}$. To provide the model with positional context, a learnable positional embedding $\mathbf{P} \in \mathbb{R}^{T \times d_{\text{emb}}}$ is added to the input embeddings:

$$\tilde{\mathbf{Z}} = \mathbf{Z} + \mathbf{P}. \tag{3.3}$$

To condition the denoising behavior on the current timestep $t \in \mathbb{N}$ of the diffusion process, the model introduces a continuous scalar embedding. This is constructed by applying a sinusoidal frequency encoding to the timestep:

$$\mathbf{e}_t = \left[\sin(2^k t), \cos(2^k t)\right]_{k=0}^{d_s/2-1} \in \mathbb{R}^{d_s}, \tag{3.4}$$

where d_s is the embedding dimension of the scalar token. This timestep embedding is either appended as an additional token to the input sequence or concatenated to the flattened input vector before tokenization.

The core of the architecture consists of a Transformer with L layers. Each layer alternates between multi-head self-attention and a position-wise feedforward network, wrapped in residual connections and layer normalization. Let $\mathbf{H}^{(0)} = \tilde{\mathbf{Z}}$ be the input to the Transformer. The computation proceeds as follows:

$$\mathbf{A}^{(\ell)} = \text{SelfAttention}(\text{LayerNorm}(\mathbf{H}^{(\ell)})), \tag{3.5}$$

$$\mathbf{H}^{(\ell+1)} = \mathbf{H}^{(\ell)} + \mathbf{A}^{(\ell)} + \text{MLP}(\text{LayerNorm}(\mathbf{H}^{(\ell)} + \mathbf{A}^{(\ell)})), \tag{3.6}$$

for $\ell = 0, \ldots, L-1$, yielding a final sequence representation $\mathbf{H}^{(L)} \in \mathbb{R}^{T \times d_{\text{emb}}}$.

Each token embedding in $\mathbf{H}^{(L)}$ is then decoded by a corresponding MLP decoder $f_t^{\text{dec}} : \mathbb{R}^{d_{\text{emb}}} \to \mathbb{R}^{d_t}$ to recover a prediction $\hat{\mathbf{x}}_t$ for the original parameter chunk:

$$\hat{\mathbf{x}}_t = f_t^{\text{dec}}(\mathbf{h}_t), \quad \text{for } t = 1, \ldots, T, \tag{3.7}$$

where $\mathbf{h}_t \in \mathbb{R}^{d_{\text{emb}}}$ denotes the corresponding row of $\mathbf{H}^{(L)}$. The final reconstructed parameter vector is obtained by concatenation:

$$\hat{\mathbf{x}} = \text{concat}(\hat{\mathbf{x}}_1, \hat{\mathbf{x}}_2, \ldots, \hat{\mathbf{x}}_T). \tag{3.8}$$

This architecture is designed to be both modular and expressive. The decision to tokenize the parameter vector allows the model to scale to large MLPs and to isolate structurally distinct parts of the network, such as layers or parameter types. The use of per-token encoders and decoders enables flexibility in handling variable-sized chunks, while the Transformer backbone allows the model to capture non-local dependencies and global structure across the parameter space. The sinusoidal embedding of the timestep provides temporal conditioning without the need for recurrent mechanisms or time-specific layers.

This formulation offers several advantages. First, by operating directly in weight space,

the method circumvents the need for encoder-decoder architectures or intermediate latent manifolds, simplifying the pipeline and eliminating potential information bottlenecks. Furthermore, the architecture is agnostic to the spatial dimensionality of the input. The same pipeline can model 3D or 4D fields without architectural changes; only the positional encoding scheme is adapted to the dimensionality of the domain. The use of a Transformer is particularly well suited to this setup, as it can model long-range dependencies and structured correlations within the weight vector—especially across parameters belonging to different layers.

In contrast to latent diffusion models, which often require a compact representation space and suffer from decoder fidelity limitations, *HyperDiffusion* treats the MLP weights themselves as the generative target. This eliminates the risk of latent collapse or poor decoder generalization. Moreover, because the output is a continuous function, mesh resolution can be adapted at inference time without retraining.

Nevertheless, one notable limitation of this architecture is its ignorance of the permutation symmetries inherent in the weight space of MLPs. Since permuting the neurons of a hidden layer—along with corresponding adjustments to adjacent weight matrices—does not alter the function the MLP computes, many distinct parameter vectors represent the same underlying function. The current model treats these functionally equivalent representations as distinct and as a result, it must learn to be invariant to such permutations implicitly, which may reduce sample efficiency or degrade generative consistency.

The model also does not incorporate any explicit geometric or topological priors about the surfaces it represents. During training, it learns purely from MLP weight vectors and does not explicitly reason about shape connectivity, surface smoothness, or valid topology. This absence of geometric regularization may limit the quality of the generated surfaces, especially early in the reverse diffusion trajectory when the sampled weights may not yet encode a valid field.

Finally, the computational demands of training are nontrivial. The diffusion model is trained for thousands of iterations with long noise schedules and a large Transformer backbone. This results in high memory and runtime requirements, which may preclude its use in resource-constrained settings.

To address these challenges, in the following sections we will introduce symmetry-aware models that explicitly incorporate MLP permutation invariance into the learning process. While this represents a promising direction, it is not the only possible path forward. Alternative approaches may involve different architectural choices, training objectives, or inductive priors tailored to the structure of implicit shape representations.

3.2. MLPs as graphs

In multilayer perceptrons, symmetries naturally arise due to the exchangeability of hidden neurons within a layer. Specifically, for any permutation of neurons in a hidden layer, the output of the network remains unchanged if the associated weights and biases are permuted consistently. This defines a symmetry group acting on the weight space of MLPs, with each hidden layer $\ell \in \{1, \ldots, L-1\}$ having an associated symmetric group S_{d_ℓ}, where d_ℓ is the number of neurons in that layer.

The primary contribution of the work is to develop a generative model with a neural architecture that is *equivariant* to these symmetry transformations. Given a representation of an MLP (e.g., its weights and biases encoded in some structured form), the learned function Φ should satisfy:

$$\Phi(g \cdot \theta) = g \cdot \Phi(\theta), \quad \forall g \in G, \tag{3.9}$$

where θ denotes the MLP's parameter vector, and $g \cdot \theta$ is the group action applied to the parameters. This ensures that symmetrically equivalent networks (i.e., those differing only by a permutation of neurons) are mapped to symmetrically equivalent representations by the model.

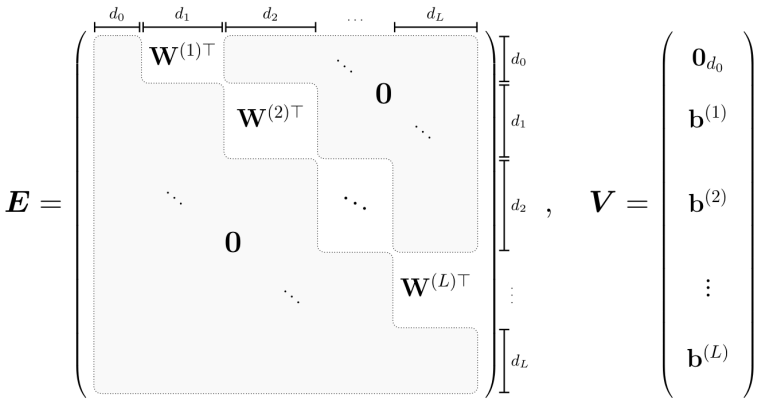


Figure 3.2: Neural graph representation of an MLP with L layers, showing edge and vertex matrices.

To facilitate this, the method constructs a graph representation of the MLP, referred to as a *neural graph*, in which each neuron corresponds to a node, and each weight to a directed edge. Importantly, this graph is invariant to the ordering of neurons within each layer, provided that the permutation is applied consistently to the source and target indices of the edges. That is, any permutation $g \in G$ of the hidden neurons induces a relabeling of

the graph's nodes, without changing the structural identity of the graph:

$$(V, E) \mapsto (\rho(g)V, \rho(g)^\top E\rho(g)), \tag{3.10}$$

where V is the matrix of node features, E is the tensor of edge features, and $\rho(g)$ is the block-diagonal permutation matrix encoding the neuron-wise permutation.

The neural graph is then processed by an equivariant architecture that preserves equivariance under such permutations by construction. This means that the output representation of the MLP remains consistent under neuron relabeling, and therefore reflects its function rather than its parameter instantiation.

By treating the input as a structured object and enforcing equivariance to its natural symmetry group, this framework avoids the redundancy and ambiguity that arise from arbitrary neuron orderings. This is crucial in applications such as generative modeling or meta-learning over MLPs, where function-level understanding and symmetry-consistent operations are essential.

3.3. Permutation-aware architecture

In this section, we describe the architectures used to learn and generate representations of MLPs in a permutation-aware manner. The overall pipeline encodes an MLP as a graph, applies a relational graph-based model to extract meaningful features, and projects these features back to the weight space of the target MLP. This process is conditioned on a diffusion timestep, enabling integration with diffusion-based generative models over neural network parameters.

3.3.1. Dense GNN

We introduce a neural network architecture designed to model the space of multilayer perceptrons by learning transformations over their parameters using a graph-based representation as discussed in the previous section 3.2. At its core, the model treats each MLP as a structured graph and applies a relational graph neural network (Section 1.3) to this representation. The resulting node and edge embeddings are then projected back into parameter space to predict a new set of MLP weights. This process is conditioned on a diffusion timestep, enabling the model's integration within diffusion-based generative frameworks.

Let $x \in \mathbb{R}^{B\times D}$ be a batch of flattened weight vectors, where B is the batch size and D is

the total number of parameters of an MLP defined by a layer layout

$$\ell = [d_0, d_1, \ldots, d_L],$$

with d_0 input neurons, d_L output neurons, and $L-1$ hidden layers. The goal of the model is to predict a new set of MLP parameters $\hat{x} \in \mathbb{R}^{B\times D}$ from an input x and a timestep index $t \in \{0, \ldots, T\}$.

The first step is to transform each MLP into a directed graph $G = (V, E)$, where nodes represent neurons and directed edges represent weights between layers. The total number of nodes is

$$N = \sum_{i=0}^{L} d_i. \tag{3.11}$$

For each adjacent layer pair (d_i, d_{i+1}), edges are constructed between all pairs of neurons from layer i to layer $i+1$. The edge index tensor $\texttt{edge\_index} \in \mathbb{N}^{2\times|E|}$ is formed via Cartesian products:

$$E_i = \{(u, v) \mid u \in \text{layer } i, v \in \text{layer } i+1\}. \tag{3.12}$$

The full edge index is then given by

$$\texttt{edge\_index} = \bigcup_{i=0}^{L-1} E_i, \tag{3.13}$$

A graph constructor module $\mathcal{G}$ is used to project the flattened MLP weights $x \in \mathbb{R}^{B\times D}$ into a pair of initial node and edge features:

$$(\mathbf{v}, \mathbf{e}) = \mathcal{G}(x), \tag{3.14}$$

where $\mathbf{v} \in \mathbb{R}^{B\times N\times d_{\text{hid}}}$ are node features and $\mathbf{e} \in \mathbb{R}^{B\times|E|\times d_{\text{hid}}}$ are edge features. Each feature dimension is of size d_{hid}, which serves as the hidden dimension throughout the GNN backbone.

To incorporate the timestep $t \in \{0, \ldots, T-1\}$, an embedding layer maps each scalar index to a vector:

$$\mathbf{t}_{\text{emb}} = \text{Embed}(t) \in \mathbb{R}^{B\times d_{\text{hid}}}. \tag{3.15}$$

This embedding is added to the node and edge features:

$$\mathbf{v} \leftarrow \mathbf{v} + \mathbf{t}_{\text{emb}}, \tag{3.16}$$

$$\mathbf{e} \leftarrow \mathbf{e} + \mathbf{t}_{\text{emb}}. \tag{3.17}$$

The node and edge features are processed by a message-passing GNN. Let $\mathcal{F}_{\text{GNN}}$ denote the GNN:

$$(\mathbf{v}', \mathbf{e}') = \mathcal{F}_{\text{GNN}}(\mathbf{v}, \mathbf{e}, \texttt{edge\_index}), \tag{3.18}$$

where the output tensors $\mathbf{v}' \in \mathbb{R}^{B\times N\times d_{\text{hid}}}$, $\mathbf{e}' \in \mathbb{R}^{B\times |E|\times d_{\text{hid}}}$ represent updated features after multiple rounds of message passing and aggregation.

The processed features are linearly projected back to one-dimensional outputs:

$$\mathbf{v}_{\text{out}} = \text{Proj}_{\text{node}}(\mathbf{v}') \in \mathbb{R}^{B\times N\times 1}, \tag{3.19}$$

$$\mathbf{e}_{\text{out}} = \text{Proj}_{\text{edge}}(\mathbf{e}') \in \mathbb{R}^{B\times |E|\times 1}. \tag{3.20}$$

The projections are applied via two-layer MLPs with ReLU activations, shared across all nodes and edges respectively.

Finally, the projected scalar edge weights and node biases are decoded back into a flattened MLP parameter vector using a deterministic inverse mapping. Let $\texttt{Flatten}(\cdot)$ denote this reassembly procedure:

$$\hat{x} = \texttt{Flatten}(\mathbf{v}_{\text{out}}, \mathbf{e}_{\text{out}}) \in \mathbb{R}^{B\times D}. \tag{3.21}$$

This operation reorganizes the edge weights and node biases according to the original MLP layer structure, optionally applying per-layer normalization using stored mean and variance statistics.

The full forward pass is thus summarized as:

$$\hat{x} = \texttt{Flatten}\left(\text{Proj}_{\text{node}}\left(\mathbf{v}'\right), \text{Proj}_{\text{edge}}\left(\mathbf{e}'\right)\right), \quad \text{where } (\mathbf{v}', \mathbf{e}') = \mathcal{F}_{\text{GNN}}(\mathbf{v}+\mathbf{t}, \mathbf{e}+\mathbf{t}). \tag{3.22}$$

This architecture enables flexible, permutation-equivariant modeling of MLP weight spaces via graph representations, and supports conditional generation of MLP parameters as required by downstream tasks such as denoising diffusion.

With regard to the backbone used, we describe the forward pass of the PNA-based graph neural network architecture [15] used as a backbone in the context of learning over MLP parameter graphs. The architecture follows the general GNN design pattern of message

passing, feature transformation, normalization, and edge updates. Let us denote the input node features as $x \in \mathbb{R}^{N \times d_{\text{in}}}$ and the edge features as $e \in \mathbb{R}^{M \times d_e}$, where N is the number of nodes and M the number of edges in the graph. The edge indices are provided in $\texttt{edge\_index} \in \mathbb{N}^{2 \times M}$.

The forward pass proceeds as follows. For each layer $\ell \in \{1, \ldots, L\}$, where L is the number of GNN layers, the node embeddings $x^{(\ell)}$ are updated via a PNAConv module:

$$x^{(\ell+1)} = \texttt{PNAConv}^{(\ell)}(x^{(\ell)}, \texttt{edge\_index}, e^{(\ell)}), \tag{3.23}$$

where edge features $e^{(\ell)}$ may also be updated depending on the configuration.

Each `PNAConv` layer performs message passing by aggregating transformed messages from neighbors. The message m_{ij} sent from node j to node i is computed as:

$$m_{ij} = h_\theta^{(\ell)}(x_i, x_j, e_{ij}), \tag{3.24}$$

where $h_\theta^{(\ell)}$ is an MLP whose input includes the concatenation of source and target node embeddings and possibly the edge feature.

Messages from all neighbors are aggregated using a composite scheme of aggregators and scalers:

$$a_i = \texttt{Aggregate}(\{m_{ij}\}_{j \in \mathcal{N}(i)}), \tag{3.25}$$

where aggregation includes statistical functions (e.g., mean, min, max, std) scaled according to the node degree.

The output embedding is then:

$$x_i^{(\ell+1)} = \gamma_\theta(x_i^{(\ell)}, a_i), \tag{3.26}$$

where γ_θ is another MLP that combines the original embedding and the aggregated message.

Between layers, optional activation functions, normalization, and dropout are applied:

$$x_i^{(\ell+1)} \leftarrow \texttt{Dropout}(\texttt{Norm}(\sigma(x_i^{(\ell+1)}))). \tag{3.27}$$

If edge updates are enabled, edge features are also updated at each layer (except possibly

the last):

$$e_{ij}^{(\ell+1)} = \texttt{EdgeMLP}(x_i^{(\ell+1)}, x_j^{(\ell+1)}, e_{ij}^{(\ell)}). \tag{3.28}$$

This design allows for expressive message passing while preserving compatibility with edge-conditioned processing, degree-aware scaling [15], and multilayer feature mixing.

3.3.2. Relational Transformer

We describe the attention mechanism of the *Relational Transformer* [17], a Transformer-based architecture designed to process graphs where both node and edge features are critical. The architecture generalizes standard attention by integrating directed edge features into the attention computation and allowing for layer-wise updates of both node and edge representations.

Fistly, given node features $X \in \mathbb{R}^{B\times N\times d_{\text{node}}}$ and edge features $E \in \mathbb{R}^{B\times N\times N\times d_{\text{edge}}}$, linear projections are applied:

$$Q^n, K^n, V^n = W^{\text{node}}X, \quad Q^e, K^e, V^e = W^{\text{edge}}E, \tag{3.29}$$

where $Q^n, K^n, V^n \in \mathbb{R}^{B\times N\times d}$ and $Q^e, K^e, V^e \in \mathbb{R}^{B\times N\times N\times d}$.

These are combined to define edge-specific queries, keys, and values:

$$Q_{ij} = Q_i^n + Q_{ij}^e, \quad K_{ij} = K_j^n + K_{ij}^e, \quad V_{ij} = V_j^n + V_{ij}^e. \tag{3.30}$$

Attention scores are computed using scaled dot-products:

$$A_{ij} = \frac{1}{\sqrt{d}}\langle Q_{ij}, K_{ij}\rangle, \tag{3.31}$$

followed by softmax normalization over incoming nodes:

$$\alpha_{ij} = \frac{\exp(A_{ij})}{\sum_k \exp(A_{ik})}. \tag{3.32}$$

The output representation for each node is obtained by aggregating value vectors:

$$O_i = \sum_j \alpha_{ij} V_{ij}, \tag{3.33}$$

which is then linearly projected back to the node feature dimension.

This formulation allows the attention mechanism to be sensitive to both the node states and the edge relations between them, enabling rich relational reasoning over neural network graphs.

4 | Latent Diffusion

Directly modeling the distribution of neural network weights with diffusion models presents both theoretical and practical difficulties. The high dimensionality of MLP weight vectors, their sensitivity to permutations of neurons, and the lack of an explicit inductive bias toward functional similarity make weight-space diffusion a brittle approach. Small perturbations in weight space often correspond to disproportionately large changes in the MLP's output function, complicating generative modeling and reducing sample quality.

Initial experiments with symmetry-aware architectures aimed to address these issues by treating weight vectors as structured graphs and incorporating equivariance constraints. However, these models struggled to scale to realistic network sizes due to the large number of edges and vertices in the neural graphs, which imposed heavy computational burdens and limited their expressive capacity. As a result, the generated weights often failed to capture functionally coherent behavior at scale.

To overcome these limitations, this chapter introduces a two-stage generative pipeline based on the latent diffusion paradigm proposed by Rombach et al. [53]. In this framework, generative modeling is performed in a compressed latent space rather than directly in the high-dimensional input space, improving scalability and efficiency. We adapt this approach to the weight space of MLPs: the first stage uses a variational autoencoder, introduced in Section 1.4.2, to map MLP parameter vectors into a compact latent space where distances reflect meaningful functional differences. In the second stage, we train a diffusion model, described in Section 1.4.3, to learn a generative process over these latent codes, which are then decoded back into full MLP weights.

This design offers several key advantages. By training the VAE to reconstruct the functional behavior of MLPs—rather than their exact weight values—the latent representation becomes robust to weight-level noise and neuron permutations. As a result, the latent space becomes a more stable and semantically meaningful domain for diffusion-based generation. Moreover, operating in a lower-dimensional space significantly reduces the computational complexity of diffusion training and sampling.

The rest of this chapter is structured as follows: Section 4.1 introduces the variational

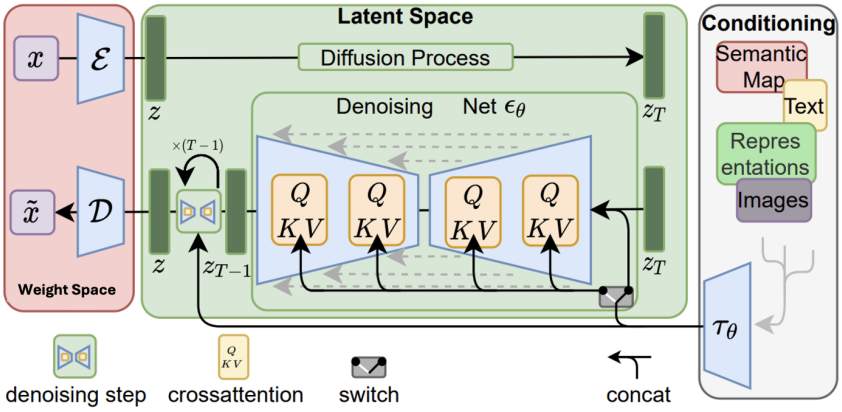


Figure 4.1: Illustration of a Latent Diffusion Model (LDM) architecture, where the input is encoded into a latent space, iteratively denoised through a neural network. Optionally, it is possible to condition with cross-attention the denoising process on various inputs such as semantic maps, text, or images. Adapted from [53].

autoencoder used for function-preserving MLP reconstruction, describing its architecture and training objective. Section 4.2 details how a diffusion model is trained and applied in the latent space of the VAE, enabling the generation of novel MLPs that retain functional coherence.

4.1. A VAE for Functional Weight Reconstruction

The architecture under consideration extends the standard variational autoencoder (VAE) framework to operate directly in the space of neural network weights. Unlike conventional VAEs, which reconstruct the input data itself (e.g., images or point clouds), this model reconstructs the *functional behavior* of an input MLP by matching its outputs on a predefined set of evaluation coordinates.

Formally, let $\theta \in \mathbb{R}^D$ denote the flattened weight vector of an MLP, and let $\mathbf{x} \in \mathbb{R}^{N \times d}$ be a batch of N input coordinates (e.g., 3D points). The encoder $q_\phi(z \mid \theta)$ maps θ to a latent posterior distribution. A latent sample $z \sim q_\phi(z \mid \theta)$ is then decoded via $\hat{\theta} = f_\psi(z)$, producing a new set of MLP weights. To assess reconstruction quality, the model evaluates the original and reconstructed MLPs over the same set of coordinates:

$$\mathbf{y}_{\text{in}} = \text{MLP}_\theta(\mathbf{x}), \quad \mathbf{y}_{\text{out}} = \text{MLP}_{\hat{\theta}}(\mathbf{x}), \tag{4.1}$$

The loss function used in training comprises two terms: a cross-entropy loss between the

predictions of the original and reconstructed MLPs, and the KL divergence between the posterior and a unit Gaussian prior:

$$\mathcal{L}_{\text{VAE}} = \mathbb{E}_{z\sim q_\phi(z|\theta)}\left[\mathcal{L}_{\text{CE}}(\mathbf{y}_{\text{out}}, \mathbf{y}_{\text{in}})\right] + \beta \cdot D_{\text{KL}}\left(q_\phi(z \mid \theta) \,\|\, \mathcal{N}(0, I)\right), \tag{4.2}$$

where β is a scalar value adjusted to weight regularization over reconstruction.

This design differs from traditional VAEs in two critical ways. First, the reconstruction loss is applied not in weight space but in the MLP output space, allowing the model to preserve the function encoded by the original weights rather than their exact parameter values. Second, the comparison is performed using binary cross-entropy between the predicted and target occupancy logits

This functional reconstruction criterion improves robustness allowing the VAE to focus on preserving semantic content rather than memorizing exact weight configurations.

The variational autoencoder is composed of a Transformer-based encoder-decoder pair designed to operate directly on flattened MLP weight vectors. Similarly to *Hyperdiffusion*, the encoder architecture is based on a Transformer that operates over a sequence of tokenized MLP parameters. Each input sample consists of a flattened MLP parameters vector whose structure is determined by a specified layer layout. The encoder first segments this vector into tokens using a predefined split that separates weights and biases per layer. Depending on whether bias and weight parameters are aggregated jointly or separately, the total number of tokens is either equal to the number of layers or twice that number. Each token is projected into a fixed-dimensional embedding space $\mathbb{R}^{d_{\text{emb}}}$ through a token-specific linear projecting layer. These embeddings are then augmented with learned positional embeddings and processed by a standard Transformer stack composed of multi-head self-attention and feedforward layers. The final output of the encoder is a sequence of contextualized embeddings, which can be pooled or aggregated to produce a latent representation for downstream use in the variational autoencoder. This design allows the encoder to attend across all layers of the MLP simultaneously, capturing complex interactions between weights and biases across the architecture. The decoder performs the inverse operation: given a latent vector sampled from this distribution, it replicates it across the token dimension and projects it back into the model embedding space. After adding positional encodings, this sequence is processed by the same architecture of transformer layers as the encoder. Each output token is then passed through a token-specific linear projection head that reconstructs its corresponding segment of MLP weights or biases. The concatenated result yields a complete reconstruction of the original parameter vector.

4.2. Diffusion in the Latent Space of a VAE

Training a diffusion model to generate neural network parameters directly in weight space poses several practical and conceptual challenges. Raw parameter vectors of multilayer perceptrons tend to be high-dimensional, redundant, and structured in ways that are not easily amenable to generative modeling. To address this, we perform diffusion in the latent space of a variational autoencoder, which learns a lower-dimensional representation that preserves the functional properties of the original MLP.

Let $\theta \in \mathbb{R}^D$ denote the vector of MLP weights. The encoder network $q_\phi(z \mid \theta)$ maps these weights to a latent variable $z \in \mathbb{R}^d$, where typically $d \ll D$. The decoder $f_\psi(z)$ reconstructs the MLP weights from the latent sample. Importantly, the VAE is trained such that the decoded weights $\hat{\theta} = f_\psi(z)$ produce the same output function as θ when evaluated on fixed coordinates, rather than matching θ exactly. This ensures that the latent space captures semantically meaningful variations in the function space of neural networks.

During diffusion training, we treat z as the data distribution to model. Specifically, we normalize each latent code via

$$\tilde{z} = \frac{z - \mu}{\sigma}, \tag{4.3}$$

and train a denoising network ε_θ to predict Gaussian noise injected at varying timesteps. At each training iteration, a latent code is perturbed via the forward diffusion process:

$$\tilde{z}_t = \sqrt{\bar{\alpha}_t}\tilde{z} + \sqrt{1 - \bar{\alpha}_t}\varepsilon, \tag{4.4}$$

where $\varepsilon \sim \mathcal{N}(0, I)$ and $\bar{\alpha}_t$ is the cumulative product of noise schedule coefficients. The denoising network is then trained to minimize the mean squared error (MSE) between the predicted noise and the ground-truth:

$$\mathcal{L}_{\text{diff}} = \mathbb{E}_{t,z,\varepsilon} \left\| \varepsilon_\theta(\tilde{z}_t, t) - \varepsilon \right\|^2. \tag{4.5}$$

Performing diffusion in the latent space brings several advantages. First, it reduces the dimensionality of the generative modeling problem, leading to more stable and data-efficient training. Second, by modeling latents instead of raw parameters, the diffusion model inherits the inductive bias of the VAE: any sample generated by the denoising network can be decoded into a valid MLP whose output function is semantically meaningful.

Thus, combining a VAE with a diffusion model enables the synthesis of plausible MLPs

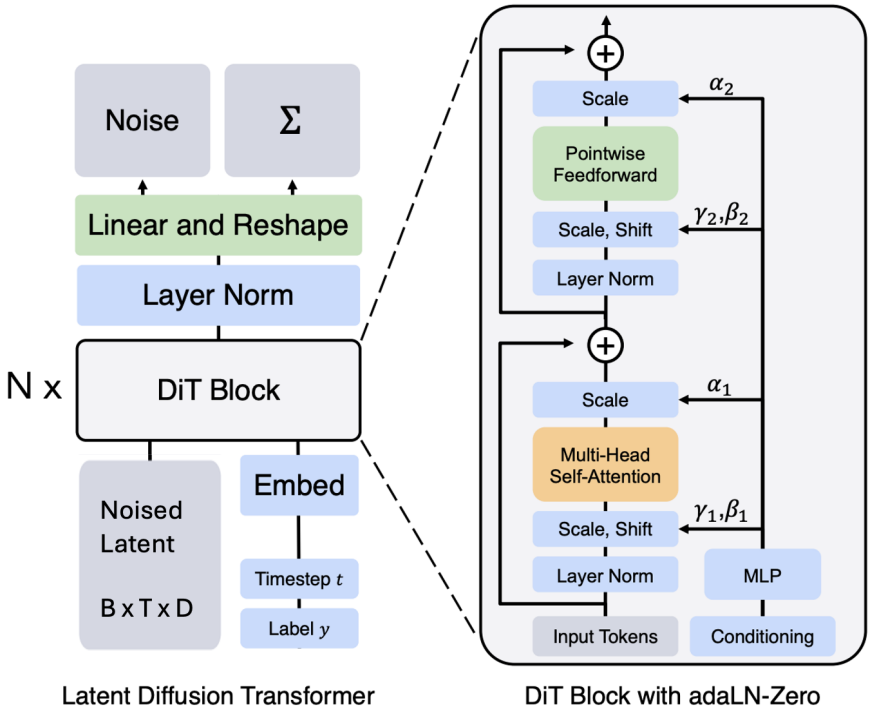


Figure 4.2: The Diffusion Transformer (DiT) architecture. Left: We train conditional latent DiT models. The input latent is already decomposed into a sequence of embedding tokens of the encoded neural network layers and processed by several DiT blocks. Right: Details of the DiT blocks.

not by directly generating weights, but by generating latent codes that decode into functionally coherent neural networks.

4.2.1. Diffusion Transformer in Latent Space

The diffusion model is implemented as a transformer-based network, DiT [50], which operates in the latent space learned by a variational autoencoder. Let $x \in \mathbb{R}^{B \times T \times D}$ denote a sequence of latent tokens obtained from the encoder, where B is the batch size, T the number of tokens, and D the latent dimensionality. The model aims to predict the noise added to these latent variables at various timesteps in a diffusion process.

The input sequence x is first embedded using a linear projection and then summed with a fixed sinusoidal positional embedding:

A separate embedding is computed for the diffusion timestep $t \in \mathbb{N}$ using sinusoidal

embeddings followed by a two-layer MLP:

$$\tau = \text{MLP}(\text{SinCos}(t)) \in \mathbb{R}^H. \tag{4.6}$$

This conditioning vector τ is used to modulate each transformer block via adaptive layer normalization.

Each DiT block contains a self-attention mechanism and a feedforward MLP, both modulated using the conditioning vector. Let $x_\ell \in \mathbb{R}^{B\times T\times H}$ denote the hidden state at layer ℓ. Each block applies two conditional transformations:

$$\text{AdaLN}_{\text{attn}}(\tau) = (\Delta^{(1)}_{\text{shift}}, \Delta^{(1)}_{\text{scale}}, g^{(1)}), \tag{4.7}$$
$$\text{AdaLN}_{\text{mlp}}(\tau) = (\Delta^{(2)}_{\text{shift}}, \Delta^{(2)}_{\text{scale}}, g^{(2)}), \tag{4.8}$$

where the shifts, scales, and gates are learned via a linear transformation of τ.

The output is updated as follows:

$$h^{\text{attn}}_\ell = x_\ell + g^{(1)} \cdot \text{SelfAttn}(\text{LN}(x_\ell) \cdot (1 + \Delta^{(1)}_{\text{scale}}) + \Delta^{(1)}_{\text{shift}}), \tag{4.9}$$
$$x_{\ell+1} = h^{\text{attn}}_\ell + g^{(2)} \cdot \text{MLP}(\text{LN}(h^{\text{attn}}_\ell) \cdot (1 + \Delta^{(2)}_{\text{scale}}) + \Delta^{(2)}_{\text{shift}}) \tag{4.10}$$

This adaptive conditioning mechanism is known as AdaLN-Zero, and it ensures that the influence of τ is applied in a learned and controlled manner.

After the sequence of transformer blocks, a final layer projects the hidden states back to the output space.

By operating in the latent space of a pretrained VAE, the DiT model benefits from a compact and structured representation of the original data. This improves training efficiency and robustness, while enabling high-fidelity generation with fewer steps. Moreover, diffusion in latent space reduces the dimensionality of the denoising problem, making it more tractable for high-resolution or high-dimensional generative tasks.

5 | Results

This chapter presents the experimental setup used to evaluate the different approaches introduced in previous chapters. We begin by describing the dataset employed for training and evaluation, followed by an overview of the loss functions and evaluation metrics used to assess the quality and diversity of the generated 3D shapes. These elements are essential for validating the effectiveness of the proposed methods, ensuring that the generated outputs are both accurate and representative of the target distribution. Finally, we report and analyze the results obtained.

5.1. Dataset

The ShapeNet dataset [9] is a large-scale collection of 3D object models widely used in deep learning and computer vision research. It covers 55 common object categories with about 51,300 unique 3D models, making it a key resource for tasks like 3D reconstruction, shape generation, and classification. The core subset, ShapeNetCore, consists of clean, manually verified models with consistent annotations, ensuring high-quality training data for neural networks. The dataset provides 3D models in multiple formats, including meshes and voxel grids, along with semantic labels and part segmentations for more detailed analysis. ShapeNet has played a crucial role in standardizing benchmarks for 3D deep learning, enabling the development and evaluation of models for shape understanding and synthesis.

Due to the absence of implicit representations in the dataset, it was required to train for every object its corresponding implicit representation. The training pipeline is described in Section 2.2.

To quantitatively evaluate the quality of the reconstructed shapes, we use the Intersection over Union (IoU) metric defined as follows:

$$\text{IoU} = \frac{|V_{\text{gt}} \cap V_{\text{gen}}|}{|V_{\text{gt}} \cup V_{\text{gen}}|}, \tag{5.1}$$

where V_{gt} denotes the set of occupied voxels in the ground truth voxel grid, and V_{gen} denotes the occupied voxels in the generated voxel grid.

Specifically, the ground truth mesh provided by ShapeNet is voxelized into a binary occupancy grid, and the same process is applied to the mesh obtained by applying the Marching Cubes algorithm [38] to the output of the trained implicit neural representation. The IoU is then computed as the ratio between the intersection and the union of the occupied voxels in the two grids. This provides a robust, resolution-invariant measure of how accurately the generated surface approximates the ground truth geometry. By aggregating IoU scores across the dataset, we obtain class-level statistics such as mean and standard deviation, which help identify performance trends and variations across different object categories.

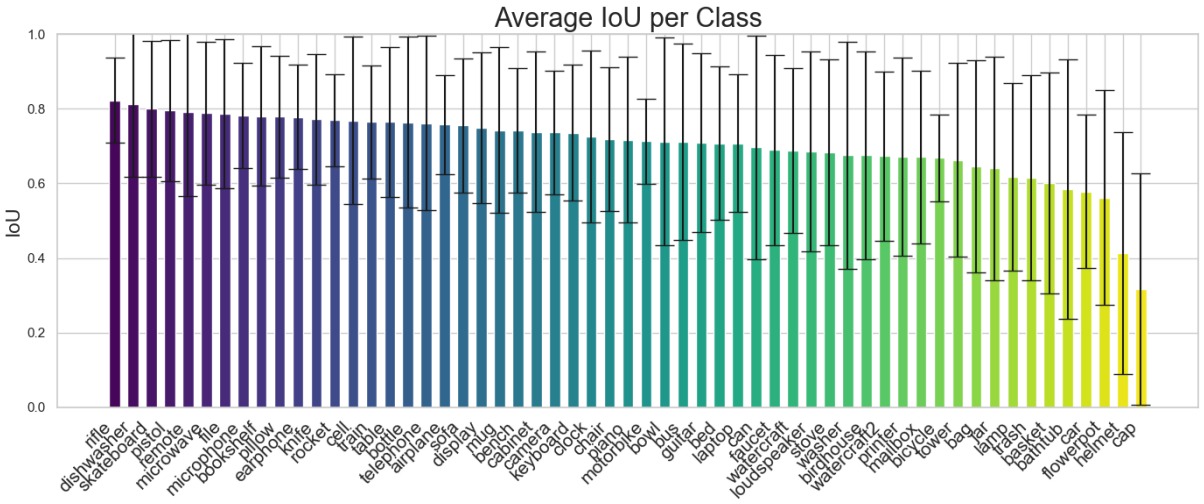
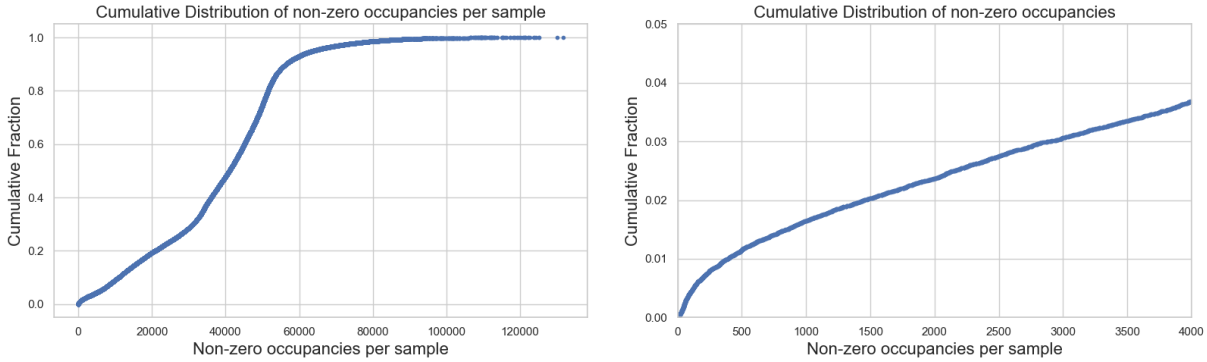


Figure 5.1: Average Intersection over Union (IoU) per class on the ShapeNet dataset. For each object category, the mean IoU between the ground truth and generated meshes is shown with standard deviation error bars. This visualization highlights the variation in reconstruction quality across categories.

To further analyze the dataset characteristics, we examine the distribution of the number of non-zero occupancy targets, denoted by o_i, across all samples. This quantity represents the number of points labeled as "inside" the shape during training, based on binary occupancy values. For each shape, a dataset of 200,000 3D spatial points is sampled, and the corresponding occupancy values (either 0 or 1) are computed to indicate whether each point lies inside or outside the mesh. Of those 200,000 point, 100,000 are sampled uniformly in the space while the remaining 100,000 are sampled from the surface of the mesh with addition of small gaussian noise. The total number of o_i thus measures how

much of the sampled space is occupied by the object and serves as a proxy for the shape's volumetric complexity or density.

Figure 5.2 presents the cumulative distribution function (CDF) of o_i aggregated over the entire dataset. The left panel shows the full distribution, capturing the long tail of high-density samples, while the right panel provides a zoomed-in view restricted to the interval $[0, 4{,}000]$. This dual visualization reveals both the overall spread and the detailed structure of the distribution, offering insights into the geometric variability of the dataset and the relative difficulty of learning implicit representations for different shapes. Shapes with a low o_i typically correspond to geometries with thin surfaces, hollow interiors, or sparse volume occupancy. Such conditions may result in weak training signals and pose challenges for learning accurate implicit representations, potentially leading to poorer reconstruction quality.



(a) Full range of o_i values

(b) Zoom into $[0, 4000]$

Figure 5.2: Cumulative distribution function (CDF) of o_i values across the dataset. In order to train every single shape, 200k points are sampled from the ground truth mesh.

For this reason, it is valuable to investigate the relationship between o_i and the Intersection over Union (IoU) between the ground truth and generated meshes. A strong correlation would suggest that occupancy sparsity directly impacts the model's ability to learn faithful 3D representations, highlighting a possible avenue for data filtering strategies.

To explore the relationship between the complexity of the input data and reconstruction accuracy, Figure 5.3 shows a scatter plot of o_i versus IoU, for visualization purposes for the airplane class only. A linear regression curve is overlaid to highlight the general trend.

The scatter plot reveals a general trend where samples with higher numbers of non-zero occupancy points (o_i) tend to achieve higher Intersection over Union (IoU) values, suggesting that richer training data supports better implicit shape reconstruction. However, notable outliers exist: some samples with large o_i still exhibit relatively low IoU. This discrepancy

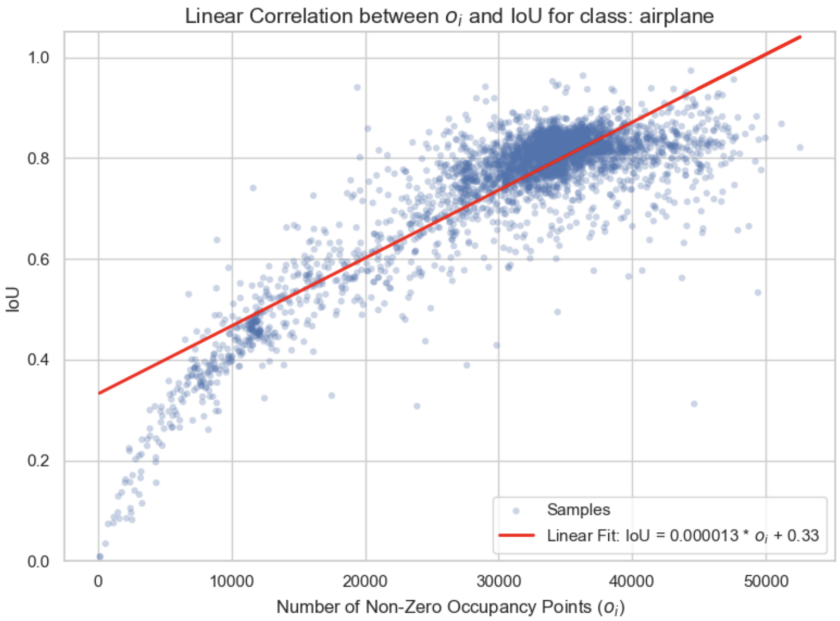


Figure 5.3: Scatter plot of o_i versus IoU with an overlapped linear regression curve.

could be attributed to the inherent stochasticity of the training process, including random initialization, optimization noise, and potential local minima, which occasionally lead to suboptimal implicit representations despite abundant training points. Such variability highlights the complexity of the training pipeline and motivates further investigation into training stability and robustness.

Additionally, alternative sampling strategies for training implicit representations are possible. For instance, one could sample points uniformly in the 3D space and then sample points on the surface of the mesh, and only optionally adding noise. These different approaches may impact the quality and robustness of the learned implicit functions, suggesting further avenues for exploration.

In the end, we report the amount of implicit representations available for each class in the dataset (Table 5.1).

The full dataset will be released completely open and publicly available on Hugging Face.

Class	# Samples
table	8353
car	6953
chair	6676
airplane	4013
sofa	3123
rifle	2372
lamp	2286
watercraft	1922
bench	1798
cabinet	1552
loudspeaker	1550
watercraft2	1126
display	1080
telephone	1040
bus	935
guitar	797
bathtub	787
faucet	738
clock	645
flowerpot	551
cell	523
jar	486
bottle	472
bookshelf	451
laptop	451
knife	422
train	388
motorbike	337

Class	# Samples
trash	327
pistol	307
file	292
bed	252
piano	239
stove	217
mug	208
printer	165
helmet	152
bowl	152
skateboard	152
microwave	148
washer	136
tower	131
camera	113
can	106
pillow	96
mailbox	93
dishwasher	92
basket	89
rocket	85
bag	81
earphone	73
birdhouse	67
microphone	67
remote	66
keyboard	65
bicycle	59
cap	53

Table 5.1: Number of samples available for each class in the dataset, split into two tables for readability.

5.2. Metrics

In this section, we report the metrics used both during training and evaluation of the quality of the generated 3D samples. The metrics for point clouds are applied to point clouds extracted from the meshes defined by the implicit representations and reference point clouds kept for metrics computation.

Precision, Recall, and F1-score

Precision, recall, and F1-score are commonly used metrics to evaluate the performance of classification models, especially in scenarios with class imbalance or when false positives and false negatives carry different consequences.

Precision quantifies the proportion of correctly predicted positive instances among all instances predicted as positive. It is defined as:

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}, \tag{5.2}$$

where TP (true positives) is the number of correctly predicted positive samples, and FP (false positives) is the number of incorrectly predicted positive samples.

Recall (also called sensitivity or true positive rate) measures the proportion of actual positives that were correctly predicted:

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}, \tag{5.3}$$

where FN (false negatives) is the number of positive samples that were incorrectly classified as negative.

F1-score is the harmonic mean of precision and recall. It provides a single score that balances both concerns, and is particularly useful when precision and recall are both important:

$$\text{F1-score} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}. \tag{5.4}$$

The F1-score reaches its best value at 1 and worst at 0. It is especially informative when dealing with imbalanced datasets or when it is important to minimize both false positives and false negatives.

Chamfer Distance (CD)

Chamfer Distance measures how close two point sets are by computing the sum of squared distances between each point in one set and its closest point in the other set. It is asymmetric and does not take the geometric structure of the point clouds into account.

Mathematically, for two sets of points $P = \{p_1, p_2, \ldots, p_n\}$ and $Q = \{q_1, q_2, \ldots, q_m\}$, the Chamfer Distance is defined as:

$$\text{CD}(P, Q) = \frac{1}{|P|} \sum_{p \in P} \min_{q \in Q} \|p - q\|^2 + \frac{1}{|Q|} \sum_{q \in Q} \min_{p \in P} \|q - p\|^2, \tag{5.5}$$

where $\|p - q\|$ represents the Euclidean distance between points p and q.

A smaller Chamfer Distance means the point clouds are closer to each other, indicating better alignment but it does not capture the structure or the distribution of the points well. It only focuses on the nearest neighbors, which might be problematic for uneven distributions or points with large gaps.

Coverage (COV)

The Coverage metric is used to assess the diversity of generated shapes in comparison to a reference set, which typically comes from the training data. It measures the extent to which the generated shapes cover the variety of shapes present in the reference set. Specifically, it calculates the fraction of reference shapes that can be associated with at least one shape from the generated set. The matching between shapes is determined using a distance metric, which evaluates how similar or different two shapes are.

The Coverage score ranges between 0 and 1, where a higher score indicates that the generated shapes exhibit more diversity, as they cover a broader range of the reference shapes. Formally, this metric is represented as:

$$\text{COV}(G, R) = \frac{|\{g \in G : \min_{r \in R} d(g, r)\}|}{|R|}, \tag{5.6}$$

In this formula, G represents the generated set of shapes, R is the reference set, and $d(g, r)$ denotes the distance between a generated shape g and a reference shape r.

Coverage is valuable for detecting mode collapse in generative models, where the model consistently generates the same or similar outputs, even though they look realistic. By measuring the variety of shapes generated, Coverage can reveal whether the model is

producing diverse outputs or just repeating the same ones. However, it's important to note that this metric does not provide any insight into the quality or realism of the generated shapes themselves.

Minimum Matching Distance (MMD)

Minimum matching distance is a metric evaluating the generation quality. It is often used as a complementary metric to the Coverage, so to evaluate both quality and diversity. As Coverage, it takes as input a generated and a reference set and it relies on a distance function. It has a simple formulation, as it is defined as the average distance of reference shapes to their closest shape in the generated set:

$$\text{MMD}(G, R) = \frac{1}{|R|} \sum_{r \in R} \min_{g \in G} d(g, r), \tag{5.7}$$

where G is the generated set of shapes, R is the reference set, and $d(g, r)$ represents the distance between a generated shape g and a reference shape r.

1-Nearest Neighbor Accuracy (1-NNA)

The 1-NNA (1-Nearest Neighbor Accuracy) metric provides an assessment of both the realism and variety of samples produced by a generative model when compared against a known set of examples. The evaluation involves merging the set of generated samples G with the reference set R and labeling them (for instance, assigning 0 to generated samples and 1 to real samples). A 1-nearest neighbor classifier is then applied to this combined set. For every sample, the classifier finds the closest sample given a distance metric such as the Chamfer Distance (5.5), and the predicted label is determined by that nearest neighbor. The overall performance of the classifier is calculated by the formula:

$$\text{1-NNA}(G, R) = \frac{\sum_{g \in G} I[N_g \in G] + \sum_{r \in R} I[N_r \in R]}{|G| + |R|}, \tag{5.8}$$

where $I[\cdot]$ is an indicator function that equals 1 if the condition is met (i.e., if the nearest neighbor comes from the same set as the sample being evaluated) and 0 otherwise, and N_i represents the closest neighbor to sample i.

An accuracy value near 50% indicates that the generated shapes are nearly indistinguishable from the real shapes, reflecting a successful generation process in terms of both fidelity and diversity. This metric is particularly valuable because it assesses the over-

all distribution similarity rather than focusing solely on individual features or marginal distributions.

Fréchet Inception Distance (FID) for 3D Shapes

The Fréchet Inception Distance (FID) is a widely adopted metric for evaluating the quality and diversity of generative models. Originally developed for images [26], FID compares the distributions of real and generated data in a feature space defined by a pretrained neural network. The key assumption is that the extracted features follow a multivariate Gaussian distribution. The FID between two such distributions is computed as:

$$\text{FID} = \|\mu_r - \mu_g\|^2 + \text{Tr}\left(\Sigma_r + \Sigma_g - 2\left(\Sigma_r\Sigma_g\right)^{1/2}\right), \tag{5.9}$$

where μ_r, Σ_r and μ_g, Σ_g are the empirical means and covariances of the feature representations of real and generated samples, respectively.

While FID is traditionally computed on features extracted from 2D images using networks such as InceptionV3 [60], it can be extended to 3D data. In our setting, 3D shapes are first converted into point clouds sampled from their mesh surfaces. These point clouds are then passed through a pretrained network designed for 3D point cloud classification (PointNet++ [51]), which serves as the feature extractor. The resulting features are used to estimate the mean and covariance for both real and generated sets, and the FID is computed as in Equation 5.9.

This adaptation enables the use of FID to assess the realism and diversity of generated 3D shapes, based on their geometric and structural properties encoded in the point cloud representations.

5.3. Experiments and Results

5.3.1. Symmetry-aware models

In this section, we present the experimental results of our proposed symmetry-aware generative models: the **Dense GNN** and the **Relational Transformer** (Section 3.3). These models are designed to incorporate permutation symmetry into the generation process of MLP weights that represent 3D implicit surfaces. To evaluate their performance, we follow a pipeline that begins with sampling MLP weights from pure Gaussian noise through the learned reverse diffusion process. The resulting weights define a occupancy

Table 5.2: Model configurations of PNA backbone with varying embedding dimensions and number of layers, along with parameter counts.

Model backbone	Embedding Dim.	# Layers	# Parameters
PNA	16	2	17.2K
PNA	16	3	22.4K
PNA	32	2	52.6K
PNA	32	3	71.4K
PNA	64	2	174K
PNA	64	3	243K

Table 5.3: Evaluation results of PNA models with varying embedding dimensions and number of layers. For comparison the HyperDiffusion baseline is added.

Model	1-NN-CD-acc ↓	FID ↓	COV ↑
PNA-16-2	1.000	178.61	1.7
PNA-16-3	1.000	172.91	5.0
PNA-32-2	0.992	176.15	3.3
PNA-32-3	0.992	164.13	5.0
PNA-64-2	0.992	167.64	8.3
PNA-64-3	1.000	175.36	5.0
HyperDiffusion	**0.69**	**3.5**	**49**

field, which we decode into a 3D mesh using the marching cubes algorithm. We then compute evaluation metrics, as introduced in the previous section, to quantify the quality and fidelity of the generated shapes.

To assess the effectiveness of the symmetry-aware Dense GNN model, we conducted a series of experiments varying its two primary hyperparameters: the *embedding dimension* of the graph nodes and the *number of message-passing layers*. These parameters directly influence the model's expressivity and capacity. Specifically, we trained the GNN with embedding dimensions of 16, 32, and 64, and for each dimension, we evaluated both 2-layer and 3-layer configurations. This resulted in six different GNN variants, whose model sizes (in terms of trainable parameters) ranged from 17.2K to 243K. The full set of configurations is summarized below:

These configurations allow us to analyze how increased representational power affects generation quality, and whether deeper or wider graph encoders provide tangible benefits in modeling the weight space of implicit surfaces.

Despite the incorporation of symmetry-aware message passing, the Dense GNN model consistently underperforms compared to the baseline. As a result, and for the sake of clarity, we report only the FID and 1-NN accuracy metrics for this model.

Model	1-NN-CD-acc ↓	FID ↓	COV ↑
RT	1.000	128.20	28.3
HyperDiffusion	**0.69**	**3.5**	**49**

Table 5.4: Quantitative results for the Relational Transformer model. The architecture is configured with a graph node embedding dimension of 16, edge embedding dimension of 32, 1 relational transformer layer with 2 attention heads and attention dimension 32. Only one configuration is reported due to the overall unsatisfactory performance of the model.

A closer inspection of the training dynamics reveals that the Dense GNN exhibits early saturation of the training loss, suggesting a limited capacity to model the complex target distribution. Increasing either the embedding dimension or the number of layers improves performance slightly, but not substantially. This points to a fundamental bottleneck in the model's ability to capture the intricacies of the weight space geometry required to generate accurate implicit surfaces.

Moreover, scaling up the GNN model is computationally challenging due to the cost of running message passing on large input graphs, making it difficult to have a model able to encode and propagate the necessary geometric information during message passing. In our setting, each graph corresponds to an MLP, and contains one node for each neuron and an edge for each parameter (bias excluded). For our implicit representation architecture, which includes three hidden layers of 128 neurons each and a scalar output, this results in over 300 graph nodes and 30,000 edges.

This makes it difficult to scale the model in a straightforward manner to close the performance gap with more expressive baselines.

Similarly, the Relational Transformer model yielded unsatisfactory quantitative results across all evaluated metrics. This outcome suggests inherent limitations in the model's capacity to capture the underlying data distribution effectively, given the current training setup and computational constraints. The architecture combines attention-based mechanisms with structural priors from the relational graph, and its key parameters include the graph embedding dimension, the number of transformer layers, and the number of attention heads. Despite extensive experimentation with different configurations of these parameters, no significant improvement in generation quality was observed. As a result, we report only the outcome of a representative run for completeness, while acknowledging the broader challenges encountered in training this model effectively in our setting.

To provide an intuitive understanding of the reported metric values, we include a quali-

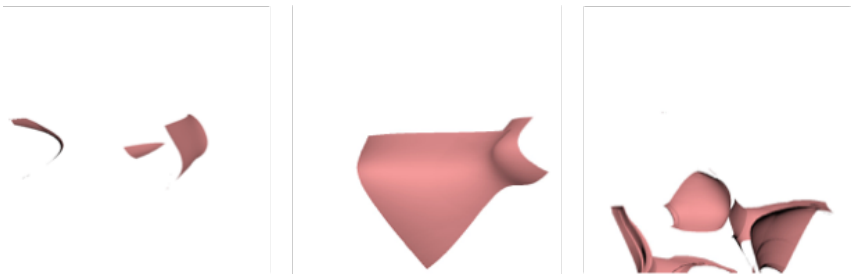


Figure 5.4: Three examples of samples generated by the GNN model. The outputs consist mainly of noisy, incoherent fragments of surfaces, reflecting the limited generative quality and structural coherence of the model's predictions.

tative visualization of a representative sample generated by the GNN model. This visualization serves to contextualize the numerical results by illustrating the appearance and characteristics of the generated samples.

Given the observed limitations in expressivity and the significant computational challenges associated with scaling the symmetry-aware models, we shifted our focus away from these architectures. In particular, we explored a model architecture inspired by the baseline Hyperdiffusion approach. Unlike the baseline, which relies on large, layer-wise projecting MLP networks (Eq. 3.2), our proposed model employs a shared graph neural network (PNA-based) encoder. This encoder is designed to produce an equivariant representation, which is subsequently processed by transformer layers.

We will refer to this model as the **GNN-Transformer**. It retains the important parameters of the GNN, namely the embedding dimension and the number of message-passing layers, while inheriting the hyperparameters of the transformer component. This approach allows us to experiment with combining an equivariant encoder with the powerful sequence modeling capabilities of transformers. Our objective is to assess whether the increased computational cost and the higher number of parameters involved in this hybrid architecture translate into improved generation quality.

To evaluate the effectiveness of this design, we conducted a series of experiments by varying both the GNN and transformer components. In particular, we explored whether increasing the embedding dimension of the transformer or scaling up the GNN encoder yields improvements in generation quality. The configurations considered in our study are summarized in Table 5.5.

The performance of the GNN-Transformer models is summarized in Table 5.6. We focus our analysis on two key metrics: validation FID, which assesses the overall distributional similarity between generated and real samples, and 1-NN-CD-accuracy, which evaluates

Model	GNN Dim.	# MP Layers	Total Params	Transformer Config.
GNN_T	16	1	5.2M	2-2-128-128
GNN_T	16	1	11.2M	2-2-256-256
GNN_T	16	1	23.4M	2-2-512-512
GNN_T	32	1	5.3M	2-2-128-128
GNN_T	64	1	5.4M	2-2-128-128

Table 5.5: Configurations of the GNN-Transformer models used in our experiments. Each model combines a GNN encoder with a transformer, where the Transformer configuration column field reports the number of layers, attention heads, node feature dimension, and edge feature dimension respectively.

Model	1-NN-CD-acc ↓	FID ↓	COV ↑
GNN_T-64-1-128	1.000	105.88	1.7
GNN_T-32-1-128	1.000	103.91	1.7
GNN_T-16-1-512	1.000	91.43	1.7
GNN_T-16-1-256	1.000	114.08	1.7
GNN_T-16-1-128	0.992	97.59	3.3
HyperDiffusion	**0.69**	**3.5**	**49**

Table 5.6: Quantitative results for GNN-Transformer variants. The models differ by GNN embedding dimension and transformer configuration.

point-wise similarity through nearest neighbor matching.

From the Table 5.6, it can be observed that all configurations yield very high 1-NN-CD-accuracy scores (close to or exactly 1.000), indicating that the model can almost perfectly distinguish generated samples from real ones. This suggests that the generated shapes are easily recognizable as synthetic and do not closely resemble real samples, at least in terms of their point-wise characteristics. Therefore, a high 1-NN-CD-accuracy score reflects poor realism rather than similarity, highlighting a limitation in the fidelity of the generated shapes.

More informative is the behavior of the FID metric across different architectures. When increasing the transformer embedding dimension (from 128 to 256 and 512, keeping the GNN fixed), we observe a consistent reduction in FID, reaching its lowest value (91.43) with a transformer embedding of 512. This suggests that the transformer component is indeed capable of learning better global representations and refining the generative process, especially when granted sufficient capacity.

In contrast, increasing the GNN embedding dimension (from 16 to 32 and 64, while keeping the transformer configuration fixed at 2 layers with 128-dimensional embeddings)

Figure 5.5: Generated samples from the **GNN-Transformer** model for the *airplane* class. While the fuselage is consistently reproduced, the finer details and class-specific structures (e.g., wings and tails) are mostly either missing, noisy or poorly formed, indicating limited generalization capability.

leads to only marginal changes in FID. While the model with a 32-dimensional GNN achieves slightly better FID (103.91) than the 64-dimensional one (105.88), the differences are minor and do not support a clear trend in favor of larger GNN encoders. This indicates that improvements in generation quality are more sensitive to the expressivity of the transformer layers than to the encoding capacity of the GNN.

The persistent high values of 1-NN-CD-acc across all configurations, despite variations in FID, suggest that the generated samples are easily distinguishable from real ones. This indicates that the models struggle to produce realistic outputs that align well with the data manifold.

This observation becomes particularly evident when inspecting the generated samples for the airplane class. As shown in Figure 5.5, the model consistently learns to generate the fuselage — a feature shared across all instances in the class — but fails to generalize to the finer, class-defining components such as wings, tails, or engines. This suggests that the model overfits to the most prominent common shape components while lacking the capacity or inductive bias to represent and generate more diverse structural variations. Nonetheless, the ability to reliably produce at least the fuselage contributes to a lower FID compared to the one obtained with the only GNN-based model, which fails to capture even this core structure.

Given the outcomes of the previous experiments, we further explored the **GNN-Transformer** architecture by scaling the transformer component while keeping the graph encoder fixed to a smaller configuration with embedding dimension 16 and a single message-passing layer. The goal was to assess whether increasing the representational capacity of the

Model	1-NN-CD-acc ↓	FID ↓	COV ↑
GNN_T-16-1-256	1.000	**88.37**	1.7

Table 5.7: Quantitative results for the GNN-Transformer model with a small GNN encoder (embedding dimension 16, 1 layer) and a large transformer module (8 layers, 8 heads, embedding dimension 256).

transformer alone could lead to improvements in generative performance. As anticipated, this configuration resulted in a lower FID score, indicating a better match between the distribution of generated and real samples in feature space. However, despite this quantitative improvement (see Table 5.7), the model continued to suffer from the same limitations observed in previous experiments. In particular, the 1-NN-CD accuracy remained at 1, and the visual inspection of generated samples revealed no significant gains in output quality or diversity. This suggests that the transformer alone cannot compensate for the lack of expressiveness in the graph-based encoding.

5.3.2. Latent Diffusion models

Given the results of the previous experiments, we decided to move away from generative architectures incorporating symmetry-aware components, such as GNN-based encoders. Despite their theoretical advantages in capturing structural priors, the empirical performance of these models was consistently unsatisfactory, both quantitatively and qualitatively. As a consequence, we opted to explore a more expressive and scalable generative approach grounded in recent advances in latent diffusion modeling. Specifically ss described in Section 4.2, we adopt a latent diffusion pipeline, which decouples the generative task into two distinct stages: (1) a Variational Autoencoder is first trained to compress the data into a lower-dimensional latent space that preserves its essential features, and (2) a diffusion model is trained to generate samples in this latent space. This decomposition not only alleviates the high computational cost of operating diffusion models directly in data space but also provides a more compact and semantically meaningful representation to guide the generation process. In the following sections, we first describe the experimental setup and results obtained from training the VAE, and then detail the training and evaluation of the diffusion model operating in its latent space.

In training the VAE, particular attention was devoted to the influence of the β parameter in the loss function (Equation 1.14), as it directly controls the trade-off between accurate reconstruction and latent space regularization. A high value of β encourages disentanglement and more structured latent spaces by strengthening the KL divergence term, but

may hinder the model's ability to reconstruct fine details. Conversely, a low β prioritizes reconstruction at the expense of regularity in the latent representation.

To explore this trade-off, we conducted experiments using two types of β scheduling strategies. The first adopts a constant β value throughout training, while the second employs a cyclical annealing schedule as introduced in prior literature [21]. The cyclical schedule periodically increases β from zero to a maximum value, thereby allowing the model to initially focus on reconstruction and progressively shift toward enforcing a regular latent space. We evaluated the effect of both strategies using several maximum β values: 1×10^{-6}, 1×10^{-5}, and 1×10^{-4}. This experimental setup was designed to gain insight into how regularization strength and scheduling dynamics influence the VAE's learning behavior and downstream generative performance.

It is important to emphasize that in our VAE framework, we are working with implicit representations, where each sample in the dataset is encoded as the weights of a neural network that models an occupancy function. This function takes a spatial coordinate as input and outputs the probability that the point belongs to a 3D shape. As such, there is no need to reconstruct the exact same architecture that produced the original representation. Instead, the VAE is only required to learn a set of weights that define a network capable of representing a plausible occupancy field. This flexibility allows us to significantly reduce the complexity of the decoder network without compromising the learning objective.

To take advantage of this property and reduce computational cost during the initial experiments, we introduced a *reducing factor*, defined as the ratio between the number of hidden units in the input (original) network and those in the output (reconstructed) network. For these preliminary experiments, we set the reducing factor to 2, effectively halving the size of the hidden layers in the reconstructed MLPs. This setup allows for a larger batch size training and so an overall faster training while still enabling the decoder to model valid occupancy fields.

Regarding evaluation, we focus exclusively on the reconstruction capabilities of the VAE, rather than its generative performance. Since our objective at this stage is to assess how effectively the encoder–decoder pair can compress and reconstruct a given implicit representation, we evaluate the model using reconstruction metrics only. Specifically, as the reconstruction task is formulated as a binary classification problem over spatial coordinates (predicting whether a point belongs to a shape or not), we adopt the F1 score as the primary metric. This choice captures the balance between precision and recall, which is particularly relevant in settings with class imbalance (e.g., few occupied points

Table 5.8: Results of VAE training with different β scheduling strategies. All values are extracted from the checkpoint with the highest F1 score during training.

Beta Schedule	FID (decoded) ↓	F1 Score ↑	Std Latent	Mean Latent
CyclicalScheduler_1e-06	78.96	**0.7392**	0.5193	-0.0263
CyclicalScheduler_1e-05	78.89	0.7333	0.8015	-0.0124
CyclicalScheduler_0.0001	85.18	0.7365	0.9304	-0.0033
ConstantValue_1e-06	**78.63**	0.7381	0.9156	-0.0139
ConstantValue_1e-05	83.09	0.7330	0.9804	**-0.0002**
ConstantValue_0.0001	90.02	0.7027	**0.9955**	0.0020

in a large 3D grid).

Importantly, we do not evaluate the model's generative ability via sampling from a standard Gaussian prior and decoding from random latent vectors. Instead, we delegate the generative role to the second stage of the pipeline, where a diffusion model is trained to operate in the latent space learned by the VAE. This separation of concerns allows us to focus the VAE on learning a compact, meaningful latent representation with high fidelity reconstruction, while the diffusion model handles generation from noise.

Table 5.8 summarizes the reconstruction and latent statistics for VAE models trained with different β scheduling strategies. Each entry corresponds to the checkpoint that achieved the highest F1 score during training.

From a reconstruction perspective, measured via the F1 score, the best performing model uses a `CyclicalScheduler` with a maximum β of 1e$-$6, reaching an F1 of 0.7392. Interestingly, this setting also results in a relatively low reconstruction FID (78.96), suggesting that cyclical annealing with a small regularization pressure can lead to both high fidelity and better alignment with the ground-truth shapes. Among the constant schedule strategies, the best F1 score (0.7381) is comparable and achieved with $\beta = 1e{-}6$, also showing a low FID and stable latent distribution statistics.

When analyzing the latent space, we focus on two indicators: the mean value of the latent vector and the average standard deviation across latent dimensions. These values reflect how well the posterior distribution $q(z|\theta)$ aligns with the standard Gaussian prior $\mathcal{N}(0, I)$. As expected, higher β values enforce stronger regularization, increasing the mean standard deviation (up to 0.99) and pushing the mean values closer to zero. However, this often comes at the cost of poorer reconstruction, as seen in the constant schedule with $\beta = 1e{-}4$, which shows the lowest F1 (0.7027) and the highest FID (90.02).

Overall in our setting, small β values with any scheduler strike a good balance, preserving

Table 5.9: Evaluation metrics for the best-performing VAE on the *car*, *chair*, and *airplane* classes. The metrics refer to the decoded validation samples.

Model	1-NN-CD Acc ↓	FID ↓	COV ↑	MMD ↓
VAE_cars	0.758	13.49	23.7	5.2
VAE_chairs	0.608	17.58	53.3	23.7
VAE_airplanes	0.500	8.95	56.7	3.2

Table 5.10: Reconstruction score and latent space statistics for the best-performing VAE on the *car*, *chair*, and *airplane* classes. The metrics refer to the decoded validation samples.

Model	F1 Score ↑	Std Latent	Mean Latent
VAE_cars	0.724	0.840	-0.0390
VAE_chairs	0.305	0.230	-0.0104
VAE_airplanes	0.734	0.982	-0.0054

reconstruction performance while maintaining meaningful structure in the latent space.

After experimenting with different configurations of hyperparameters, we now proceed to evaluate the best-performing VAE model on specific object categories. In particular, we focus on the *airplanes*, *cars*, and *chairs* classes—three categories that are among the most commonly studied in the 3D shape literature due to their structural variability and representation challenges. The goal of this evaluation is to assess whether the trends observed in the aggregated metrics generalize across individual shape categories, and to better understand the reconstruction capabilities of the VAE in category-specific settings.

All the VAEs models in Table 5.9 and 5.10 share the same overall architecture design described in 4.1, with variations in encoder and decoder depth to best suit each class. For the airplanes class, the encoder consists of 8 transformer layers with 4 attention heads and a hidden dimension of 512, while the decoder uses 4 layers with the same number of heads and hidden dimension. For the cars class, a more compact architecture is used, with 2 encoder layers (2 heads) and 2 decoder layers (4 heads), both with a hidden dimension of 512. The chairs class employs a deeper configuration, using 8 layers for both encoder and decoder, each with 4 heads and a hidden dimension of 512. All final VAEs do not employ a *reducing factor*.

No comparison with the baseline is available since the results concern the reconstructed samples, while the baseline concerns generated samples from gaussian noise.

In the following it is possible to find examples of meshes extracted from reconstructed implicit representations for the validation set.

Figure 5.6: Example of the VAE reconstruction ability over time for a sample in the car class.

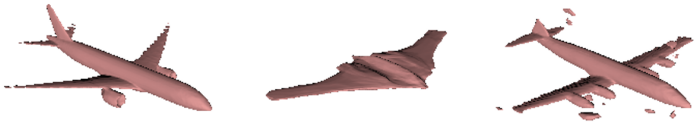


Figure 5.7: Example of the VAE reconstruction ability for the airplanes class.

Taking for example Figure 5.7, the reconstructed samples show that the VAE successfully learns to reproduce a diverse set of airplane geometries, capturing key structural variations across the dataset. However, some artifacts are still visible in the outputs. These imperfections may arise both from limitations in the original implicit representation and from reconstruction inaccuracies introduced by the VAE during training.

As shown in the examples of decoded meshes in Figure 5.8 and supported by the quantitative results in Table 5.9, the VAE trained on the *chair* class, despite its ability to capture the diversity of shapes, suffers from significant reconstruction failures. In particular, the model often produces incomplete geometries, especially in regions with thin surfaces, where parts of the shape are missing or distorted. This issue does not stem from limitations in the original implicit representation or from poor sampling of the training points since the occupancy labels are available and accurate but rather from the inability of the VAE decoder to learn to output a set of weights that reproduces a correct occupancy field, as also reflected in the low F1 score.

This highlights a limitation of the proposed methodology: while the VAE performs well on certain classes such as *cars* and *airplanes*, its failure to reconstruct complex or highly varied structures like *chairs* suggests that the approach may not generalize well across all shape categories. In particular, the inability to learn an accurate occupancy field for

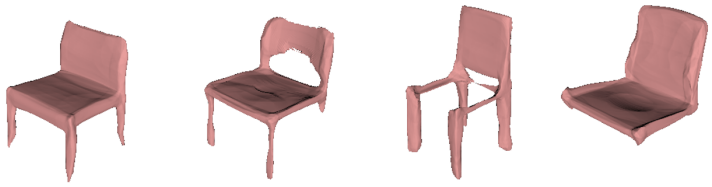


Figure 5.8: Example of the VAE reconstruction ability for the chairs class.

chairs, despite having valid supervision, indicates that further refinement of the architecture, training procedure, or representation strategy is needed. Addressing this generalization gap and improving robustness to structural variability is an important direction for future work.

We now shift our focus from the Variational Autoencoders to the Latent Diffusion Model. As discussed, while the VAE is effective at encoding shapes into a latent space and reconstructing them, it offers no guarantee that arbitrary samples from the standard Gaussian prior can be reliably mapped to meaningful shapes through the decoder. This is a well-known limitation of VAEs, where the learned latent distribution often deviates from the assumed prior, making direct sampling unreliable. To address this issue and enable controlled generative modeling, we introduce a diffusion model trained in the latent space. The goal of this model is to learn how to map noise drawn from a standard Gaussian distribution to the VAE's latent space distribution, enabling high-quality shape generation from pure noise.

To understand how model capacity impacts this mapping process, we conducted a series of experiments where we scaled the embedding dimension of the latent diffusion model while keeping the number of layers and attention heads fixed. As expected, models with larger embedding dimensions (and thus more parameters) achieve lower validation MSE losses, indicating a greater capacity to model the latent space distribution. However as it can be seen in Table 5.11, this does not consistently lead to better generative quality, as measured by FID or 1-NN-CD accuracy. In particular, the smallest model achieves competitive or even better performance in terms of FID, suggesting that a smaller architecture can still be effective for high-quality generation.

We now report in Table 5.12 the final results for the best-performing latent diffusion model on the *airplane*, *car*, and *chair* categories. For each class, we select the model configuration that achieves the best 1-NN-CD accuracy. These results are then compared with the

Table 5.11: Evaluation metrics for latent diffusion models on the *car* class with varying embedding dimensions. All models share the same architecture configuration (DiT) except for the embedding dimension. The model name format is `DiT_<n_layers>_<n_heads>_<hidden_dim>`, indicating the number of transformer layers, attention heads, and embedding (hidden) dimension, respectively.

Model	# Parameters	1-NN-CD Acc ↓	FID ↓	COV ↑	MMD ↓
DiT_12_6_96	2.6M	0.775	6.27	**28.3**	5.5
DiT_12_6_192	10.0M	0.783	7.26	26.7	5.6
DiT_12_6_384	39.8M	**0.767**	**6.24**	25.0	**5.5**

Table 5.12: Comparison between our best-performing latent diffusion models and the results from the *HyperDiffusion* paper for the *airplane*, *car*, and *chair* categories.

Class	Method	Params	1-NN-CD Acc ↓	FID ↓	COV ↑	MMD ↓
Airplane	LDM (ours)	127.9M	**0.633**	13.40	41.7	4.10
	HyperDiffusion	1B	0.693	**3.5**	**49**	**3.4**
Car	LDM (ours)	81.4M	0.733	6.15	26.7	**3.03**
	HyperDiffusion	1B	**0.731**	**2.6**	**36**	3.4
Chair	LDM (ours)	131.7M	0.817	29.34	26.7	**3.92**
	HyperDiffusion	1B	**0.541**	**1.7**	**53**	7.1

HyperDiffusion baseline to assess the relative performance of our approach. As discussed in Section 4.2 the LDMs are based on DiT architectures. Specifically, the best models used in our experiments are `DiT_12_6_384` for *airplanes*, and `DiT_8_4_384` for both *cars* and *chairs*, where the model name `DiT_<L>_<H>_<D>` indicates the number of transformer layers L, the number of attention heads H, and the hidden dimension D. While the LDM models consistently operate with significantly fewer parameters—ranging from 81M to 132M compared to over 1B for HyperDiffusion—they still achieve competitive results in terms of generation quality except for the *chairs* class due to the limited reconstruction capability of its VAE.

6 | Conclusions

This thesis presented a comprehensive study on the generation of 3D shapes using implicit neural representations and diffusion-based generative models. The work focused on developing a two-stage pipeline capable of synthesizing complex 3D geometries with high fidelity and diversity. The proposed approach leverages the compactness and expressiveness of implicit representations, where each 3D shape is encoded as a neural network that approximates an occupancy function.

The initial part of the research explored symmetry-aware generative models, motivated by the theoretical benefits of respecting the permutation symmetries of multilayer perceptrons. Despite promising motivations, these models proved difficult to scale and integrate effectively within a diffusion framework. Their performance did not surpass simpler, non-equivariant baselines, leading to the conclusion that while symmetry-aware architectures are conceptually appealing, their practical benefits in this context remain limited under current resource and data constraints.

Motivated by these findings, the work transitioned to a more scalable and effective latent diffusion framework. A variational autoencoder was trained to embed implicit functions into a structured latent space, and a diffusion model was subsequently trained within this space to learn a generative prior. This two-stage design enabled sampling from Gaussian noise while maintaining consistency and expressiveness in the decoded 3D surfaces.

The pipeline was evaluated on several ShapeNet categories, including cars, chairs, and airplanes. Results demonstrated that the latent diffusion approach can match or outperform the *HyperDiffusion* baseline on several metrics, especially in terms of fidelity and diversity of generated samples. Moreover, performance varied across categories, with some classes (such as chairs) remaining particularly challenging to model.

Future Work

This work opens up several avenues for future research:

- **Data quality and implicit training.** The quality of the dataset plays a fun-

damental role in the success of the generative model. Since the dataset consists of neural networks trained to represent 3D shapes, improving the training procedure of these implicit representations—e.g., via better sampling, loss functions, or architecture choices—can have a direct impact on generation performance.

- **Scalable GNN-based diffusion.** As diffusion frameworks evolve, it will be important to monitor the development of graph-based diffusion models capable of processing large graphs efficiently. This could revive interest in symmetry-aware approaches, especially if future models are designed to support weight-space equivariance in a scalable manner.
- **Multiclass generative modeling.** While this work focused on class-specific models, extending the pipeline to support multi-class generation would increase scalability and enable richer conditional modeling. This may require architectural modifications or the introduction of class conditioning in both the encoder and the diffusion model.
- **Improved loss functions.** The current pipeline relies primarily on a reconstruction loss (e.g., binary cross-entropy or MSE). However, this may be insufficient to ensure perceptually plausible outputs. Future work could explore perceptual losses computed from pretrained classifiers trained on the dataset, which would guide the model toward more semantically meaningful reconstructions, particularly for underperforming categories like chairs. Additionally, combining the diffusion loss with a generative adversarial component (i.e., GAN loss) may improve sharpness and diversity.

Overall, this thesis highlights the potential of combining implicit representations with latent diffusion to generate 3D shapes, while also identifying key limitations and suggesting promising directions for continued exploration in this fast-evolving field.

Bibliography

[1] P. Achlioptas, O. Diamanti, I. Mitliagkas, and L. Guibas. Learning representations and generative models for 3d point clouds, 2018. URL https://arxiv.org/abs/1707.02392.

[2] M. Arjovsky, S. Chintala, and L. Bottou. Wasserstein gan, 2017. URL https://arxiv.org/abs/1701.07875.

[3] D. Bau, J.-Y. Zhu, J. Wulff, W. Peebles, H. Strobelt, B. Zhou, and A. Torralba. Seeing what a gan cannot generate, 2019. URL https://arxiv.org/abs/1910.11626.

[4] J. Brea, B. Simsek, B. Illing, and W. Gerstner. Weight-space symmetry in deep networks gives rise to permutation saddles, connected by equal-loss valleys across the loss landscape, 2019. URL https://arxiv.org/abs/1907.02911.

[5] A. Brock, T. Lim, J. M. Ritchie, and N. Weston. Generative and discriminative voxel modeling with convolutional neural networks, 2016. URL https://arxiv.org/abs/1608.04236.

[6] M. M. Bronstein, J. Bruna, T. Cohen, and P. Veličković. Geometric deep learning: Grids, groups, graphs, geodesics, and gauges, 2021. URL https://arxiv.org/abs/2104.13478.

[7] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[8] R. Cai, G. Yang, H. Averbuch-Elor, Z. Hao, S. Belongie, N. Snavely, and B. Hariharan. Learning gradient fields for shape generation, 2020. URL https://arxiv.org/abs/2008.06520.

[9] A. X. Chang, T. Funkhouser, L. Guibas, P. Hanrahan, Q. Huang, Z. Li, S. Savarese, M. Savva, S. Song, H. Su, et al. Shapenet: An information-rich 3d model repository. *arXiv preprint arXiv:1512.03012*, 2015.

[10] H. Chen, J. Gu, A. Chen, W. Tian, Z. Tu, L. Liu, and H. Su. Single-stage diffusion nerf: A unified approach to 3d generation and reconstruction, 2023. URL https://arxiv.org/abs/2304.06714.

[11] S. Cheng, M. Bronstein, Y. Zhou, I. Kotsia, M. Pantic, and S. Zafeiriou. Meshgan: Non-linear 3d morphable models of faces, 2019. URL https://arxiv.org/abs/1903.10384.

[12] Y.-C. Cheng, H.-Y. Lee, S. Tulyakov, A. Schwing, and L. Gui. Sdfusion: Multimodal 3d shape completion, reconstruction, and generation, 2023. URL https://arxiv.org/abs/2212.04493.

[13] A. Choromanska, M. Henaff, M. Mathieu, G. B. Arous, and Y. LeCun. The loss surfaces of multilayer networks, 2015. URL https://arxiv.org/abs/1412.0233.

[14] T. S. Cohen, M. Geiger, J. Koehler, and M. Welling. Spherical cnns, 2018. URL https://arxiv.org/abs/1801.10130.

[15] G. Corso, L. Cavalleri, D. Beaini, P. Liò, and P. Veličković. Principal neighbourhood aggregation for graph nets, 2020. URL https://arxiv.org/abs/2004.05718.

[16] P. Dhariwal and A. Nichol. Diffusion models beat gans on image synthesis. *Advances in neural information processing systems*, 34:8780–8794, 2021.

[17] C. Diao and R. Loynd. Relational attention: Generalizing transformers for graph-structured tasks, 2023. URL https://arxiv.org/abs/2210.05062.

[18] L. Dinh, R. Pascanu, S. Bengio, and Y. Bengio. Sharp minima can generalize for deep nets, 2017. URL https://arxiv.org/abs/1703.04933.

[19] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv: 2010.11929*, 2020.

[20] Z. Erkoç, F. Ma, Q. Shan, M. Nießner, and A. Dai. Hyperdiffusion: Generating implicit neural fields with weight-space diffusion, 2023. URL https://arxiv.org/abs/2303.17015.

[21] H. Fu, C. Li, X. Liu, J. Gao, A. Celikyilmaz, and L. Carin. Cyclical annealing schedule: A simple approach to mitigating kl vanishing, 2019. URL https://arxiv.org/abs/1903.10145.

[22] F. B. Fuchs, D. E. Worrall, V. Fischer, and M. Welling. Se(3)-transformers: 3d roto-

translation equivariant attention networks, 2020. URL https://arxiv.org/abs/2006.10503.

[23] L. Gao, J. Yang, T. Wu, Y.-J. Yuan, H. Fu, Y.-K. Lai, and H. Zhang. Sdm-net: Deep generative network for structured deformable mesh, 2019. URL https://arxiv.org/abs/1908.04520.

[24] L. Gao, T. Wu, Y.-J. Yuan, M.-X. Lin, Y.-K. Lai, and H. Zhang. Tm-net: Deep generative networks for textured meshes, 2021. URL https://arxiv.org/abs/2010.06217.

[25] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative adversarial nets. *Advances in neural information processing systems*, 27, 2014.

[26] M. Heusel, H. Ramsauer, T. Unterthiner, B. Nessler, and S. Hochreiter. Gans trained by a two time-scale update rule converge to a local nash equilibrium, 2018. URL https://arxiv.org/abs/1706.08500.

[27] G. E. Hinton and R. S. Zemel. Autoencoders, minimum description length and helmholtz free energy. In *Proceedings of the 7th International Conference on Neural Information Processing Systems*, NIPS'93, page 3–10, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.

[28] J. Ho, A. Jain, and P. Abbeel. Denoising diffusion probabilistic models. *Advances in neural information processing systems*, 33:6840–6851, 2020.

[29] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[30] H. Jun and A. Nichol. Shap-e: Generating conditional 3d implicit functions, 2023. URL https://arxiv.org/abs/2305.02463.

[31] J. Kim, J. Yoo, J. Lee, and S. Hong. Setvae: Learning hierarchical composition for generative modeling of set-structured data, 2021. URL https://arxiv.org/abs/2103.15619.

[32] D. P. Kingma. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[33] D. P. Kingma and M. Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[34] M. V. Koroteev. Bert: a review of applications in natural language processing and understanding. *arXiv preprint arXiv:2103.11943*, 2021.

[35] A. R. Kosiorek, H. Strathmann, D. Zoran, P. Moreno, R. Schneider, S. Mokrá, and D. J. Rezende. Nerf-vae: A geometry aware 3d scene generative model, 2021. URL https://arxiv.org/abs/2104.00587.

[36] M. Li, Y. Duan, J. Zhou, and J. Lu. Diffusion-sdf: Text-to-shape via voxelized diffusion, 2023. URL https://arxiv.org/abs/2212.03293.

[37] Z. Liu, Y. Feng, M. J. Black, D. Nowrouzezahrai, L. Paull, and W. Liu. Meshdiffusion: Score-based generative 3d mesh modeling, 2023. URL https://arxiv.org/abs/2303.08133.

[38] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. In *Seminal graphics: pioneering efforts that shaped the field*, pages 347–353. 1998.

[39] A. Luo, T. Li, W.-H. Zhang, and T. S. Lee. Surfgen: Adversarial 3d shape synthesis with explicit surface discriminators, 2022. URL https://arxiv.org/abs/2201.00112.

[40] S. Luo and W. Hu. Diffusion probabilistic models for 3d point cloud generation, 2021. URL https://arxiv.org/abs/2103.01458.

[41] Z. Lyu, J. Wang, Y. An, Y. Zhang, D. Lin, and B. Dai. Controllable mesh generation through sparse latent point diffusion models, 2023. URL https://arxiv.org/abs/2303.07938.

[42] X. Mao, Q. Li, H. Xie, R. Y. K. Lau, Z. Wang, and S. P. Smolley. Least squares generative adversarial networks, 2017. URL https://arxiv.org/abs/1611.04076.

[43] B. Mildenhall, P. P. Srinivasan, M. Tancik, J. T. Barron, R. Ramamoorthi, and R. Ng. Nerf: Representing scenes as neural radiance fields for view synthesis, 2020. URL https://arxiv.org/abs/2003.08934.

[44] N. Müller, Y. Siddiqui, L. Porzi, S. R. Bulò, P. Kontschieder, and M. Nießner. Diffrf: Rendering-guided 3d radiance field diffusion, 2023. URL https://arxiv.org/abs/2212.01206.

[45] A. Navon, A. Shamsian, I. Achituve, E. Fetaya, G. Chechik, and H. Maron. Equivariant architectures for learning in deep weight spaces, 2023. URL https://arxiv.org/abs/2301.12780.

[46] T. Nguyen-Phuoc, C. Li, L. Theis, C. Richardt, and Y.-L. Yang. Hologan: Unsupervised learning of 3d representations from natural images, 2019. URL https://arxiv.org/abs/1904.01326.

[47] A. Nichol, P. Dhariwal, A. Ramesh, P. Shyam, P. Mishkin, B. McGrew, I. Sutskever, and M. Chen. Glide: Towards photorealistic image generation and editing with text-guided diffusion models, 2022. URL https://arxiv.org/abs/2112.10741.

[48] A. Nichol, H. Jun, P. Dhariwal, P. Mishkin, and M. Chen. Point-e: A system for generating 3d point clouds from complex prompts, 2022. URL https://arxiv.org/abs/2212.08751.

[49] J. J. Park, P. Florence, J. Straub, R. Newcombe, and S. Lovegrove. Deepsdf: Learning continuous signed distance functions for shape representation. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 165–174, 2019.

[50] W. Peebles and S. Xie. Scalable diffusion models with transformers, 2023. URL https://arxiv.org/abs/2212.09748.

[51] C. R. Qi, H. Su, K. Mo, and L. J. Guibas. Pointnet: Deep learning on point sets for 3d classification and segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 652–660, 2017.

[52] N. Rahaman, A. Baratin, D. Arpit, F. Draxler, M. Lin, F. A. Hamprecht, Y. Bengio, and A. Courville. On the spectral bias of neural networks, 2019. URL https://arxiv.org/abs/1806.08734.

[53] R. Rombach, A. Blattmann, D. Lorenz, P. Esser, and B. Ommer. High-resolution image synthesis with latent diffusion models, 2022. URL https://arxiv.org/abs/2112.10752.

[54] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[55] T. Shen, J. Gao, K. Yin, M.-Y. Liu, and S. Fidler. Deep marching tetrahedra: a hybrid representation for high-resolution 3d shape synthesis, 2021. URL https://arxiv.org/abs/2111.04276.

[56] D. W. Shu, S. W. Park, and J. Kwon. 3d point cloud generative adversarial network based on tree structured graph convolutions, 2019. URL https://arxiv.org/abs/1905.06292.

[57] J. R. Shue, E. R. Chan, R. Po, Z. Ankner, J. Wu, and G. Wetzstein. 3d neural field generation using triplane diffusion, 2022. URL https://arxiv.org/abs/2211.16677.

[58] J. Sohl-Dickstein, E. Weiss, N. Maheswaranathan, and S. Ganguli. Deep unsupervised learning using nonequilibrium thermodynamics. In *International conference on machine learning*, pages 2256–2265. PMLR, 2015.

[59] Y. Song, J. Sohl-Dickstein, D. P. Kingma, A. Kumar, S. Ermon, and B. Poole. Score-based generative modeling through stochastic differential equations. *arXiv preprint arXiv:2011.13456*, 2020.

[60] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. Rethinking the inception architecture for computer vision, 2015. URL https://arxiv.org/abs/1512.00567.

[61] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need. *Advances in Neural Information Processing Systems*, 2017.

[62] T. Wang, B. Zhang, T. Zhang, S. Gu, J. Bao, T. Baltrusaitis, J. Shen, D. Chen, F. Wen, Q. Chen, and B. Guo. Rodin: A generative model for sculpting 3d digital avatars using diffusion, 2022. URL https://arxiv.org/abs/2212.06135.

[63] M. Wiatrak, S. V. Albrecht, and A. Nystrom. Stabilizing generative adversarial networks: A survey, 2020. URL https://arxiv.org/abs/1910.00927.

[64] J. Wu, C. Zhang, T. Xue, W. T. Freeman, and J. B. Tenenbaum. Learning a probabilistic latent space of object shapes via 3d generative-adversarial modeling, 2017. URL https://arxiv.org/abs/1610.07584.

[65] X. Zeng, A. Vahdat, F. Williams, Z. Gojcic, O. Litany, S. Fidler, and K. Kreis. Lion: Latent point diffusion models for 3d shape generation, 2022. URL https://arxiv.org/abs/2210.06978.

[66] X.-Y. Zheng, Y. Liu, P.-S. Wang, and X. Tong. Sdf-stylegan: Implicit sdf-based stylegan for 3d shape generation, 2022. URL https://arxiv.org/abs/2206.12055.

[67] J. Zhou, G. Cui, S. Hu, Z. Zhang, C. Yang, Z. Liu, L. Wang, C. Li, and M. Sun. Graph neural networks: A review of methods and applications, 2021. URL https://arxiv.org/abs/1812.08434.

[68] L. Zhou, Y. Du, and J. Wu. 3d shape generation and completion through point-voxel diffusion, 2021. URL https://arxiv.org/abs/2104.03670.

A | Appendix A: Training and inference algorithms

Algorithm A.1 Training the Variational Autoencoder (VAE) on Implicit Representations

Require: Dataset of implicit MLPs $\{\theta_i\}$ with associated point samples $\{(x_j)\}$
Require: Encoder $q_\phi(z|\theta)$, Decoder $p_\psi(\hat{\theta}|z)$
Require: Epochs E, batch size B, KL weight β

for epoch = 1 to E **do**
  for each batch $\{\theta_1, \ldots, \theta_B\}$ **do**
    Encode: $z_i \sim q_\phi(z|\theta_i)$
    Decode: $\hat{\theta}_i = p_\psi(z_i)$
    for each $\hat{\theta}_i$ **do**
      Sample spatial points $\{x_j\}$ from training distribution
      Evaluate predicted occupancies: $\hat{o}_j = \hat{\theta}_i(x_j)$
      Get ground truth occupancies: $o_j = \theta_i(x_j)$
      Compute $\mathcal{L}_{\text{rec}} = \text{BinaryCrossEntropy}(\hat{o}_j, o_j)$
    end for
    Compute $\mathcal{L}_{\text{KL}} = D_{\text{KL}}(q_\phi(z|\theta_i) \,||\, \mathcal{N}(0, I))$
    Total loss: $\mathcal{L} = \mathcal{L}_{\text{rec}} + \beta \cdot \mathcal{L}_{\text{KL}}$
    Backpropagate and update parameters ϕ, ψ
  end for
end for

Algorithm A.2 Training the Latent Diffusion Model in VAE Latent Space

Require: Trained VAE encoder $q_\phi(z|\theta)$
Require: Diffusion model $\epsilon_\theta(z_t, t)$
Require: Noise schedule β_t, total steps T
Initialize: Compute μ, σ = mean and std of latent vectors from first batch
for epoch = 1 to E **do**
  for each batch $\{\theta_1, \ldots, \theta_B\}$ **do**
    Encode: $z_i \sim q_\phi(z|\theta_i)$
    Normalize: $z_i \leftarrow (z_i - \mu)/\sigma$
    Sample timestep $t \sim \mathcal{U}(1, T)$
    Sample noise $\epsilon \sim \mathcal{N}(0, I)$
    Generate $z_t = \sqrt{\bar{\alpha}_t} z_i + \sqrt{1 - \bar{\alpha}_t}\epsilon$
    Predict noise: $\hat{\epsilon} = \epsilon_\theta(z_t, t)$
    Compute loss: $\mathcal{L} = \|\epsilon - \hat{\epsilon}\|^2$
    Backpropagate and update model
  end for
end for

Algorithm A.3 Inference: Generating Implicit Representations with Latent Diffusion

Require: Trained diffusion model ϵ_θ, VAE decoder p_ψ
Require: Latent normalization stats: mean μ, std σ
Require: Diffusion steps T, noise schedule β_t
Sample $z_T \sim \mathcal{N}(0, I)$
for $t = T$ down to 1 **do**
  Predict noise: $\hat{\epsilon}_t = \epsilon_\theta(z_t, t)$
  Estimate z_{t-1}:
    $z_{t-1} = \frac{1}{\sqrt{\alpha_t}}\left(z_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\hat{\epsilon}_t\right) + \sigma_t \cdot \eta$
    (with $\eta \sim \mathcal{N}(0, I)$ if $t > 1$)
end for
Denormalize: $z_0 \leftarrow z_0 \cdot \sigma + \mu$
Decode: $\hat{\theta} = p_\psi(z_0)$
Return $\hat{\theta}$ as generated implicit function

List of Figures

List of Tables

Ils arrivèrent au bord de la Seine, en grande banlieue.

Kathe dit à Jules :
Si tu veux rentrer à Paris à temps pour dîner avec cet ami, tu peux prendre le train à cette gare, Jules.

Et elle stoppa au carrefour.

Jules descendit, vint à la portière.
Elle l'embrassa gravement. Puis elle lui dit, les yeux brillants :
Regarde-nous bien, Jules !

Et elle démarra, emmenant Jim.
Au lieu de prendre à droite la belle route du bord de l'eau,
elle continua tout droit et s'engagea sur le pont étroit qui était en réparation.

Jim allait lui demander :
Pourquoi prends-tu par là ?

Mais après tout, cela lui était égal.
Jules les regardait.