Take a Seat on our Generated Sofa

Sidhika Balachandar `sidhikab`
Samaksh Goyal `sagoyal`

Abstract

Recently, 3D point clouds have become the preferred data for several geometric modeling and computer vision problems. They provide an accurate yet sparse sampling of the physical world and provide easy integration to traditional image processing pipelines. They have also been rigorously explored in the deep learning community. In our project we compare GAN and flow-based networks for point cloud generation. We evaluate the performance of our models qualitatively and quantitatively using the Chamfer Distance metric. As found by others in the field, we see that generative models trained on a latent representation of the data produce better results than those trained on the raw data. These latent models also satisfy the permutation invariance property of point cloud data. Additionally, we find that both the latent GAN and flow models produce realistic point clouds.

1 Introduction

3D representations of real-life objects are important for problems involving computer vision, robotics, medicine, augmented reality, and virtual reality. In our project we plan to investigate the 3D world represented as point clouds. Point clouds are a scattering of 3D points on the surface of an object and are created with depth sensor tools such as LIDAR or RGB-D cameras, see 1. A point cloud is made up of an unordered list of (x, y, z) points. Point clouds are a popular form of representing 3D data particularly for deep learning tasks because they are the simplest representation of 3D objects. Point clouds only contain information about points in 3D space, they do not contain any information about connectivity between points. An alternative representation such as voxel based methods have been attempted in the past but the resolution of voxels can change depending on the sampling pattern of a scanner so this is not used in practice. Furthermore storing an entire mesh for a 3D object can become cumbersome as then face/edge/vertex connectivity also needs to be stored.

Deep learning based networks for point cloud generation have attracted a lot of attention recently. In particular, point cloud generation is helpful for video game designers. When creating a virtual world, artists do not want all of the furniture to look exactly the same. Therefore generative models are helpful to create various designs which can then be further tweaked by the designer.

In our project we plan to compare the performance of various GAN and flow-based networks for point cloud generation. Specifcally, we are investigative four different models: 1) the r-GAN model which is based on the raw point cloud, 2) the l-GAN model which is based on a latent representation of the point cloud specified by PointNet++, 3) the Wasserstein l-GAN which uses the Wassertstein loss, 4) an l-MAF normalizing flow model. We will compare the performance of each model both qualitatively and quantitatively. Qualitatively we will generate new pointclouds from the "sofa" class and compare their visualizations to real sofa data. Quantitatively we will compare the models using the Chamfer Distance metric presented in [1].

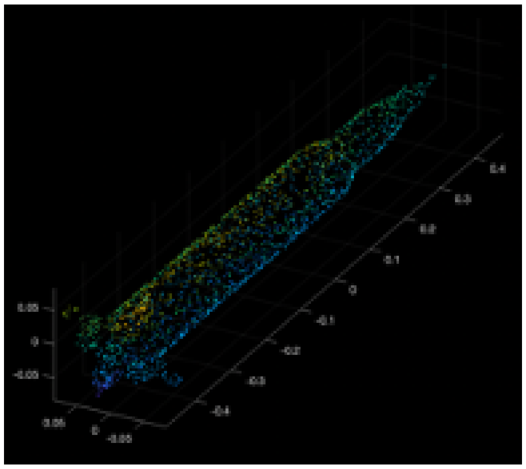


Figure 1: Point cloud representation of a rocket.

2 Related Work

The first GANs for point cloud generation are presented in [1]. This paper presents two GANs: r-GAN and l-GAN. r-GAN is the most simple constructiton of a GAN for point clouds. The generator network is a fully connected network that takes in the raw point cloud (hence the name r-GAN). The l-GAN modifies the generator network to instead operate in a latent representation of a point cloud. This latent representation represents the encoding specified by the trained PointNet++ model [2]. In our project we replicate the r-GAN and l-GAN networks. We do not use the exact same architecture presented in the paper, this architecture, particularly for r-GAN is bulky and requires parallelization across multiple GPUs. Thus instead we create a simplified version of the r-GAN and l-GAN that achieves similar performance.

We also create the first MAF based flow network for point cloud generation. The use of flow based generative models for point clouds has been explored in works such as [3]. The PointFlow model presents a continuous normalizing flow based network. Again in an effort to simplify the construction of our network, we choose to create a simple discrete normalizing flow network based on the MAF model.

There has been a lot of recent work that has improved on the above GAN and flow based models. In particular, many papers have improved the generator network through the incorporation of graph convolutional layers (GCN). Examples of these GCN based GANS include Tree-GAN [4] and GraphCNN-GAN [5]. Even newer research has improved on the GCN based generator by allowing for individual part generation and modifcation within an entire generated object [6] and by using knowledge from the mathematics of spherical geometry [7].

3 Approach

3.1 Generative Adversarial Networks

In this project we create three Generative Adversarial Networks (GANs). The basic architecture of a GAN is based on an adversarial game between a generator and a discriminator, see 2. The generator tries to generate samples that look indistinguishable when compared to real data samples. The generator generates a sample by drawing a random seed $z \sim p_z$, where p_z designates some noise distribution. z is then passed through the generator network. Ultimately the generator learns a distribution p_G, and we train the generator to learn a p_G that is as close as possible to the real p_{data}. The discriminator is tasked with distinguishing the real data samples $x \sim p_{data}$ from the fake generated samples $x' \sim p_G$.

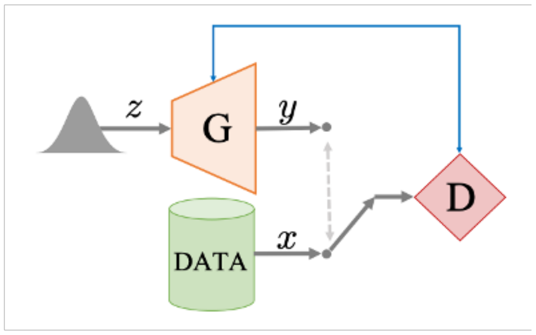


Figure 2: A GAN is based on an adversarial game between a generator (G) and a discriminator (D). Image from [1].

We create three different GANs: 1) the r-GAN model which is based on the raw point cloud, 2) the l-GAN model which is based on a latent representation of the point cloud specified by PointNet++, and 3) the Wasserstein l-GAN which uses the Wassertstein loss. We will start by describing the difference between the r-GAN and the l-GAN.

3.1.1 r-GAN versus l-GAN

The r-GAN and the l-GAN differ in what they model as the data distribution, p_{data}. The r-GAN models p_{data} as the distribution of all raw point clouds. In our case a raw point cloud is a list of 2048 (x, y, z) points. p_{data} assigns a probability to all possible lists of 2048 (x, y, z). Therefore the model is called the raw-GAN or r-GAN. On the other hand, the l-GAN models p_{data} as the distribution of latent representations of point clouds. In our case a latent representation of a point cloud is a vector of size 128. p_{data} assigns a probability to all possible vectors of size 128. Therefore the model is called the latent-GAN or l-GAN.

3.1.2 PointNet AutoEncoder

The latent representation of a point cloud used in the l-GAN is obtained using the PointNet AutoEncoder's encoder network. Due to the importance of the latent representation, we will now briefly explain the PointNet AutoEncoder model and how it differs from other autoencoders.

An autoencoder is a machine learning model used to learn efficient data encodings. It is made up of two parts: an encoder, ϕ, and a decoder, ψ, such that

$$\phi : \mathcal{X} \rightarrow \mathcal{F} \tag{1}$$

$$\psi : \mathcal{F} \rightarrow \mathcal{X} \tag{2}$$

Here $\mathcal{X}$ represents the set of all 3D point clouds and $\mathcal{F}$ represents the set of all latent representations of elements in $\mathcal{X}$. Then the encoder can be thought of as a function mapping an input 3D point cloud to a latent representation of the input. The decoder can be thought of as a function mapping a latent representation to it's 3D point cloud. Since an autoencoder feeds the output of the encoder to the decoder, ideally the output of an autoencoder model is equivalent to the 3D point cloud fed in as input.

One of the quirks of point cloud data is that it has no ordering. In other words, point cloud data is permutation invariant, the points can be reordered or permuted and they will still represent the same object. Thus when building models for point cloud data, such as an autoencoder or a generative model, we want to ideally construct a permutation invariant model. In other words, the output of the model is the same no matter the ordering of the input. The PointNet AutoEncoder is a permutation invariant autoencoder. The permutation invariance property is enforced through the input representation of the point cloud and through the loss function used to train the autoencoder model.

Permutation Invariant Input Representation

In order to enforce the permutation invariance property, we cannot immediately run the encoder network on an input point cloud. In order to make sure that the output of the encoder network is permutation invariant, we must first obtain a permutation invariant representation of the input point cloud. Then we can run the encoder network on this permutation invariant representation. In order to obtain a permutation invariant representation of the input, we can either sort the input into a canonical order or use a simple symmetric function. The first strategy requires us to preprocess our data such that each point cloud matches the canonical representation. The second strategy does not require any preprocessing, and thus it is the option we will use.

A symmetric function is a function that is permutation invariant. This means that it is invariant to the order of the inputs to the function. For example, + and $\times$ are invariant functions since they are commutative operators. We will approximate a symmetric function f using functions g and h in the following way:

$$f(\{x_1, \ldots, x_n\}) \approx g(h(x_1), \ldots, h(x_n)) \tag{3}$$

In equation 3, $\{x_1, \ldots, x_n\}$ represents the input set of n points and each $x_i \in \mathbb{R}^3$. h will be approximated as a Conv1d function and $h : \mathbb{R}^3 \rightarrow \mathbb{R}^d$. We will call $\mathbb{R}^d$ the channel dimension. Then g will be approximated as a max function over the channel dimension and $g : \mathbb{R}^{n \times d} \rightarrow \mathbb{R}^d$.

Since the max function is commutative, it is clear that g is a symmetric or permutation invariant function. We will now discuss why we model h with a Conv1d function. Suppose we start with a point cloud with 1000 data points. If h is the identity function, the max function reduces the dimensionality from 1000×3 to 3 and thus throws out a lot of valuable data. Thus we must first increase the dimensionality of the input from 1000×3 to $1000 \times d$, where $d >> 3$. Then when we take the max, we end up with an output of dimension d instead of 3. We use the Conv1d function to increase the dimensionality. We choose a Conv1d funtion over a simple linear layer because the Conv1d uses fewer parameters.

Permutation Invariant Chamfer Loss

One of the most standard loss functions is the MSE loss. For a pointcloud autoencoder, the MSE loss would calculate the mean squared error between each point in the input point cloud and the output point cloud. In particular, the ith point in the input point cloud is compared to the ith point in the output point cloud. Since point cloud data is permutation invariant, it is undesirable to enforce this pairing between input and output points. Thus for our model, instead of using MSE loss, we use a loss function based on the Chamfer Distance.

The Chamfer distance (CD), like MSE loss, calculates the error between two sets of points. CD measures the squared distance between each point in one set to its nearest neighbor in the other set. The equation for the Chamfer-loss is provided below

$$d_{CH}(S_1, S_2) = \frac{1}{N} \sum_{x \in S_1} \min_{y \in S_2} \|x - y\|_2^2 + \frac{1}{N} \sum_{y \in S_2} \min_{x \in S_1} \|x - y\|_2^2 \tag{4}$$

In simpler terms, CD takes a point x in the input point cloud. It then tries to find the point y in the output point cloud that is most likely to match the input point. Note that MSE loss chooses the matching output point to be the point at the same index as the input point. CD instead searches over all points in the output point cloud and chooses the point closest in distance to the input point. After pairing the input point with an output point, the error is then calculated using the L2 norm, similar to the MSE loss. Also similar to the MSE loss, in the end in order to find the loss with respect to the entire input point cloud we calculate the mean error across all points in the input point cloud. This represents the first summation term in equation 4.

The second summation term repeats this process but instead takes each point in the output point cloud and finds the corresponding closest point in the input point cloud. For MSE loss, since the input and output points are paired by index, the mapping is equivalent whether evaluated with respect to the input or the output points. However, for CD, there is no guarantee that we will have the same pairings. In other words, CD is antisymmetric, and thus we must calculate the loss with respect to the input points and the loss with respect to the output points. Our final loss is the sum of these two terms.

3.1.3 GAN Loss versus Wasserstein Loss

The r-GAN and l-GAN were trained using the standard non-saturating GAN loss function. Our last model, the Wasserstein l-GAN uses the Wasserstein loss instead of the standard GAN loss. To review from lecture, the standard GAN loss function is a minimax loss over the generator and discriminator. Mathematically, the loss is

$$\min_G \max_D L(G, D) = E_{x \sim p_{data}} [\log D(x)] + E_{x \sim p_G} [\log (1 - D(x))] \tag{5}$$

However, this form of the loss suffers from the vanishing graident problem. Therefore for the non-saturating loss, the discriminator's loss stays the same, however the generator's loss is the following:

$$L_G = -E_{x \sim p_G} [\log D(x)] \tag{6}$$

This GAN algorithm can be thought of as minimizing the divergence between a data distribution p_{data} and a model distribution p_G. When both p_{data} and p_G place probability mass on only a very small part of the domain, the loss goes to infinity. In order to avoid this instability, in our third model we use the Wasserstein loss with gradient penalty. This loss is defined as the following:

$$L_D = \mathbb{E}_{x \sim p_G} [D(x)] - \mathbb{E}_{x \sim p_{data}} [D(x)] + \lambda \mathbb{E}_{x \sim r} \left[(\|\nabla D(x)\|_2 - 1)^2 \right] \tag{7}$$

$$L_G = -\mathbb{E}_{x \sim p_G(x)} [D(x)] \tag{8}$$

Here r is defined by sampling $\alpha \sim Uniform([0, 1]), x_1 \sim p_\theta(x)$, and $x_2 \sim p_{data}(x)$, and returning $\alpha x_1 + (1 - \alpha) x_2$. The WGAN enables stable training by penalizing functions whose derivative is too large. The hyperparameter λ controls the strength of the penalty.

3.2 Normalizing Flow Model

An alternative to the GAN based generative model, we also have a normalizing flow based model to create samples of our shape class. A normalizing flow based model is one such that the change of variables gives a normalized density after applying an invertible transformation and that these transformation can be composed with each other to create more complex invertible functions. Below $\mathbf{f}_\theta$ represents such a function:

$$p_X(\mathbf{x}; \theta) = p_Z \left(\mathbf{f}_\theta^{-1}(\mathbf{x}) \right) \left| det \left(\frac{\partial \mathbf{f}_\theta^{-1}(\mathbf{x})}{\partial \mathbf{x}} \right) \right| \tag{9}$$

Our Normalizing Flow Model was MAF (Masked Autoregressive Flow) where the forward mapping is an autoregressive model, where sampling is sequential and slow yet training is fast and parallel.

3.3 Sampling

Recall our objective is to transform a sample z from a normal distribution into point cloud representation of a shape. In our r-GAN the generator satisfies this by outputting a 2048 points at the end of the neural network. In all of our latent generators (l-GAN, Wasserstein l-GAN, and l-MAF), once a sample z is passed through the generator, it is then passed through our saved Autoencoder's decoder block to recreate the point cloud. Thus for each each shape class there are two models that need to be trained, the first is the Autoencoder for that class and the second is the generative model for that class.

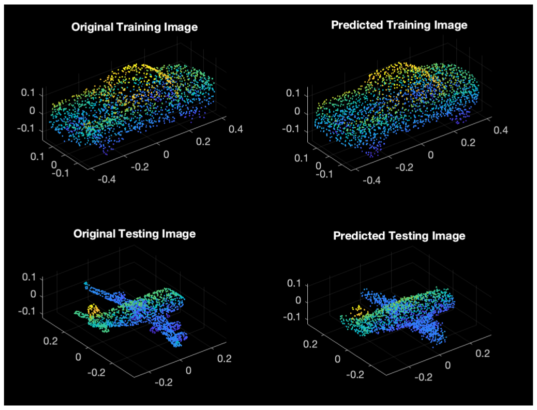


Figure 3: Comparison of decoded training vs testing point cloud using pointnet model.

4 Results

In order to recreate l-GAN from [1] we first create an AE and assessed the quality of the latent space. Our next steps will be to use this latent space as the input to the generator and discriminator in a GANs framework. We will then sample point clouds by take a vector produced by the generator and passing it into the AE's decoder. Below we describe AE results.

4.1 AE Validation

The autoencoder model we used is based of the model in the paper by [1]. Their model is structured as follows: The encoder contains 1D convolutions with batch normalization and ReLUs, with kernel size of 1 and stride of 1. This allows us to treat each 3D point independently. The decoder contains fully connected layers with FC-ReLUs. An Adam optimizer is used with an initial learning rate of 0.0005, β_1 of 0.9 and a batch size of 50 to train all AEs.

We created three different models:

1) The baseline autoencoder architecture has a bottleneck layer of 128 (which was found to be the most generalizable in [1]). Our layers for the encoder take as input flattened 2048×3 points in the point cloud, followed by layers of size 4096, 2048, 1024, 128, and the layers for the decoder are the same in reverse order. We also add batch normalization and leaky ReLU to ensure back propagation of gradients. The baseline model was also trained using a mean square error (MSE) loss.

2) In our next model we used the same architecture but substitute the MSE loss with the Chamfer loss.

3) For our final model, the architecture we used was inspired by [3] to incorporate permutation invariance. For our encoder we used three sets of 1 dimensional convolutions, batchnorm, and leaky relu to increase the dimension of each point from 3 (x,y, z) to 512. Then we took a max over all the points and used 2 fully connected layer to produce the same hidden dimension size as the baseline, 128. For the decoder we use the same architecture as our baseline. We also use Chamfer loss.

Our Pointnet based architecture that uses Chamfer loss and permutation invariance provides significant visual similarity. To best reflect the advancements we also train our model on two different categories: cars and planes.

When we visualized the results for a car point cloud that was present in the train set, we are able to almost fully recover the original shape. Most interestingly we conserve small domain specific details like the curvature of the wheels of the car, however we lose the the rigid structure of the truck bed which now has a softer profile. As a test we try to visualize an airplane and the model is able to get a reasonable approximation that matches the general frame and span of the wings. However during testing we lose many details like the nose angle and the 4 engines beneath the wings. We can see

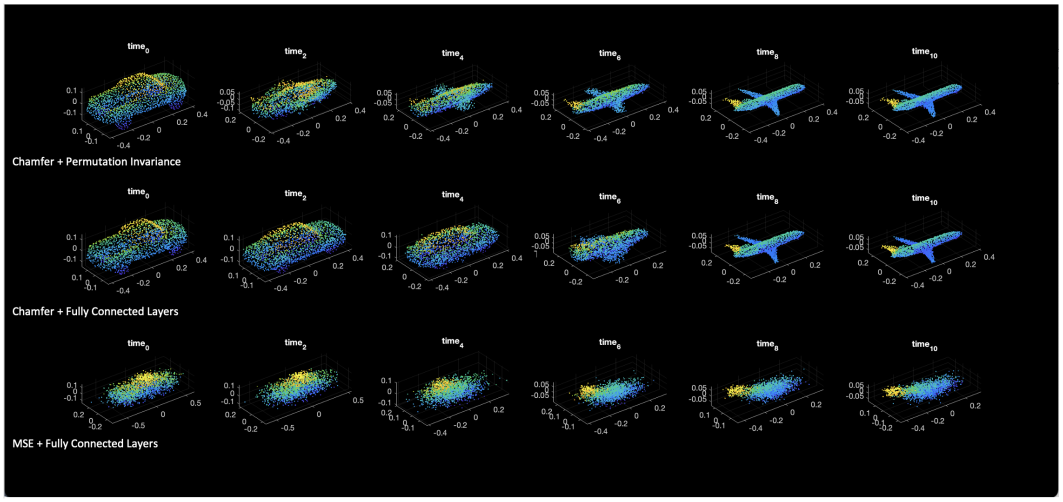

Chamfer + Permutation Invariance
Chamfer + Fully Connected Layers
MSE + Fully Connected Layers
time$_0$
time$_2$
time$_4$
time$_6$
time$_8$
time$_{10}$

Figure 4: Interpolation in latent space between car and airplane for three different models.

that the autoencoder is struggling with this since it produces a heavier set of point clouds where the engines should be, however it does not recover their cylindrical shape.

4.2 Latent Space Analysis

To better compare the latent space learned by our three models, figure 1 provides the interpolated representations between two interpolants: a car and an airplane.

In figure 1, the baseline model consisting of fully connected layers and MSE loss is on the bottom row. Because the encodings of the two interpolants are extremely inaccurate, so are their intermediate interpolations.

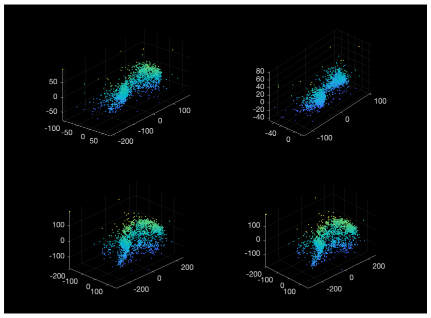
In figure 1, our second model in which retain our fully connected layers but add in Chamfer loss, is on the middle row. The encoding is of greater visual quality for both interpolants. We see that using a permutation invariant loss function has considerable advantages. Since Chamfer distance is attempting to minimize the misalignment of each point from the other point, there is a serious penalty if even one of the x,y,z coordinates aren't close to the target.

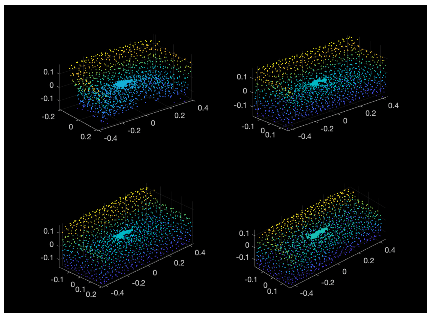
An interesting note in the interpolation is that we have a sharp transition between time 4 and time 5 of the middle row. Recall that since we are interpolating in the latent space and then decoding back into the point cloud space, what we might possibly be seeing here is a 128 dimensional hyperplane separating the spaces of the cars and planes. Once enough of the individual dimensions are close to a specific typical "car" or "airplane" vector then the decoder recognizes this transition and transforms it into that dimension.

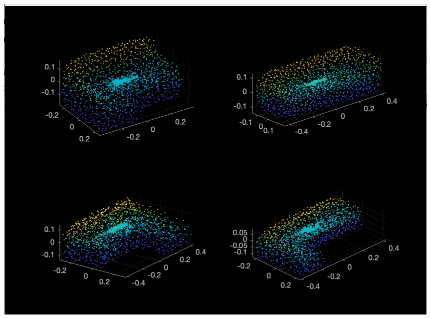
Our last model, for which we replace the fully connected layers with a permutation invariant layer, is shown in the top row. Time steps 2 and 4 represents a more hybrid shape of the mixture of a car and a plane. This seems to suggest that the latent space learned by adding permutation invariance is more continuous. Furthermore step 2 is quite intriguing because it shows the formation of wing like structures on the car while also preserving the the hood and trunk of a pickup truck. Thus showing a smoother transfer of information than compared to the previous model.

Lastly, recall the goal of interpolating in the latent space: to measure the degree to which our compressed latent dimension is able to capture the geometric topology of the space represented by the car-plane shapes. Based on figure 1 it seems that compressability was largely improved by using Chamfer distance, while the smoothness of the geometric topology was improved by permutation invariance.

4.3 Generated Sofa Samples


(a)


(b)


(c)

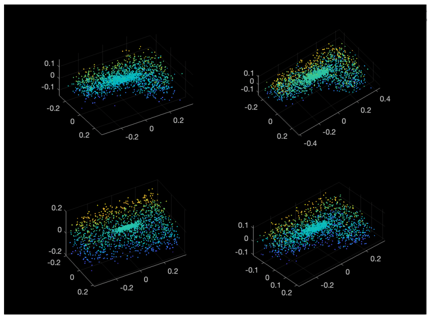

(d)

Figure 5: a: r-GAN, b: l-GAN, c: l-MAF, d:Wasserstein l-GAN

We now evaluate the visual quality of the generated samples by passing 4 random seeds from a multivariate normal distribution. As the figure shows r-GAN remains unable to sample point clouds that resemble any form of a sofa, suggesting that a point cloud is too rich of a representation to learn from. This could also be due to the number of layers and specific convolutions applied,[1] and all were infact able to reconstruct crude representations of the target shape. l-GAN and l-MAF adequately samples sofas, which indicates using AE latent representations as proxy for the richer point cloud data is effective. Wasserstein l-GAN results in fuzzy sofas, could be due to the strength of λ and norm selection.

Note that interestingly the sofas generated are topologically varying. We see that L-shaped couches and once with different back, shoulder support and different arm rest configurations. This indicates that the model is able to sample from a spectrum of couches, yet the ability to model the full distribution of couches must be shown quantitatively.

The concentrated blue patch in center possibly corresponds to generator always learning to scatter points on the middle seat of a sofa (as all sofas are within unit sphere).

4.4 Chamfer Distance Sample Distribution

As a quantitative analysis we examine the distribution of the average Chamfer distance (CD) for 100 generated samples. The average CD for one generated sample is calculated by taking the average of the CDs between the generated sample and 100 real samples. “Data” represents the average CD between two distinct sets of 100 real samples. The “Data” distribution is skewed-right which means that the real distribution of sofas has high variety. None of the generative models can capture the same amount of variety, but the l-GAN does the best. In other words, the l-GAN is the most skewed-right. Typically, a lower CD indicates a better model. These results suggest that CD is not the best comparison metric.

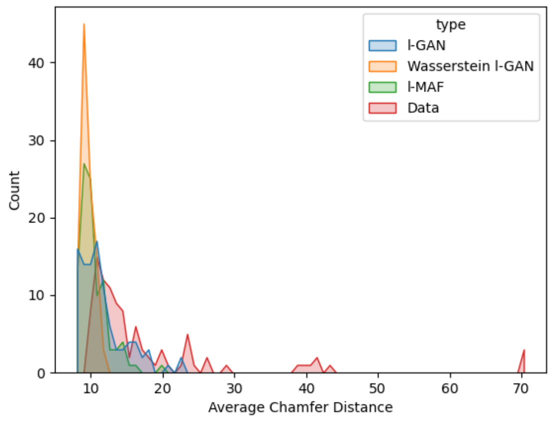


Figure 6: Understanding the Distribution of Sofas

5 Conclusion

Recall that our goal was to compare various generative models for point cloud generation. Our results showed, and matched [1] that point clouds themseleves are too complex and inconsistently sampled to develop systematic algorithms on top off. Thus it is easier to treat their latent representation from an AE model as proxy for data distribution which leads to better application of generative models. This was seen by the visual generations of the sofa in l-GAN and l-MAF.

Through our quantitative evalution which looks at the average CD distances between pairs of generated and data point cloud, l-GAN appears to do the best job at capturing the diversity of the real sofa distribution. However a more rigorous analysis is needed to fully justify this claim. The inherent problem with the current approach is that we are taking samples from two different underlying distributions and trying to make judgements based on some statistic based on those samples. The proper way of comparing the generative and data distributions would be just that, to directly compare the distributions. In this effort recent papers establish new metrics such as the Frechet Point Cloud Distance [8], which determine the difference between generated probability measures and real probability measures.

An obvious extension of our work would be generalize our uni-class sampling strategy to a multi-class problem. For example, we could devise a two step sampling scheme that first sample a shape category, such (as motorcycle, car, sofa, etc.) and then samples a point cloud given a shape category as seen in [3].

References

[1] Panos Achlioptas, Olga Diamanti, Ioannis Mitliagkas, and Leonidas J Guibas. Learning representations and generative models for 3d point clouds. *arXiv preprint arXiv:1707.02392*, 2017.

[2] Charles R Qi, Li Yi, Hao Su, and Leonidas J Guibas. Pointnet++: Deep hierarchical feature learning on point sets in a metric space. *arXiv preprint arXiv:1706.02413*, 2017.

[3] Guandao Yang, Xun Huang, Zekun Hao, Ming-Yu Liu, Serge Belongie, and Bharath Hariharan. Pointflow: 3d point cloud generation with continuous normalizing flows. 2019.

[4] Dong Wook Shu, Sung Woo Park, and Junseok Kwon. 3d point cloud generative adversarial network based on tree structured graph convolutions. *CoRR*, abs/1905.06292, 2019.

[5] Diego Valsesia, Giulia Fracastoro, and Enrico Magli. Learning localized generative models for 3d point clouds via graph convolution. In *International Conference on Learning Representations*, 2019.

[6] Rinon Gal, Amit Bermano, Hao Zhang, and Daniel Cohen-Or. MRGAN: multi-rooted 3d shape generation with unsupervised part disentanglement. *CoRR*, abs/2007.12944, 2020.

[7] Ruihui Li, Xianzhi Li, Ka-Hei Hui, and Chi-Wing Fu. SP-GAN: sphere-guided 3d shape generation and manipulation. *CoRR*, abs/2108.04476, 2021.

[8] Dong Wook Shu, Sung Woo Park, and Junseok Kwon. 3d point cloud generative adversarial network based on tree structured graph convolutions. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 3859–3868, 2019.