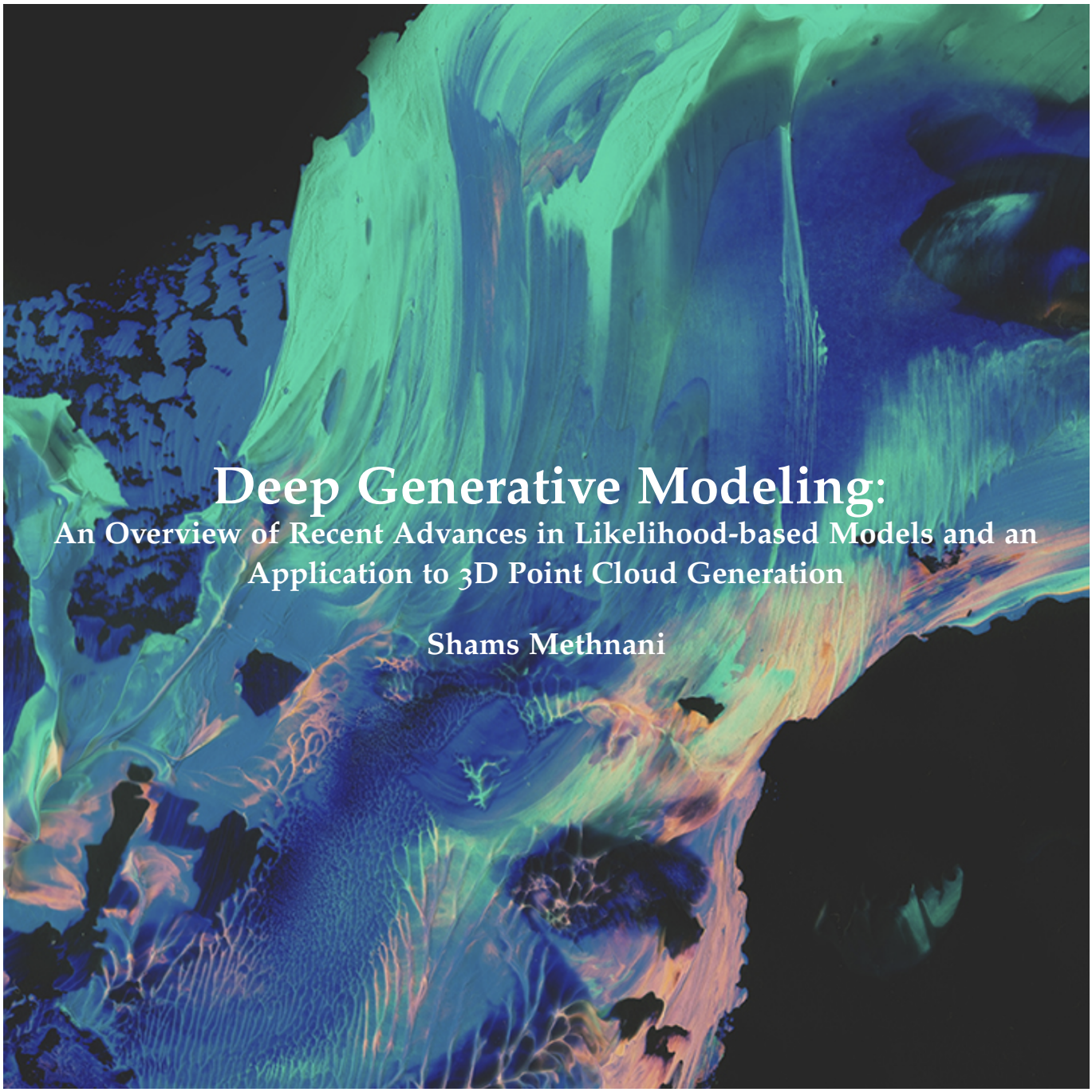
UMEÅ UNIVERSITY



Deep Generative Modeling:

An Overview of Recent Advances in Likelihood-based Models and an Application to 3D Point Cloud Generation

Shams Methnani

Master of Science in Mathematical Statistics, 120 ECTS
Department of Mathematics and Mathematical Statistics
Spring 2023

Abstract

Deep generative modeling refers to the process of constructing a model, parameterized by a deep neural network, that learns the underlying patterns and structures of the data generating process which produced the samples in a given dataset, in order to generate novel samples that resemble those in the original dataset. Deep generative models for 3D shape generation hold significant importance to various fields including robotics, medical imaging, manufacturing, computer animation and more. This work provides a pedagogical overview of likelihood-based models, namely variational autoencoders, flow-based models, diffusion models and rectified flows and investigates the effect on generation quality of replacing the Chamfer loss with a sliced Wasserstein loss in an application to shape generation with 3D point clouds.

Keywords: *Generative modeling, deep learning, variational autoencoders, flow-based models, diffusion models, rectified flows, flow matching*

Sammanfattning

Djup generativ modellering hänvisar till processen att konstruera en modell, parametriserad av ett djupt neuralt nätverk, som lär sig de underliggande mönstren och strukturerna för den datagenererande process som producerade proverna i en given datamängd, för att producera nya prover som liknar dem i den ursprungliga datamängden. Djupa generativa modeller för 3D-formgenerering har stor betydelse för olika områden, inklusive robotik, medicinsk bildbehandling, industriell tillverkning, datoranimation osv. Detta arbete ger en pedagogisk översikt av sannolikhetsbaserade modeller, nämligen variational autoencoders, flödesbaserade modeller, diffusionsmodeller och likriktade flöden och undersöker effekten på generationskvaliteten av att ersätta Chamfer-lossfunktionen med en sliced Wasserstein-lossfunktion i en applikation för att forma formgenerering med 3D-punktmoln.

Nyckelord: *Generativ modellering, djupinlärning, variational autoencoders, flödesbaserade modeller, diffusionsmodeller, likriktade flöden, flödesmatchning*

Acknowledgments

I would first like to express gratitude to my supervisor, Axel Flinth, for his guidance and mentorship, which have shaped this project and greatly enriched my understanding of the subject matter. I would also like to extend appreciation to my examiner, Jun Yu, for his constructive feedback and evaluation, which has contributed to the quality and refinement of this work.

I am deeply grateful to my parents, Amal and Mabrouk, for their endless patience and unwavering belief in my abilities. Their love and support have been pillars of strength throughout my life. I am also grateful to my wonderful brother, Nour, and incredible sister, Leila, whose affection and friendship have been constant sources of happiness.

I am thankful to Nouf, Cassie, Lubna and my climbing friends at MX for their continuous encouragement and camaraderie. Their presence has provided both inspiration on the wall and comfort on the ground throughout the highs and lows of this endeavor.

Finally, I would like to extend my sincere gratitude to the gracious crew at Mirzam for their warm hospitality and words of encouragement during the course of this journey. Despite my often over-extended stays, I was consistently embraced and made to feel welcome and I looked forward every day to sitting in the cocoa nibs corner to write.

CONTENTS

1 INTRODUCTION

The surge of deep generative models has ushered in a new era of artificial intelligence. Displaying notable success across a variety of tasks, these models have garnered significant attention in recent years from both researchers and the public, alike. Deep generative models hold compelling promise across a diverse range of domains, with direct applications including image generation, audio and music synthesis, text generation and drug design. In the context of 3D shape generation, generative models offer the potential to significantly reduce costs in generating assets for digital environments, designing prototypes for manufacturing, training robots for perception tasks, visualizing medical imaging data and more.

Deep generative modeling belongs to the broader field of generative modeling. A generative model seeks to learn a representation of the process which generated a dataset. Through this learned representation, the model is able to synthesize novel data samples that retain characteristics of the original dataset. Leveraging the advances in deep learning, the field of deep generative models has witnessed a dramatic rate of progress in a short period of time.

Despite their success, these models are not without their challenges and a variety of techniques have been proposed to address them. The aim of this work is to provide an overview of methods and recent advances in deep generative modeling. In addition, we aim to investigate the effect on the quality of samples produced in an application on 3D point cloud generation by altering the loss function used in a base model. The research questions we seek to answer are:

1. What are the underlying principles of likelihood-based deep generative models?
2. What is the effect of the replacing a Chamfer loss function with a sliced Wasserstein loss function on the quality of samples generated by a state-of-the-art generative model for 3D point clouds?

This work is structured as follows:

- Chapter 2 briefly introduces neural networks, which we consider to be the backbone of deep generative models.
- Chapter 3 introduces deep generative models, the fundamental concepts behind them and the various approaches used to optimize them. In particular, we pay close attention to likelihood-based models, focusing on variational autoencoders, flow-based models, diffusion models and rectified flows.
- Chapter 4 shifts attention to representation learning and generative modeling in the 3D case, which presents unique challenges. Specifically, we focus on 3D point cloud data and the special considerations that need to be extended to them.

- Chapter 5 presents an application to shape generation with 3D point clouds. In particular, we conduct an experiment to investigate the effect on the quality of samples generated in one step by replacing the objective function used in a state-of-the-art model from the Chamfer loss to the sliced Wasserstein loss.

We expect the reader to be familiar with concepts in basic probability and rudimentary calculus but we briefly recall the relevant concepts as they present themselves.

2

NEURAL NETWORKS

Neural networks have enjoyed noteworthy dominance in modern machine learning on account of the impressive results achieved over a wide range of tasks, from image recognition to natural language generation. Inspired by the structure of the human brain, neural networks consist of layers of interconnected nodes, with each node performing simple computations on its inputs. The output of each node is passed through a non-linear *activation function*, allowing the model to approximate non-linear functions. When a neural network consists of multiple layers, it is called a *deep neural network*. Through the composition of these computations, deep neural networks can learn representations of complex data, making them an attractive choice for parameterizing generative models.

2.1 FITTING A MODEL

In the context of generative modeling, we are given a dataset of samples $D = \{x^{(i)}, i \in N\}$, where each sample is generated by some underlying distribution, $x^{(i)} \sim p_D(x)$. Each $x^{(i)}$ is typically a vector, which could represent the pixel values in an image or the position of a point in a point cloud. The true data distribution is unknown to us as we only have access to the samples in D. The goal is to approximate the true data distribution with a parameterized neural network $p_\theta(x)$ using the given data samples with the hope that the resulting approximation is close to the true underlying data distribution. This amounts to finding the parameters θ of our neural network which best fit the given data set D.

Maximum Likelihood Estimation

A natural approach to finding the best parameters θ for a model is to select those which make the observed data most probable. A likelihood function is defined as the joint probability of observed data given the model parameters. The difference between a probability function and a likelihood function is that a probability function is a function of the data, with the parameters considered fixed while a likelihood function is a function of the parameters, with the data considered fixed. We maximize the likelihood because we are searching through different possible parameters given fixed observed data. Under the assumption that the data samples are independent and identically distributed (i.i.d.), which we will consider to be the case throughout this work, the likelihood can be written as:

$$p(x^{(1)}, \ldots, x^{(N)}|\theta) = \prod_{i=1}^{N} p_\theta(x^{(i)}).$$

It is more convenient to instead maximize the log-likelihood function, which is equivalent to maximizing the likelihood function as the logarithm is a monotonically increasing function. The product of probabilities becomes a sum of log-probabilities, which provides both analytical convenience as well as numerical stability during computation since terms with probabilities close to zero will not cause excessive numerical errors. The log-likelihood function can be expressed as follows:

$$\log p(x^{(1)}, \ldots, x^{(N)}|\theta) = \log \prod_{i=1}^{N} p_\theta(x^{(i)}) = \sum_{i=1}^{N} \log p_\theta(x^{(i)}).$$

By convention, machine learning optimization problems are typically formulated as minimization problems. The maximum likelihood estimation (MLE) objective function is therefore often expressed as minimizing the negative log-likelihood:

$$\mathcal{L}_{MLE}(\theta) = -\sum_{i=1}^{N} \log p_\theta(x^{(i)})$$

The optimal parameters θ found by MLE are those which minimize $\mathcal{L}_{MLE}(\theta)$:

$$\theta_{MLE} = \operatorname*{argmin}_\theta \mathcal{L}_{MLE}(\theta).$$

Kullback-Leibler Divergence

Another approach to finding the best parameters for the approximation is to minimize a distance between the target distribution and approximation. One way to measure a notion of distance between distributions is by information content. The information carried by a random variable x can be quantified by the following expression,

$$I(x) = -\log p(x),$$

where $p(x)$ is the probability of x. When x is continuous, $p(x)$ is a probability density. Recall that the logarithm of a value between 0 and 1 is negative, with those values closer to 0 having very large negative logarithmic values and those closer to 1 having very small negative logarithmic values. This tells us that unlikely events carry high amounts of information, while events that are almost certain carry very little information. The entropy of a distribution $p(x)$ is defined as:

$$\begin{aligned} H(x) &= \mathbb{E}_p\left[-\log p(x)\right] \\ &= \begin{cases} -\sum_{x \in X} p(x) \log p(x) & \text{if } x \text{ is discrete.} \\ -\int_X p(x) \log p(x) & \text{if } x \text{ is continuous,} \end{cases} \end{aligned}$$

where X is the set on which random variable x is distributed. This can be thought of as the expected information content of a random event drawn from $p(x)$ and is a way to quantify uncertainty in probability distributions. Distributions for which the outcome is almost certain have low entropy, while distributions where the possible outcomes are more uniformly probable have high entropy, i.e. high uncertainty [8]. The Kullback-Leibler (KL) divergence captures dissimilarity between two distributions $p(x)$ and $q(x)$ by measuring their relative entropy as follows,

$$\begin{aligned} D_{KL}(p \parallel q) &= \mathbb{E}_p\left[\log p(x) - \log q(x)\right] \\ &= \begin{cases} \sum_{x \in X} \log \frac{p(x)}{q(x)} & \text{if } x \text{ is discrete.} \\ \int_X p(x) \log \frac{p(x)}{q(x)} dx & \text{if } x \text{ is continuous.} \end{cases} \end{aligned}$$

KL divergence has the following properties:

1. $D_{KL}(p \parallel q) \geqslant 0$
2. $D_{KL}(p \parallel q) \neq D_{KL}(q \parallel p)$

The first property states that the KL divergence is non-negative and the second property states that the KL divergence is asymmetric. While the KL divergence is often thought of as a type of distance between probability distributions, it is not a proper distance in the mathematical sense as it is not symmetric and does not satisfy the triangle inequality.

In the context of optimizing parameters θ of a model which approximates a data distribution $p_D(x)$, the KL divergence minimization can be expanded as follows:

$$\begin{aligned} \operatorname*{argmin}_\theta D_{KL}(p_D \parallel p_\theta) &= \operatorname*{argmin}_\theta \mathbb{E}_{p_D}\left[\log \frac{p_D(x)}{p_\theta(x)}\right] \\ &= \operatorname*{argmin}_\theta \mathbb{E}_{p_D}\left[\log p_D(x) - \log p_\theta(x)\right] \\ &= \operatorname*{argmin}_\theta \underbrace{\mathbb{E}_{p_D}\left[\log p_D(x)\right]}_{\text{does not depend on } \theta} - \mathbb{E}_{p_D}\left[\log p_\theta(x)\right] \\ &= \operatorname*{argmin}_\theta - \mathbb{E}_{p_D}\left[\log p_\theta(x)\right]. \end{aligned}$$

Given a set of finite samples $D = \{x^{(i)}, i \in N\}$ drawn from $p_D(x)$, we can approximate the expectation with Monte Carlo estimation:

$$\begin{aligned}\underset{\theta}{\operatorname{argmin}} - \mathbb{E}_{p_D}[\log p_\theta(x)] &\approx \underset{\theta}{\operatorname{argmin}} - \sum_{i=1}^{N} \log p_\theta(x^{(i)}) \\ &= \theta_{MLE}.\end{aligned}$$

We can see from this derivation that minimizing the KL divergence from the data distribution to the model is equivalent to maximizing the likelihood of the data.

2.2 OPTIMIZATION ALGORITHMS

Once a particular loss or objective function is chosen, an optimization procedure is required in order to update the model parameters such that the loss function is minimized. One such algorithm is *gradient descent*. The idea behind gradient descent is to iterate through the data, apply the loss function with the current parameters and take a step towards the direction of steepest descent in hopes of eventually ending up in some local minima. The direction of steepest descent is characterized by the negative gradient of the loss function with respect to (w.r.t.) the parameters of the model. Moving in this direction will minimize the loss. During each iteration, the parameters θ are updated by subtracting a scalar multiple of the gradient from their current values. This can be done through *backpropagation*, which is a method for computing how much each parameter contributed to the loss by applying the chain rule of differentiation. The scalar multiple is a hyperparameter known as the *learning rate*, which determines the step size when descending along the loss function.

Stochastic gradient descent is a more efficient variant of gradient descent which only applies the loss function on a random subset of the data during each iteration to update the parameters. A popular variant of stochastic gradient descent is the *Adam optimizer*, which adapts its learning rate based on the first and second-order moments (i.e. the mean and the variance) of the current gradients. The name refers to its adaptive moment estimation.

3

DEEP GENERATIVE MODELS

Generative models aim to capture the underlying structure of a dataset and use that knowledge to generate new data samples that adhere to the learned patterns. Consider the task of learning to generate images of cats. A generative model is built from a sample of images of cats by capturing some useful representation of the underlying space, i.e. the space of all possible images of cats. The typical building block for these models is to estimate the joint probability distribution that generated the data, which may be very complex and high-dimensional, as depicted in fig. 1. When coupled with access to vast amounts of data and computing power, deep neural networks are well-suited for approximating functions of high-dimensional data and are a popular choice for parameterizing generative models. Such models are called *deep generative models*.

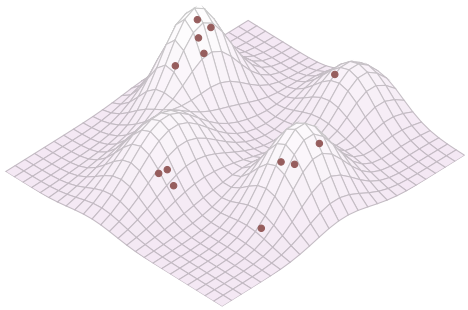


Figure 1: The data generation process is governed by some unknown data distribution which may be very complex. The generative model is only given a sample of the data, depicted here by solid points, and must learn to estimate the underlying data distribution in order to evaluate probabilities of points and generate new samples.

Deep generative models can be broadly classified into two categories based on their learning approach. The first category is likelihood-based models, which include variational autoencoders, normalizing flows, and diffusion models. These models are trained by maximizing the likelihood of the observed data, aiming to find the most probable parameters that explain the samples. The second is likelihood-free models, the most prominent of which is generative adversarial networks (GAN). GANs follow a different approach, in which a two-player minimax game is played between a generator and a discriminator. The generator aims to produce realistic samples to deceive the discriminator, while the discriminator aims to distinguish between real and generated samples. Through this adversarial process, the generator learns to produce high-quality synthetic data capable of deceiving the discriminator.

3.1 AUTOENCODERS

An autoencoder is neural network that learns to reconstruct its input after compressing it through a hidden bottleneck layer. It consists of two networks: an encoder network, which learns an encoder function $f_\phi(x)$ that transforms the input into a latent representation z and a decoder network, which learns a decoder function $g_\theta(z)$ to transform a latent code z back into the input. This architecture is depicted in fig. 2.

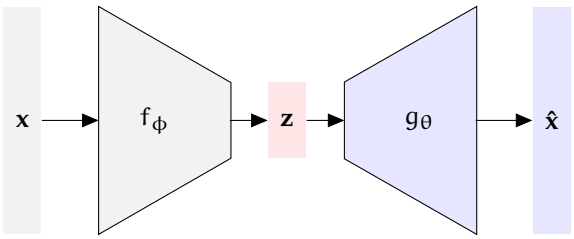


Figure 2: The encoder transforms input into a latent representation which the decoder transforms back into the input.

The autoencoder simultaneously learns the parameters ϕ and θ to transform and reconstruct the input data. The input is first transformed into another representation $z = f_\phi(x)$ and is then reconstructed into $\hat{x} = g_\theta(z)$. The objective is then to minimize the difference between x and $\hat{x}$, which can be accomplished by minimizing the mean squared error (MSE) between them:

$$\mathcal{L}_{AE}(\phi, \theta) = \frac{1}{N} \sum_{i=1}^{N} \|x^{(i)} - g_\theta(f_\phi(x^{(i)}))\|_2^2.$$

Autoencoders were initially introduced as a dimensionality reduction method [9]. In this formulation, the bottleneck layer is of lower dimension than the input, forcing the model to learn a lower-dimensional representation of the data. The object of interest in such cases is the bottleneck layer $z = f_\phi(x)$, which is the compressed representation of the input, called the latent representation or code. The original data can then be reconstructed using just the latent representation and the decoder network. Autoencoders cannot generate novel data in themselves but are popular building blocks for representation learning in generative models.

Denoising Autoencoders

A denoising autoencoder (DAE) [25] is an autoencoder which first corrupts the input data before feeding it into the encoder network and reconstructs the original, uncorrupted data as the output of the decoder. The idea is to feed a neural network corrupted versions of samples for which we have the uncorrupted version, so that it may learn to recover uncorrupted versions of samples it has not seen yet. This also allows the bottleneck layer to be of higher dimension than the input data while still forcing the model to learn something

useful, namely repairing or *denoising* data. This architecture is depicted in fig. 3. For larger networks with many more parameters than the input dimension, this also serves to avoid overfitting and improve robustness.

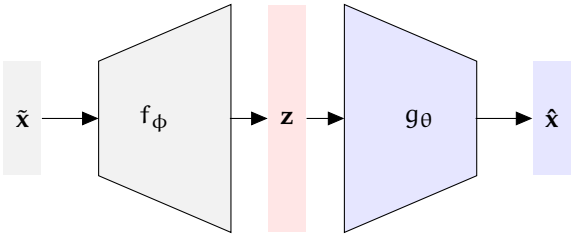


Figure 3: The encoder transforms corrupted input into a latent representation which the decoder transforms back into uncorrupted input.

The same objective can be used, with input to the encoder replaced with the corrupted data as follows:

$$\tilde{x} \sim C(\tilde{x}|x)$$
$$\mathcal{L}_{DAE}(\phi, \theta) = \frac{1}{N} \sum_{i=1}^{N} \|x^{(i)} - g_\theta(f_\phi(\tilde{x}^{(i)}))\|_2^2,$$

where $C(\tilde{x}|x)$ is some corruption process on x, typically a process that randomly removes values or adds noise. A DAE can then be thought of as model which takes a corrupted input and generates a denoised sample.

3.2 VARIATIONAL AUTOENCODERS

The contents of this section are based on [22].

The goal of a generative model is to capture the distribution of a set of observed data variables. These variables can contain elements which are highly dependent on one another. It can be useful to infer an underlying structure to the data in order to disentangle those dependencies. This allows dependencies between any pair of elements from the observed variables to be captured indirectly by their respective dependencies with the latent variables that generated them [8]. A variational autoencoder (VAE) is a generative model which introduces latent variables $z \sim p_Z(z)$, that generate the observed data x. The latent variable is typically of lower dimension than the data. The dependency graph is illustrated as a graphical model in fig. 4.

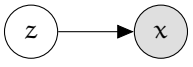


Figure 4: Observed data x is generated by latent variables z.

The motivation behind this is that the data that we are trying to model, which may be very high-dimensional, could be explained by a smaller number of underlying factors which a model can learn and represent in a compressed manner. For example, a million-pixel image of a dog could be described by fewer variables such as whether it has paws, pointy ears etc. The idea then is to model the joint distribution $p_\theta(x, z)$ and integrate out the latent variable in order to obtain the marginal distribution of the data, $p_\theta(x)$. By introducing $p_Z(z)$, the relationship between the variables in our model can be described by:

$$p_\theta(x, z) = p_\theta(x|z)p_Z(z)$$
$$p_\theta(x) = \int_Z p(x, z)dz = \int_Z p_\theta(x|z)p_Z(z)dz,$$

where $p_\theta(x|z)$ is a conditional model, parameterized by a neural network, that learns to reconstruct x given a latent variable z and $p_Z(z)$ is typically a fixed prior, though it can also be learned. Only the data x is observed and latent variables z are unknown to us.

We will make the assumption that $p_Z(z)$ will be chosen so that it is easy to sample from and that the learned conditional model $p_\theta(x|z)$ is also easy to sample from. The generative process for new data can then be summarized by the following procedure:

1. Sample z from $p_Z(z)$.
2. Sample new data point x from the conditional distribution $p_\theta(x|z)$.

The task is then to find the optimal parameters θ which maximize the probability of the observed data $D = \{x^{(1)}, \ldots, x^{(N)}\}$:

$$\theta_{MLE} = \underset{\theta}{\operatorname{argmin}} - \sum_{i=1}^{N} \log p_\theta(x^{(i)})$$
$$p_\theta(x^{(i)}) = \int_Z p_\theta(x^{(i)}|z)p_Z(z)dz.$$

The integral which defines the marginal probability $p_\theta(x^{(i)})$ of each data point can be intractable to compute, as it involves summing over all possible configurations of z. A typical approach to solve this integral is Monte Carlo integration:

$$\begin{aligned} p_\theta(x^{(i)}) &= \int_Z p_\theta(x^{(i)}|z)p_Z(z)dz \\ &= \mathbb{E}_{z\sim p_Z}\left[p_\theta(x^{(i)}|z)\right] \\ &\approx \frac{1}{K}\sum_{k=1}^{K} p_\theta(x^{(i)}|z_k^{(i)}), \end{aligned}$$

where $z_k^{(i)} \sim p_Z(z)$ is the k^{th} randomly sampled z for estimating the expectation of the i^{th} data point $x^{(i)}$. However, each $x^{(i)}$ will be generated by only a few of the possible latent

variable values. This means that for most randomly sampled $z_k^{(i)}$, the value of $p_\theta(x^{(i)}|z_k^{(i)})$ will be almost 0 and an unreasonably large number of samples K is required for an accurate estimate of the integral. This problem is exacerbated if z is high-dimensional. Importance sampling offers a more efficient approach to Monte Carlo estimation.

Importance Sampling

Importance sampling is a technique for approximating properties of one distribution by working with a different distribution. This is useful when direct sampling from the distribution we are interested in is either difficult or inefficient. Suppose we would like to compute the expectation of a function of a random variable x with respect to a distribution $p(x)$, which is either difficult or, as in the scenario described above, inefficient to sample from. We can introduce another distribution $q(x)$, which we know to be easy to sample from, and rewrite the expectation as follows:

$$\begin{aligned}\mathbb{E}_p\left[f(x)\right] &= \int_X f(x)p(x)dx \\ &= \int_X \frac{f(x)p(x)q(x)}{q(x)}dx \\ &= \mathbb{E}_q\left[\frac{f(x)p(x)}{q(x)}\right].\end{aligned}$$

We can see that the expectation is now expressed with respect to q.

The expectation in the original training objective can now be replaced with the importance sampling approximation and we can rewrite a revised objective as follows:

$$\begin{aligned}\theta_{MLE} &= \underset{\theta}{\operatorname{argmin}} - \sum_{i=1}^{n} \log p_\theta(x^{(i)}) \\ p_\theta(x^{(i)}) &= \int_Z p_\theta(x^{(i)}|z)p_Z(z)dz \\ &\approx \frac{1}{K}\sum_{k=1}^{K} \frac{p_Z(z_k^{(i)})}{q(z_k^{(i)})} p_\theta(x^{(i)}|z_k^{(i)}), \quad z_k^{(i)} \sim q(z_k^{(i)}).\end{aligned}$$

The ratio $\frac{p_Z(z_k^{(i)})}{q(z_k^{(i)})}$ will adjust for the importance of each sample under $q(z_k^{(i)})$ based on how well it matches $p_Z(z_k^{(i)})$. If we sample a particular $z_k^{(i)}$ which under q is very likely but under p_Z is very unlikely, then the contribution will be scaled down. The idea is to amplify the contribution of a well-matched sample when $p_\theta(x^{(i)}|z_k^{(i)})$ is large. In practice,

regular variational autoencoders use $K = 1$ samples for this approximation. When $K > 1$, the model is known as an *importance-weighted variational autoencoder*.

Choosing a candidate for q(z)

While importance sampling is valid for any choice of $q(z)$, a choice which produces well-matched samples is much more efficient. We would like a distribution which produces a sample z that is likely given a particular $x^{(i)}$. A good choice for $q(z)$ is then the posterior distribution $p_\theta(z|x^{(i)})$, which produces likely samples of latent variables z, given a particular data point $x^{(i)}$. Bayes' rule tells us that:

$$p_\theta(z|x^{(i)}) = \frac{p_\theta(x^{(i)}|z)p_Z(z)}{p_\theta(x^{(i)})}.$$

However, the denominator $p_\theta(x^{(i)})$ is precisely the term that we don't know how to compute, due to the integral involving all of the configurations of z, and it therefore isn't clear how to sample from this distribution. We can borrow methods from variational inference to find a suitable distribution $q(z)$ that is as close as possible to $p_\theta(z|x^{(i)})$.

The Variational Approach

Variational inference is a collection of methods to approximate a complex distribution by optimizing the parameters ϕ of a simpler distribution. This allows us to sample from a distribution $q(z)$, which we know to be tractable, such that:

$$q(z) \approx p_\theta(z|x^{(i)}).$$

For simplicity, we will consider the case where $q(z)$ is chosen to be a Gaussian distribution to illustrate an example, but other distributions can also be chosen. A Gaussian distribution is parameterized by a mean vector μ and a covariance matrix Σ. For convenience, a diagonal covariance matrix is often assumed in practice and the inference network will instead output a vector σ such that $\Sigma = \sigma^2 I$. Our goal is then find the optimal parameters $\phi = \{\mu, \sigma\}$ to minimize the dissimilarity between this approximate distribution $q_\phi(z)$ and the true posterior distribution $p_\theta(z|x)$, which can be done by minimizing their KL divergence:

$$\phi^{(i)} = \underset{q(z)}{\operatorname{argmin}}\, D_{KL}\left(q(z) \parallel p_\theta(z|x^{(i)})\right).$$

Note that each data point $x^{(i)}$ leads to a different set of parameters $\phi^{(i)}$. It will typically be infeasible to estimate a different distribution for every data point. VAEs instead model a single neural network which outputs the parameters of $q(z)$ given a particular data sample

$x^{(i)}$, which we denote $q_\phi(z|x^{(i)})$. The network learns by taking a sum of the objectives of each $x^{(i)}$ and infers the distribution parameters over the entire data set:

$$\phi^* = \underset{\phi}{\operatorname{argmin}} \sum_{i=1}^{N} D_{KL}\left(q_\phi(z|x^{(i)}) \parallel p_\theta(z|x^{(i)})\right).$$

Our model now consists of two neural networks: an *inference network* q_ϕ which infers the parameters of a distribution that approximates the posterior $p_\theta(z|x)$, and a *generator network* p_θ which takes latent variables as input and produces a new sample $\hat{x}$. The model is illustrated in fig. 5.

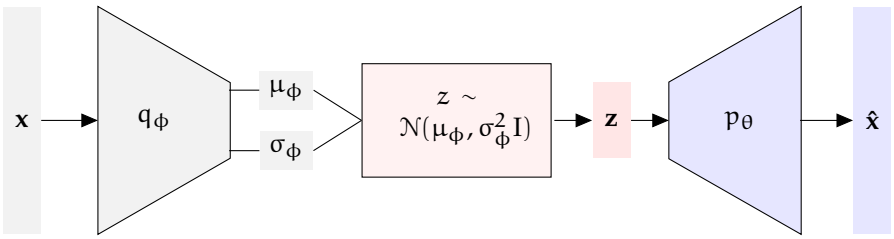


Figure 5: The inference network q_ϕ produces the parameters for the distribution from which to sample z. The generator network p_θ takes z as input and produces a sample $\hat{x}$.

Minimizing KL divergence via the Evidence Lower Bound

It is not directly clear how to minimize a KL divergence which includes terms that we cannot compute. We can rewrite the KL divergence from $q_\phi(z|x)$ to $p_\theta(z|x)$ as follows,

$$
\begin{aligned}
D_{KL}\left(q_\phi(z|x) \parallel p_\theta(z|x)\right) &= -\mathbb{E}_{q_\phi}\left[\log \frac{p_\theta(z|x)}{q_\phi(z|x)}\right] \\
&= -\mathbb{E}_{q_\phi}\left[\log \frac{\frac{p_\theta(x|z)p_Z(z)}{p_\theta(x)}}{q_\phi(z|x)}\right] \\
&= -\mathbb{E}_{q_\phi}\left[\log \frac{p_\theta(x|z)p_Z(z)}{q_\phi(z|x)} \cdot \frac{1}{p_\theta(x)}\right] \\
&= -\mathbb{E}_{q_\phi}\left[\log \frac{p_\theta(x|z)p_Z(z)}{q_\phi(z|x)} + \log \frac{1}{p_\theta(x)}\right] \\
&= -\mathbb{E}_{q_\phi}\left[\log \frac{p_\theta(x|z)p_Z(z)}{q_\phi(z|x)} - \log p_\theta(x)\right] \\
&= -\mathbb{E}_{q_\phi}\left[\log \frac{p_\theta(x|z)p_Z(z)}{q_\phi(z|x)}\right] + \mathbb{E}_{q_\phi}\left[\log p_\theta(x)\right] \\
&= -\mathbb{E}_{q_\phi}\left[\log \frac{p_\theta(x|z)p_Z(z)}{q_\phi(z|x)}\right] + \log p_\theta(x) \underbrace{\mathbb{E}_{q_\phi}\left[1\right]}_{\text{equal to } 1} \\
&= -\mathbb{E}_{q_\phi}\left[\log \frac{p_\theta(x|z)p_Z(z)}{q_\phi(z|x)}\right] + \log p_\theta(x)
\end{aligned}
$$

Rearranging the terms, we have:

$$
\underbrace{\log p_\theta(x)}_{\text{fixed quantity}} = \underbrace{D_{KL}\left(q_\phi(z|x) \parallel p_\theta(z|x)\right)}_{\geqslant 0} + \underbrace{\mathbb{E}_{q_\phi}\left[\log \frac{p_\theta(x|z)p_Z(z)}{q_\phi(z|x)}\right]}_{\mathcal{L}}.
$$

The key insight from this derivation is that the KL divergence term can be minimized indirectly by maximizing $\mathcal{L}$ since, once x is given, $\log p_\theta(x)$ is a fixed quantity w.r.t. the parameters of q_ϕ. As the KL divergence is always non-negative, $\mathcal{L}$ is a lower bound on $\log p_\theta(x)$ and is known as the variational lower bound or the evidence lower bound (ELBO). Note that we can compute all of the terms in $\mathcal{L}$:

- $q_\phi(z|x)$ is our chosen Gaussian distribution for importance sampling with parameters $\mu_\phi(x)$ and $\sigma_\phi(x)$ given by the inference network q_ϕ.
- $p_\theta(x|z)$ is the output of the generator network p_θ.
- $p_Z(z)$ is our prior which we chose to be easy to sample from.

The ELBO can then be used as a proxy objective and the model can be trained to find optimal parameters ϕ and θ with familiar gradient-based optimization methods. However, there is an additional problem with optimizing the inference network q_ϕ: the latent variables z are generated by random sampling which is non-differentiable and so the gradi-

ents w.r.t. ϕ cannot be backpropagated. To address this, we can use the *reparameterization trick*.

Reparameterization Trick

The reparameterization trick rewrites the random variable x as some deterministic function of noise. For the case when the random variable is distributed according to a Gaussian distribution, $x \sim \mathcal{N}(\mu, \sigma^2 I)$, it can be rewritten as follows:

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

We can therefore reparameterize our samples z according to the parameters learned by our inference network such that:

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

This isolates the random component of the network, allowing the gradients w.r.t. ϕ to be computed and the VAE to be optimized using stochastic gradient descent or any variant.

The name "variational autoencoders" comes from the fact that we use the variational method of maximizing the ELBO and that the model is structured similarly to an autoencoder. We can see this more clearly if we expand the ELBO term as follows:

$$\begin{aligned}
\mathcal{L} &= \mathbb{E}_{q_\phi(z|x)}\left[\log \frac{p_\theta(x|z)p_Z(z)}{q_\phi(z|x)}\right] \\
&= \mathbb{E}_{q_\phi(z|x)}\left[\log p_\theta(x|z) + \log \frac{p_Z(z)}{q_\phi(z|x)}\right] \\
&= \mathbb{E}_{q_\phi(z|x)}\left[\log p_\theta(x|z)\right] + \mathbb{E}_{q_\phi(z|x)}\left[\log \frac{p_Z(z)}{q_\phi(z|x)}\right] \\
&= \underbrace{\mathbb{E}_{q_\phi(z|x)}\left[\log p_\theta(x|z)\right]}_{\text{reconstruction term}} - \underbrace{D_{KL}\left(q_\phi(z|x) \parallel p_Z(z)\right)}_{\text{prior-matching term}}
\end{aligned}$$

Like deterministic autoencoders, VAEs learn a latent representation of input data by pushing it through a bottleneck layer and minimizing a reconstruction loss. The training objective in VAEs has an additional KL term which can be seen as a regularization term encouraging the distribution of the latent space to match a prior. Instead of encoding input as a single point in latent space, VAEs encode input as a distribution $q_\phi(z|x)$ over the latent space. We can therefore think of the inference network $q_\phi(z|x)$ as an encoder. The generator network $p_\theta(x|z)$ is simultaneously learned, which transforms a given latent z into x and can be treated as a decoder. A graphical representation of the model is depicted in fig. 6.

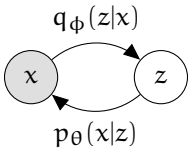


Figure 6: Encoder $q_\phi(z|x)$ encodes input x as a distribution over the latent variables z and decoder $p_\theta(x|z)$ decodes the latent variables z back into observations x.

VAEs maximize the likelihood of the data indirectly by maximizing the ELBO. This means that VAEs cannot evaluate exact likelihood of samples. Flow-based models represent the exact likelihood and are explored in the next section.

3.3 FLOW-BASED MODELS

The contents of the following section are based on [12, 18].

Flow-based models are a class of generative models that transform one distribution to another by constructing a sequence of transformations which "flow" samples from the initial distribution to the target. The initial distribution, which we call the *base distribution*, is typically a simple one (such as a standard Gaussian) and the target is typically a more complex distribution such as the one which generated samples in a dataset. Flow-based models are also called *normalizing flows* in the literature, referring to the notion that the flow "normalizes" a complex distribution to a simple one and is particularly well-suited when the base distribution is a standard Gaussian i.e. a standard *normal* distribution. The main idea of flow-based models is to express a random variable x from the target distribution as some transformation of a random variable z[1] from the known base distribution $p_Z(z)$:

$$\begin{aligned} x &= f(z) \\ z &\sim p_Z(z). \end{aligned}$$

The transformation f is typically a sequence of transformations that iteratively morph the sample from the base distribution to our target distribution. Under some restrictions on the type of transformations permitted, the density of the target distribution can be learned through learning the sequence of transformations from the initial distribution. The goal of flow-based models is then to learn this transformation f (or its inverse f^{-1}), which we will consider to be parameterized by a deep neural network.

[1] While this is sometimes referred to as a latent variable in the normalizing flows literature, this terminology is ill-suited as highlighted by [18] since upon observing x, the corresponding $z = f^{-1}(x)$ is uniquely determined and thus not really latent. This variable also differs from the latent variable previously introduced in that its dimension must be equal to that of the variable x.

The central component of flow-based models relies on the change of variables formula. We recall it here, briefly. Given a random variable $z \sim p_Z(z)$ and a differentiable, invertible function f which maps z to x, the change of variables formula tells how to compute the probability density of x:

$$\begin{aligned} x &= f(z) \\ p_\theta(x) &= p_Z(f^{-1}(x)) \left| \frac{\partial f^{-1}(x)}{\partial x} \right| \\ \left| \frac{\partial f^{-1}(x)}{\partial x} \right| &= |\det \mathbf{J}_{f^{-1}}(x)|, \end{aligned}$$

where $|\det \mathbf{J}_{f^{-1}}(x)|$ is the absolute value of the Jacobian determinant of $f^{-1}(x)$ and

$$\mathbf{J}_{f^{-1}}(x) = \begin{bmatrix} \frac{\partial f_1^{-1}}{\partial x_1} & \cdots & \frac{\partial f_1^{-1}}{\partial x_D} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_D^{-1}}{\partial x_1} & \cdots & \frac{\partial f_D^{-1}}{\partial x_D} \end{bmatrix}, \tag{1}$$

where D is the dimension of x. This term accounts for the change in volume caused by f.

Restrictions on the Sequence of Transformations

The defining property of flow-based models is the restriction placed on the types of transformations permitted in the sequence. Each transformation must be invertible and both the transformation and its inverse must be differentiable. Such transformations are known as *diffeomorphisms*. A useful property of diffeomorphisms is that they are composable. A composition of two diffeomorphic transformations is also diffeomorphic. Suppose f_1 and f_2 are diffeomorphic transformations. Their composition $f_2 \circ f_1$ is also diffeomorphic and its inverse and Jacobian determinant are well-defined:

$$\begin{aligned} (f_2 \circ f_1)^{-1} &= f_1^{-1} \circ f_2^{-1} \\ \det \mathbf{J}_{f_2 \circ f_1}(z) &= \det \mathbf{J}_{f_2}(f_1(z)) \det \mathbf{J}_{f_1}(z). \end{aligned}$$

This property allows for complex transformations to be constructed through a composition of simpler transformations while satisfying the invertibility and differentiability

requirements. A flow from z to x can then be defined as chain of T transformations as follows:

$$\begin{aligned} f &= f_T \circ \cdots \circ f_1 \\ x_t &= f_t(x_{t-1}) \quad \text{for } t = 1, \cdots, T \\ x_0 &= z \\ x_T &= x. \end{aligned}$$

The flow defines a path that samples from $p_Z(z)$ traverse as they are transformed by the sequence. The transformation of the density is illustrated in fig. 7.

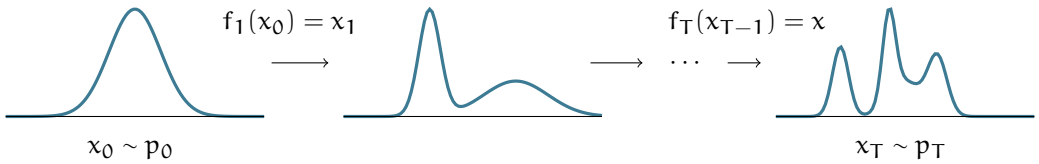


Figure 7: The flow transforms $x_0 = z$ to $x_T = x$ through a chain of transformations f.

As each f_t is restricted to be diffeomorphic, the density of x is well-defined by the change-of-variables formula and can be computed by the product of the density under the base distribution and the change in volume induced by the sequence of transformations, which is the Jacobian determinant of the composed transformation:

$$\begin{aligned} x = x_T &= f_T \circ \cdots \circ f_1(x_0) \\ p_\theta(x) &= p_T(x_T) \\ &= p_0(x_0) \left|\det J_{f_T \circ \cdots \circ f_1}(x_0)\right|^{-1} \\ &= p_0(x_0) \prod_{t=1}^{T} \left|\det J_{f_t}(x_{t-1})\right|^{-1} \\ \log p_\theta(x) &= \log p_0(x_0) - \sum_{t=1}^{T} \log \left|\det J_{f_T}(x_{T-1})\right| \end{aligned}$$

Training then amounts to maximizing the log-likelihood of the data:

$$\theta_{MLE} = \operatorname*{argmin}_\theta - \sum_{n=1}^{N} \log p_\theta(x).$$

While the density is well-defined, it assumes the ability to efficiently compute the inverse transformation f^{-1} and its Jacobian determinant. Generating a new sample from $p_\theta(x)$ amounts to sampling from the base distribution $p_Z(z)$ and pushing it through the transformation f. Thus, this operation assumes the ability to easily sample from $p_Z(z)$ and compute the transformation f. These two main operations have different computational requirements which may be at odds. The choice of whether to model f or f^{-1} will depend on the desired application. A large amount of the research effort in normalizing flows is dedicated to designing neural network architectures that ensure invertibility and tractability of the Jacobian determinant.

Continuous Normalizing Flows

Normalizing flows construct flows by chaining together a sequence of T one-step transformations such that $x_t = f_t(x_{t-1})$ for $t = 1, \ldots, T$. Continuous normalizing flows adopt the strategy of constructing flows in continuous time by learning the continuous dynamics of the flow and applying integration to find the transformation. This amounts to defining an ordinary differential equation (ODE) describing the evolution of the flow in time:

$$\begin{aligned} dx_t &= v_\theta(x_t, t)dt \\ x_{t_0} &= z \\ x_{t_1} &= x, \end{aligned}$$

where v_θ is a function parameterized by a neural network and time t runs continuously from t_0 to t_1. The benefit of this formulation is that unlike normalizing flows, the learnable function v_θ has minimal requirements, namely that it is continuous in t and differentiable. The transformation $x = f(z)$ is defined by applying the learned dynamics forward in time through integration:

$$\begin{aligned} x &= f(z) \\ &= z + \int_{t=t_0}^{t_1} v_\theta(x_t, t)dt. \end{aligned}$$

The inverse transformation is defined by:

$$\begin{aligned} z &= f^{-1}(x) \\ &= x + \int_{t=t_1}^{t_0} v_\theta(x_t, t)dt \\ &= x - \int_{t=t_0}^{t_1} v_\theta(x_t, t)dt, \end{aligned}$$

which is simply running the dynamics backwards in time by changing the sign of the integral. This property means that the transformation and its inverse have the same computational complexity, thus bypassing the design choice.

Analogous to Jacobian determinant which accounts for the change in volume in normalizing flows, the change in log density for continuous-time flows is defined by:

$$\frac{d \log p(x_t)}{dt} = -\text{Tr}\left(\mathbf{J}_{v_\theta}(x_t, t)\right),$$

where $\text{Tr}(\cdot)$ is the trace operator and $\mathbf{J}_{v_\theta}(x_t, t)$ is the Jacobian of v_θ evaluated at (x_t, t).

While computing the trace of a D-dimensional matrix may not be tractable if D is large, it can be approximated with:

$$\text{Tr}\left(\mathbf{J}_{v_\theta}(x_t, t)\right) = v^T \mathbf{J}_{v_\theta}(x_t, t) v,$$

where v is any D-dimensional random vector with zero mean and unit covariance. This is known as the *Hutchinson's trace estimator* and is efficient even when D is large.

The log density of the target distribution can then be computed by integrating the change in log density:

$$\log p_\theta(x) = \log p_Z(z) - \int_{t=t_0}^{t_1} \text{Tr}\left(\mathbf{J}_{v_\theta}(x_t, t)\right) dt.$$

There is typically no analytical solution for v_θ and a numerical solver must be used in practice.

Euler's method for Solving ODEs

Decades of research effort has been devoted to developing numerical solvers for ODEs and many of the existing methods can be leveraged to solve for v_θ. One of simplest methods is known as Euler's method. To numerically solve an ODE, a discretization scheme must be defined in order for the dynamics to be simulated in a computer. Euler's method is one such scheme which discretizes the ODE using some small step size $\Delta t > 0$ and transforming the state of the variable in time as follows:

$$x_{t+\Delta t} = x_t + \Delta t \, v_\theta(x_t, t).$$

The ODE can then be simulated by running this scheme from $x_{t_0} = z$ to $x_{t_1} = x$.

We can see that the closer Δt is to zero, the more accurate the solution will be. While a very small step-size Δt will lead to a more accurate solution, it will require more computations. However, when Δt is large, the discretization will incur errors and is not numerically stable.

Optimizing Continuous Normalizing Flows

Finding the optimal parameters of our model then amounts to maximizing the log likelihood of the data. However, in order to train our model to learn the parameters θ of v, we need to backpropagate the gradients of the objective with respect to the intermediate states x_t through the ODE solver which is computationally demanding. It can be shown that the gradient of some objective function $\mathcal{L}(\theta, x)$ w.r.t. the intermediate states of the flow x_t is characterized by the following ODE:

$$\frac{d}{dt}\left(\frac{\partial \mathcal{L}}{\partial x_t}\right) = -\left(\frac{\partial \mathcal{L}}{\partial x_t}\right)^T \frac{\partial v_\theta(x_t, t)}{\partial x_t},$$

known as the *adjoint sensitivity method*, we refer the reader to [18] for more information. The gradient w.r.t. the parameters θ is defined by:

$$\frac{\partial \mathcal{L}}{\partial \theta} = \int_{t=t_1}^{t_0} \frac{\partial \mathcal{L}}{\partial x_t} \frac{\partial v_\theta(x_t, t)}{\partial \theta} dt,$$

and can be thought of as backpropagation in continuous time. While continuous normalizing flows are capable of modeling arbitrary continuous-time paths for probability distributions, training them by running these numerical ODE simulations is computationally expensive.

Flow-based models use deterministic trajectories to transform latent variables to the data space. Stochastic trajectories can also be used and are the basis of diffusion models, which is the topic of the next section.

3.4 DIFFUSION MODELS

The contents of the following section are based on [16].

Drawing inspiration from non-equilibrium thermodynamics [20], diffusion models are a class of generative models that iteratively corrupt data samples into random noise during a *forward process* and learn the *reverse process* to generate novel data samples starting from random noise. Specifically, each data point x_0 is transformed into x_T by passing it through T steps of a probabilistic encoder $q(x_t|x_{t-1})$. Given a large enough T, x_T should look like random noise, as enough noise will have been injected to destroy any structure in the data. It is analytically convenient to use additive Gaussian noise as the encoder such that $x_T \sim \mathcal{N}(0, \mathbf{I})$. We can then take a random sample from $\mathcal{N}(0, \mathbf{I})$ and pass it through T steps of a decoder $p_\theta(x_{t-1}|x_t)$ to get x_0. This process is illustrated in fig. 8.

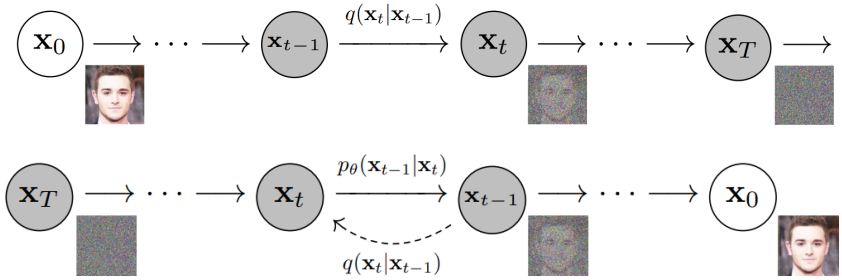


Figure 8: Illustration of how diffusion models work [10].

In this section, we will explore two prominent formulations of diffusion models, denoising diffusion probabilistic models and score-based generative models.

Denoising Diffusion Probabilistic Models

Denoising Diffusion Probabilistic Models (DDPM) [10] formulate a diffusion model as two parameterized Markov chains:

- a forward chain with a fixed transition kernel that iteratively corrupts the data to noise.
- a reverse chain with a transition kernel parameterized by a deep neural network that learns to transform noise back into data.

The predefined forward Markov transition kernel is parameterized by Gaussian distribution, allowing closed-form evaluation. Given an initial data distribution, $q(x_0)$ and data sample $x_0 \sim q(x_0)$, the forward process is defined with the Markov transition kernel $q(x_t|x_{t-1})$ such that:

$$q(x_t|x_{t-1}) = \mathcal{N}\left(\sqrt{1-\beta_t}x_{t-1}, \beta_t I\right),$$

which iteratively adds noise to sample x_{t-1} at each time step t, according to some chosen variance schedule with $\beta_t \in (0,1)$. By the chain rule of probability and the Markov property, the joint distribution of $x_1, x_2, \ldots, x_T$ conditioned on x_0 can be factorized into:

$$q(x_{1:T}|x_0) = \prod_{t=1}^{T} q(x_t|x_{t-1}).$$

The analytical convenience of parameterizing the transitions with a Gaussian distribution presents itself when we try to sample x_t at any arbitrary time step t.

The random variable x_t can be expressed as a transformation of x_{t-1} and standard Gaussian noise:

$$x_t = \sqrt{1-\beta_t}x_{t-1} + \sqrt{\beta_t}\epsilon_t, \quad \epsilon_t \sim \mathcal{N}(0, I).$$

Let $\alpha_t = 1 - \beta_t$, $\bar{\alpha}_t = \prod_{i=1}^{t} \alpha_i$:

$$\begin{aligned} x_t &= \sqrt{\alpha_t} x_{t-1} + \sqrt{1-\alpha_t}\epsilon_t \\ x_{t-1} &= \sqrt{\alpha_{t-1}} x_{t-2} + \sqrt{1-\alpha_{t-1}}\epsilon_{t-1}, \end{aligned}$$

where $\epsilon_t \sim \mathcal{N}(0, I)$. We can substitute x_{t-1} in the first equation to obtain:

$$x_t = \sqrt{\alpha_t \alpha_{t-1}} x_{t-2} + \sqrt{\alpha_t (1-\alpha_{t-1})}\epsilon_{t-1} + \sqrt{1-\alpha_t}\epsilon_t,$$

where $\sqrt{\alpha_t (1-\alpha_{t-1})}\epsilon_{t-1} + \sqrt{1-\alpha_t}\epsilon_t$ is the sum of two Gaussians distributed as $\mathcal{N}(0, \alpha_t (1-\alpha_{t-1}) I)$ and $\mathcal{N}(0, \alpha_t (1-\alpha_{t-1}) I)$, respectively. Using the sum of independent Gaussians property, this can be expressed as a single Gaussian with mean equal to the sum of the means and covariance equal to the sum of the covariances:

$$\begin{aligned} x_t &= \sqrt{\alpha_t \alpha_{t-1}} x_{t-2} + \sqrt{\alpha_t (1-\alpha_{t-1}) + 1 - \alpha_t}\,\bar{\epsilon} \\ &= \sqrt{\alpha_t \alpha_{t-1}} x_{t-2} + \sqrt{1-\alpha_t \alpha_{t-1}}\,\bar{\epsilon}, \end{aligned}$$

where $\bar{\epsilon} \sim \mathcal{N}(0, I)$. All $t-1$ intermediate steps can be marginalized out in this way to obtain a closed form solution for $q(x_t|x_0)$:

$$\begin{aligned} x_t &= \sqrt{\alpha_t} x_{t-1} + \sqrt{1-\alpha_t}\epsilon_t \\ &= \sqrt{\alpha_t \alpha_{t-1}} x_{t-2} + \sqrt{1-\alpha_t \alpha_{t-1}}\,\bar{\epsilon} \\ &\vdots \\ &= \sqrt{\prod_{i=1}^{t} \alpha_i}\, x_0 + \sqrt{1 - \prod_{i=1}^{t} \alpha_i}\,\epsilon \\ &= \sqrt{\bar{\alpha}_t} x_0 + \sqrt{1-\bar{\alpha}_t}\epsilon, \end{aligned}$$

where $\epsilon \sim \mathcal{N}(0, I)$. Thus, given x_0, x_t is distributed as follows:

$$q(x_t|x_0) = \mathcal{N}\left(\sqrt{\bar{\alpha}_t} x_0, (1-\bar{\alpha}_t) I\right).$$

The reverse transformation is also parameterized by Gaussian distributions, with mean $\mu_\theta(x_t, t)$ and variance $\Sigma_\theta(x_t, t)$ at each time step t, learned by a deep neural network. This reverse process transforms a sample $x_T \sim \mathcal{N}(0, I)$ from a standard Gaussian prior by gradually removing noise at each step t from $t = T$ to $t = 0$. The reverse Markov transition kernel is then be defined as:

$$p_\theta(x_{t-1}|x_t) = \mathcal{N}\left(\mu_\theta(x_t, t), \Sigma_\theta(x_t, t)\right),$$

which iteratively removes noise from sample x_t at each time step t, according to the learned parameters. As with the forward process, the joint distribution of $x_T, x_{t-1}, \ldots, x_0$ can be factorized into:

$$p_\theta(x_{0:T}) = \prod_{t=1}^{T} p_\theta(x_{t-1}|x_t)p(x_T),$$

where $p(x_T)$ is the Gaussian prior which does not depend on θ.

Thus at each time step, the neural network is approximating a reversal of the amount of noise added in the equivalent time step during the forward process.

The task is then to learn the parameters of transition kernel defining the reverse process which approximate the noise added during the forward process. This can be done by minimizing the KL divergence between distributions defining the respective transition kernels:

$$\begin{aligned} D_{KL}\left(q(x_{1:T}|x_0) \parallel p_\theta(x_{0:T}|x_0)\right) &= \mathbb{E}_{q(x_{1:T}|x_0)}\left[\log \frac{q(x_{1:T}|x_0)}{p_\theta(x_{0:T})/p_\theta(x_0)}\right] \\ &= \mathbb{E}_{q(x_{1:T}|x_0)}\left[\log \frac{q(x_{1:T}|x_0)}{p_\theta(x_{0:T})} + \log p_\theta(x_0)\right] \\ &= \mathbb{E}_{q(x_{1:T}|x_0)}\left[\log \frac{q(x_{1:T}|x_0)}{p_\theta(x_{0:T})}\right] + \mathbb{E}_{q(x_0)}\left[\log p_\theta(x_0)\right]. \end{aligned}$$

Rearranging the terms, we have:

$$\underbrace{\mathbb{E}_{q(x_0)}\left[\log p_\theta(x_0)\right]}_{\text{log likelihood}} = \underbrace{D_{KL}\left(q(x_{1:T}|x_0) \parallel p_\theta(x_{0:T}|x_0)\right)}_{\geqslant 0} - \underbrace{\mathbb{E}_{q(x_{1:T}|x_0)}\left[\log \frac{q(x_{1:T}|x_0)}{p_\theta(x_{0:T})}\right]}_{\mathcal{L}}$$

As in the case with VAEs, we have derived an ELBO which we can maximize in order to indirectly minimize the KL divergence between the transition kernels. We therefore can use $\mathcal{L}$ as a proxy objective to learn our transition kernel.

It can be shown that the ELBO $\mathcal{L}$ decomposes into the following terms [10]:

$$\mathcal{L} = \mathbb{E}_q\left[\underbrace{D_{KL}\left(q(x_T|x_0) \parallel p(x_T)\right)}_{\text{constant w.r.t. } \theta} + \sum_{t=2}^{T} \underbrace{D_{KL}\left(q(x_{t-1}|x_t, x_0) \parallel p_\theta(x_{t-1}|x_t)\right)}_{\text{comparisons between Gaussians}} - \log p_\theta(x_0|x_1)\right]$$

Writing the ELBO in this way allows the KL terms to be computed in closed form. The last term $\log p_\theta(x_0|x_1)$ is evaluated separately using $\mathcal{N}(\mu_\theta(x_1, t), \Sigma_\theta(x_1, 1))$.

Score-based Generative Models

By the rules of probability, in order for a probability density function $p(x)$ to be valid, it must satisfy certain properties:

1. $p(x) \geqslant 0$
2. $\int_X p(x)dx = 1$.

An arbitrary function $f(x)$ can be transformed into a valid probability distribution in the following manner:

$$p(x) = \frac{e^{f(x)}}{Z}$$
$$Z = \int_X e^{f(x)}dx,$$

as exponentiation ensures positivity and normalizing by the constant Z ensures that $\int_X p(x)dx = 1$.

The benefit of this framework is that it is very general, as any function $f(x)$ that outputs a scalar can be used to parameterize the model. We will of course parameterize $f(x)$ with a neural network $f_\theta(x)$. However, the normalizing constant Z_θ of the neural network poses a problem as it is typically intractable to compute. The main idea behind score-based generative models [21] is to instead model the *score function* of the density rather than directly learning the distribution of the data. The score function of a probability density function $p(x)$ is defined as the gradient of the log probability density $\nabla_x \log p(x)$. This gradient is with respect to the data x and not the parameters of the model. It is a vector field pointing towards the direction where data density grows the most.

Modeling the score function has the desirable property of avoiding the normalizing constant Z_θ altogether as it does not depend on the input data x and therefore $\nabla_x Z_\theta = 0$. The score function of $p_\theta(x)$ can then be simplified:

$$\begin{aligned}\nabla_x \log p_\theta(x) &= \nabla_x \log \frac{e^{f_\theta(x)}}{Z_\theta} \\ &= \nabla_x f_\theta(x) - \underbrace{\nabla_x \log Z_\theta}_{\text{equal to 0}} \\ &= \nabla_x f_\theta(x).\end{aligned}$$

Score Matching

The task is then to minimize the dissimilarity between the score function $\nabla_x \log p_\theta(x)$ and the true score function $\nabla_x \log p_D(x)$. This can be done by minimizing the *Fisher Divergence*:

$$D_F(p_D \parallel p_\theta) = \mathbb{E}_{p_D}\left[\frac{1}{2}\|\nabla_x \log p_D(x) - \nabla_x \log p_\theta(x)\|^2\right]$$

The Fisher divergence captures the MSE between the gradient vector fields of $\log p_D(x)$ and learned distribution $\log p_\theta(x)$ w.r.t. the input. The optimal parameters θ^* are those which minimize this divergence:

$$\theta^* = \underset{\theta}{\operatorname{argmin}}\, D_F\left(p_D \parallel p_\theta\right)$$

However we don't know true value of $\nabla_x \log p_D(x)$.

The idea behind score matching is to rewrite the Fisher divergence equation without the unknown distribution $p_D(x)$. It can be shown that the minimizing the Fisher divergence is equivalent to the following [11]:

$$\theta^* = \underset{\theta}{\operatorname{argmin}}\, \mathbb{E}_{p_D}\left[\operatorname{Tr}(\mathbf{J}_x \nabla_x \log p_\theta(x)) + \frac{1}{2}\|\nabla_x \log p_\theta(x)\|_2^2\right]$$

This revised objective function does not contain the problematic $p_D(x)$ term. However, computing the trace of the Jacobian poses another challenge as it requires $O(D^2)$ time where D is the dimensionality of the input data. This can be prohibitively expensive when D is large, which is often the case in reality. Additionally, this objective function requires that $\log p_D(x)$ satisfies the regularity condition that it is continuously differentiable and finite everywhere, which often does not hold in practice.

Denoising Score Matching

To alleviate the condition of continuous differentiability, noise can be added to data points such that $\tilde{x} = x + \epsilon, \epsilon \sim p(\epsilon)$. Provided that $p(\epsilon)$ is smooth, the resulting noisy distribution $q(\tilde{x}) = \int q(\tilde{x}|x)p_D(x)dx$ is also smooth. Thus, the Fisher divergence can be used as a valid objective:

$$D_F\left(q \parallel p_\theta\right) = \mathbb{E}_{q(\tilde{x})}\left[\frac{1}{2}\|\nabla_{\tilde{x}} \log p_\theta(\tilde{x}) - \nabla_{\tilde{x}} \log q(\tilde{x})\|_2^2\right].$$

This however still requires the trace of the Jacobian. Denoising score matching [24] proposes a scalable approximation by showing that:

$$\begin{aligned} D_F\left(q \parallel p_\theta\right) &= \mathbb{E}_q\left[\frac{1}{2}\|\nabla_{\tilde{x}} \log p_\theta(\tilde{x}) - \nabla_{\tilde{x}} \log q(\tilde{x})\|_2^2\right] \\ &= \mathbb{E}_q\left[\frac{1}{2}\|\nabla_{\tilde{x}} \log p_\theta(\tilde{x}) - \nabla_{\tilde{x}} \log q(\tilde{x}|x)\|_2^2\right] + \text{constant}, \\ &\approx \frac{1}{2}\,\mathbb{E}_q\left[\|s_\theta(\tilde{x}) - \frac{(x-\tilde{x})}{\sigma^2}\|_2^2\right] + \text{constant}, \end{aligned}$$

where $s_\theta(\tilde{x}) = \nabla_{\tilde{x}} \log p_\theta(\tilde{x})$ and $\nabla_x \log q(\tilde{x}|x) = \nabla_x \log \mathcal{N}(\tilde{x}|x, \sigma^2 I) = \frac{(x-\tilde{x})}{\sigma^2} + \text{constant}$.

The underlying intuition is that following the gradient of the log density at some noisy point $\tilde{x}$ should move towards the uncorrupted point x, i.e. the directional term $x - \tilde{x}$. The objective is therefore regressing the score function on this directional vector.

A data point can be generated by first drawing a sample from the corrupted distribution and moving along the gradient of the density:

$$\tilde{x}_t = \tilde{x}_{t-1} + \alpha s_\theta(\tilde{x}) + \sqrt{\alpha}\epsilon_t, \quad \epsilon_t \sim \mathcal{N}(0, I),$$

where α is the step size and ϵ_t is random noise drawn at every time step to avoid getting stuck in local optima. This is known as *Langevin sampling* and can be seen as a form of gradient ascent along the probability density of the data.

The question then becomes how much noise to add during the corruption process. Levels of noise which are too high will lead to a bad approximation, while levels which are too low will make it difficult for the Langevin sampling to traverse the between modes of the distribution, particularly if there are disjoint regions. Score-matching generative models [27] propose corrupting the data with different scales of Gaussian noise, yielding different distributions:

$$q(\tilde{x}|x) = \mathcal{N}(\tilde{x}|x, \sigma^2 I)$$
$$q(\tilde{x}) = \int p_D(x) q(\tilde{x}|x) dx.$$

To perform Langevin sampling, a sample is draw from the distribution with the highest noise scale first, then smoothly reduces the magnitudes of the noise scales until reaching the smallest scale. This is known as *annealed Langevin dynamics*. In practice, a single neural network $s_\theta(\tilde{x}, \sigma)$ is used which approximates the distribution for each noise scale σ and is optimized with a weighted sum of the score matching objectives. Using the denoising score matching objective, the weighted objective function can be written as:

$$\mathcal{L}(\theta, \sigma_{1:T}) = \sum_{t=1}^{T} \lambda_t \mathcal{L}(\theta, \sigma_t)$$
$$\mathcal{L}(\theta, \sigma_t) = \frac{1}{2}\mathbb{E}_q \left[\| s_\theta(\tilde{x}, \sigma_t) + \frac{(\tilde{x} - x)}{\sigma_t^2} \|_2^2 \right],$$

where $\lambda_t > 0$ is a weighting term for the denoising score matching objective function associated with σ_t and $\sigma_1 > \sigma_2 > \cdots > \sigma_T$.

Diffusion Models as SDEs

DDPMs and score-based generative models can be interpreted as discretizations of a stochastic differential equation (SDE) describing a diffusion process of the following form:

$$dx_t = f(x_t, t)dt + g(t)dw,$$

where $f(x_t, t)$ is known as the *drift coefficient*, $g(t)$ is called the *diffusion coefficient* and w is standard Brownian motion.

Analagous to the ODE case, sampling from this equation requires drawing an initial sample x_0 from the source distribution and simulating its trajectory with a numerical solver. The Euler-Maruyama method is an extension of Euler's method for discretizing SDEs:

$$x_{t+\Delta t} = x_t + \Delta t\, f(x_t, t) + \sqrt{\Delta t}\, \mathcal{N}(0, I)\, g(t),$$

where Δt is the step size. The SDE can then be simulated from $t = 0$ to $t = 1$.

In order to generate samples from this model, we must be able to reverse the SDE. It can be shown that any SDE of the form described above has the following reverse-time SDE [3]:

$$dx_t = \left[f(x_t, t) - g(t)^2 \nabla_x \log p_t(x_t) \right] dt + g(t) d\bar{w},$$

where $\bar{w}$ is standard Brownian motion when time flows backwards, dt is an infinitesimal negative time step and $\nabla_x \log p_t(x_t)$ is the score function, which can be approximated with a neural network $s_\theta(x_t, t)$ trained by denoising score matching. Setting $f(x_t, t) = \frac{1}{2}\beta(t)$ and $g(t) = \sqrt{\beta(t)}$, novel samples can be generated by drawing an initial sample x_t from standard Gaussian noise and evolving it according to the Euler-Maruyama scheme:

$$x_{t-1} = x_t + \frac{1}{2}\beta(t) \left[x_t + 2 s_\theta(x_t, t) \right] \Delta t + \sqrt{\beta(t) \Delta t}\, \mathcal{N}(0, I).$$

Probability Flow ODE

To improve the sampling speed of diffusion models, an ODE variant of the SDE was proposed by removing the stochastic term, deemed the *probability flow ODE*:

$$dx_t = h(x_t, t) dt = \left[f(x_t, t) - \frac{1}{2} g(t)^2 \nabla_x \log p_t(x_t) \right] dt,$$

where $\nabla_x \log p_t(x_t)$ is again the score function which can be approximated by a neural network $s_\theta(x_t, t)$. Setting $f(x_t, t) = \frac{1}{2}\beta(t)$ and $g(t) = \sqrt{\beta(t)}$, novel samples can be generated by drawing an initial sample x_t from standard Gaussian noise and evolving it according to Euler's scheme:

$$x_{t-1} = x_t + \frac{1}{2}\beta(t) \left[x_t + s_\theta(x_t, t) \right] \Delta t.$$

It can be shown that samples from this ODE results in samples from the same marginal distribution as the SDE variant. We can see that this is a special case of a continuous normalizing flow model, where the model learns to approximate the score function.

3.5 RECTIFIED FLOWS

The content in this section is based on [14].

The SDE formulation of diffusion models highlights the source of the limitation on sampling speed. As the SDE must be simulated in order to generate a sample, large step sizes incur large discretization errors, resulting in poor samples. For this reason, sufficiently small steps are required for samples of high quality to be generated which necessitates slow sampling speed if the trajectory contains high curvature. It would then be beneficial to place a preference for trajectories which are straight, as straight paths incur no discretization error when solved numerically. This is the main idea behind rectified flows [14].

Given two distributions, there are typically infinitely many maps which transform one to the other. The goal of a generative model is to learn one of these mappings (usually from a noise distribution to a target data distribution). Rectified flows are continuous normalizing flow models which enforce certain properties on the map learned, namely that it is straight.

Given a source distribution p_0 and a target distribution p_1, rectified flows learn a mapping implicitly by constructing an ODE:

$$\begin{aligned} dx_t = v(x_t, t)dt, \quad t \in [0, 1], \\ x_0 \sim p_0 \\ x_1 \sim p_1. \end{aligned}$$

The key insight is that the desired drift v can be constructed by arbitrary interpolations of points from the source to the target. Specifically, the desired ODE should traverse the linear interpolation of points from p_0 to p_1.

Let x_0 and x_1 be samples drawn from p_0 and p_1 respectively. The linear interpolation x_t for any arbitrary $t \in [0, 1]$ of x_0 and x_1 is defined as:

$$x_t = tx_1 + (1 - t)x_0, \quad t \in [0, 1].$$

An ODE $dx_t = v_\theta(x_t, t)dt$, parameterized by a deep neural network, that traverses the desired path can be learned by minimizing its MSE with the direction $x_1 - x_0$:

$$\begin{aligned} \mathcal{L}_{rec}(\theta) = \|(x_1 - x_0) - v_\theta(x_t, t)\|^2 \\ v_\theta(z, t) = \mathbb{E}\left[x_1 - x_0 | x_t = z\right], \end{aligned}$$

where $v_\theta(z, t)$ can be seen as the average direction of the lines passing through point z at time t. The objective function $\mathcal{L}_{rec}(\theta)$ can be minimized with an off-the-shelf optimizer such as stochastic gradient descent or a variant.

This is effectively a simulation-free approach for training continuous normalizing flows. Instead of expensive simulations of the ODE to compute the gradients w.r.t. the intermediate variables induced by the flow, the proposed objective function directly regresses

on a target vector field that generates the desired flow. This training technique was also proposed in concurrent works, namely [13], which refers to this method as *flow matching* and [2], which describes the method as building normalizing flows with *stochastic interpolants*.

Reflow for Fast Generation

Rectified flow can be recursively applied to produce a k-rectified flow, yielding progressively straighter flows. Let the rectified flow $X = \{x_t : t \in [0,1]\}$, induced from (x_0, x_1) be denoted as $X = \text{Rectflow}((x_0, x_1))$. Then the $(k+1)$-rectified flow can be defined as:

$$X^{k+1} = \text{Rectflow}((x_0^k, x_1^k)),$$

where $(x_0^0, x_1^0) = (x_0, x_1)$. In practice this means a rectified flow model is recursively fine-tuned with samples (x_0^k, x_1^k) drawn from the k-th rectified flow by using them as couplings for training objective (i.e. replacing (x_0, x_1) with (x_0^k, x_1^k)). This is referred to as a *reflow* procedure. With increasing values of k, the straightness of trajectories should improve but the optimal value of k will depend on the application.

Distillation for One-Step Prediction

The sampling speed of a k-th rectified flow can be further improved by distillation. Using the k-th rectified flow as a "teacher" model. We can train a new "student" neural network $\hat{T}(x_0) = x_0 + v_\theta(z_0, 0)$ to directly predict x_1^k from x_0^k without simulating the ODE. This can be done by minimizing the following loss:

$$\mathcal{L}_{\texttt{distill}}(\theta) = \mathbb{E}\left[\|(x_1^k - x_0^k) - v_\theta(x_0^k, 0)\|^2\right]$$

Rectified flows aim to leverage straight paths for fast sampling. If an ODE $dx_t = v_\theta(x_t, t)dt$ has perfectly straight paths then:

$$x_t = x_0 + tv_\theta(x_0, 0),$$

which can be solved exactly in a single step. The neural network in the distillation stage is trying to predict the solution at $t = 1$.

Rectified flows is therefore a framework which proposes three stages of training:

- A flow stage, during which a continuous normalizing flow is trained in a simulation-free manner to learn the desired flow, enforcing a straightness property.
- A reflow stage, during which the rectified flow is recursively applied k times, with couplings sampled from the previous rectified flow, in order to further straighten the paths.
- A distill stage, in which a neural network is trained to predict x_1^k directly from x_0^k.

CONCLUSION

In summary, we presented a survey of methods and underlying principles in likelihood-based deep generative modeling.

VAEs jointly optimize an inference network and a generator network by maximizing the likelihood of the data indirectly through maximizing the ELBO.

Flow-based models place restrictions on the types of transformations applied to a known source distribution, allowing the explicit distribution to be tracked as it is transformed and permitting the exact likelihood to be represented. The constraints on the types of transformations also restrict the types of neural network architectures permitted, limiting their ability to benefit from progress in scalable architectures.

Diffusion models add random noise to destroy structure in data in a fixed forward process and learn a reversal of the process to generate novel samples from noise. This can be interpreted as learning a time-reversed SDE which is trained by matching the forward and reverse vector fields. Sampling can be done by discretizing the learned time-reversed SDE in order to evolve a sample drawn from noise to a point in the target distribution using an SDE solver. The sampling speed is limited by the curvature in the SDE trajectory, as larger step sizes incur larger error. Rectified flows expand on this observation and enforce a straightness property on the forward and reverse trajectories by restricting intermediate states to be linear interpolations of points from the source to the target. The straighter trajectories allow for faster sampling as larger step sizes in the ODE discretization scheme will incur relatively smaller errors. This process of obtaining rectified flows can be recursively applied to a model to further straighten trajectories. Given that the trajectories are straight enough, a distillation stage can be applied for fast sampling by directly predicting the target point from the source point.

In the next chapter, we explore the task of 3D point cloud representation and generation. We then present an application of rectified flows for fast shape generation with 3D point clouds and conduct an experiment investigating the effect on sample quality of replacing the loss function used during the distillation stage of training.

4 SHAPE GENERATION WITH 3D POINT CLOUDS

4.1 POINT CLOUDS

A point cloud is a disordered set of discrete points. In the context of 3D shape generation, the point cloud can be seen as a set of samples from the surface of the shape. An example is shown in fig. 9. Point clouds are a popular choice for representing and generating 3D shapes on account of them of being the direct output of widely-used depth sensors, such as LiDAR sensors.

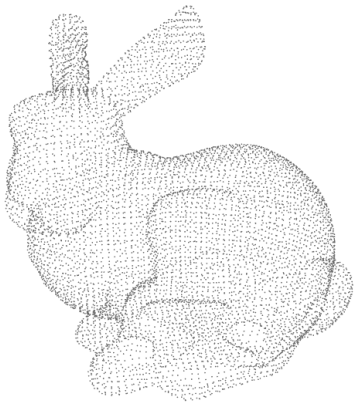


Figure 9: A point cloud representation of the Stanford Bunny [23].

The various deep generative models discussed in the preceding sections have enjoyed massive success in the realm of 2D generative tasks such as image generation. However, the neural network architectures employed in these models are typically designed to process regular grid data (like images) and cannot be naively applied to irregular structures such as point clouds.

4.2 PROCESSING POINT CLOUDS

Historically, methods for processing point cloud data can be categorized into two approaches:

- *Voxel-based* methods transform the irregular data into a regular 3D grid (or **vo**lume pi**xel**s) in order to leverage the widely used techniques applied to 2D grids (such as convolutional neural networks). As with the 2D case, the quality of the representation is limited by the resolution of the grid. Low resolutions lead to information

loss, while high resolutions permit fine details to be captured. In the 3D case however, memory-requirements for voxel-based representations grow cubically w.r.t. the input resolution, presenting a memory-intensive challenge when fine details are required.

- *Point-based* methods, on the other hand, work directly on the points to extract features locally by aggregating neighbouring features of each point. By leveraging the inherent sparsity of point cloud data, this bypasses the high memory demands of voxel-based methods but the irregularity results in the relevant data being non-contiguous (i.e. points and their neighbours are not directly next to each other in memory) and requires random memory accesses during the nearest-neighbour search, which is computationally expensive. This results in a large portion of the computation-time being dedicated to finding neighbours in these methods as opposed to extracting features [15].

A Hybrid Approach

Point-voxel convolutional neural networks (PVCNN) [15] combine the advantages of both approaches, representing the input data as points to leverage the low memory footprint of sparse representation and performing feature extraction on voxels to exploit the contiguous memory access patterns of grid structures. This is accomplished through separation of fine-grained feature transformation in a *points branch* and coarse-grained neighbour aggregation in a *voxel branch*. This separation is depicted in fig. 10.

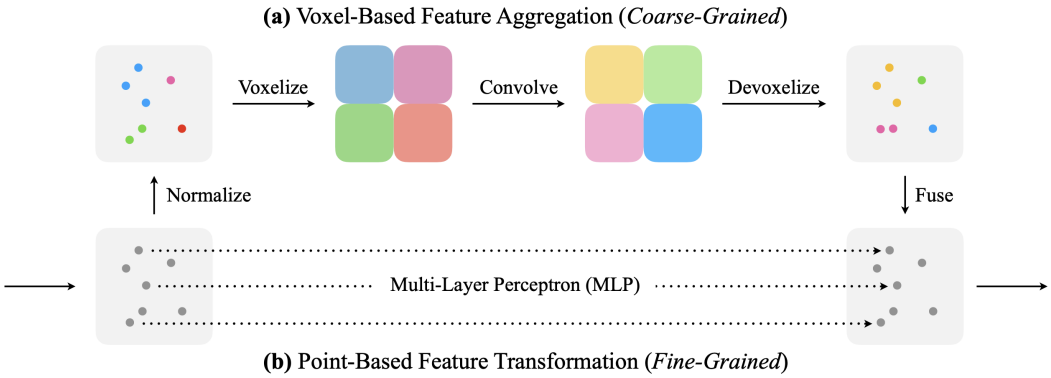


Figure 10: PVCNN separates fine-grained feature transformation and coarse-grained neighbourhood feature aggregation [15].

4.3 MEASURING DISTANCES BETWEEN POINT CLOUDS

The contents of this section are based on [17, 1].

An important component of learning representation and generation of 3D point clouds is choosing a metric to measure the dissimilarity between two point clouds. As point

clouds are permutation-invariant, meaning that any reordering of the points should yield the same point cloud, the metric should also be permutation invariant. The most popular choices in the literature are the *Chamfer discrepancy* and the *Earth Mover's Distance* [17], which are both permutation-invariant.

Chamfer Discrepancy

For two point clouds P and Q, the Chamfer discrepancy (CD) between them is defined as:

$$D_{CD}(P, Q) = \frac{1}{|P|} \sum_{x \in P} \min_{y \in Q} \|x - y\|_2^2 + \frac{1}{|Q|} \sum_{y \in Q} \min_{x \in P} \|x - y\|_2^2.$$

For each point in point cloud P, the distance to its nearest neighbour in point cloud Q, shown in fig. 11, is computed and squared. The squared distances are then averaged across all points in P. This process is repeated for all points in point cloud Q, computing the distance to its nearest neighbour in point cloud P and taking the average of the squared distances. The two averages are then summed to find the CD. Minimizing the CD between a reference P and reconstructed sample Q is the preferred choice in representation learning for 3D point cloud data as it is easy and efficient to compute. However, it is not invariant to rotation and scaling of one of the point clouds and as long as supports of P and Q are close, then the CD will be small even if their corresponding distributions are different.

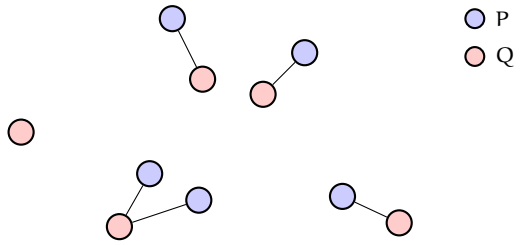


Figure 11: The CD calculates the average squared distance from a point in each point cloud to its nearest neighbour in the other point cloud. The lines connect the points in P to its nearest neighbour in Q.

Earth Mover's Distance

The Earth Mover's distance (EMD) between point clouds P and Q, containing an equal number of points, is defined as:

$$D_{EMD}(P, Q) = \min_{T: P \to Q} \sum_{x \in P} \|x - T(x)\|_2,$$

where T is a one-to-one mapping or *bijection* between P and Q. This can be thought of as minimizing a notion of the cost, as captured by $\|\cdot\|$, required to transport mass from P to Q. A example is illustrated in fig. 12.

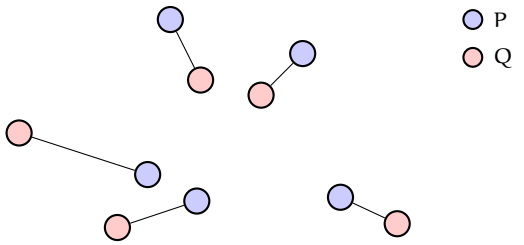


Figure 12: The EMD finds the optimal bijection between points in P and points in Q which minimizes the distance between them.

In the context of point clouds, the EMD has been shown to be better than the CD as a loss function for reconstruction tasks [1, 6]. It can also be shown that the EMD is an upper bound on the CD [17, lemma 1]:

$$D_{CD}(P, Q) \leqslant 2K\, D_{EMD}(P, Q),$$

where K is the diameter of the convex hull bounding the support of P and Q. This suggests that minimizing the EMD will also minimize the CD, but minimizing the CD does not necessarily minimize the EMD.

It is however more expensive to compute the EMD as its computational complexity grows cubically w.r.t. to the size of the input [17].

The EMD, as defined above, is a special case of the *Wasserstein distance*, which is a measure of the minimal cost, specified by some distance d, required to transform one probability distribution to another and can be used to quantify the distance between them. More formally, let μ, ν be probability measures on Ω and $\pi \in \Pi(\mu, \nu)$ be the set of probability measures on the product space $\Omega \times \Omega$ such that:

$$\pi(\cdot, \Omega) = \mu$$
$$\pi(\Omega, \cdot) = \nu,$$

i.e. the set of joint distributions for which the marginals are distributed as μ and ν. For $p \geqslant 1$, the p-Wasserstein distance is defined as:

$$W_p(\mu, \nu) = \inf_{\pi \in \Pi(\mu,\nu)} \left\{ \mathbb{E}_\pi \left(d(x, y)^p \right) \right\}^{\frac{1}{p}},$$

where x and y are random variables marginally distributed as μ and ν, respectively. When $p = 1$, the Wasserstein distance is known as the Earth Mover's Distance, and in the one-dimensional case where $\Omega = \mathbb{R}$ and $d(x, y) = \|x - y\|$, has a closed-form solution:

$$W_p(\mu, \nu) = \left(\int_0^1 |F_X^{-1}(t) - F_Y^{-1}(t)|^p \, dt \right)^{\frac{1}{p}},$$

where F_X and F_Y are the cumulative distribution functions of x and y respectively. There is no closed-form solution when the dimension is larger than one.

Sliced Wasserstein Distance

The sliced Wasserstein distance is an approximation of the Wasserstein distance obtained by taking an average of several EMDs in one dimension. This allows us to leverage the closed-form solution of the one-dimensional EMD, making it far more computationally efficient. The key idea behind the sliced Wasserstein distance to project the measures onto a random direction on the unit sphere and compute the EMD between the projections.

More formally, let μ and ν be the target measures and their projections on the direction θ on the unit sphere be denoted as $\pi_\theta \# \mu$ and $\pi_\theta \# \nu$, respectively. The sliced Wasserstein distance of order p is then defined as:

$$SW_p(\mu, \nu) = \left(\int_{S^{d-1}} W_p \left(\pi_\theta \# \mu, \pi_\theta \# \nu \right) d\theta \right)^{\frac{1}{p}}.$$

The integral defining this distance is intractable in practice and is typically estimated using Monte Carlo integration:

$$\begin{aligned} SW_p(\mu, \nu) &= \left(\int_{S^{d-1}} W_p \left(\pi_\theta \# \mu, \pi_\theta \# \nu \right) d\theta \right)^{\frac{1}{p}} \\ &\approx \left(\frac{1}{N} \sum_{i=1}^{N} W_p \left(\pi_{\theta_i} \# \mu, \pi_{\theta_i} \# \nu \right) \right)^{\frac{1}{p}}, \end{aligned}$$

where d is the dimensionality of the point cloud and $\{\theta_i, i \in N\}$ are N uniformly sampled directions on the unit sphere S^{d-1}. The number of sampled projections or "slices", N, is a hyperparameter that can be tuned. For 3D point clouds, the authors of [17] suggest $N = 100$ as sufficient.

4.4 EVALUATING GENERATION QUALITY

Various methods for measuring quality of point cloud generation have been proposed. The most-often reported metrics are those proposed in [1], namely the Jensen-Shannon divergence(JSD), coverage (COV) and minimum matching distance (MMD) in which two sets of point clouds A and B are compared. The typical setting is that A is a set of generated point clouds and B is a set of reference point clouds, like a held-out test set.

Jensen-Shannon Divergence

Assuming a canonical voxel grid and axis-aligned point cloud data, the Jensen-Shannon divergence measures the degree to which the point clouds tend to occupy similar locations.

$$\begin{aligned} JSD(P_A \parallel P_B) &= \frac{1}{2} D_{KL}(P_A \parallel M) + \frac{1}{2} D_{KL}(P_B \parallel M) \\ M &= \frac{1}{2}(P_A + P_B), \end{aligned}$$

where P_A, P_B are the empirical distributions obtained by counting the number of points lying within each voxel across all point clouds in A and B respectively and $D_{KL}(\cdot \parallel \cdot)$ is the KL divergence. A low JSD indicates that the point clouds tend to occupy similar locations.

Coverage

Coverage measures the degree to which one set of point clouds includes the diversity represented in another set of point clouds. For each point cloud in set A, its closest neighbour in set B is found using either the CD or the EMD. The coverage is then computed as the fraction of point clouds in set B that were matched as the nearest neighbour of at least one point cloud in set A. A high coverage score indicates that most of set B is represented in set A.

Minimum Matching Distance

As coverage doesn't measure how well the covered samples are represented in set A (only whether different point clouds in B were matched as the nearest neighbour is considered, not how close they were), minimum matching distance is a method proposed to measure fidelity of the generated set of point clouds. It is the average distance from a point cloud in the reference set to its nearest neighbour in the generated set:

$$MMD(A, B) = \frac{1}{|B|} \sum_{Y \in B} \min_{X \in A} D(X, Y),$$

where $|B|$ denotes the cardinality of set B and $D(\cdot, \cdot)$ is either the CD or the EMD. A lower MMD indicates better fidelity.

1-Nearest Neighbor Accuracy

The metrics described thus far have been criticized for their various limitations, most notably by [27]. The JSD only considers the marginal point distributions and not the distribution of the individual shapes, meaning that a model which always outputs the

average shape can obtain a perfect JSD score without learning any useful representation. Coverage is criticized for not capturing generation quality at all, as distances between point clouds in the reference set and generated set can be arbitrarily large. MMD is limited in that it only requires a single point cloud in a generated set to be close to the average of the points clouds in the reference set in order to achieve a relatively good score. To this end, the 1-Nearest neighbour (1-NN) accuracy was proposed as a better metric for evaluating generation quality for point clouds [27]. 1-NN accuracy is a method for assessing whether two distributions are identical. Let $S_{-X} = A \cup B - \{X\}$, i.e. the union of the sets A and B with the point cloud X removed, and N_X be the nearest neighbour of X in S_{-X}. The 1-NN accuracy is defined as:

$$1 - \text{NN}(A, B) = \frac{\sum_{X \in A} \mathbb{I}\left[N_X \in A\right] + \sum_{Y \in B} \mathbb{I}\left[N_Y \in B\right]}{|A| + |B|},$$

where $\mathbb{I}[\cdot]$ is the indicator function. This uses a 1-NN classifier to determine, for each sample, whether it comes from set A or set B according to the label of its nearest neighbour. If A and B are identical, the accuracy of this classifier should be 50% as a sample should be equally likely to be classified to either set. Therefore the closer the accuracy is to 50%, the more similar are A and B. The nearest neighbour calculation is computed using either the CD or the EMD.

4.5 FAST POINT CLOUD GENERATION WITH STRAIGHT FLOWS

Point-Voxel Diffusion (PVD) [28] is a DDPM model for 3D point cloud generation, parameterized by a PVCNN. The proposed model is able to generate samples of high quality, reporting state-of-the-art results for 1-NN accuracy performance. However, the model suffers from a slow sampling speed, as is the case with diffusion-based models. Point Straight Flows (PSF) [26] is a model proposed for fast sampling of 3D point clouds. It builds on the PVD model, replacing the diffusion component with flow matching to leverage rectified flows and distillation for one-step generation, vastly improving the sampling speed. We experiment with this model in the next chapter.

5

EXPERIMENTS ON SHAPE GENERATION WITH 3D POINT CLOUDS

5.1 EXPERIMENTAL SETUP

We base our experiments on the Point Straight Flows (PSF) model [26], which applies flow matching to learn a mapping from standard Gaussian noise to 3D point clouds. The neural network architecture parameterizing the model is a PVCNN as described in [28]. PSF is a model trained to improve the sampling speed of 3D point cloud generation. The model accomplishes this by rectifying the flow of the learned mapping from noise to data to improve the straightness of the trajectories. The flow is then distilled by minimizing the CD between a sample generated after one step of the ODE solver and the sample generated after N steps of the ODE solver.

Motivation

The authors of [17] advocate using the sliced Wasserstein distance in place of the CD for representation learning in 3D point cloud tasks. The justification for this is under the assumption that [17, Lemma 1] is true, minimizing the EMD will minimize the CD, while minimizing CD will not necessarily minimize the EMD and that the sliced Wasserstein distance is an estimate of the EMD, with computational complexity similar to the CD. The aim of this experiment is to investigate the effect of using a sliced-Wasserstein loss in place of the Chamfer loss on the quality of the samples generated by the PSF model.

Dataset

ShapeNet [5] is a large-scale data repository containing 3D models from 55 shape categories. The ShapeNet dataset used in this section is provided by the public code repository of [27], containing 15,000 sampled points for each shape. We use the shape categories of *airplane*, *chair* and *car* and sample 2,048 points for both training and testing stages. All point clouds are normalized during pre-processing by centering their bounding box to the origin and scaling them to fit within a unit cube.

Model

The code is based on the publicly available codebase for the Point Straight Flows model [26]. We leave the neural network architecture unchanged.

Let the initial state $x_0 \in \mathbb{R}^{M\times 3}$ be a sample drawn from standard Gaussian noise and $x_1 \in \mathbb{R}^{M\times 3}$ be a point cloud sampled from the dataset. In all our experiments, the number

of points M in the point cloud is 2048. The model is parameterized by a neural network, denoted v_θ which represents the velocity force of the following ODE process:

$$dx_t = v_\theta(x_t, t)dt \quad t \in [0, 1],$$

where x_t is the intermediate point cloud at time t.

Training

Recall the training stages described in the rectified flows section of chapter 2. The model in our experiment is accordingly trained in those three stages.

- **Flow stage**: For sample x_1 drawn from the data and x_0 drawn from standard Gaussian noise, the objective function is the MSE between the output of the network $v_\theta(x_t, t)$ and the desired direction $x_1 - x_0$, which is a straight line from the noise to the data:
$$\mathcal{L}_{rec}(\theta) = \|(x_1 - x_0) - v_\theta(x_t, t)\|^2,$$
where $x_t = tx_1 + (1-t)x_0$ is the linear interpolation between the data and the noise, and t is uniformly sampled between $[0, 1]$. The loss is minimized using an Adam optimizer with a learning rate of 2×10^{-4} for 1500 epochs.

- **Reflow stage**: We then generate 45,000 samples to create couplings for the second training stage to rectify the flow. To generate the samples, first x_0 is drawn from standard Gaussian noise and x_1 is the result of simulating the following ODE from $t = 0$ to $t = 1000$, starting with the initial state x_0, using an Euler-based solver:
$$x_{t+1} = x_t + v_\theta(x_t, t)\Delta t,$$
where $\Delta t = \frac{1}{1000}$ and $v_\theta(x_t, t)$ denotes the output of the pretrained model from the first stage of training. For each pair (x_0, x_1) generated, the model is trained with the same MSE loss as the first stage and is minimized with an Adam optimizer for 200 epochs, using a learning rate of 2×10^{-5}. Note that the x_0 in this stage is not redrawn from standard Gaussian noise during training but is the Gaussian noise that was used when generating the samples and simulated to reach x_1.

- **Distillation stage**: In the third stage, we train two separate models with the different loss functions of interest. The loss functions are constructed as follows:
$$\mathcal{L}_{distill}(\theta) = D(\hat{x}_1, x_1)$$
$$\hat{x}_1 = x_0 + v_\theta(x_0, 0),$$
where v_θ is the pretrained model from the second stage, (x_0, x_1) are the samples generated for the second stage and $D(\cdot, \cdot)$ is the distance function. The distance function used for the first model is the CD, provided in the *pytorch3d* library [19] and the distance used for the second model is the sliced Wasserstein distance with $N = 100$ projections, as suggested in [17]. We use the sliced Wasserstein distance implementation provided by the *Python Optimal Transport* library [7].

Both losses are minimized with an Adam optimizer with a learning rate of 2×10^{-5} for 200 epochs.

Evaluation Metrics

Following the convention in prior works on 3D point clouds [4, 27, 26, 28], we report the COV, MMD, JSD and 1-NN accuracy as the metrics for generation quality.

5.2 RESULTS

After the two models are trained in the distillaton stage, we evaluate each using the 10,000 samples in the test set as the reference points. The results of 1-NN accuracy test are shown in table 1.

Table 1: 1-NN accuracy test results. Both CD and EMD as the distance metric are calculated. Lower scores indicate better quality and diversity. The numbers are reported after multiplying them by 10^2.

Loss Type	1-NN-CD	1-NN-EMD
Chamfer	**72.02**	64.44
SWD	78.82	**58.73**

We find that distilling the model with a sliced Wasserstein distance achieves a better 1-NN accuracy with the EMD as the distance metric, with a margin of 5.71. The Chamfer model achieves a better 1-NN accuracy with the CD as the distance metric, with a margin of 6.8. In both cases, the margin between the accuracy scores is between 5 and 7, which we consider to be significant in a practical sense.

Table 2: COV test results. Both the CD and the EMD as the distance metric are calculated. Higher scores indicate better diversity. The numbers are reported after multiplying them by 10^2.

Loss Type	COV-CD	COV-EMD
Chamfer	**45.73**	48.13
SWD	43.48	**53.27**

The results of the COV test are reported in table 2. The Chamfer loss model achieves better results with the CD, while the sliced Wasserstein model achieves better results with the EMD as the distance metric. Recall that the COV metric only indicates the fraction of the reference set which was considered the nearest neighbour to at least one point cloud in the generated set. The results when using the CD as the distance metric in table 2 are very close, meaning that only a relatively few more point clouds in the reference set were considered to be the nearest-neighbour to at least one point cloud generated by each model. When using EMD as the distance metric, the margins are slightly more significant.

The results of the MMD test capture the average distance of the nearest neighbour as measured by the CD and the EMD and are reported in table 3.

Table 3: MMD test results. Both the CD and the EMD as the distance metric are calculated. Lower scores indicate better quality. The EMD numbers are scaled by 10^3 and the CD numbers are scaled by 10^4.

Loss Type	MMD-CD	MMD-EMD
Chamfer	**14.85**	10.50
SWD	15.42	**10.30**

Looking at the results in table 3, the Chamfer loss model performs better when the CD is used as the distance metric and the sliced Wasserstein loss model performs better when the EMD is used as the distance metric. When paired with the COV results in table 2, we can see that while a higher number of point clouds in the reference set were considered to be the nearest neighbour of at least one point cloud in the generated set for the sliced Wasserstein model when the EMD is used as the distance metric, the average distance to the nearest neighbour in the reference set for both models is very similar for both distance metrics.

Table 4: JSD test results. Lower scores indicate better quality. The numbers are scaled by 10^3.

Loss Type	JSD
Chamfer	**14.26**
SWD	14.51

The results of the JSD test, reported in table 4, show very similar results, indicating that on average the generated shapes tend to occupy similar locations as those in the reference set. The results of the COV, MMD and JSD tests all indicate that replacing the Chamfer loss with a sliced Wasserstein loss does not degrade those quality metrics and that both models perform similarly.

Qualitative results generated by the Chamfer loss model and sliced Wasserstein loss model are displayed in fig. 13 and fig. 14, respectively.

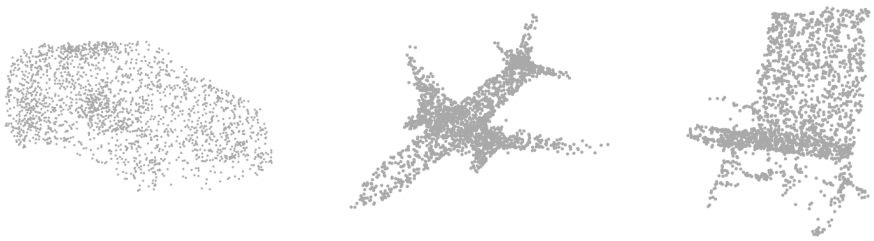


Figure 13: Samples generated by Chamfer loss model.

Figure 14: Samples generated by SWD loss model.

The samples generated by the sliced Wasserstein loss model appear noisier than those generated by the Chamfer loss model. This is particularly noticeable in the airplane example in fig. 14. A reason for this may be that the reflow samples used for the distillation stage may contain some noise which the Wasserstein metric considers as an intrinsic part of the distribution of the reference point cloud, while the CD is only concerned with the nearest neighbour in the reference point cloud, ignoring any outliers like noise. While the assessment is subjective, it could be argued that the sliced Wasserstein model generated the better-looking chair, which could suggest that the chair shape is easier for the sliced Wasserstein model to learn than the airplane shape. The cars produced by both models appear to be of similar quality and don't appear to be noisy, which could also suggest that car shape is easier to learn than both the chair and airplane shapes. This could be because the car shape has fewer variations, while more variation exists chair and airplane shapes, making them more difficult to learn.

6

CONCLUSION

We sought to investigate the effect of replacing the Chamfer loss with a sliced Wasserstein loss on the quality of the samples generated by the PSF model. The results of the experiment indicate that the switch leads to better performance on quality metrics when the EMD distance metric is used. We expected the sliced Wasserstein model to perform better in the 1-NN accuracy test when both the CD and EMD were used as the distance metric, but found it only performed better when the EMD was used as the distance metric. A notable limitation of this experiment is the accuracy of the sliced Wasserstein distance as an estimate of the EMD. The accuracy of the estimate is dependent on the number of projections used, as well as the method of sampling the projection directions. Due to resource limitations, we used the suggested $N = 100$ projections in our computation but future work can investigate the effect of tuning this hyperparameter. We used uniform sampling on the unit sphere to obtain projection directions, but future work can also investigate other methods to sample the projection directions in order to minimize information loss during projection.

An additional limitation is that the distillation stage is heavily dependent on the samples generated by the pretrained flow model from the first stage of training. The metric used in the distillation stage is simply measuring the discrepancy between the sample generated in one step of the ODE solver and the same sample simulated for 100 steps. The reference point clouds are therefore not sampled from true data distribution, but are sampled from the distribution approximated by the flow model, which may itself be noisy or of low quality. The tests, on the other hand, use samples from the true data distribution, which is after all the intended distribution for the PSF model to approximate.

There are other avenues by which to improve the quality of the one-step generator. In particular, if straighter paths can be found in the first flow stage, then subsequent stages will benefit. There has been recent interest in altering the sampling method to create the couplings for flow matching. The proposed method samples the couplings independently. However, some couplings could have a higher probability of generating straighter trajectories than others. This line of investigation is also a promising direction for future work.

REFERENCES

[1] P. Achlioptas, O. Diamanti, I. Mitliagkas, and L. J. Guibas. Learning representations and generative models for 3d point clouds. In *International Conference on Machine Learning*, 2018. 10.48550/arXiv.1707.02392. [Cited on pages 36, 38, and 39.]

[2] M. S. Albergo and E. Vanden-Eijnden. Building normalizing flows with stochastic interpolants. In *The Eleventh International Conference on Learning Representations*, 2023. 10.48550/arXiv.2209.15571. [Cited on page 33.]

[3] B. D. Anderson. Reverse-time diffusion equation models. *Stochastic Processes and their Applications*, 1982. 10.1016/0304-4149(82)90051-5. [Cited on page 31.]

[4] R. Cai, G. Yang, H. Averbuch-Elor, Z. Hao, S. Belongie, N. Snavely, and B. Hariharan. Learning gradient fields for shape generation. In *Proceedings of the European Conference on Computer Vision (ECCV)*, 2020. 10.48550/arXiv.2008.06520. [Cited on page 44.]

[5] A. X. Chang, T. A. Funkhouser, L. J. Guibas, P. Hanrahan, Q.-X. Huang, Z. Li, S. Savarese, M. Savva, S. Song, H. Su, J. Xiao, L. Yi, and F. Yu. Shapenet: An information-rich 3d model repository. *CoRR*, 2015. 10.48550/arXiv.1512.03012. [Cited on page 42.]

[6] H. Fan, H. Su, and L. J. Guibas. A point set generation network for 3d object reconstruction from a single image. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017. 10.48550/arXiv.1612.00603. [Cited on page 38.]

[7] R. Flamary, N. Courty, A. Gramfort, M. Z. Alaya, A. Boisbunon, S. Chambon, L. Chapel, A. Corenflos, K. Fatras, N. Fournier, L. Gautheron, N. T. Gayraud, H. Janati, A. Rakotomamonjy, I. Redko, A. Rolet, A. Schutz, V. Seguy, D. J. Sutherland, R. Tavenard, A. Tong, and T. Vayer. Pot: Python optimal transport. *Journal of Machine Learning Research*, 2021. URL `http://jmlr.org/papers/v22/20-451.html`. [Cited on page 43.]

[8] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. URL `http://www.deeplearningbook.org`. [Cited on pages 8 and 12.]

[9] G. E. Hinton and R. Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 2006. 10.1126/science.1127647. URL `https://classes.cs.uoregon.edu/17S/cis607bddl/papers/Hinton2006.pdf`. [Cited on page 11.]

[10] J. Ho, A. Jain, and P. Abbeel. Denoising diffusion probabilistic models. *Neural Information Processing Systems*, 2020. 10.48550/arXiv.2006.11239. [Cited on pages 25 and 27.]

[11] A. Hyvärinen. Estimation of non-normalized statistical models by score matching. *Journal of Machine Learning Research*, 2005. URL `http://jmlr.org/papers/v6/hyvarinen05a.html`. [Cited on page 29.]

[12] I. Kobyzev, S. D. Prince, and M. A. Brubaker. Normalizing flows: An introduction and review of current methods. *IEEE Transactions on Pattern Analysis & Machine Intelligence*, 2021. 10.48550/arXiv.1908.09257. [Cited on page 19.]

[13] Y. Lipman, R. T. Q. Chen, H. Ben-Hamu, M. Nickel, and M. Le. Flow matching for generative modeling. In *The Eleventh International Conference on Learning Representations*, 2023. 10.48550/arXiv.2210.02747. [Cited on page 33.]

[14] X. Liu, C. Gong, and Q. Liu. Flow straight and fast: Learning to generate and transfer data with rectified flow. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*, 2023. 10.48550/arXiv.2209.03003. [Cited on page 32.]

[15] Z. Liu, H. Tang, Y. Lin, and S. Han. Point-voxel cnn for efficient 3d deep learning. In *Advances in Neural Information Processing Systems*, 2019. 10.48550/arXiv.1907.03739. [Cited on page 36.]

[16] K. P. Murphy. *Probabilistic Machine Learning: Advanced Topics*. MIT Press, 2023. URL `http://probml.github.io/book2`. [Cited on page 24.]

[17] T. Nguyen, Q.-H. Pham, T. Le, T. Pham, N. Ho, and B.-S. Hua. Point-set distances for learning representations of 3d point clouds. *2021 IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 10458–10467, 2021. 10.48550/arXiv.2102.04014. [Cited on pages 36, 37, 38, 39, 42, and 43.]

[18] G. Papamakarios, E. Nalisnick, D. J. Rezende, S. Mohamed, and B. Lakshminarayanan. Normalizing flows for probabilistic modeling and inference. *J. Mach. Learn. Res.*, 2021. 10.48550/arXiv.1912.02762. [Cited on pages 19 and 24.]

[19] N. Ravi, J. Reizenstein, D. Novotny, T. Gordon, W.-Y. Lo, J. Johnson, and G. Gkioxari. Accelerating 3d deep learning with pytorch3d. *arXiv:2007.08501*, 2020. 10.48550/arXiv.2007.08501. [Cited on page 43.]

[20] J. N. Sohl-Dickstein, E. A. Weiss, N. Maheswaranathan, and S. Ganguli. Deep unsupervised learning using nonequilibrium thermodynamics. *International Conference On Machine Learning*, 2015. 10.48550/arXiv.1503.03585. [Cited on page 24.]

[21] Y. Song and S. Ermon. Generative modeling by estimating gradients of the data distribution. *Neural Information Processing Systems*, 2019. 10.48550/arXiv.1907.05600. [Cited on page 28.]

[22] J. M. Tomczak. *Deep Generative Modeling*. Springer Nature, 2022. 10.1007/978-3-030-93158-2. [Cited on page 12.]

[23] G. Turk and M. Levoy. Zippered polygon meshes from range images. In *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '94, page 311–318. Association for Computing Machinery, 1994. 10.1145/192161.192241. [Cited on page 35.]

[24] P. Vincent. A connection between score matching and denoising autoencoders. *Neural Computation*, 2011. URL `https://www.iro.umontreal.ca/~vincentp/Publications/DenoisingScoreMatching_NeuralComp2011.pdf`. [Cited on page 29.]

[25] P. Vincent, H. Larochelle, Y. Bengio, and P.-A. Manzagol. Extracting and composing robust features with denoising autoencoders. In *International Conference on Machine Learning proceedings*. ACM, 2008. URL `https://www.cs.toronto.edu/~larocheh/publications/icml-2008-denoising-autoencoders.pdf`. [Cited on page 11.]

[26] L. Wu, D. Wang, C. Gong, X. Liu, Y. Xiong, R. Ranjan, R. Krishnamoorthi, V. Chandra, and Q. Liu. Fast point cloud generation with straight flows. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2023. 10.48550/arXiv.2212.01747. [Cited on pages 41, 42, and 44.]

[27] G. Yang, X. Huang, Z. Hao, M.-Y. Liu, S. Belongie, and B. Hariharan. Pointflow: 3d point cloud generation with continuous normalizing flows. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, 10 2019. 10.48550/arXiv.1906.12320. [Cited on pages 30, 40, 41, 42, and 44.]

[28] L. Zhou, Y. Du, and J. Wu. 3d shape generation and completion through point-voxel diffusion. *2021 IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 5806–5815, 2021. 10.48550/arXiv.2104.03670. [Cited on pages 41, 42, and 44.]