Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Matematyczny
specjalność: ogólna

Michał Stypułkowski

Representing Point Clouds with Generative Conditional Invertible Flow Networks

Praca magisterska
napisana pod kierunkiem
dr. hab. Jana Chorowskiego

Wrocław 2020

Abstract

We propose a simple yet effective method to represent point clouds as sets of samples drawn from a cloud-specific probability distribution. This interpretation matches the intrinsic characteristics of point clouds: the number of points and their ordering within a cloud is not important as all points are drawn from the proximity of the object boundary. We postulate to represent each cloud as a parameterized probability distribution defined by a generative neural network. To exploit similarities between same-class objects and to improve model performance, we turn to weight sharing: networks that model densities of points belonging to objects in the same family share all parameters with the exception of a small, object-specific embedding vector.

As a result, our model offers competitive or superior quantitative results on benchmark datasets, while enabling unprecedented capabilities to perform cloud manipulation tasks, such as point cloud reconstruction and alignment.

Prezentujemy prostą, lecz efektywną metodę reprezentacji chmur punktów, jako zbiorów punktów pochodzących z rozkładu prawdopodobieństwa charakterystycznego dla danej chmury. Ta interpretacja jest zgodna z cechami chmur punktów: liczba punktów i ich kolejność w chmurze nie są ważne, ponieważ wszystkie punkty pochodzą z bliskiego otoczenia powierzchni obiektu. Proponujemy, aby reprezentować każdą chmurę jako sparametryzowany rozkład prawdopodobieństwa, definiowany przez generatywną sieć neuronową. W celu wykorzystania podobieństwa pomiędzy obiektami tej samej klasy i poprawienia działania modelu, używamy współdzielenia wag: sieci modelujące rozkłady punktów należących do tej samej rodziny, współdzielą wszystkie parametry za wyjątkiem niewielkiego wektora, będącego zanurzeniem danego kształtu.

W rezultacie, nasz model osiąga zbliżone lub lepsze wyniki w eksperymentach, w porównaniu do wiodących metod, jednocześnie umożliwiając wykonywanie zadań na chmurach, takich jak rekonstrukcja i wyrównywanie.

Acknowledgements

The author wishes to thank Jan Chorowski for supervising this work, Kacper Kania, Tomasz Trzciński, Maciej Zamorski, and Maciej Zięba for collaboration on the publication, prof. Małgorzata Bogdan for valuable conversations, Tooploox for the possibility to create it during the AI Residency program, and the PLGrid project for computational resources on the Prometheus cluster.

Contents

Chapter 1

Introduction

People have a strong ability to recognize objects with little prior knowledge - given a couple of pictures of bikes, we can easily learn their common features and how they look like in general. When we see some new bike model, we can correctly and instantly classify it. Let's say we are given a piece of paper and a pencil. We are asked to draw a chair. When we finish, we draw a chair again. The process can be repeated an infinite number of times. Each of the drawings would be different, because we don't know the structure of just one chair, but we understand them in general.

Human beings are able to quickly learn many distributions of objects around us. We can tell if an unseen so far vehicle is indeed a bike by calculating the likelihood of it in bikes distribution. We can also draw an infinite number of chairs simply by sampling from learned chairs distribution.

Richard Feynman, an American theoretical physicist, said *What I cannot create, I do not understand.* This famous quote matches the intuition behind the purpose of generative models. The development of generative models is believed to be a promising step towards generalization in tasks tackled by artificial intelligence algorithms. We would like them to act like humans - know the whole structure of data, not only specific samples. These models are being trained in an unsupervised manner. They require no labeled data, and they can explore and discover patterns on their own.

Point clouds are one of the most popular ways of representing three-dimensional data. The main benefits of modeling point clouds can be noticed in applications of LIDAR, depth sensors, and time-of-flight cameras, where this representation of the surrounding environment is common. However, the raw point cloud is difficult to process due to intrinsic noise during data gathering. Moreover, these representations are often difficult to process. Our model provides hidden representations of input point clouds that can be efficiently analyzed. Additionally, the model is sound from the probability theory perspective since it models parametrized distribution, which resembles the original distribution of the data. It is also able to overcome most of

the difficulties regarding point clouds modeling described above.

Chapter 2

Background

2.1 Latent variable methods

In generative approach to the data modelling, we want to learn distribution of the data $p(x)$. However, we can do it explicitly only for objects sampled from very simple distributions, adjusting their parameters, e.g. mean and variance in Gaussian distribution. Another approach is to fit conditional probability $p(x|z)$, where z is a hidden variable, also called *latent variable*. Using conditional probability formula, we have:

$$p(x) = \int p(x, z)dz = \int p(x|z)p(z)dz \tag{2.1}$$

Now, if we are able o integrate over all possible z's, we can calculate the direct likelihood of the data.

2.1.1 Expectation-Maximization

One of the simple latent variable methods is the Expectation-Maximization (EM) algorithm. The latent variable z is usually discrete, e.g. indicates the distribution in Gaussian Mixture Model (GMM). In this case, $p(x)$ is tractable. The construction of the EM method allows us to easily calculate likelihood (2.1) in the Expectation step.

Let $x = (x_1, x_2, \ldots, x_n)$ be n independent identically distributed data points. We want to fit a model $p(x, z)$ with set of parameters θ, where z is an unobservable latent variable with values in $\{1, 2, \ldots, k\}$, with the log-likelihood expressed as:

$$\log p(x) = \sum_{m=1}^{n} \log p(x_m) = \sum_{m=1}^{n} \log \sum_{i=1}^{k} p(x_m, z_m = i). \tag{2.2}$$

Note that standard maximum likelihood estimation would not be trivial due to the sum inside of the logarithm. Instead, let us propose iterative approach to find optimal set of parameters. The EM algorithm allows us to construct a lower bound on $\log p(x)$ in the E-step and optimize it in the M-step.

Let us derive the lower bound. To do that, we will use Jensen's inequality.

Jensen's inequality Let X be a random variable with values in R_X. Let f be a convex function on R_X, such that $\mathbb{E}[f(X)]$ and $f(\mathbb{E}[X])$ are finite. Then:

$$\mathbb{E}[f(X)] \geq f(\mathbb{E}[X]). \tag{2.3}$$

Note that equality $\mathbb{E}[f(X)] = f(\mathbb{E}[X])$ holds if and only if f is strictly convex (i.e. $f''(x) > 0$ for all x), and X is constant. We will make use of Jensen's inequality for concave functions. In that case the direction of the inequality is reversed, i.e. $\mathbb{E}[f(X)] \leq f(\mathbb{E}[X])$. Equality holds for strictly concave functions.

We can now construct the lower bound of the log-likelihood:

$$\log p(x) = \sum_{m=1}^{n} \log \sum_{i=1}^{k} p(x_m, z_m = i) \tag{2.4}$$

$$= \sum_{m=1}^{n} \log \sum_{i=1}^{k} \frac{p(x_m, z_m = i)}{q_m(z_m = i)} q_m(z_m = i) \tag{2.5}$$

$$\geq \sum_{m=1}^{n} \sum_{i=1}^{k} \log \left[\frac{p(x_m, z_m = i)}{q_m(z_m = i)} \right] q_m(z_m = i), \tag{2.6}$$

where $q_m(z_m)$ are some distributions over z's. The last transition used Jensen's inequality for strictly concave function $f(x) = \log x$. $\sum_{i=1}^{k} \frac{p(x_m, z_m = i)}{q_m(z_m = i)} q_m(z_m = i)$ is an expected value of $\frac{p(x_m, z_m)}{q_m(z_m)}$ with respect to $z_m \sim q_m(z_m)$.

Now, we would like to choose $q_m(z_m)$'s to make the lower bound tight. Since $\log x$ is strictly concave, we need $\frac{p(x_m, z_m)}{q_m(z_m)}$ to be constant:

$$q_m(z_m) \propto p(x_m, z_m). \tag{2.7}$$

Since $q_m(z_m)$ is a probability distribution, we can write:

$$q_m(z_m) = \frac{p(x_m, z_m)}{\sum_{i=1}^{k} p(x_m, z_m = i)} \tag{2.8}$$

$$= \frac{p(x_m, z_m)}{p(x_m)} \tag{2.9}$$

$$= p(z_m | x_m). \tag{2.10}$$

Thus, in the Expectation step we simply choose $q_m(z_m)$'s to be posterior distribution of the z_m's given x_m's. Now, having the lower bound of the log-likelihood derived, we update θ to maximize it. This will be the Maximization step.

We will now show that this algorithm converges. Let $\theta^{(t)}$ and $\theta^{(t+1)}$ be the parameters from iterations t and $t+1$, respectively. Let $l(\theta^{(t)})$ be the log-likelihood calculated using parameters $\theta^{(t)}$. We will prove that $l(\theta^{(t)}) \leq l(\theta^{(t+1)})$:

$$l(\theta^{(t+1)}) \geq \sum_{m=1}^{n}\sum_{i=1}^{k} \log\left[\frac{p^{(t+1)}(x_m, z_m = i)}{q_m^{(t)}(z_m = i)}\right] q_m^{(t)}(z_m = i) \tag{2.11}$$

$$\geq \sum_{m=1}^{n}\sum_{i=1}^{k} \log\left[\frac{p^{(t)}(x_m, z_m = i)}{q_m^{(t)}(z_m = i)}\right] q_m^{(t)}(z_m = i) \tag{2.12}$$

$$= l(\theta^{(t)}). \tag{2.13}$$

The first step holds because:

$$l(\theta) \geq \sum_{m=1}^{n}\sum_{i=1}^{k} \log\left[\frac{p(x_m, z_m = i)}{q_m(z_m = i)}\right] q_m(z_m = i) \tag{2.14}$$

is true for any choice of θ and $q_m(z_m)$. The second step comes from the fact that the parameters $\theta^{(t+1)}$ were chosen to maximize $\sum_{m=1}^{n}\sum_{i=1}^{k} \log\left[\frac{p(x_m, z_m=i)}{q_m^{(t)}(z_m=i)}\right] q_m^{(t)}(z_m = i)$, and thus value of this expression calculated at $\theta^{(t+1)}$ has to be greater or equal than at $\theta^{(t)}$.

Gaussian Mixture Model (GMM)

For simplicity, we will introduce GMM for just one dimension, but the algorithm works analogically for higher dimensional data. Let $x = (x_1, x_2, \ldots, x_n)$ be n identically distributed 1-dimensional data points. We assume that they come from a mixture of k Gaussian distributions:

$$p(x_m) = \sum_{i=1}^{k} \pi_i f(x_m; \mu_i, \sigma_i), \tag{2.15}$$

where $\pi_i = p(z_m = i)$ is a probability of the i-th component of the mixture, $\sum_{i=1}^{k} \pi_i = 1$, and $f(x_m; \mu_i, \sigma_i) = p(x_m | z_m = i) = \frac{1}{\sqrt{2\pi\sigma_i^2}} \exp\left(-\frac{(x_m-\mu_i)^2}{2\sigma_i^2}\right)$ is a density function of a Gaussian distribution $\mathcal{N}(\mu_i, \sigma_i)$.

Our goal is to maximize data likelihood, or equivalently log-likelihood:

$$\log p(x) = \log \prod_{m=1}^{n} p(x_m) = \sum_{m=1}^{n} \log p(x_m) \tag{2.16}$$

$$= \sum_{m=1}^{n} \log \sum_{i=1}^{k} \pi_i f(x_m; \mu_i, \sigma_i). \tag{2.17}$$

Note that standard maximum likelihood estimation would not be trivial due to the sum inside of the logarithm. Instead, let us propose an iterative approach to find optimal set of parameters $\theta = \{\pi, \mu, \sigma\}$.

Expectation step

In the E step, for each data point x_m and each mixture i we calculate the posterior probability that x_m belongs to the i-th component $q_m(z_m = i) = p(z_m = i|x_m)$. We can use Bayes' rule to rewrite it as:

$$q_m(z_m = i) = p(z_m = i|x_m) = \frac{p(x_m|z_m = i)p(z_m = i)}{\sum_{j=1}^{k} p(x_m|z_m = j)p(z_m = j)} \tag{2.18}$$

$$= \frac{\pi_i f(x_m; \mu_i, \sigma_i)}{\sum_{j=1}^{k} \pi_j f(x_m; \mu_j, \sigma_j)}. \tag{2.19}$$

Maximization step

Now we want to find parameters θ that maximize the lower bound of the log-likelihood found in the E step:

$$\log p(x) \geq \sum_{m=1}^{n} \sum_{i=1}^{k} \log \left[\frac{p(x_m, z_m = i)}{q_m(z_m = i)} \right] q_m(z_m = i) \tag{2.20}$$

$$= \sum_{m=1}^{n} \sum_{i=1}^{k} (\log \pi_i + \log f(x_m; \mu_i, \sigma_i)) q_m(z_m = i). \tag{2.21}$$

In order to find optimal π, we need to use Lagrange multipliers under the constraint $\sum_{i=1}^{k} \pi_i = 1$. The solution is:

$$\pi_i^* = \frac{1}{n} \sum_{m=1}^{n} q_m(z_m = i). \tag{2.22}$$

Furthermore, we can calculate μ and σ that maximize $\mathbb{E}_{z \sim q_m(z_m)}[p(x)]$:

$$\mu_i^* = \frac{\sum_{m=1}^{n} q_m(z_m = i) x_m}{\sum_{m=1}^{n} q_m(z_m = i)}, \tag{2.23}$$

$$\sigma_i^{2*} = \frac{\sum_{m=1}^{n} q_m(z_m = i)(x_m - \mu_i^*)^2}{\sum_{m=1}^{n} q_m(z_m = i)}. \tag{2.24}$$

Both expectation and maximization steps are performed iteratively until convergence. Finally, we can calculate the likelihood of the data using found parameters:

$$p(x) = \prod_{m=1}^{n} \sum_{i=1}^{k} \pi_i^* f(x_m; \mu_i^*, \sigma_i^*). \tag{2.25}$$

2.1.2 Variational Autoencoders

Variational Autoencoder (VAE) [1] tries to overcome issue with continuous latent variables. Let's assume our data is generated from underlying unobserved variable

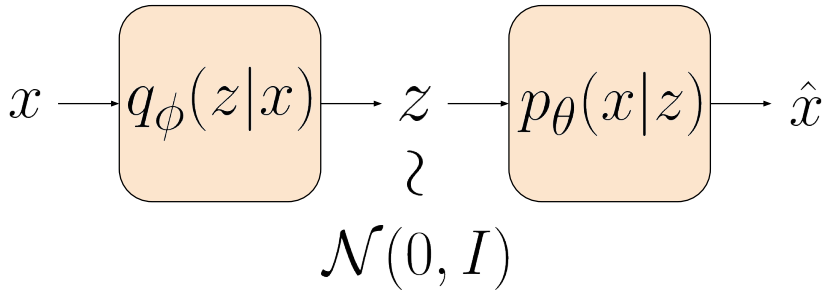

Figure 2.1: Architecture of VAE. Encoder q transforms input data into latent code z, which is normally distributed. Decoder p reconstruct the initial object x.

z with prior distribution $p_{\theta^*}(z)$, i.e. $x \sim p_{\theta^*}(x|z)$ and $z \sim p_{\theta^*}(z)$, where θ^* are true parameters of the model which we would like to estimate. We can approximate $p_{\theta^*}(x|z)$ with $p_\theta(x|z)$, where θ is a set of learnable parameters and choose prior latent distribution $p_\theta(z)$ to be simple, e.g. Gaussian. Function $p_\theta(x|z)$ will be called *decoder*.

We want to maximize likelihood of the data:

$$p_\theta(x) = \int p_\theta(x|z)p_\theta(z)dz. \tag{2.26}$$

In practice, this is intractable for a continuous variable z, because it requires integration over all possible values of z. We are also unable to calculate the posterior distribution:

$$p_\theta(z|x) = \frac{p_\theta(x|z)p_\theta(z)}{p_\theta(x)} \tag{2.27}$$

because of the likelihood in denominator.

The main idea of the VAE is to approximate the posterior density with a variational distribution $q_\phi(z|x)$, which we will call the *encoder*, where ϕ is another set of trainable parameters. The entire data flow is presented in Figure 2.1.

Having encoder and decoder, we can transform logarithm of the data likelihood into more convenient form:

$$\log p_\theta(x) = \log \int p_\theta(x,z)dz \tag{2.28}$$
$$= \log \int p_\theta(x,z)\frac{q_\phi(z|x)}{q_\phi(z|x)}dz \tag{2.29}$$
$$= \log \mathbb{E}_{z\sim q_\phi(z|x)}[\frac{p_\theta(x,z)}{q_\phi(z|x)}] \tag{2.30}$$
$$\geq \mathbb{E}_{z\sim q_\phi(z|x)}[\log \frac{p_\theta(x,z)}{q_\phi(z|x)}] \qquad \text{(Jensen's inequality)} \tag{2.31}$$
$$= \mathbb{E}_{z\sim q_\phi(z|x)}[\log \frac{p_\theta(x|z)p_\theta(z)}{q_\phi(z|x)}] \tag{2.32}$$
$$= \mathbb{E}_{z\sim q_\phi(z|x)}[\log p_\theta(x|z)] + \mathbb{E}_{z\sim q_\phi(z|x)}[\log \frac{p_\theta(z)}{q_\phi(z|x)}] \tag{2.33}$$
$$= \mathbb{E}_{z\sim q_\phi(z|x)}[\log p_\theta(x|z)] + D_{KL}(q_\phi(z|x)||p_\theta(z)) \tag{2.34}$$
$$= \mathcal{L}(x,\theta,\phi), \tag{2.35}$$

where D_{KL} is Kullback–Leibler divergence. $\mathcal{L}(x,\theta,\phi)$ is called *variational lower bound* or *ELBO*. We cannot optimize the likelihood directly, but we can find θ^* and ϕ^* that maximize ELBO, i.e.:

$$\theta^*, \phi^* = \underset{\theta,\phi}{\operatorname{argmax}}\, \mathcal{L}(x,\theta,\phi). \tag{2.36}$$

Intuitively, $\mathbb{E}_z[\log p_\theta(x|z)]$ maximizes the reconstruction accuracy. In case of images, it is negative L_2 distance between input sample and its reconstruction. Kullback–Leibler divergence measures how much information is missing between empirical distribution $q_\phi(z|x)$ and prior $p_\theta(z)$).

VAE can be seen as a continuous version of the EM algorithm. In the EM latent variable z is discrete, allowing us to explicitly calculate the likelihood of the data. The number of all possible z's is relatively small, thus integration over latent space is tractable. In VAEs, expectation step can be associated with minimization of $D_{KL}(q_\phi(z|x)||p_\theta(z|x))$, thus with optimization of ϕ. The maximization step, on the other hand, improves parameters θ, given expected distribution on the latent space. In contrast to EM, we are not able to set $q_\phi(z|x)$ explicitly to make the lower bound (2.31) tight with respect to θ. Instead, VAEs are trained using gradient methods and both ϕ and θ are optimized simultaneously.

2.1.3 Normalizing Flows

Discrete Normalizing Flows

Let us suppose we are given a bijective function f, such that $z = f(x)$ and $x = f^{-1}(z)$, and prior distribution $p(z)$. We can express the likelihood of the data $p(x)$

using f and $p(z)$ by change of variable formula:

$$p(x) = p(f(x))\left|\det\left(\frac{\partial f}{\partial x}\right)\right|, \tag{2.37}$$

where $\frac{\partial f}{\partial x}$ is a Jacobian matrix of the function f with respect to x. Note that in order to compute the Jacobian, the f's image needs to have the same dimensionality as the support, i.e. if $x \in \mathbb{R}^D$ then $z \in \mathbb{R}^D$.

To enable generation capabilities, normalizing flows seek such a function f. Having that, we can inverse the mapping process between x and z. Sampling z from prior distribution $p(z)$ and transforming it by f^{-1}, we get $x \sim p(x)$. Consequently, we can distinguish two operations:

- Inference

$$x \sim p(x) \tag{2.38}$$
$$z = f(x), \tag{2.39}$$

- Sampling

$$z \sim p(z) \tag{2.40}$$
$$x = f^{-1}(z). \tag{2.41}$$

Finding such a transformation is not trivial, especially in an experimental environment. There are several constraints. First of all, f should be easily invertible. Data in machine learning models often have many dimensions. For example, if we think about f as a linear transformation in high dimension space, the process of inverting is time and resource consuming. Secondly, the determinant of the Jacobian needs to be easy to compute, most preferably it should be expressed explicitly in a closed-form. Calculating or approximating derivatives is expensive in a numerical framework, thus we want to limit that as much as possible.

In [2], authors proposed real-valued non-volume preserving transformation (Real NVP). We can express function f as a combination of simple transformations f_i called *coupling layers*, i.e. $f = f_n \circ f_{n-1} \circ \cdots \circ f_1$. If we know inverse functions f_i^{-1}, we can find f^{-1}:

$$f^{-1} = (f_n \circ f_{n-1} \circ \cdots \circ f_1)^{-1} = f_1^{-1} \circ f_2^{-1} \circ \cdots \circ f_n^{-1}. \tag{2.42}$$

We can use another property of combination of functions to derive a Jacobian of f using the Jacobians of f_i:

$$\frac{\partial f}{\partial x} = \frac{\partial f_1}{\partial x} \cdot \frac{\partial f_2}{\partial f_1(x)} \cdot \cdots \cdot \frac{\partial f_n}{\partial f_{n-1} \circ \cdots \circ f_1(x)}. \tag{2.43}$$

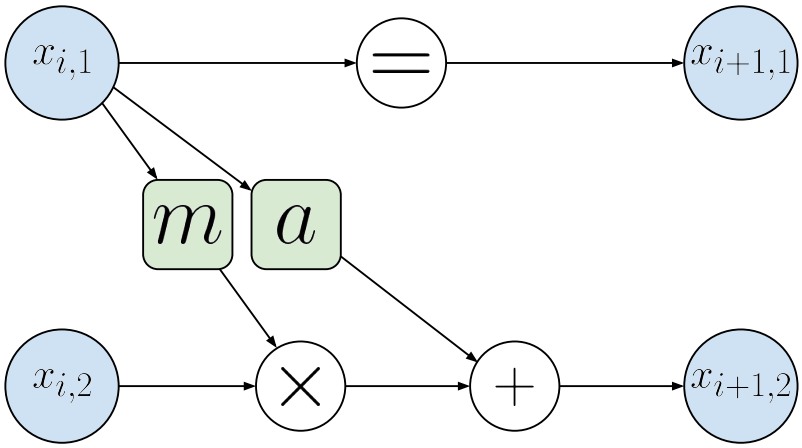

Figure 2.2: Coupling layer as a base block of Real NVP. Input data x_i is split into two parts: $x_{i,1}$ and $x_{i,2}$. The first one is passed forward without any change. The second one is transformed by functions of $x_{i,1}$: m and a. The output x_{i+1} is a concatenation of $x_{i+1,1}$ and $x_{i+1,2}$.

Let $x_i \in \mathbb{R}^D$ be an input of the f_i and $(x_{i,1}, x_{i,2})$ some partition of x_i. Most often $x_{i,1}$ is a vector containing half of the x_i's dimensions and $x_{i,2}$ the other half. Coupling layer f_i, presented in Figure 2.2, is a transformation defined as:

$$x_{i+1,1} = x_{i,1} \tag{2.44}$$

$$x_{i+1,2} = x_{i,2} \odot \exp[m_i(x_{i,1})] + a_i(x_{i,1}), \tag{2.45}$$

where $(x_{i+1,1}, x_{i+1,2})$ partition of x_{i+1} same as $(x_{i,1}, x_{i,2})$ with respect to x_i. $xi + 1$ is the output of the coupling layer f_i. m_i and a_i are some functions with trainable parameters. Of course, $x_1 = x$ and $x_{n+1} = z$ are input of the first layer and output of the last one, respectively.

Inverse transformation f_i^{-1} is easy to derive and given by:

$$x_{i,1} = x_{i+1,1} \tag{2.46}$$

$$x_{i,2} = (x_{i+1,2} - a_i(x_{i+1,1})) \odot \exp[-m_i(x_{i+1,1})]. \tag{2.47}$$

Let us calculate the determinant of the Jacobian of the f_i:

$$\det\left(\frac{\partial f_i}{\partial x}\right) = \det\begin{pmatrix} \mathbb{I} & 0 \\ \frac{\partial x_{i+1,2}}{\partial x_{i,1}} & \frac{\partial x_{i+1,2}}{\partial x_{i,2}} \end{pmatrix} = \det\begin{pmatrix} \mathbb{I} & 0 \\ \frac{\partial x_{i+1,2}}{\partial x_{i,1}} & \operatorname{diag}(\exp[m_i(x_{i,1})]) \end{pmatrix} \tag{2.48}$$

$$= \prod_{j=1}^{d} \exp[m_i(x_{i,1})_j] = \exp\Big[\sum_{j=1}^{d} m_i(x_{i,1})_j\Big], \tag{2.49}$$

where $\operatorname{diag}(\exp[m_i(x_{i,1})])$ is a diagonal matrix with vector $\exp[m_i(x_{i,1})]$ on its diagonal. d is the dimension of the $x_{i,2}$.

We can now derive log-likelihood of the data using: (2.37)

$$\log p(x) = \log p(f(x)) + \log \left| \det \left(\frac{\partial f}{\partial x} \right) \right| \tag{2.50}$$

$$= \log p(f(x)) + \log \left| \prod_{i=1}^{n} \exp \Big[\sum_{j=1}^{d} m_i (x_{i,1})_j \Big] \right| \tag{2.51}$$

$$= \log p(f(x)) + \log \left| \exp \Big[\sum_{i=1}^{n} \sum_{j=1}^{d} m_i (x_{i,1})_j \Big] \right| \tag{2.52}$$

$$= \log p(f(x)) + \sum_{i=1}^{n} \sum_{j=1}^{d} m_i (x_{i,1})_j. \tag{2.53}$$

Our optimization goal is to maximize $\log p(x)$.

Continuous Normalizing Flows

So far we have discussed normalizing flows that learn the mapping from complex data-driven distribution to a simple one as a stack of discrete transformations. We can approach this task with another group of flow-based models - continuous normalizing flows [3, 4].

Let us now define transformation f by time-dependent dynamics:

$$\frac{\partial z(t)}{\partial t} = f(z(t), t), \tag{2.54}$$

where f is a parametrized function, with initial condition $z(t_0) = z_0$. z_0 is a starting point sampled from prior distribution $p(z_0)$, i.e. Gaussian. $z(t_1) = x$ is an observable data point.

We can write continuous normalizing flow as:

$$x = z(t_0) + \int_{t_0}^{t_1} f(z(t), t) dt, \tag{2.55}$$

which describes transformation between samples from prior and data distributions. It can be inverted to obtain normalized data points:

$$z_0 = x + \int_{t_1}^{t_0} f(z(t), t) dt. \tag{2.56}$$

Let us assume that f is continuous in $[t_0, t_1]$ and uniformly Lipschitz continuous in z. Then, we can derive change of log-density as:

$$\frac{\partial \log p(z(t))}{\partial t} = - \operatorname{Tr} \Big(\frac{df}{dz(t)} \Big). \tag{2.57}$$

Proof can be found in [3]. Finally, total change of log-likelihood can be computed using integration over time:

$$\log p(x) = \log p(z(t_0)) - \int_{t_0}^{t_1} \operatorname{Tr} \Big(\frac{df}{dz(t)} \Big) dt. \tag{2.58}$$

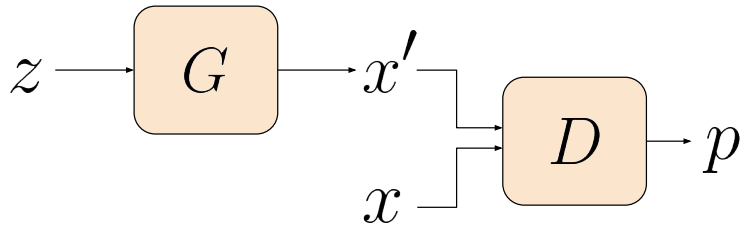

Figure 2.3: Structure of GAN. Generator G transforms noise sampled from prior distribution into realistically looking data x'. Discriminator tries to distinguish between real and fake objects. p is the probability that data is real.

2.1.4 Generative Adversarial Networks

Generative Adversarial Networks (GANs) [5] are yet another approach to data generation using latent variable models. When introduced, they were considered as heuristic due to no strong probability-based theory behind them. Currently, the theory is being investigated by researchers. They are still believed to be one of the biggest breakthroughs in deep learning, especially in generative modeling. GANs currently achieve state-of-the-art results in qualitative image generation and are used with other data types, e.g. videos, audio, or point clouds.

Let $x \sim p(x)$ be data sampled from unknown distribution $p(x)$ and let $z \sim p(z)$ be a latent variable (also called *noise*) with known prior distribution $p(z)$, e.g. Gaussian. GANs consist of two parametrized functions:

- **Generator (G)** responsible for generating realistically looking data samples $x' = G(z)$ from noise z,
- **Discriminator (D)** that has to distinguish between real data x and samples generated by the Generator $G(z)$ - it calculates the probability that given object is real.

Generator and Discriminator are competing against each other in a minimax game described by the function $V(G, D)$:

$$\min_G \max_D V(G, D) = \mathbb{E}_{x \sim p(x)}[\log D(x)] + \mathbb{E}_{z \sim p(z)}[\log(1 - D(G(z)))]. \tag{2.59}$$

In practice, direct optimization of $V(G, D)$ is not trivial. Instead, we can create two separate objectives for G and D:

$$D^* = \operatorname*{argmax}_D V(G, D) \tag{2.60}$$

$$G^* = \operatorname*{argmax}_G \mathbb{E}_{z \sim p(z)}[\log D(G(z))]. \tag{2.61}$$

In this approach, we update D and G alternately. Hence, treating D as constant obtained in the step 2.60, we can approximate minimization of $V(G, D)$ with maximization of $\mathbb{E}_{z\sim p(z)}[\log D(G(z))]$. More precisely:

$$\begin{aligned}\underset{G}{\operatorname{argmin}} V(G, D) &= \underset{G}{\operatorname{argmin}} \mathbb{E}_{x\sim p(x)}[\log D(x)] + \mathbb{E}_{z\sim p(z)}[\log(1 - D(G(z)))] &(2.62)\\ &= \underset{G}{\operatorname{argmin}} \mathbb{E}_{z\sim p(z)}[\log(1 - D(G(z)))] &(2.63)\\ &\approx \underset{G}{\operatorname{argmax}} \mathbb{E}_{z\sim p(z)}[\log D(G(z))]. &(2.64)\end{aligned}$$

Let us now discuss an intuition behind the optimization steps. $V(G, D)$ is maximized by the Discriminator when it always marks samples generated by the Generator as 0 and real samples as 1. It is the best scenario for the Discriminator and it cannot be fooled by the Generator then. On the other hand, Generator that maximizes $\mathbb{E}_{z\sim p(z)}[\log D(G(z))]$ is a perfect one, meaning that it always generates such samples that the Discriminator is not able to recognize the real ones. Optimal situation for the whole model is when the output of the Discriminator is always $\frac{1}{2}$, i.e. $D(x) = D(G(z)) = \frac{1}{2}$ for every data sample x and random noise z.

The training process of GANs is very unstable and it requires heavy hyperparameters tuning. The initial evaluation is possible only by visual investigation of generated samples. In addition, vanilla GANs do not have encoding capabilities, thus they cannot learn any useful representation of the data. Later introduced extensions, such as BiGAN [6], solve this problem by extending the architecture by additional modules, e.g. encoder that maps data back into the latent space.

2.2 Neural Networks

Neural networks (Figure 2.4) are algorithms inspired by animal brain construction. Biological neurons are connected in an enormous network, that transfers signal by neural activation. Similarly, neural networks consist of neurons grouped in layers, which process signal iteratively through the whole architecture using linear operations and activation functions. Neural networks are trained to perform a specific task. Similarly to the learning process of a child, they try to improve based on the good and bad decisions they made in the past.

The development of neural networks and the rapid growth of the hardware industry in the 21st century started the era of deep learning. These methods found their place in both scientific and commercial worlds, outperforming classic machine learning models or even solving problems that could not be overcome back then.

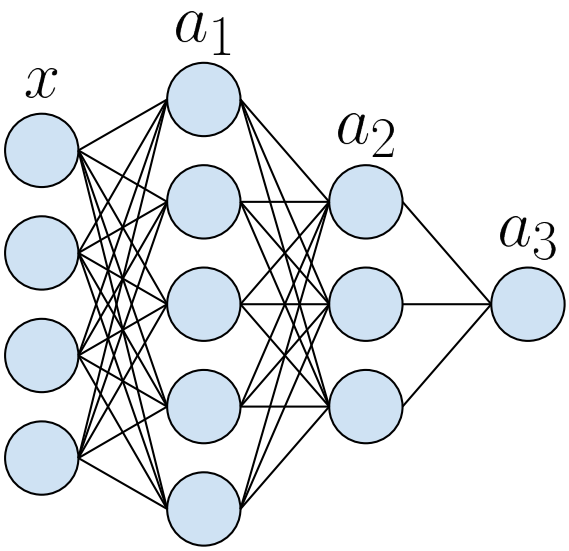

Figure 2.4: Simple neural network with 3 layers. Visualization made for one data point x.

2.2.1 Overview

We can divide learning process of neural network into two parts: *forward propagation* and *backward propagation*. First one is responsible for processing the signal from input and calculating value of a *loss function* which measures how well the model perform on a given task. Second one makes adjustment on the neural network's parameters based on the entire step of forward pass in order to improve performance in next iterations.

Forward propagation

Let $X \in \mathbb{R}^{n \times d_0}$ be training data. More precisely, X is a matrix containing n d_0-dimensional data points. Linear neural network, also called Multilayer Perceptron (MLP), has forward propagation defined as:

$$Z_1 = XW_1 + b_1 \tag{2.65}$$
$$A_1 = f_1(Z_1) \tag{2.66}$$
$$\vdots \tag{2.67}$$
$$Z_l = A_{l-1}W_l + b_l \tag{2.68}$$
$$A_l = f_l(Z_l), \tag{2.69}$$

where f_i are nonlinear activation functions, such as sigmoid $\sigma(x) = \frac{1}{1+e^{-x}}$, rectified linear unit $\mathrm{ReLU}(x) = \max(0, x)$ or hyperbolic tangent. $W_i \in \mathbb{R}^{d_{i-1} \times d_i}$ are weight

matrices, where d_i is a number of neurons in i-th layer, and $b_i \in \mathbb{R}^{d_i}$ bias vectors. Weights and biases are called *parameters* of the network. Number of layers, neurons and other values that are constant during the training process are called *hyperparameters*.

To measure performance of the model on a given task, we calculate value of a loss function $\mathcal{L}(\mathbf{W}, \mathbf{b}; X)$, where $\mathbf{W} = \{W_i : i = 1, \ldots, l\}$ and $\mathbf{b} = \{b_i : i = 1, \ldots, l\}$ are sets of weights and biases, respectively. Note that this form of loss function is valid only for unsupervised tasks. We do not provide any true labels which the model should learn from.

Backward propagation

To optimize the loss function, the model needs to adjust its parameters. It makes use of the influence of each parameter on the final loss value. In the backward step, gradients of the loss function with respect to every parameter are calculated. To calculate the derivative $\frac{\partial \mathcal{L}}{\partial W_k}$, we make use of the chain rule:

$$\frac{\partial \mathcal{L}}{\partial W_k} = \frac{\partial \mathcal{L}}{\partial A_l} \left(\prod_{i=0}^{l-k-1} \frac{\partial A_{l-i}}{\partial Z_{l-i}} \frac{\partial Z_{l-i}}{\partial A_{l-i-1}} \right) \frac{\partial A_k}{\partial Z_k} \frac{\partial Z_k}{\partial W_k}. \tag{2.70}$$

Analogically, we can compute gradients with respect to the bias vector b_k:

$$\frac{\partial \mathcal{L}}{\partial b_k} = \frac{\partial \mathcal{L}}{\partial A_l} \left(\prod_{i=0}^{l-k-1} \frac{\partial A_{l-i}}{\partial Z_{l-i}} \frac{\partial Z_{l-i}}{\partial A_{l-i-1}} \right) \frac{\partial A_k}{\partial Z_k}. \tag{2.71}$$

Note that there is no need to calculate $\frac{\partial Z_k}{\partial b_k}$, because it is a vector containing only ones.

Having gradients calculated, we can use them to update parameters so the value of the loss function is closer to the global optimum. The most common optimization algorithm used to make it happen is Gradient Descent (GD). In a single iteration it updates parameters as follows:

$$W_k = W_k - \alpha \frac{\partial \mathcal{L}}{\partial W_k} \tag{2.72}$$

$$b_k = b_k - \alpha \frac{\partial \mathcal{L}}{\partial b_k}, \tag{2.73}$$

where α is a *learning rate*, hyperparameter that specifies how big the backward step should be.

Gradient descent process all data points in every iteration also called *epoch*, and very often it is impossible to use it with models trained on big datasets. To overcome this issue, Stochastic Gradient Descent (SGD) was introduced. In every epoch, we randomly shuffle the data and split them into chunks, called *batches*. Then, forward and backward steps are performed only on one batch at the time,

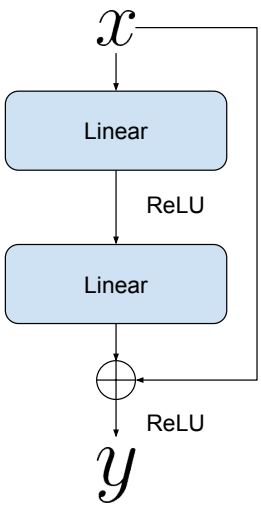

Figure 2.5: One ResNet block. Point x is transformed by linear operation followed by nonlinearity and another linear transformation. The output is added to the initial value of x and passed through final nonlinear activation.

and weights are updated accordingly by 2.72 and 2.73. In each epoch, we process all of the batches sequentially. One step of SGD performed on a small part of the data is an approximation of one GD step.

2.2.2 Residual Networks

First layers in deep neural networks often suffer from vanishing gradients during backpropagation. Because of the chain rule, accumulated gradients can be so small that the change of parameters in those layers might be insignificant. Residual Networks [7] are an architecture design that solves this problem.

ResNets introduced modification to the standard linear layers called *skip connections*. Let X be an input to a layer. Skip connection is defined as:

$$F_0 = f_0(XW_0 + b_0) \tag{2.74}$$
$$F_1 = F_0W_1 + b_1 \tag{2.75}$$
$$Y = f_1(F_1 + X), \tag{2.76}$$

where f_i are non-negative activations, e.g. ReLU defined as $\mathrm{ReLU}(x) = \max(0, x)$, W_i and b_i weights and Y output of the skip connection layer. ResNet block is shown in Figure 2.5.

In the (2.76) we add X to the output of the previous layer. Doing so we do not hurt the performance of the model. In the worst case, when weights W_1 and b_1 are zero, the last layer is an identity function. Thus no information is gained but neither is lost. During backpropagation, instead of multiplying gradients from further layers by some small values, we multiply them by 1. It preserves signal and lets weights

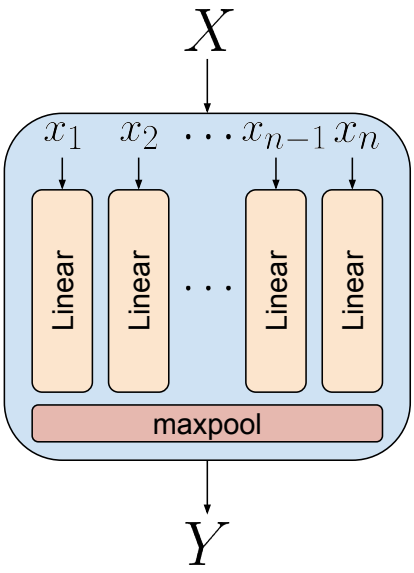

Figure 2.6: Core architecture of PointNet. Points x_i from point cloud X are transformed using simple MLP layers. The output Y is given by the max pool.

from the first layers to receive meaningful updates. In addition, ResNet strengthens the connection between the input and the output, preventing the model from losing information during the forward step.

2.2.3 PointNet

In point clouds, each object is represented by an unordered set of 3D points. This type of data structure has two main difficulties that need to be taken into account when modeling. First of all, point clouds are permutation invariant, meaning that permuting any subset of the point cloud does not change the object it represents. In this case, standard machine learning techniques or neural networks do not work well. Models adjusted to point clouds should also be comfortable with an arbitrary number of points given. Let us say that we have an object represented by 10000 points. Now, if we randomly pick 5000 or even 1000 point, the model should still be able to recognize the object.

The most popular architecture that deals well with point clouds was introduced in PointNet [8]. Let X be $n \times 3$ matrix (point cloud) and x_i i-th row of the matrix X (single point). Every x_i is treated as an independent input to MLP layers. After k transformations, global features are being extracted by max pooling, i.e.:

$$Z_1 = f_1(XW_1 + b_1) \tag{2.77}$$

$$\vdots \tag{2.78}$$

$$Z_k = f_k(Z_{k-1}W_k + b_k) \tag{2.79}$$

$$Y = \max\text{pool}(Z_k), \tag{2.80}$$

where $\max\text{pool}(Z_k)$ is a vector containing maximum of n elements in each column of Z_k. Above transformation is visualized in the Figure 2.6. Note that it is not the entire model introduced in [8], but the most significant part of it.

This construction allows us to overcome both problems with point cloud modeling. PointNet treats each point separately, making no assumptions about either ordering or cardinality. More precisely, max-pooling calculates the maximum value in each dimension, extracting global features from the whole set of points. The maximum function is invariant on arguments permutation, thus representing point clouds as unordered sets is valid. On the other hand, adding additional points from the object is also possible, because the number of points required is not fixed. Of course, adding new entries may change the extracted features.

Chapter 3

Generative models for point clouds

Recent developments in the area of autonomous vehicles increased the interest in point cloud modeling. Generative models are dominated by algorithms created for images, and most often their application to other modalities requires significant changes in data handling and architecture.

In this chapter, we will focus on metrics that evaluate the performance of generative models for point clouds and describe two current state-of-the-art methods.

3.1 Evaluation

Good models should be able to generate new point clouds that are similar to those they were trained on. In addition, we would like them not to collapse into several perfectly generated point objects. To evaluate the generative capabilities of a trained model, we need to choose evaluation metrics that are able to measure similarities between distributions defined by generated and reference samples. Ideally, they should be applicable to different types of models, e.g. flows, VAEs, and GANs, to allow comparison between them. Comparing models based on the likelihood is in general not a universal technique because some of them do not allow to explicitly calculate or even approximate the likelihood. Metrics are ought to take into consideration not only the qualitative resemblances but also the variety of samples.

3.1.1 Similarity measures

Since point clouds are represented as unordered sets of points, we cannot use standard Euclidean distance to measure how similar two objects are. Instead, there are two widely used distances to measure similarities between point clouds: earth mover's distance and Chamfer distance.

Let $x = \{x_1, x_2, \ldots, x_N\}$ and $y = \{y_1, y_2, \ldots, y_N\}$ be two point clouds, each containing N points. We can define previously mentioned distances as:

- **Earth mover's distance (EMD)**

$$\mathrm{EMD}(x, y) = \min_{\phi: x \to y} \sum_{i=1}^{N} ||x_i - \phi(x_i)||_2, \tag{3.1}$$

  where ϕ is a bijection mapping cloud x into cloud y,

- **Chamfer distance (CD)**

$$\mathrm{CD}(x, y) = \sum_{i=1}^{N} \min_{y_j \in y} ||x_i - y_j||_2^2 + \sum_{i=1}^{N} \min_{x_j \in x} ||y_i - x_j||_2^2. \tag{3.2}$$

EMD is considered to be the proper way of calculating the true distance between point clouds. However, it requires optimization algorithms because of the big number of possible bijective functions. This is the reason why CD was introduced and it is thought of as an accurate approximation of EMD.

3.1.2 Population metrics

The authors of current baseline models [9, 10] use 4 metrics to measure performance of their methods. Let G and R be sets of generated and reference samples, respectively, with the same number of objects, i.e. $|G| = |R|$. We can define:

- **Jensen-Shannon Divergence (JSD)**

$$\mathrm{JSD}(P_G||P_R) = \frac{1}{2} D_{KL}(P_G||M) + \frac{1}{2} D_{KL}(P_R||M), \tag{3.3}$$

  where $M = \frac{1}{2}(P_G + P_R)$. P_G and P_R are empirical densities of points discretized into 28^3 voxels. Note that JSD considers only point distributions of entire sets G and R, and not of each of the shapes individually.

- **Minimum matching distance (MMD)**

$$\mathrm{MMD}(G, R) = \frac{1}{|R|} \sum_{r \in R} \min_{g \in G} d(g, r), \tag{3.4}$$

  where d can be either EMD or CD. MMD is responsible for measuring quality of generated samples. It is the average of distances between every reference object and generated one that is the closest.

- **Coverage (COV)**

$$\mathrm{COV}(G, R) = \frac{1}{|R|} |\{\operatorname*{argmin}_{r \in R} d(g, r) : g \in G\}| \tag{3.5}$$

measures how diversified generated shapes are. It calculates how many unique reference objects are the least distant to generated ones. COV does not evaluate quality of samples. Therefore, together with MMD, they are currently base metrics for point cloud generation.

- **1-nearest neighbor accuracy (1-NNA)**

$$\text{1-NNA}(G, R) = \frac{1}{|G| + |R|}\Big(\sum_{g \in G} \mathbb{1}_G(N_g) + \sum_{r \in R} \mathbb{1}_R(N_r)\Big), \tag{3.6}$$

where N_x is a nearest neighbor of point cloud x in joint set $G \cup R$. 1-NNA measures the similarity between reference and generated distributions. More precisely, it calculates how many reference samples have its nearest neighbor in reference set R, and analogically how many generated samples are classified as elements of generated set G based on the label of the closest shapes. For large number of objects, 1-NNA of two sets drawn from the same distribution should converge to 50%. Values that are closer to 50% imply bigger similarity between G and R.

Each of the described metrics is not perfect on its own and has some major drawbacks. However, all of them combined, form a sufficient benchmark for generative capabilities evaluation.

3.2 Baseline models

The current state-of-the-art methods that perform best in the aforementioned metrics are l-GAN [9] and PointFlow [10]. They were created based on different approaches to generative modeling: l-GAN is adapted from standard image-oriented GAN, and PointFlow combines VAE with continuous normalizing flows. Both of them make use of PointNet [8] architecture in their modules.

3.2.1 l-GAN

Latent-space GAN (l-GAN) [9] operates on latent code learned by pre-trained autoencoder, instead of raw point clouds. Aoutoencoder is a simplified version of VAE. It consists of two parts: encoder f and decoder g. Lower dimensional latent representation of point cloud x is computed as $z = f(x)$. f follows PointNet [8] architecture. Then, z is passed to decoder g to reconstruct initial object $\hat{x} = g(z)$. The model is trained to minimize expected value of distance between x's and $\hat{x}$'s:

$$f^*, g^* = \underset{f,g}{\operatorname{argmin}}\, \mathbb{E}[d(x, \hat{x})], \tag{3.7}$$

where in case of point clouds d is either EMD or CD.

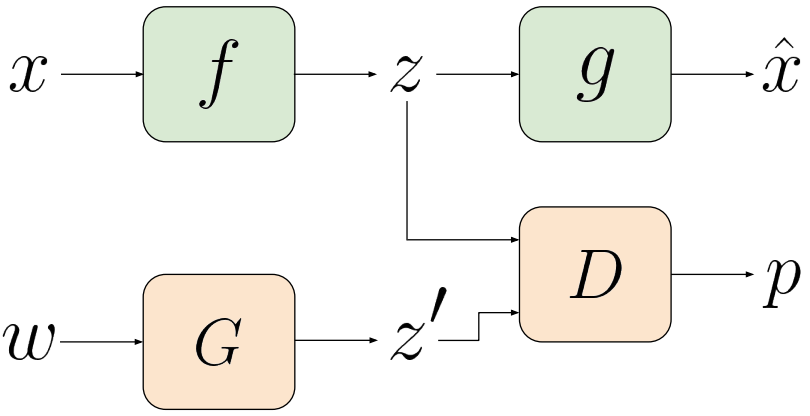

Figure 3.1: l-GAN generates latent vectors z''s that are ought to come from the same distribution as z's (orange modules), that are latent codes learned by pre-trained autoencoder (green modules).

l-GAN learns to generate samples from distribution of latent representations discovered by autoencoder. More precisely, given initial values w sampled from known prior distribution, e.g. Gaussian, generator G transforms it into fake latent samples z'. Discriminator D is responsible for distinguishing between generated codes and true ones. Generative process can be written as:

$$w \sim \mathcal{N}(0, I) \tag{3.8}$$

$$z' = G(w). \tag{3.9}$$

The training objective it to find such G and D that satisfy:

$$\min_G \max_D V(G, D) = \mathbb{E}_{z \sim p(z)}[\log D(z)] + \mathbb{E}_{z \sim p(z)}[\log(1 - D(z'))]. \tag{3.10}$$

Achieving better performance on standard benchmarks than GAN trained on raw point clouds, l-GAN has still some disadvantages. Autoencoders do not generalize to data distribution as well as VAEs, because of the deterministic nature of their latent coding. Hence, l-GAN has limited generative capabilities. Another drawback of this model is the fact that we can only generate point clouds with a fixed number of points. The decoder is implemented as a simple neural network and does not allow any changes in the output dimension. In addition, the training process requires using EMD, which adds additional difficulties with optimization and slowing the whole procedure, or CD, which is stated as a heuristic measure of the distance between point clouds. Finally, because of the GAN framework, we cannot measure or approximate the likelihood of the date, having no numerical data about the learned distribution.

3.2.2 PointFlow

PointFlow [10] is the first model utilizing normalizing flows to model point clouds. More precisely, it uses two modules of continuous normalizing flows. The key idea is to think about point clouds as samples from some shape distribution. Then, each of them represents point-level distribution as well. This approach allows us to model each data point separately, creating the possibility of generating any number of points from sampled shape. In contrary to previous methods, the training dataset may contain shapes of unrestricted cardinality.

Let us recall ELBO for Variational Autoencoder. Let $p_\theta(x|z)$ be decoder and $q_\phi(z|x)$ encoder. We can write the lower bound of the data likelihood as:

$$\log p_\theta(x) \geq \mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x|z)] - D_{KL}(q_\phi(z|x)||p_\psi(z)). \tag{3.11}$$

We can associate $p_\psi(z)$ with distribution of shapes, and $p_\theta(x|z)$ with distribution of points over the surface of the shape represented by given z.

Let us assume that all of the points in each of the shapes are conditionally independent. Reconstruction likelihood can be written as:

$$\log p_\theta(x|z) = \sum_{x_i \in x} \log p_\theta(x_i|z). \tag{3.12}$$

To model $\log p_\theta(x_i|z)$, continuous normalizing flow is used. Let us create mapping between data point x_i and some point $y_i(t_0)$ sampled from the prior distribution $p(y_i) = \mathcal{N}(0, I)$:

$$x_i = G_\theta(y_i(t_0); z) = y_i(t_0) + \int_{t_0}^{t_1} g_\theta(y_i(t), t, z)dt. \tag{3.13}$$

Inverse function is easily computed:

$$y_i(t_0) = G_\theta^{-1}(x_i; z) = x_i + \int_{t_1}^{t_0} g_\theta(y_i(t), t, z)dt. \tag{3.14}$$

Finally, we can write continuous version of change of variable formula:

$$\log p_\theta(x_i|z) = \log p(G_\theta^{-1}(x_i; z)) - \int_{t_0}^{t_1} \mathrm{Tr}\left(\frac{\partial g_\theta}{\partial y_i(t)}\right)dt. \tag{3.15}$$

To enable more flexible training over shapes, we can now rewrite the KL divergence term as:

$$-D_{KL}(q_\phi(z|x)||p_\psi(z)) = \mathbb{E}_{q_\phi(z|x)}[\log p_\psi(z)] + H[q_\phi(z|x)], \tag{3.16}$$

where H is the differential entropy defined as $H(x) = -\mathbb{E}[\log p(x)]$, where $p(x)$ is the density function of x. Similarly to the likelihood of each point, we make use of another CNF to map the prior over shapes into a Gaussian distribution $p(w) = \mathcal{N}(0, I)$:

$$z = F_\psi(w(t_0)) = w(t_0) + \int_{t_0}^{t_1} f_\psi(w(t), t)dt. \tag{3.17}$$

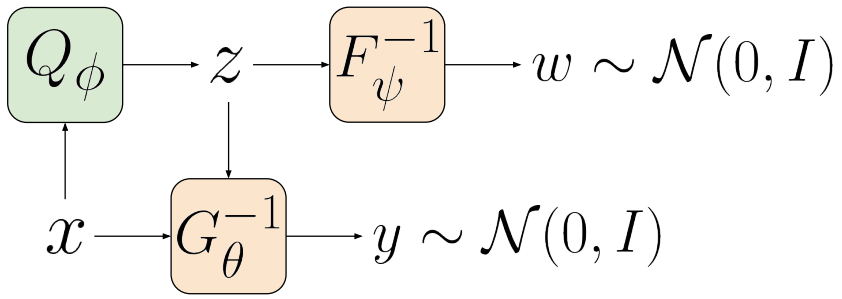

Figure 3.2: Inference procedure of PointFlow. Point cloud x is encoded into shape representation z by Q_ϕ, which is then further normalized by CNF F_ψ^{-1}. Input object x conditioned on its shape vector z, is normalized by another CNF G_θ^{-1}. During sampling, encoder is discarded. To sample new point cloud, we first sample $w \sim \mathcal{N}(0, I)$ and desired number of points $y \sim \mathcal{N}(0, I)$. Then, w is transformed into shape representation z using F_ψ. New object is given by $G_\theta(y; z)$.

Inverse transformation is given by:

$$w(t_0) = F_\psi^{-1}(z) = z + \int_{t_1}^{t_0} f_\psi(w(t), t)dt. \tag{3.18}$$

At last, log likelihood of the prior shape distribution is derived as:

$$\log p_\psi(z) = \log p(F_\psi^{-1}(z)) - \int_{t_0}^{t_1} \mathrm{Tr}\left(\frac{\partial f_\psi}{\partial w(t)}\right)dt. \tag{3.19}$$

This approach allows to learn more complex latent shape representation preserving generative capabilities.

The final training objective is defined as:

$$\begin{aligned}
\mathcal{L}(x; \phi, \psi, \theta) &= \mathbb{E}_{q_\phi(z|x)}[\log p_\psi(z) + \log p_\theta(x|z)] + H[q_\phi(z|x)] && (3.20)\\
&= \mathbb{E}_{q_\phi(z|x)}[\log p(F_\psi^{-1}(z)) - \int_{t_0}^{t_1} \mathrm{Tr}\left(\frac{\partial f_\psi}{\partial w(t)}\right)dt && (3.21)\\
&+ \sum_{x_i \in x}(\log p(G_\theta^{-1}(x_i; z)) - \int_{t_0}^{t_1} \mathrm{Tr}\left(\frac{\partial g_\theta}{\partial y_i(t)}\right)dt)] && (3.22)\\
&+ H[q_\phi(z|x)]. && (3.23)
\end{aligned}$$

In practice, $q_\phi(z|x)$ is an encoder following PointNet [8] architecture described in section 2.2.3. The inference procedure is presented in Figure 3.2. PointFlow is the current state-of-the-art in most of the metrics described in 3.1. The results of the experiments will be discussed in chapter 5. It is relatively light-weighted in terms of the number of trainable parameters but because CNFs are used, the training process takes a long time in comparison to standard normalizing flows, for instance.

Chapter 4

Conditional Invertible Flow

4.1 Model

We present Conditional Invertible Flow (CIF) as a new approach to point cloud modeling. The architecture of our model is presented in Figure 4.1. We treat individual point clouds as probability distributions in $\mathbb{R}^3$. The normalizing flow network $\mathbf{f}$ implements an invertible random variable transformation between the standard normal prior distribution $\mathcal{N}(0, I)$ and the distribution defined by points in the cloud. The flow $\mathbf{f}$ is conditioned on latent vectors e that are also normally distributed. We compute the conditioning vectors using a PointNet encoder $\mathbf{h}$ whose output, w is transformed to the standard normal prior using another normalizing flow $\mathbf{g}$. The whole model is trained jointly by direct optimization of a single loss function.

To generate new object-specific points clouds, we first sample a cloud embedding e from the standard normal prior. Then, we use the sampled embedding e to condition the model $\mathbf{f}$. Finally, we generate individual points in the cloud by sampling from the standard normal and transforming them into the data space using $\mathbf{f}^{-1}$.

Conditional Flow-based Cloud Generator

To define our training objective, we sample two subsets of points x^f, x^h from each cloud in a training batch and train the model by maximizing the likelihood of the points x^f conditioned on a cloud embedding computed on x^h. First, we put the points in x^h through a trainable PointNet encoder $\mathbf{h}$ to obtain a cloud representation w which we then map to the conditioning e using another normalizing flow network $\mathbf{g}$:

$$\begin{aligned} w &= \mathbf{h}(x^h) \\ e &= \mathbf{g}(w) \end{aligned} \tag{4.1}$$

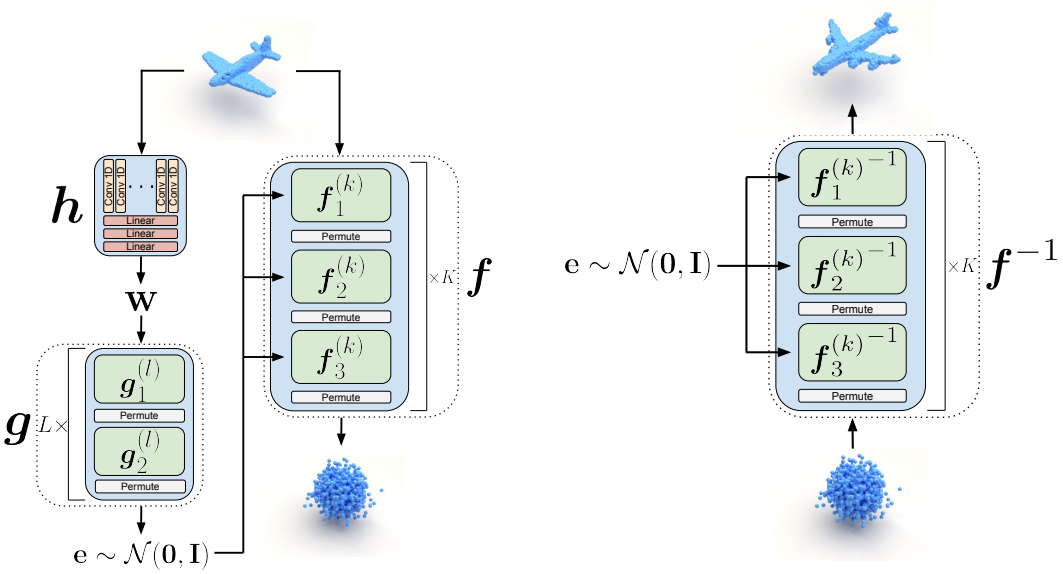

Figure 4.1: Architecture of our Conditional Invertible Flow Networks used for inference (left) and sampling (right).

Finally, we use points in x^f to compute the conditional cloud likelihood:

$$\log p(x^f|x^h) = \sum_{i=1}^{N} \log p(\mathbf{f}(x_i^f, e)) + \log \left| \det \frac{\partial \mathbf{f}(x_i^f, e)}{\partial x_i^f} \right| \tag{4.2}$$

We extend our loss function with a regularization term which promotes a normal distribution of the conditioning vectors e:

$$\log p(w|x^h) = \log p(\mathbf{g}(w)|x) + \log \left| \det \frac{\partial \mathbf{g}(e)}{\partial e} \right| \tag{4.3}$$

Our final loss function is defined to be:

$$\mathcal{L} = -\log p(x^f|x^h) - \log p(w|x^h). \tag{4.4}$$

In comparison to PointFlow [10] our model achieves good results much faster. However, CIF has about 20 times more trainable parameters. Although our model is simple, it still performs well in the benchmarks.

4.2 Architecture

We implemented networks $\mathbf{f}$ and $\mathbf{g}$ as discrete normalizing flows. Network $\mathbf{f}$ is a combination of 10 segments $\mathbf{f}^{(k)}$ for $k = 1, ..., 10$ with 3 blocks $\mathbf{f}_i^{(k)}$ for $i = 1, 2, 3$ each. Every block is defined by two ResNets. Analogically $\mathbf{g}$ has 5 segments with 2 blocks each. The encoder $\mathbf{h}$ follows architecture designs of PointNet [8]. It consists of 1D convolutional layers followed by features extraction and fully connected layers.

We chose dimensions of w and e to equal 32. Priors $p(z)$ and $p(e)$ are d-dimensional standard normal distributions, where d equals to 3 and 32, respectively. We used Adam optimizer with default parameters and learning rate 10^{-4} decaying every 10 epochs by factor 0.8.

The model was trained on 2 RTX 2080 GPUs with 12 GB memory each. Results reported in the next chapter were achieved after about 4 days of training.

Chapter 5

Results

The goal of the experiments is to provide quantitative and qualitative analysis of reconstructive and generative capabilities of the model in terms of the quality of reconstructed objects, generated samples, interpolation, and metrics including the coverage and minimum matching distance. For all experiments, we used point clouds from the ShapeNet dataset preserving the validation split from [10].

5.1 Quantitative experiments

Following the methodology proposed in [10] we evaluate the generative capabilities of the model with the criteria: Jensen-Shannon Divergence (JSD), Coverage (COV), Minimum Matching Distance (MMD), and 1-Nearest Neighbor Accuracy (1-NNA). For the last three of the mentioned measures, we consider using Chamfer (COV-CD, MMD-CD, 1-NNA-CD) and Earth-Mover's (COV-EMD, MMD-EMD, 1-NNA-EMD) distances.

We compare the results with the existing solutions including: raw-GAN [9], latent-GAN [9], PC-GAN [11] and PointFlow [10]. We train each model using point clouds from one of the three categories in the ShapeNet dataset: *airplane*, *chair*, and *car*. We sample 2048 points for each of the point cloud used for evaluation. We also report the performance of evaluation metrics for point clouds from the training set, which is considered as an upper bound since they are from the target distribution. The results of the quantitative analysis are presented in Table 5.1. The CIF model achieves competitive results compared to the reference approaches. Moreover, our approach does not require extensive training procedure, essential during the application of ODE solver for continuous flows utilized by PointFlow.

Table 5.1: Generation results. MMD-CD scores are multiplied by 10^3; MMD-EMD scores are multiplied by 10^2; JSDs are multiplied by 10^2.

Cathegory	Methods	JSD	MMD		COV		1-NNA	
			CD	EMD	CD	EMD	CD	EMD
Airplane	r-GAN	7.44	0.261	5.47	42.72	18.02	93.58	99.51
	l-GAN (CD)	4.62	0.239	4.27	43.21	21.23	86.30	97.28
	l-GAN (EMD)	**3.61**	0.269	3.29	**47.90**	50.62	87.65	85.68
	PC-GAN	4.63	0.287	3.57	36.46	40.94	94.35	92.32
	PointFlow	4.92	**0.217**	3.24	46.91	48.40	**75.68**	75.06
	CIF (ours)	4.24	0.221	**3.14**	47.57	**52.67**	77.08	**72.59**
	Training set	6.61	0.226	3.08	42.72	49.14	70.62	67.53
Chair	r-GAN	11.5	2.57	12.8	33.99	9.97	71.75	99.47
	l-GAN (CD)	4.59	2.46	8.91	41.39	25.68	64.43	85.27
	l-GAN (EMD)	2.27	2.61	7.85	40.79	41.69	64.73	65.56
	PC-GAN	3.90	2.75	8.20	36.50	38.98	76.03	78.37
	PointFlow	1.74	2.42	7.87	**46.83**	46.98	**60.88**	**59.89**
	CIF	**1.42**	**2.38**	**7.85**	44.01	**47.03**	62.71	63.39
	Training set	1.50	1.92	7.38	57.25	55.44	59.67	58.46
Car	r-GAN	12.8	1.27	8.74	15.06	9.38	97.87	99.86
	l-GAN (CD)	4.43	1.55	6.25	38.64	18.47	63.07	88.07
	l-GAN (EMD)	2.21	1.48	5.43	39.20	39.77	69.74	68.32
	PC-GAN	5.85	1.12	5.83	23.56	30.29	92.19	90.87
	PointFlow	0.87	0.91	5.22	44.03	46.59	**60.65**	62.36
	CIF	**0.79**	**0.90**	**5.12**	**44.79**	**49.24**	64.82	**61.36**
	Train set	0.86	1.03	5.33	48.30	51.42	57.39	53.27

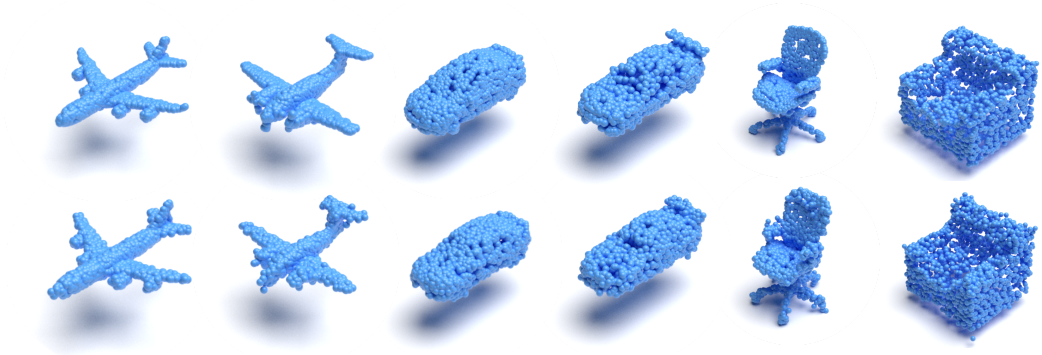

Figure 5.1: **Reconstructions** of our model obtained with an autoencoding architecture presented in Figure 5.3. The first row shows inputs to the model, and the second - their reconstructions.

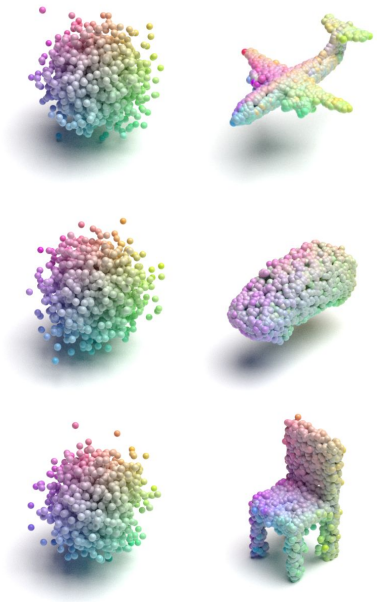

Figure 5.2: Point clouds sampled from $\mathcal{N}(0, I)$ (left) and reconstructed using our model (right). Note, how the model maps points to the closest semantic parts of objects.

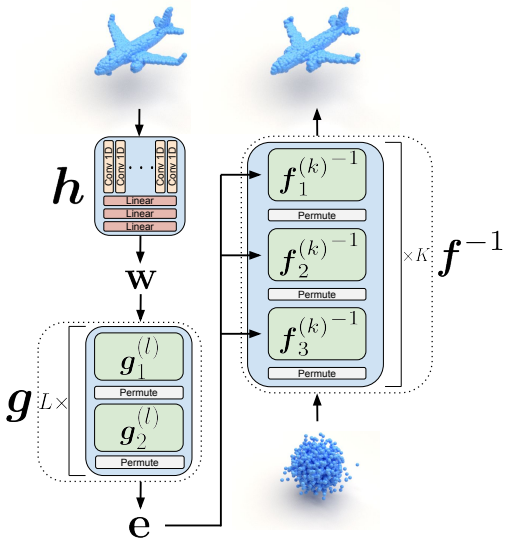

Figure 5.3: Model used during for the reconstruction task. Since we leverage invertible normalizing flows, the $\mathbf{f}$ modules can serve as a decoder of an autoencoder architecture.

5.2 Qualitative analysis

Reconstruction As the first experiment, we test the reconstruction capabilities of our model according to the procedure outlined in Figure 5.3.

First, we pass an input sample x through the encoding network $\mathbf{h}$ to obtain the feature vector w, which is further used as an input to the flow $\mathbf{g}$. As a result, we obtain an e which is embedding representing given cloud x. In order to reconstruct the object, we pass the points sampled from the $p(z) \sim \mathcal{N}(0, I)$ and the conditioning latent code e as the input to our inverted flow $\mathbf{f}^{-1}$ to produce the reconstructed point cloud.

The results are presented in the Figure 5.1. They show the ability of our model to correctly encode unseen objects into the latent space and then reconstruct them using a desired number of points.

Sampling To check our model's ability to generalize to the whole data space, we perform a sampling experiment. To sample a new point cloud, we first sample an embedding e from our prior distribution $p(e)$. With that, we obtain the conditioning term for a network $\mathbf{f}$. We then sample a chosen number of points z from a distribution $p(z)$. The embedding e and points z are then used as an input to $\mathbf{f}^{-1}$ in order to

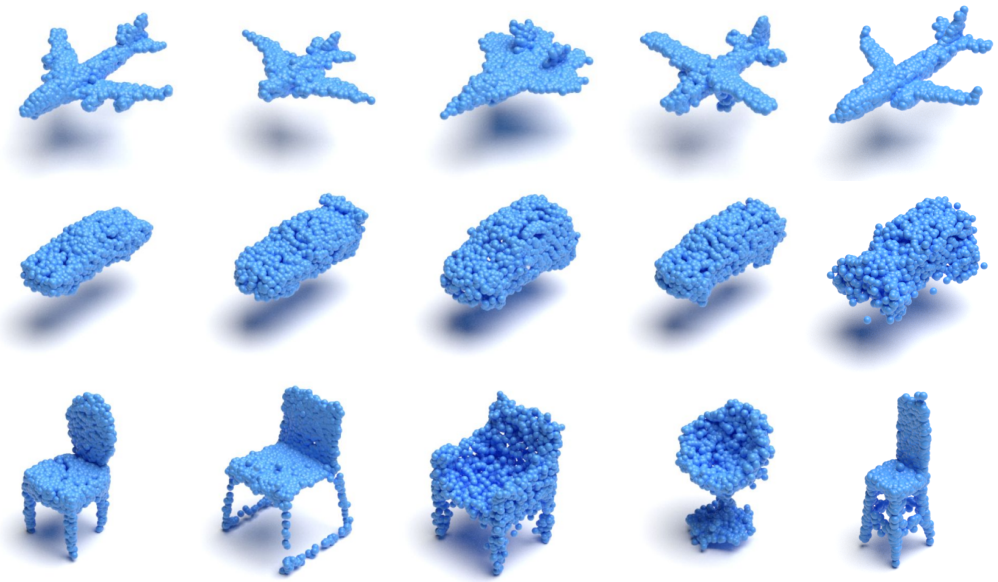

Figure 5.4: **Samples** generated with **f** normalizing flow. Rows show samples from different categories: airplanes, cars and chairs.

generate various shapes of point clouds. Figure 5.4 presents the qualitative results of the sampling experiment.

Following generation procedure in [12], we found that for cars and chairs, sampling e from the prior distribution with increased standard deviation results in better samples both in terms of quality and metrics performance. We used a standard deviation equal to 1.25 and 1.3 for cars and chairs, respectively.

Interpolation As a way to assess the embedding space E continuity, we perform the linear interpolation between the true point cloud shapes. First, we take two samples and obtain their latent space embeddings. Next, we calculate the intermediate latent codes between them. We then use those new latent space codes as an input to the $\mathbf{f}^{-1}$ to obtain interpolated point clouds.

Results presented in Figure 5.5 show that embeddings are arranged naturally and logically. Transitions between each sample are smooth, and there are no substantial visual differences.

Object alignment 3D point transformations, such as rotations or scalings can be modeled as specialized normalizing flow layers and added to the flow network **f**. In Figure 5.6 we show the possibility of recovering object pose. We extend the flow **f** with multiplication by a parametric rotation matrix. We then randomly rotate a cloud and optimize its conditional likelihood $p(x|e = 0)$ with respect to the rotation matrix. During the procedure, we use the null embedding vector $e = 0$) to deprive the model of cloud shape information. For optimization, we have used the CMA-ES

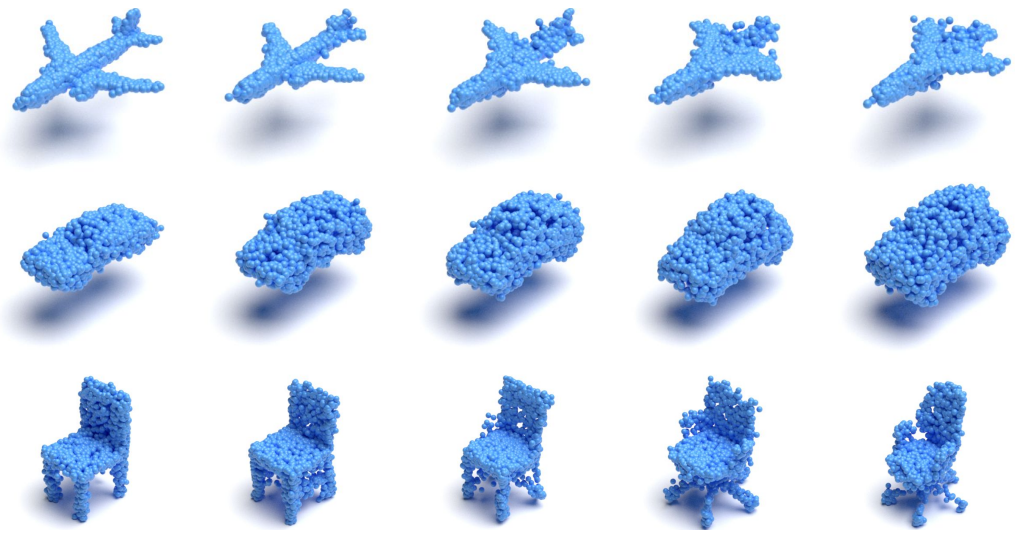

Figure 5.5: **Interpolation** of latent vectors e makes smooth transition between their reconstructions (from left to right). Each row shows interpolation on different classes of shapes.

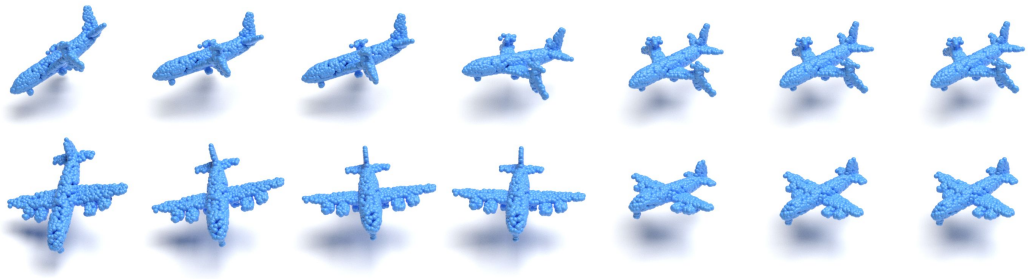

Figure 5.6: CIF is able to align objects into a canonical position. We extend the flow $\mathbf{f}$ by a multiplication by a 3D rotation matrix. We then optimize the cloud likelihood when conditioned on a null embedding vector with respect to this matrix. Each row shows subsequent steps of the optimization, starting from a random pose. The last image in each row is the object in the training set. We see that CIF is able to recover the default object pose used in the training data.

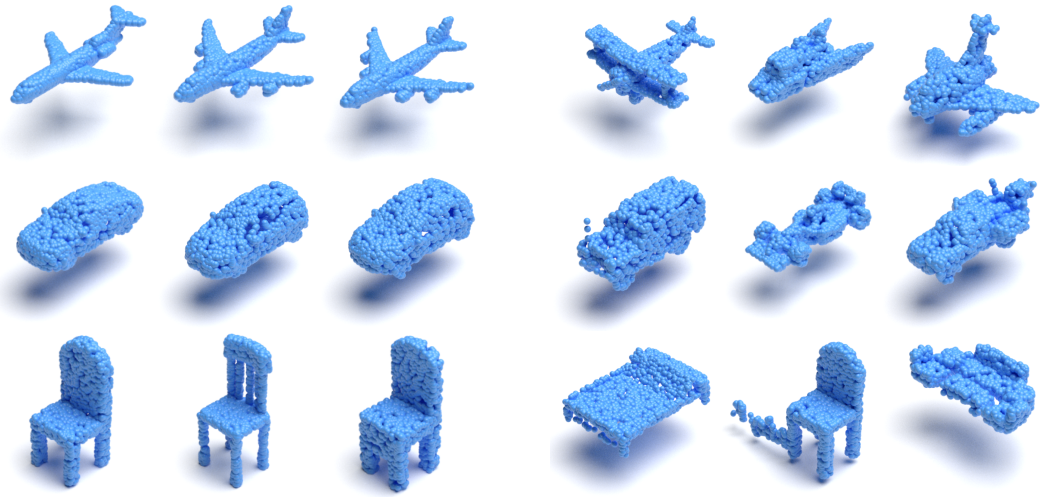

Figure 5.7: CIF is able to find the common (leftmost three clouds in each row) and rare (rightmost three) samples in the dataset by comparing the prior probability of their embeddings. Notice that using this approach, we detected erroneous samples in the dataset.

procedure [13]. We can see that this procedure is able to recover the default object pose used in the training dataset (pictured in the last column).

Determining typical objects and outliers We show that one can evaluate the likelihood $p(w|x^h)$ of the embedding vector of a given point cloud, which can be further used to determine the uniqueness of samples. We process point clouds from the dataset through **h** encoder and pass it to the **g** flow. Then, we calculate the negative likelihood of the embedding being sampled from the prior distribution $\mathcal{N}(0, I)$. The "rarest" samples in the dataset, according to the model, can be found by choosing point clouds with the top highest negative likelihood. These samples can be further investigated in the outlier detection procedure. Similarly, we can find the most typical point clouds by selecting samples producing the top lowest negative log-likelihoods. Using this method, we show in Figure 5.7 unique and typical samples from the ShapeNet dataset.

Chapter 6

Summary

We demonstrate a novel point-level, order-invariant method to encode and generate point clouds. The model treats point clouds as probability distributions and represents them using normalized flow-based models. It can be trained end-to-end efficiently, and be applied for generating new 3D point clouds that follow governing data distribution. We evaluated the performance of the method in both a qualitative and quantitative manner. Our model sets new state-of-the-art of generation quality on many parts of presented measures and performs on par with reference approaches on the rest. Moreover, we show that our model can have a variety of applications such as object alignment, typical object finding, and outlier detection.

Bibliography

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

[2] L. Dinh, J. Sohl-Dickstein, and S. Bengio, "Density estimation using real nvp," *arXiv preprint arXiv:1605.08803*, 2016.

[3] T. Q. Chen, Y. Rubanova, J. Bettencourt, and D. Duvenaud, "Neural ordinary differential equations," *CoRR*, vol. abs/1806.07366, 2018.

[4] W. Grathwohl, R. T. Q. Chen, J. Bettencourt, I. Sutskever, and D. Duvenaud, "Ffjord: Free-form continuous dynamics for scalable reversible generative models," *International Conference on Learning Representations*, 2019.

[5] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," in *Advances in neural information processing systems*, pp. 2672–2680, 2014.

[6] J. Donahue, P. Krähenbühl, and T. Darrell, "Adversarial feature learning," *arXiv preprint arXiv:1605.09782*, 2016.

[7] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *arXiv preprint arXiv:1512.03385*, 2015.

[8] C. R. Qi, H. Su, K. Mo, and L. J. Guibas, "Pointnet: Deep learning on point sets for 3d classification and segmentation," *CoRR*, vol. abs/1612.00593, 2016.

[9] P. Achlioptas, O. Diamanti, I. Mitliagkas, and L. Guibas, "Learning representations and generative models for 3d point clouds," *arXiv preprint arXiv:1707.02392*, 2017.

[10] G. Yang, X. Huang, Z. Hao, M.-Y. Liu, S. Belongie, and B. Hariharan, "Pointflow: 3d point cloud generation with continuous normalizing flows," *arXiv preprint arXiv:1906.12320*, 2019.

[11] C.-L. Li, M. Zaheer, Y. Zhang, B. Poczos, and R. Salakhutdinov, "Point cloud gan," *arXiv preprint arXiv:1810.05795*, 2018.

[12] D. P. Kingma and P. Dhariwal, "Glow: Generative flow with invertible 1x1 convolutions," in *Advances in Neural Information Processing Systems*, pp. 10215–10224, 2018.

[13] N. Hansen, "The CMA evolution strategy: A comparing review," in *Towards a New Evolutionary Computation - Advances in the Estimation of Distribution Algorithms* (J. A. Lozano, P. Larrañaga, I. Inza, and E. Bengoetxea, eds.), vol. 192 of *Studies in Fuzziness and Soft Computing*, pp. 75–102, Springer, 2006.