GAN architectures for 3D Models

Daniele Alma
University of Catania
8 PUBLICATIONS **0** CITATIONS
SEE PROFILE

Daniele Alma

GAN architectures for 3D Models

Deep Learning

Summary

Abstract

Introduction

The automatic generation of three-dimensional objects is not a trivial problem: 3D models are much more complicated than images geometrically speaking, and a certain level of detail is often needed by companies.
The manual creation of a model can take a long time, especially with lack of references. From an economical point of view, commissioning numerous models of particular objects can get really expensive, even for big companies and for relatively simple shapes.

Photogrammetry

Photogrammetry often helps in the creation of 3D models. Photogrammetry is a science with the objective of getting reliable information from physical objects and the environment through the process of registration, measure and interpretation of photographic images. Photogrammetry techniques can be used to get detailed **three-dimensional models** from a **sequence of images** of the same object by using algorithms for image processing, key points extraction and Structure from Motion.

A typical workflow for making a model with photogrammetry can require numerous steps: planning, image acquisition, reading the camera intrinsic parameters, SIFT extraction, Structure From Motion, dense point cloud computing, Delaunay triangulation and smoothing, texturing and finally post-processing. The dense point cloud computing from key points and information from the cameras is computationally expensive and is often done on the **GPU**.

GAN and 3D model generation

GAN, Generative Adversarial Networks, are a particular kind of neural network that can generate new samples of a certain domain starting from a vector in a latent space; combined with adequate encoders that can generate these vectors from a source domain, a robust conversion architecture can be obtained, compared to a simple regressor, and the generator can be used independently. For 3D models, the source domain can consist in one or many bidimensional traditional RGB images and/or depth maps and normal maps and the destination domain is a representation for three-dimensional objects.

One of the first frameworks realized for this objective is **3D-GAN** [1], that can generate 3D models from a probabilistic latent space using volumetric convolutional networks and GAN. The adversarial framework can implicitly capture object structure in the network weights.

1. GAN architectures

1.1 Definitions

A GAN is an architecture that consists in two neural networks that are trained in a competitive way with respect to a **zero-sum game**, in which the loss of an agent is the win of the other.
The two neural networks are a **generative** one, that generates candidates of the target domain, and a **discriminative** one, that evaluates the candidates to discriminate between those generated by the generative network and those coming from the dataset actual distribution.
The discriminative network is trained using the training dataset and the generator network is trained with the goal to “trick” the discriminative network, by producing candidates as realistic as possible with respect to the target domain.
Formally, given a latent space z having a prior distribution $p_z(z)$, the generative network is a function $G(z, \theta_g)$ that produces as output new data with respect to the distribution p_g with learned parameters θ_g. The discriminator is a function $D(x, \theta_d)$, in which θ_d are the model parameters and the output is the probability that x belongs to the training data distribution p_{data}.

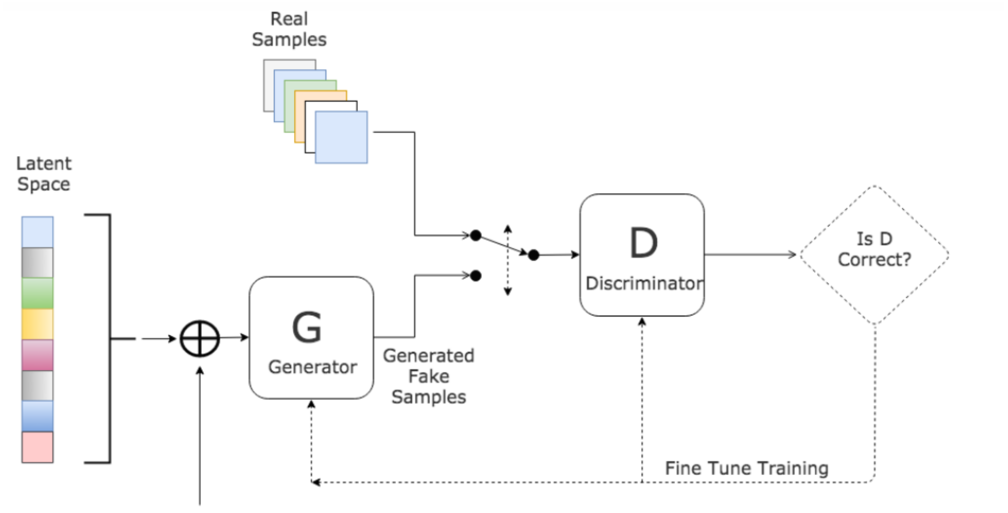


Figure 1:Structure and training of a GAN

1.2 Training

The competitive training consists in optimizing the discriminator for the classification of the training set samples (with, for example, a **cross-entropy** loss), while the generative network is optimized by trying to maximize the probability that the discriminator considers the samples produced by the generative network as coming from p_{data}.

A final objective function can be:

$$\underset{G}{min}\,\underset{D}{max}\,\mathbb{E}_{x\sim p_{data}(x)}[\log D(x)] + \mathbb{E}_{z\sim p_z(z)}\left[\log\left(1 - D\big(G(z)\big)\right)\right]$$

Usually, instead of first fully optimizing the D function parameters, k optimizations steps of D and one optimization step of G are alternated: this way, D can be kept close to its ideal parameters while the parameters of G are slowly improving.

In the 3D-GAN framework, the training uses **Stochastic Gradient Descent** this way: for each of the k training steps of the discriminator, a certain number m of training records $\{x^{(1)}, \dots, x^{(m)}\}$ are taken from the dataset with distribution p_{data} and m random noise samples $\{z^{(1)}, \dots, z^{(m)}\}$ from the generator distribution $p_g(z)$. The gradient for the discriminator is computed with these tensors as such:

$$\nabla_{\theta_d}\frac{1}{m}\sum_{i=1}^{m}\left[\log D\big(x^{(i)}\big) + \log\left(1 - D\left(G\big(z^{(i)}\big)\right)\right)\right]$$

After k optimization steps of the discriminator, a single Stochastic Gradient Descent iteration of the generator is done by taking m random noise samples $\{z^{(1)}, \dots, z^{(m)}\}$ from the distribution $p_g(z)$ and computing the gradient:

$$\nabla_{\theta_g}\frac{1}{m}\sum_{i=1}^{m}\log\left(1 - D\left(G\big(z^{(i)}\big)\right)\right)$$

It can be proven that this optimization results in the ideal p_g parameters such that $p_g \sim p_{data}$.

1.3 Cons

One of the cons of GAN architectures is the lack of a clear representation of the generator distribution $p_g(x)$ and the expertise necessary to **synchronize** the two networks D and G during training: if G is optimized too quickly compared to D, the G network will tend to exploit current D local minima, which means it overfits from outputs $G(z)$ that are classified by the discriminator as legitimate in that moment.

On the other hand, excessively training D tends to generate useless gradients for the generator.

2. 3D-GAN

2.1 Definition

The 3D-GAN architecture [1] consists in a generator G that maps a **latent vector** of 200 components z, randomly generated from a latent probabilistic space, to a $64\ x\ 64\ x\ 64$ **grid** that represents the object $G(z)$ **"voxelized"** in three dimensions: a position in the grid is called voxel and its value is equal to 1 if the voxel is occupied and 0 otherwise.

The discriminator D, like in all GAN architectures, returns the probability that a certain input is real or synthetic.

2.2 Network structure

The generator G of 3D-GAN consists in 5 volumetric convolutional layers with kernels of size $4\ x\ 4\ x\ 4$ and stride 2 (except the first layer with stride 1), with each a batch normalization layer and ReLU in between and a terminal sigmoid function.

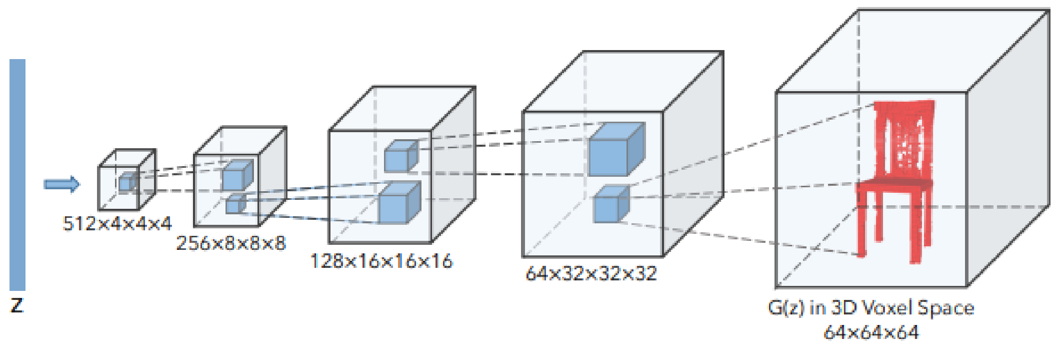


Figure 2: 3D-GAN generator

By using the sigmoid as a final activation function, the generator output can be interpreted as the probability of each voxel being occupied. The final 3D model will be created by thresholding the output with respect to 0.5.

The discriminator D has a specular structure with respect to the generator G: it takes a tensor of size $64\ x\ 64\ x\ 64$, which goes through 5 volumetric convolutional layers with Leaky ReLU with coefficient 0.2 as activation and batch normalization in between, and also a final sigmoid activation. The discriminator returns a single value in $[0,1]$, that represents the probability that a given three-dimensional object is real.

2.3 Training

Like all GAN architectures, the discriminator is privileged during training, but to avoid complicating the task for the generator which might have difficulty finding ideal values for an already good discriminator, an adaptive training is executed by updating the discriminator parameters only if its accuracy in the last batch is less than 0.8.

2.4 3D-VAE-GAN

The 3D-GAN model input is a vector with 200 components of a certain latent space: to use bidimensional images as input, it is necessary to extend the network and find a representation for them in this latent space.
3D-VAE-GAN extends 3D-GAN by introducing an image **encoder** E, which consists of 5 convolutional layers with kernel sizes $11\ x\ 11, 5\ x\ 5, 5\ x\ 5$ and $8\ x\ 8$ with strides 4, 2, 2, 2, 1 respectively. There still are intermediate layers with ReLU and batch normalization and the output of the last layer is a vector of 400 components, representing the gaussian distribution of the 200 dimensional space, where 200 values are for the mean and 200 for the logarithm of the variance. The final output vector in the latent space is obtained by sampling the gaussian distribution.
The optimization for 3D-VAE-GAN consists in the 3D-GAN adversarial loss L_{3D-GAN}, a **reconstruction loss** L_{recon} and the **Kullback-Leibler** divergence loss L_{KL}, which measures the differences between le distribution of the latent representation z and a normal multivariate distribution $p(z)$.
The last two are weighted with coefficients, set by the authors to $\alpha_1 = 5$ and $\alpha_2 = 10^{-4}$.

$$L = L_{3D-GAN} + \alpha_1 L_{KL} + \alpha_2 L_{recon}$$

$$L_{3D-GAN} = \log D(x) + \log\left(1 - D\big(G(z)\big)\right)$$

$$L_{KL} = D_{KL}(q(z|y)\ ||\ p(z))$$

$$L_{recon} = \left\|G\big(E(y)\big) - x\right\|_2$$

In which x is a 3D model of the training dataset, y is its image and $q(z|y)$ is the distribution of the latent representation z.
The training of the extended network 3D-VAE-GAN involves three steps:

1. Update the discriminator D parameters by minimizing the loss function L_{3D-GAN}.
2. Update the encoder parameters E by minimizing the loss function $L_{KL} + L_{recon}$, by using as $q(z|y)$ in the Kullback-Leibler divergence a gaussian distribution with mean and variance predicted by the encoder.
3. Update the generator G parameters by minimizing:
   $\log\big(1 - D(G(z))\big) + \left\|G\big(E(y_i)\big) - x\right\|_2$
   This involves the loss to maximize the probability that D considers the generated samples legitimate and the reconstruction loss with the encoder E.

2.5 Multiview 3D-VAE-GAN

The encoder of the 3D-VAE-GAN network can be extended to take more images as input.
Each available image of the three-dimensional model is given as input to the convolutional layers to generate a prediction in the latent space; then the predictions are put together in a single vector and passed to the generator.
To **combine predictions** in a single vector z for the generator various strategies can be used, such as pooling (in this case, since we have vectors instead of feature maps, it is intended as a combination of the components of the vectors in the same position) or a LSTM.

The Kullback-Leibler loss must now take into account each of the N points of view, to make sure the distribution is close to a gaussian, so an average is taken.

$$L_{KL-MV} = \frac{1}{N}\left(\sum_{i=0}^{N} -\frac{1}{2} * \sum\left(1 + z_{y_i} - z_{\mu_{y_i}}^2 - e^{z_{y_i}}\right)\right)$$

2.6 Implementation

The implementation was done in PyTorch and PyTorch Lightning on the Google Colab platform. The models will be projected and visualized as point clouds in matplotlib.

2.6.1 Evaluation metrics

The metrics used to evaluate three-dimensional models will be the following:

- The reconstruction loss, also optimized on the training set:
  $L_{recon} = \left\|G\left(E(y_i)\right) - x\right\|_2$
  In which x is the real model and $G\left(E(y_i)\right)$ the probability values for each voxel as predicted by the architecture.
- The **Intersection over Union** value:
  $$IoU = \frac{\left|M \cap \widehat{M}\right|}{\left|M \cup \widehat{M}\right|}$$
  In which M is the set corresponding to the voxels present in the model, while $\widehat{M}$ is the set of voxels that the architecture predicts as present (meaning a threshold with respect to 0.5 is applied).
- The SSIM value, **Structural Similarity**, which will be explained in another paragraph.
  $$SSIM(x,\hat{x}) = \frac{(2\mu_x\mu_{\hat{x}} + C_1)(2\sigma_{x\hat{x}} + C_2)}{\left(\mu_x^2 + \mu_{\hat{x}}^2 + C_1\right)\left(\sigma_x^2 + \sigma_{\hat{x}}^2 + C_2\right)}$$
  In which x is a window of the real model and $\hat{x}$ the probability values for each voxel predicted by the architecture.

The reconstruction loss and in particular the IoU value are heavily dependent on the voxel positions; a high IoU value on the training set might indicate overfitting.

2.6.2 Chosen dataset

The dataset that will be utilized for training and validation is **modelnet40v1png**, also used for MVCNN [2]. The dataset has 40 classes of objects that can be found in a house, like chairs, tables and doors. For each class, there are 80 training records and 20 validation records.
The models are in OFF format and each one of them has 12 $224x224$ images associated to them in PNG format for 12 different points of view.

The models in OFF format are converted in **BINVOX** format with a tool with the same name [3], then they are turned into numpy vectors with binvox-rw-py [4] and finally turned into tensors.

The first tests were done with three-dimensional voxelized models in a $32\ x\ 32\ x\ 32$ grid and for training a single class is chosen.

2.6.3 Adapting the model

The dataset chosen has very **few samples** per class: before training the actual models, various tests were made to see how the model fares with limited data.
Different tests were made with 150 epochs of training of the base model on the chair class, with a batch size of 20, a learning rate of 0.0025 for the generator, of 0.00001 for the discriminator and 0.0001 for the encoder; the momentum was set to 0.5.

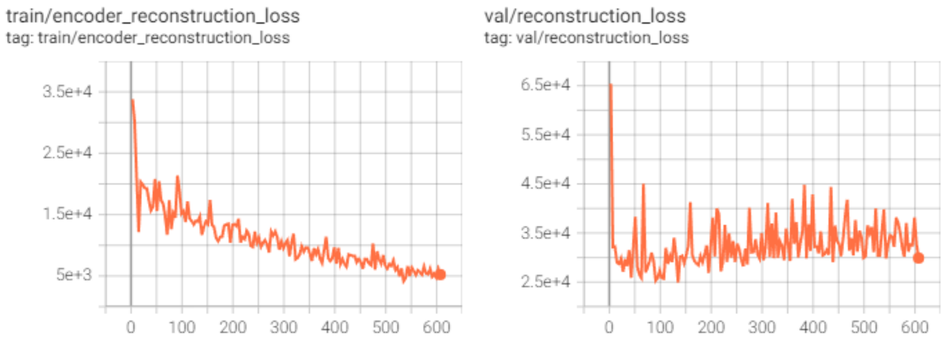


Figure 3:Reconstruction loss first model

The VAE (encoder plus generator) mainly learns to reconstruct training data and the reconstruction loss on the validation set does not decrease after a certain point.
In general, the whole models **overfits**, as can be seen by the discriminator accuracy.

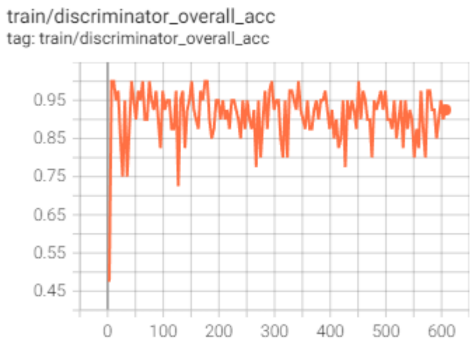
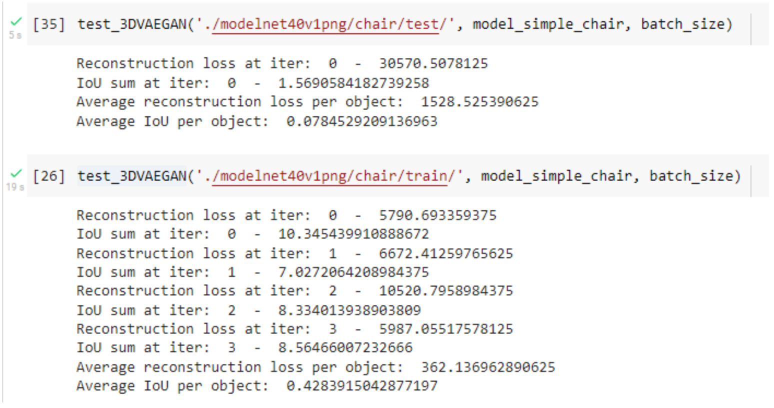


Figure 4:First model discriminator accuracy

The reconstruction loss per object on the training set is pretty small compared to the validation set. There is also a very low IoU value per object on the validation set, but the IoU value on the training set is not excessively high, which means the generator is at least not doing a direct mapping.



```
[35] test_3DVAEGAN('./modelnet40v1png/chair/test/', model_simple_chair, batch_size)

     Reconstruction loss at iter:  0  -  30570.5078125
     IoU sum at iter:  0  -  1.5690584182739258
     Average reconstruction loss per object:  1528.525390625
     Average IoU per object:  0.0784529209136963

[26] test_3DVAEGAN('./modelnet40v1png/chair/train/', model_simple_chair, batch_size)

     Reconstruction loss at iter:  0  -  5790.693359375
     IoU sum at iter:  0  -  10.345439910888672
     Reconstruction loss at iter:  1  -  6672.41259765625
     IoU sum at iter:  1  -  7.0272064208984375
     Reconstruction loss at iter:  2  -  10520.7958984375
     IoU sum at iter:  2  -  8.334013938903809
     Reconstruction loss at iter:  3  -  5987.05517578125
     IoU sum at iter:  3  -  8.56466007232666
     Average reconstruction loss per object:  362.136962890625
     Average IoU per object:  0.4283915042877197
```

Figure 5:First model test

In any case, by visualizing 3D models obtained by the inferences on the training set and the validation set the overfitting problem is clear.

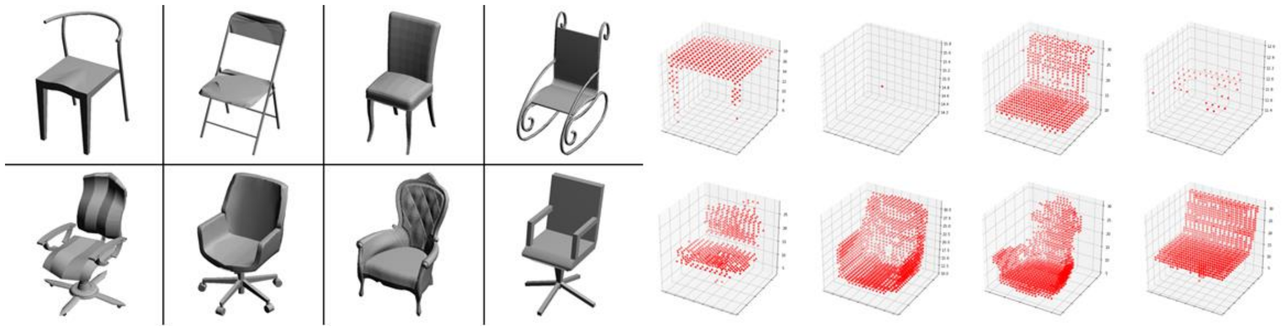


Figure 6:Reconstruction first model on the training set

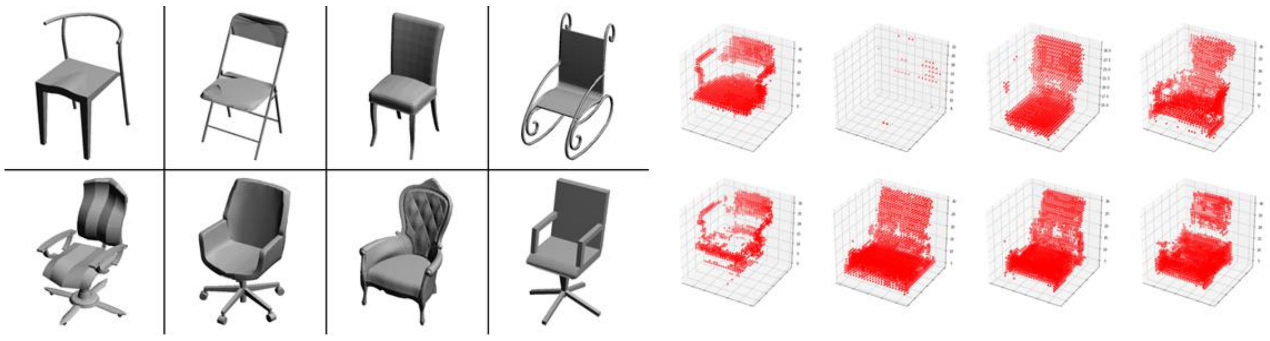


Figure 7:Reconstruction first model on the validation set

The visualization emphasizes the model difficulty in maintaining certain shapes, that results in a lot of sparse point clouds: it is probably because of **catastrophic forgetting**.
The issue is clearly the limited number of training examples: the discriminator easily **memorizes** the training set and keeps a high accuracy on real data.

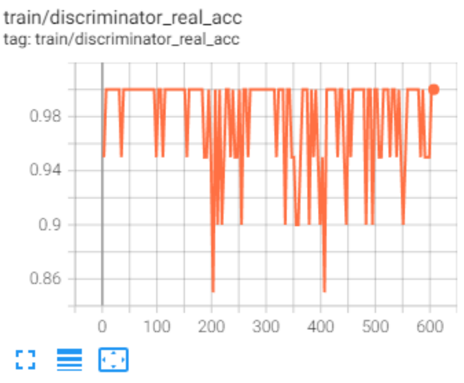


Figure 8:Real data accuracy first model

Different learning rate values for the discriminator led to similar results.

These changes to the model were then applied:

- The encoder was simplified by reducing the kernel sizes of all layers to 5, with stride 2 and padding 2.
- **Dropout** was added (in particular, Dropout3d for volumetric convolutional layers) in all layers from the first to the second to last one: in the generator the probability was set to 0.2 in the first layer and 0.1 in the successive ones, while in the discriminator it was set to 0.6 in the first layer and to 0.4 in the successive ones.
- The KL loss weight is set 3, the adversarial loss weight is set to 1 and the reconstruction loss weight is also set to 1. Higher weights for the KL loss led to too simple latent representations that did not capture all shapes of chairs, while low weights for the reconstruction loss led to **mode collapse**.
- A term for the reconstructed objects was added to the adversarial loss; those should be considered real by the discriminator.
  $$L_{adv} = \frac{1}{2}\left[\log\left(1 - D\big(G(z)\big)\right) + \log\left(1 - D\left(G\big(E(y_i)\big)\right)\right)\right]$$
- **Soft labels** were added for evaluating the adversarial loss of the discriminator and the generator: instead of considering the “real” and “fake” labels as plain 1 and 0, these are sampled from uniform distributions respectively in $[0.7,1.2]$ and $[0, 0.3]$.

After these changes, the reconstruction loss on the validation set is slightly lower.

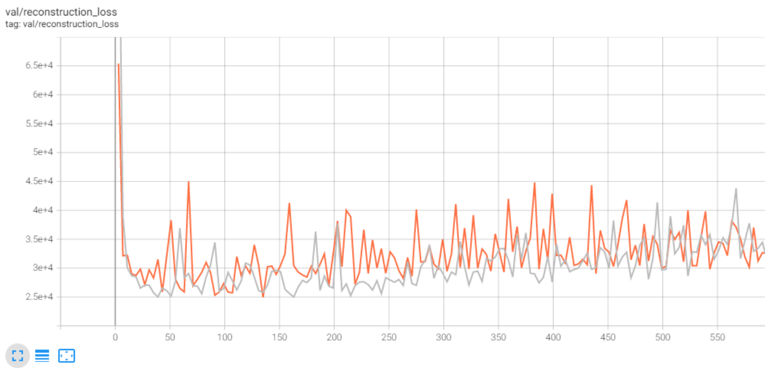


Figure 9:Reconstruction loss second model

The discriminator memorizes a bit less of the training data.

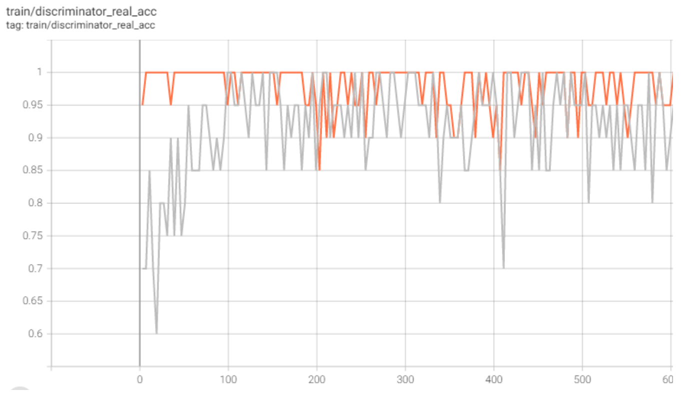


Figure 10:Real data accuracy second model

Here is a comparison of reconstructions on the validation set between the first and the second model.

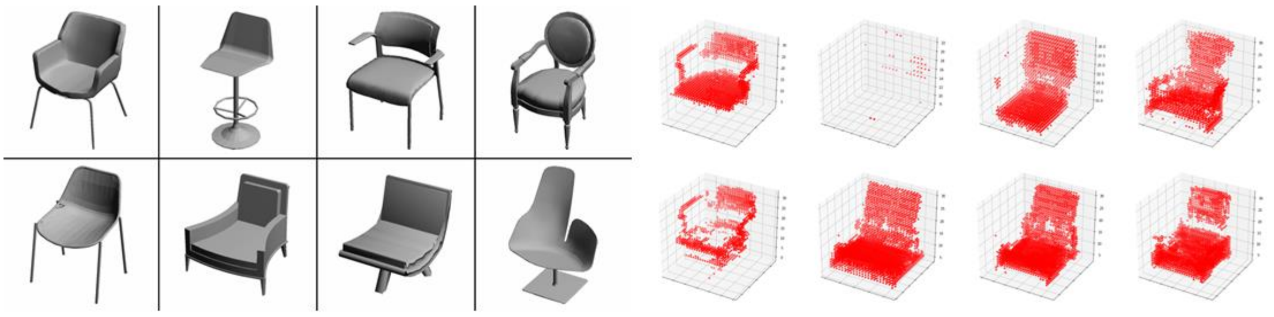


Figure 11:First model reconstruction

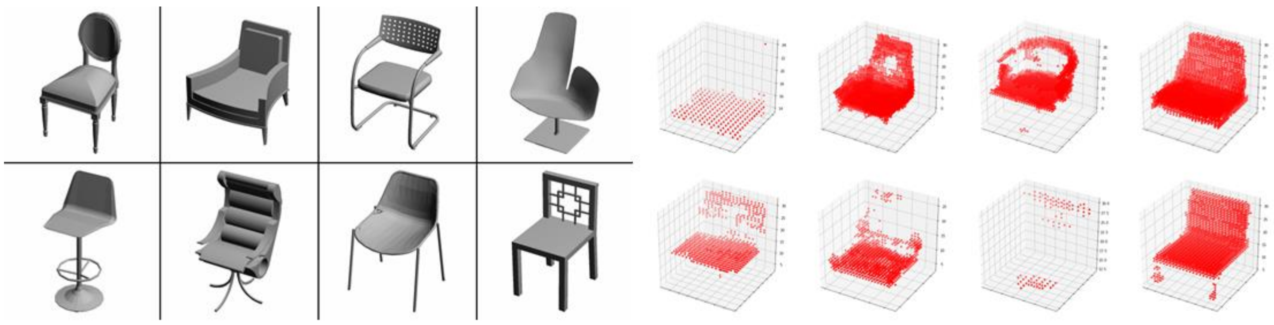


Figure 12:Second model reconstruction

The reconstructions seem slightly better, even though forgetting and the inaccuracy of shapes not in the training set persists.

2.6.4 Model for grids 32x32x32

The training for the model on voxelized chairs in grids $32\ x\ 32\ x\ 32$ was done for 340 epochs, with a batch size of 20, a learning rate of 0.0025 for the generator, of 0.00001 for the discriminator and of 0.0001 for the encoder; the momentum was set to 0.5.
The resulting loss functions graphs are:

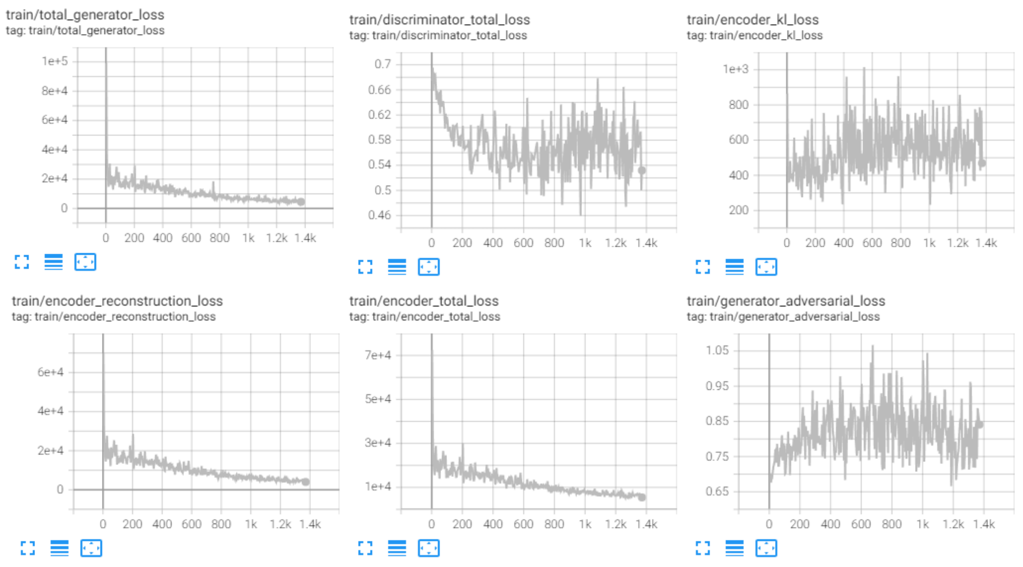


Figure 13:Graphs chair model 32x32x32

The discriminator loss is initially decreasing, then it oscillates around 0.46 and 0.66, while the generator one oscillates between 0.65 and 1.05.

The accuracy of the discriminator is:

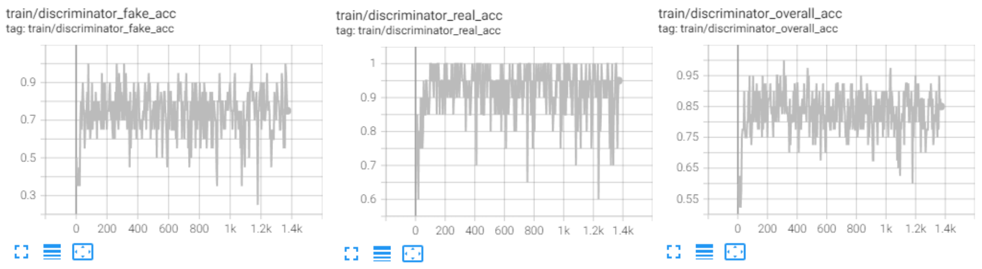


Figure 14:Discriminator accuracy chair model 32x32x32

The accuracy on real data remains pretty high with a lot of spikes, while the one on synthetic data oscillates a lot between 0.4 and 1.0.

As seen before, the reconstruction loss is not decreasing on the validation set.

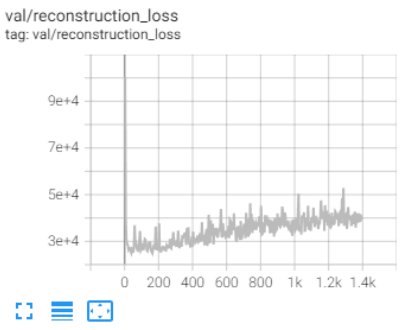


Figure 15:Reconstruction loss chair model 32x32x32

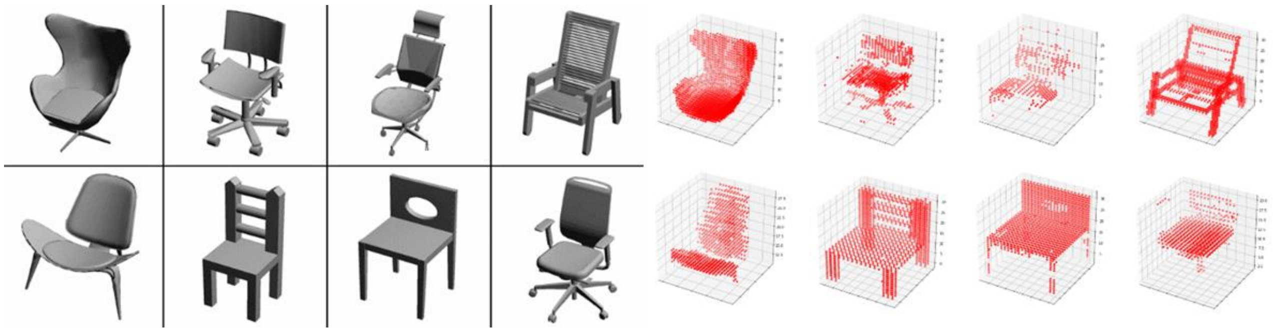


Figure 16:Chair reconstruction 32x32x32 on the training set

To see the model evolution during training, random samples and validation set reconstructions are saved every 20 epochs; with the projection of the reconstructions on the validation set the **evolution of the model** can be seen, since the input is fixed.

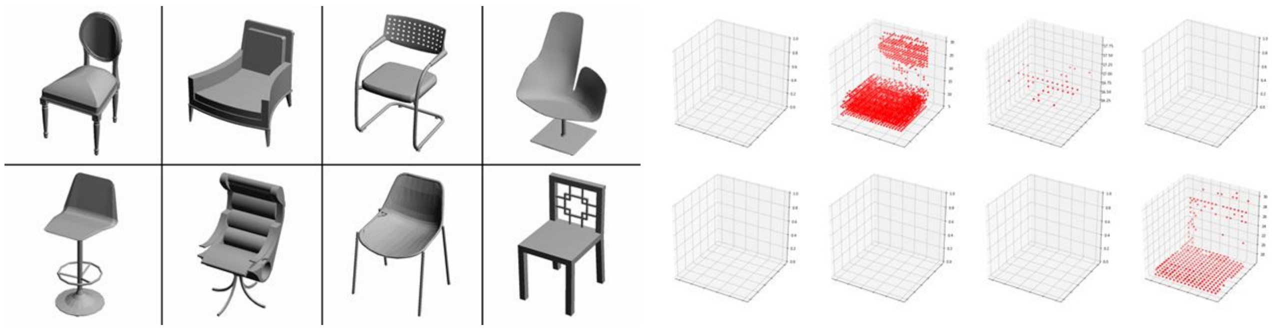


Figure 17:Reconstruction Iteration 243

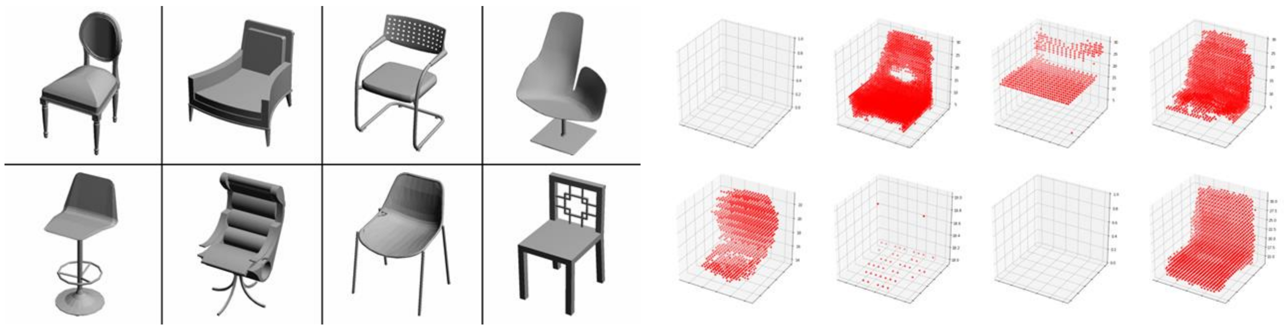


Figure 18:Reconstruction Iteration 403

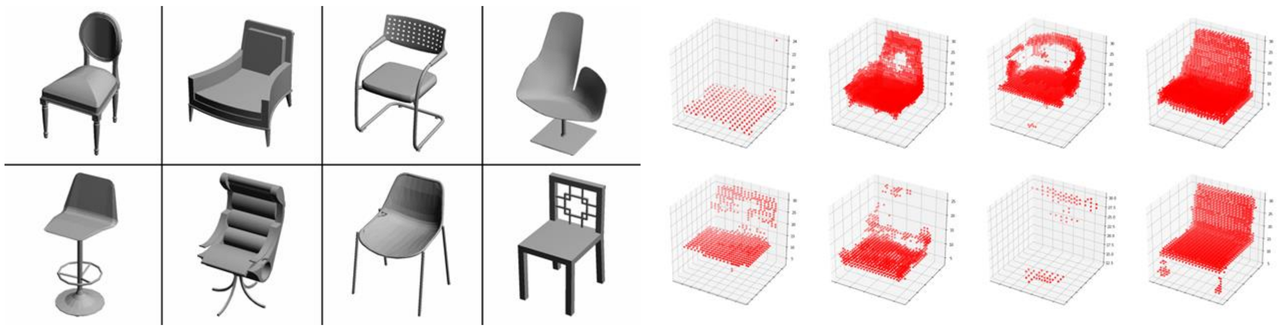


Figure 19:Reconstruction Iteration 643

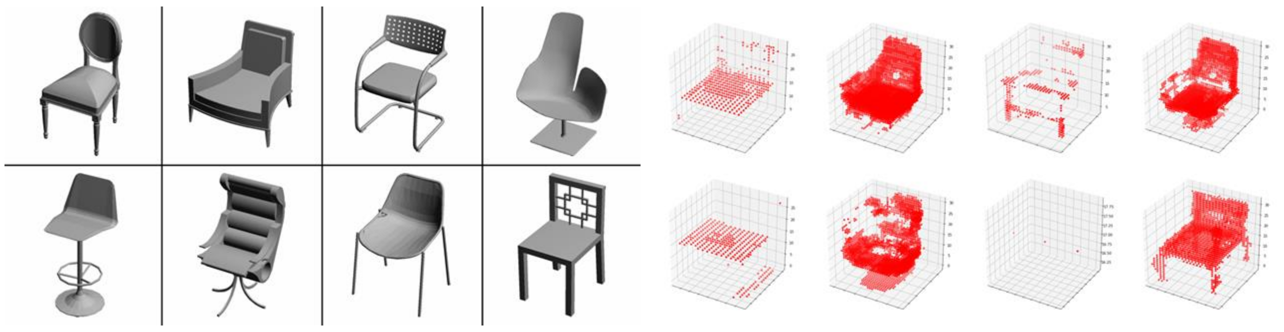


Figure 20:Reconstruction Iteration 883

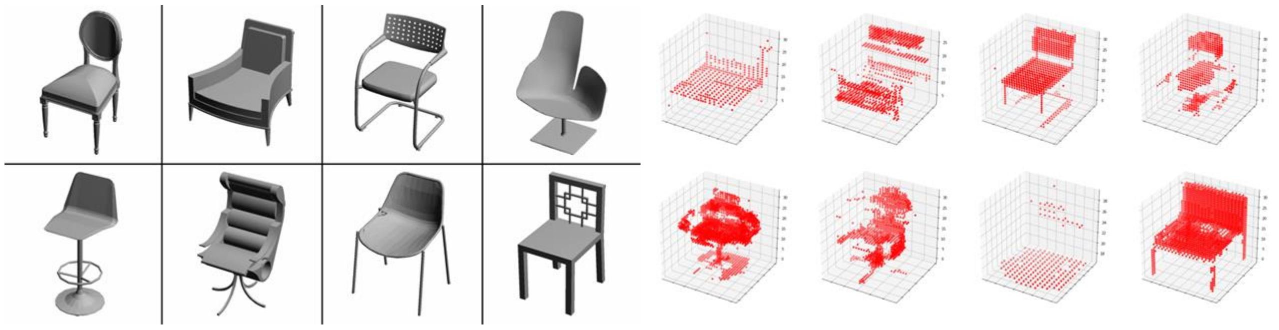


Figure 21:Reconstruction Iteration 1123

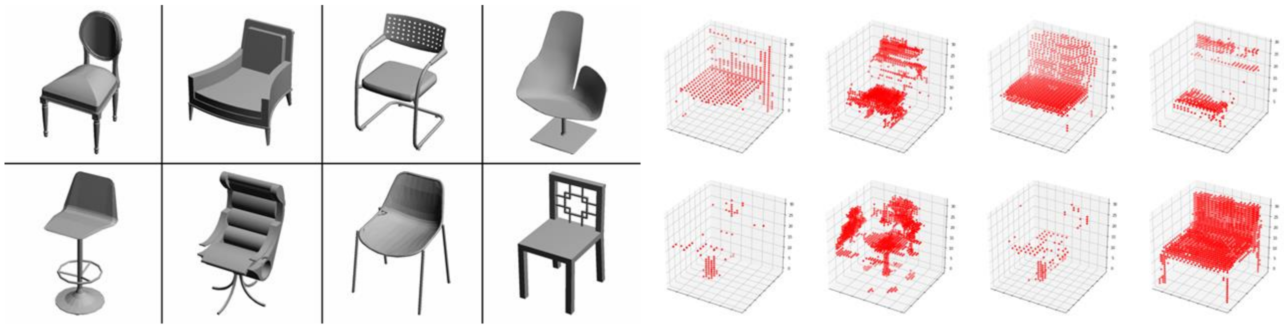


Figure 22:Reconstruction Iteration 1203

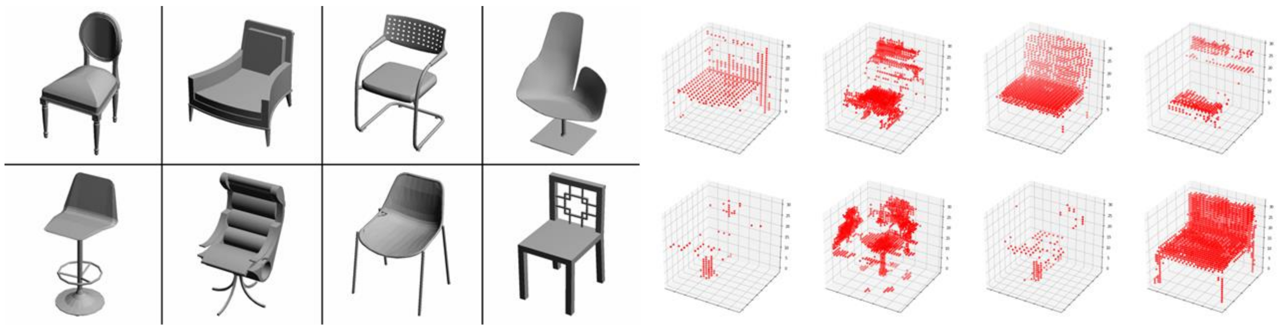


Figure 23:Reconstruction Iteration 1363

The overfitting effects are still clear, though there seems to be slightly less catastrophic forgetting.

2.6.5 SSIM

The need for a proper comparison metric between 3D models was clear from the results obtained with the first model.

For images, the **Structural Similarity** or SSIM, is often used. SSIM is computed at different scales obtained from gaussian filters. Given two windows x and y, SSIM is defined as:

$$SSIM(x,y) = \frac{(2\mu_x\mu_y + C_1)(2\sigma_{xy} + C_2)}{(\mu_x^2 + \mu_y^2 + C_1)(\sigma_x^2 + \sigma_y^2 + C_2)}$$

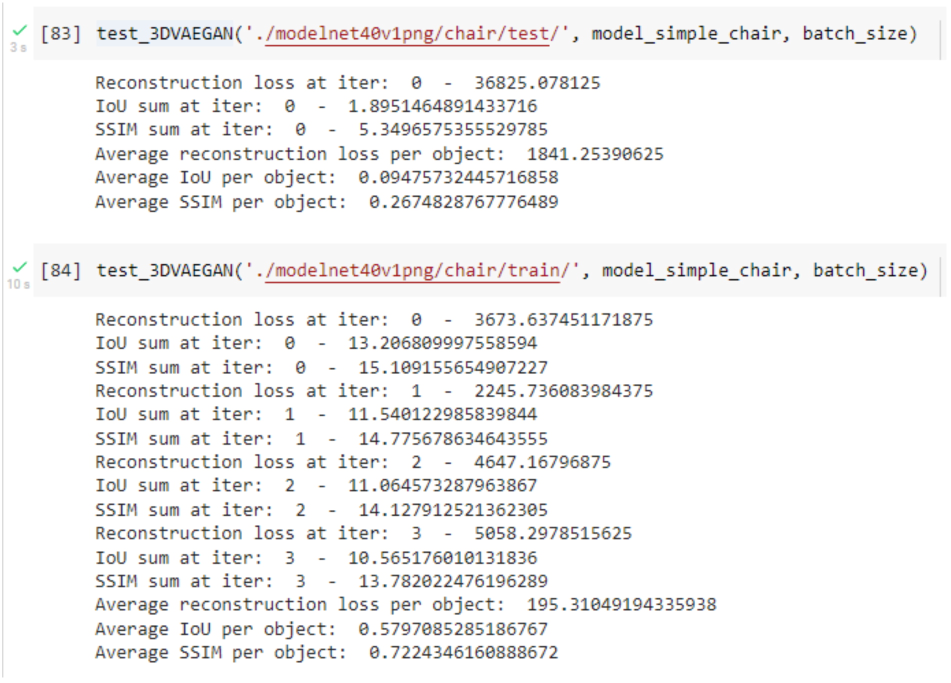
In which:

- μ_x is the mean of the window x;
- μ_y is the mean of the window y;
- σ_x^2 is the variance of the window x;
- σ_y^2 is the variance of the window y;
- σ_{xy} is the covariance of the two windows;
- $C_1 = (k_1 L)^2$ and $C_2 = (k_2 L^2)$ are needed to stabilize division in case of very small values at the denominator, with $k_1 = 0.01$ and $k_2 = 0.03$;
- L is the dynamic range of the pixel values.

The SSIM is meant to quantify the **structural information loss** between the two images, by evaluating the interdependencies between spatially close pixels. The range of admissible values is [0,1] and a high value implies more structural similarity.

SSIM can be applied to three-dimensional voxelized models simply by using three-dimensional gaussian filters.

The average SSIM on the validation set is 0.267 and on the training set 0.722. There is still a relevant difference, but it is a bit better than the difference with the IoU and reconstruction loss values.



```
[83] test_3DVAEGAN('./modelnet40v1png/chair/test/', model_simple_chair, batch_size)

     Reconstruction loss at iter:  0  -  36825.078125
     IoU sum at iter:  0  -  1.8951464891433716
     SSIM sum at iter:  0  -  5.3496575355529785
     Average reconstruction loss per object:  1841.25390625
     Average IoU per object:  0.09475732445716858
     Average SSIM per object:  0.2674828767776489
```

```
[84] test_3DVAEGAN('./modelnet40v1png/chair/train/', model_simple_chair, batch_size)

     Reconstruction loss at iter:  0  -  3673.637451171875
     IoU sum at iter:  0  -  13.206809997558594
     SSIM sum at iter:  0  -  15.109155654907227
     Reconstruction loss at iter:  1  -  2245.736083984375
     IoU sum at iter:  1  -  11.540122985839844
     SSIM sum at iter:  1  -  14.775678634643555
     Reconstruction loss at iter:  2  -  4647.16796875
     IoU sum at iter:  2  -  11.064573287963867
     SSIM sum at iter:  2  -  14.127912521362305
     Reconstruction loss at iter:  3  -  5058.2978515625
     IoU sum at iter:  3  -  10.565176010131836
     SSIM sum at iter:  3  -  13.782022476196289
     Average reconstruction loss per object:  195.31049194335938
     Average IoU per object:  0.5797085285186767
     Average SSIM per object:  0.7224346160888672
```

Figure 24:Test SSIM chair model 32x32x32

The next, obvious step was to expand the loss functions of the generator and the encoder to try and maximise the SSIM:

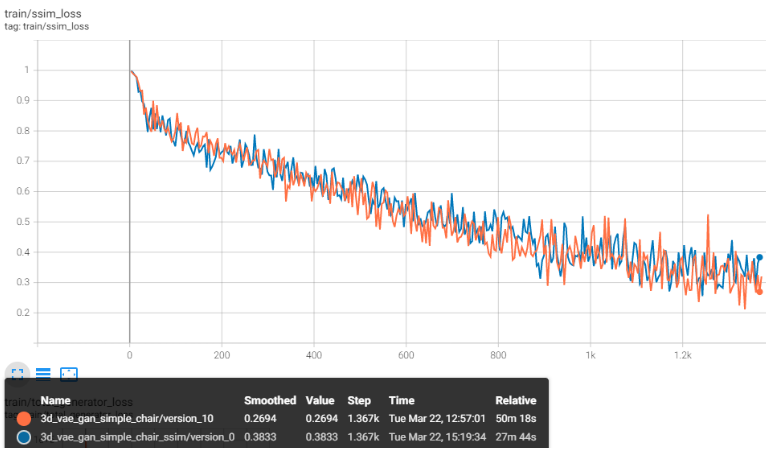
$L_{SSIM} = 1 - \mathrm{SSIM}(G(E(y_i)), x)$

$L_G = \alpha_1 L_{adv} + \alpha_2 L_{recon} + \alpha_3 \mathrm{L}_{\mathrm{SSIM}}$

$L_E = \alpha_4 L_{KL} + \alpha_2 L_{recon} + \alpha_3 L_{SSIM}$

The weight for the SSIM loss was set to 1 and the model was trained for 340 epochs with the same parameters.

In the end, all graphs followed the same behaviour. Even the SSIM loss on training set and validation set was decrementing the same way of the model in which it is not optimized.



	Name	Smoothed	Value	Step	Time	Relative
●	3d_vae_gan_simple_chair/version_10	0.2694	0.2694	1.367k	Tue Mar 22, 12:57:01	50m 18s
●	3d_vae_gan_simple_chair_ssim/version_0	0.3833	0.3833	1.367k	Tue Mar 22, 15:19:34	27m 44s

Figure 25: SSIM loss with and without optimization

We suppose that, with the chosen model sizes, the SSIM loss optimization is similar to the reconstruction loss optimization: it might be more useful in case of bigger and more complex voxelized models. The SSIM is still handy as an evaluation metric.

2.6.6 Data augmentation and data quality

As a last resort to reduce overfitting, data augmentation was attempted: we apply **random affine transformations** on both the input images and the 3D models.

Images are randomly rotated of $\theta_{img} = U(-8,8)$ degrees and randomly translated of $t_{img} = U(-0.05 * imgSize, 0.05 * imgSize)$ pixels.

Models are randomly rotated with respect to the y axis of $\theta_{models} = U(-8,8)$ degrees and translated randomly with respect to each axis of $t_{models} = U(-2,2)$ voxels.

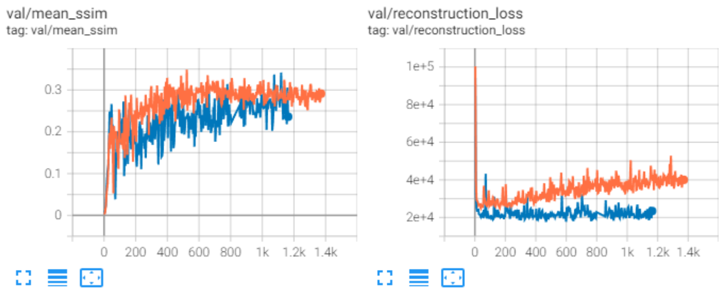


Figure 26:Validation graphs comparison with data augmentation

The reconstruction loss is a bit lower and the SSIM follows a similar trend of the previous model, even though it oscillates more.

Visually, the results are clearly worse.

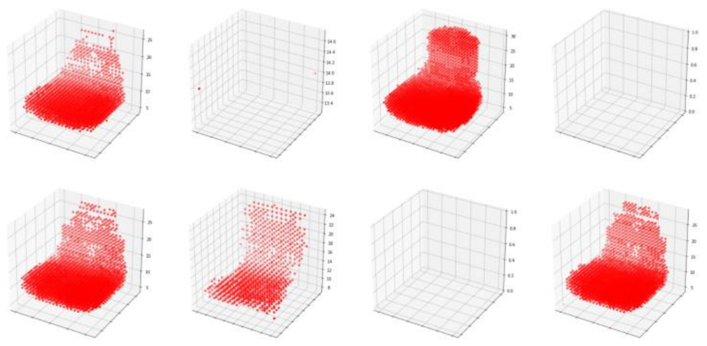


Figure 27:Reconstruction model with data augmentation

The model tends to focus on single shapes.
In the end, the main issue seems to be a **bad** 32 x 32 x 32 **voxelization** of the available chair data.
For example, these are 8 models from the training set:

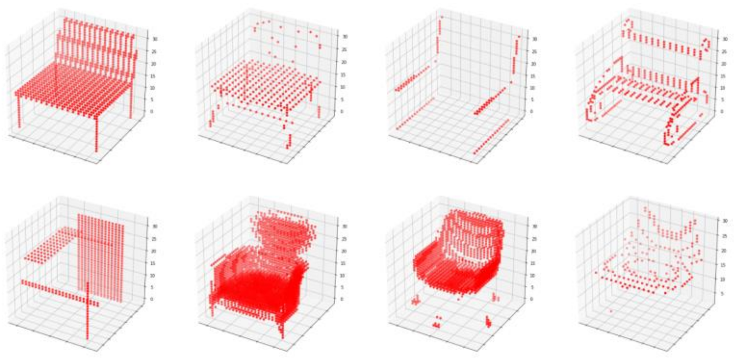


Figure 28: 32x32x32 chairs example from the training set

It can clearly be seen that the third, the fifth and the eighth models are low quality. This might depend on the software used for the conversion, binvox, but most likely on the inability to reproduce an object like a chair with a small grid.

2.6.7 Model for chairs with 64x64x64 grids

To verify the hypotheses of the previous paragraph, the network was trained on the voxelized 64 x 64 x 64 chairs for 200 epochs, with the same parameters, but without the Dropout layers and batch size was set to 10 because the training now requires a lot more memory.

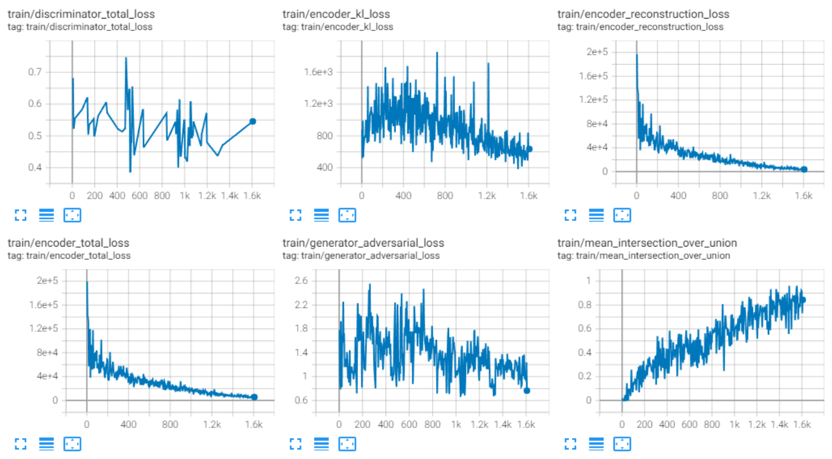


Figure 29:Training graphs model with 64x64x64 chairs

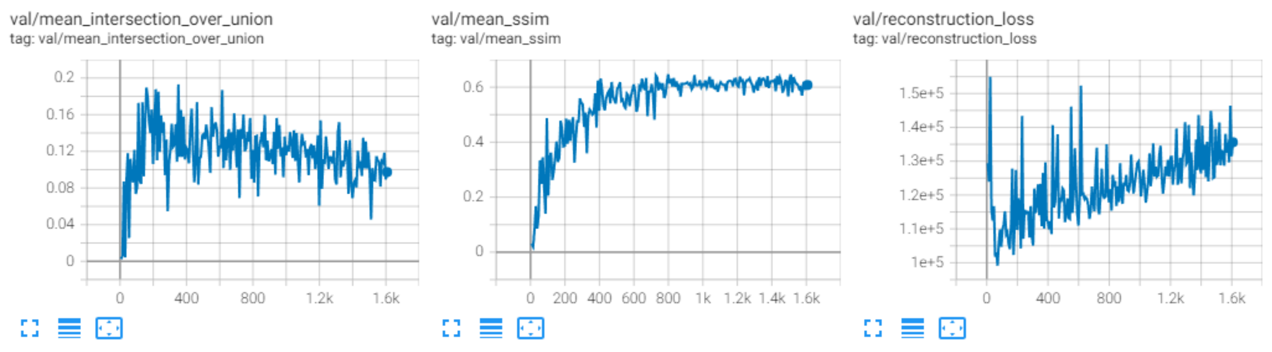


Figure 30: Validation graphs model with 64x64x64 chairs

The model generalizes much better: it has much **higher similarity** values, even with an increasing reconstruction loss.
There are better visual results on the validation set, even though overfitting and forgetting are still a bit present.

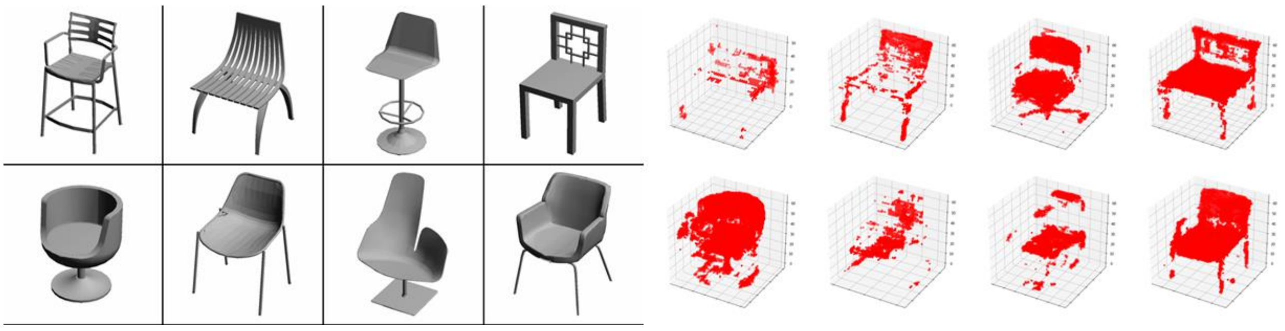


Figure 31:Validation set reconstruction for 64x64x64 chairs

Dropout is still necessary since the discriminator keeps a high accuracy on the training data.

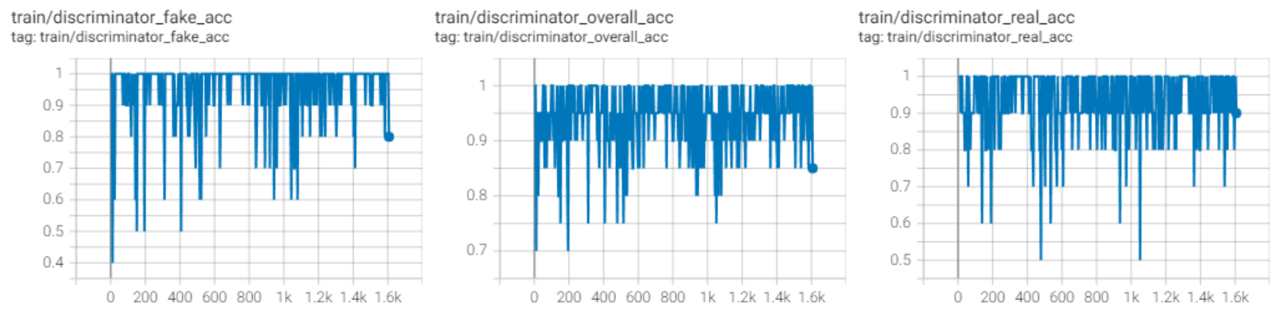


Figure 32:Discriminator accuracy chair 64x64x64 model

After all these experiments it was concluded that, other than adapting the network in case of limited data, the **grid size** choice is really important and it should be done based on the object type and the level of detail required. A wrong choice can lead to bad data and results.
There is a clear **trade-off**, because by increasing grid size we increase execution time and required memory for both training and inference. A more generalizable architecture, made by the Facebook team and not based on voxelization, will be discussed in Chapter 4.

2.6.8 Conditional VAE-GAN

The models tested so far allow us to generate objects from a single class: if objects from different classes are used as training data, generation would be ambiguous and not specialized and to arbitrarily generate models from a certain class one should manually analyse the latent vector features.

Conditional GANs are an extension of GANs [5] in which both the generator and the discriminator take the **class label** as input and make their inferences based on it.

In the conditional 3D-VAE-GAN architecture built for this project, the conditional generator and the conditional discriminator do the following steps:

1. They take the class label as input.
2. They do an **embedding** with the corresponding PyTorch module.
3. The embedding is given to a **linear layer**, which transforms it into a tensor of size equal to the input size of the base networks: 200 for the generator and equal to the number of voxels in the grid for the discriminator.
4. The output of the linear layer is multiplied **pointwise** with the actual input tensor.
5. The resulting tensor passes through the original network layers.

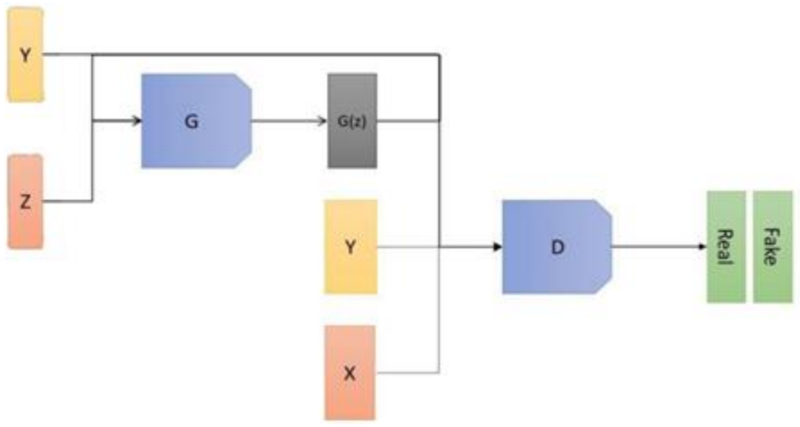


Figure 33:Conditional GAN structure

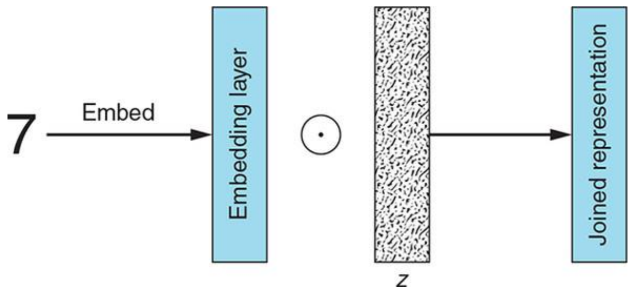


Figure 34:Label fusion with pointwise product of the tensors

The pointwise product makes it possible to exploit class label information while **preserving the dimensionality** of the input.

Conditioning the encoder with the class label information is not necessary: the encoder can still just generate the latent vector z by taking an image or multiple images as input. The conditional generator will then transform z with the point-wise product in a new vector that can properly generate a model of the corresponding class.

2.6.9 Conditional base model

The conditional model with the base encoder (that takes a single image as input) was trained for 80 epochs on the following classes: chair, lamp and car. The grid size for voxelized models is $64 \ x \ 64 \ x \ 64$.

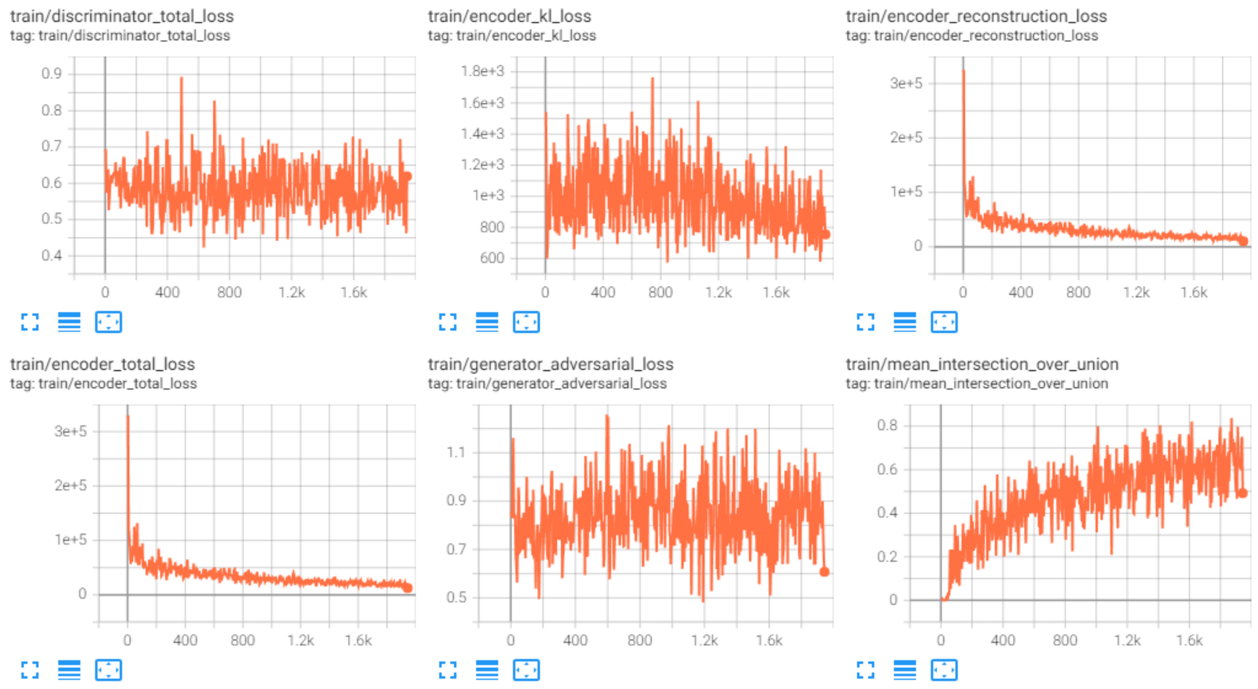


Figure 35:Conditional VAE-GAN with base encoder training graphs

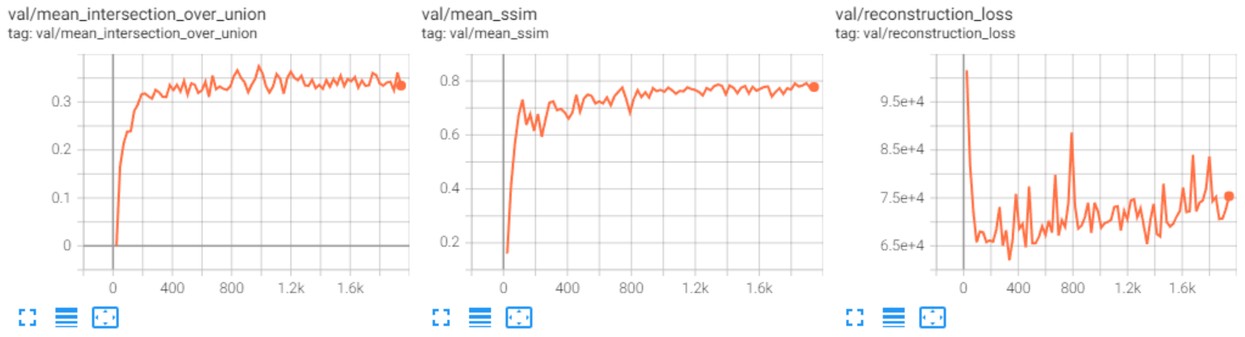


Figure 36: Conditional VAE-GAN with base encoder validation graphs

The SSIM and reconstruction loss on the validation set seem to improve a lot in the first few iterations: the model converges more quickly because it learns more useful features for reconstruction from many more objects, even if they are from different classes.

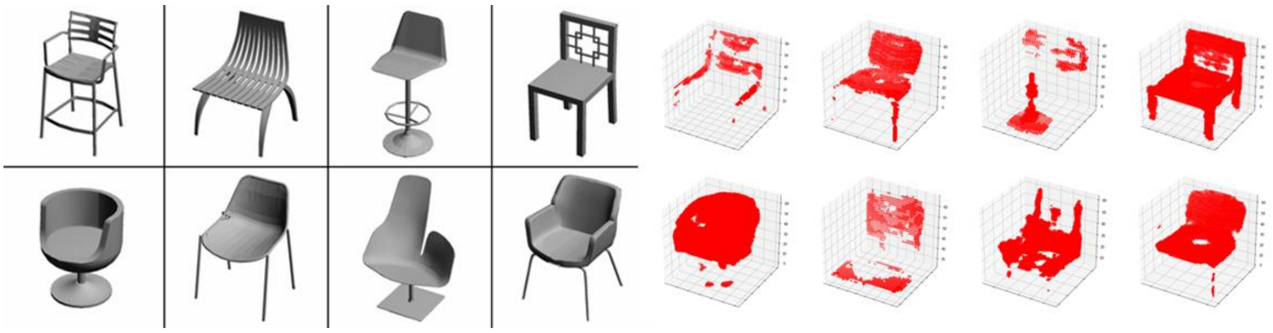


Figure 37: Validation set reconstruction conditional VAE-GAN with base encoder

Regarding **random model generation**, it seems that it requires more epochs compared to the reconstruction task: in comparison, the three-dimensional model generated by the non-conditional VAE-GAN seem to be of higher quality.

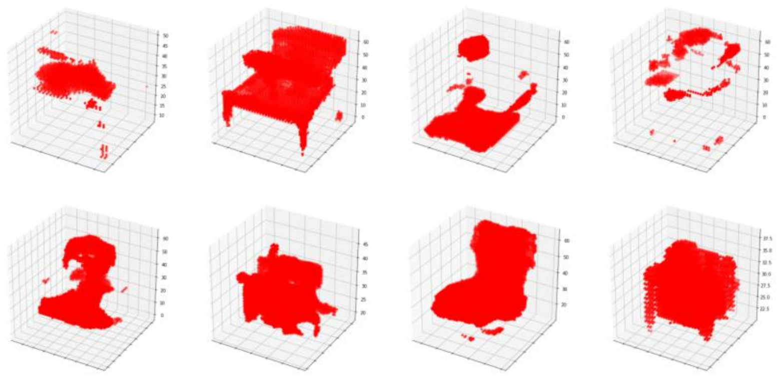


Figure 38:Random models conditional VAE-GAN (80 epochs)

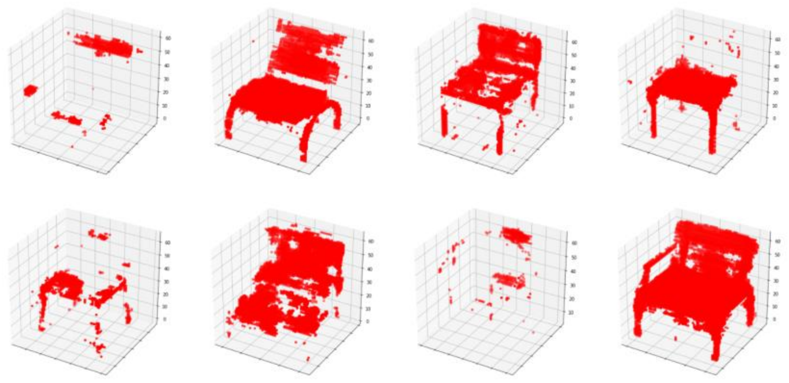
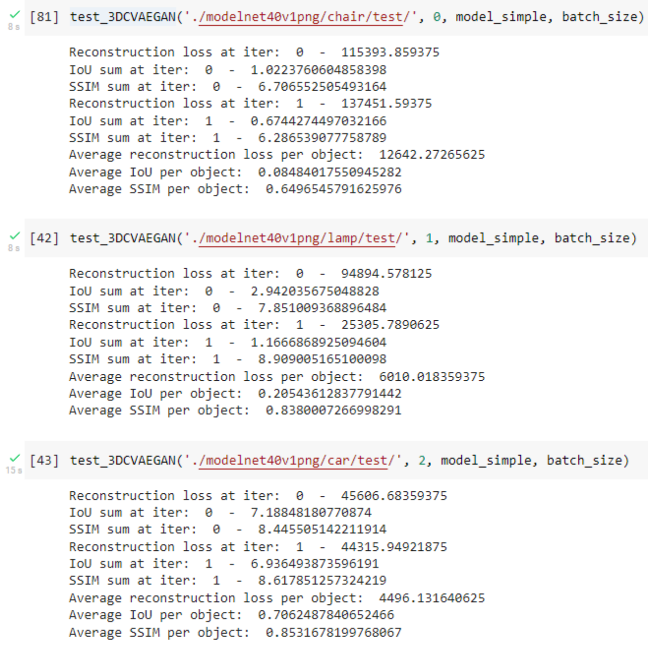


Figure 39:Random models non-conditional VAE-GAN (200 epochs)



```
[81] test_3DCVAEGAN('./modelnet40v1png/chair/test/', 0, model_simple, batch_size)

     Reconstruction loss at iter:  0  -  115393.859375
     IoU sum at iter:  0  -  1.0223760604858398
     SSIM sum at iter:  0  -  6.706552505493164
     Reconstruction loss at iter:  1  -  137451.59375
     IoU sum at iter:  1  -  0.6744274497032166
     SSIM sum at iter:  1  -  6.286539077758789
     Average reconstruction loss per object:  12642.27265625
     Average IoU per object:  0.08484017550945282
     Average SSIM per object:  0.6496545791625976

[42] test_3DCVAEGAN('./modelnet40v1png/lamp/test/', 1, model_simple, batch_size)

     Reconstruction loss at iter:  0  -  94894.578125
     IoU sum at iter:  0  -  2.942035675048828
     SSIM sum at iter:  0  -  7.851009368896484
     Reconstruction loss at iter:  1  -  25305.7890625
     IoU sum at iter:  1  -  1.1666868925094604
     SSIM sum at iter:  1  -  8.909005165100098
     Average reconstruction loss per object:  6010.018359375
     Average IoU per object:  0.20543612837791442
     Average SSIM per object:  0.8380007266998291

[43] test_3DCVAEGAN('./modelnet40v1png/car/test/', 2, model_simple, batch_size)

     Reconstruction loss at iter:  0  -  45606.68359375
     IoU sum at iter:  0  -  7.18848180770874
     SSIM sum at iter:  0  -  8.445505142211914
     Reconstruction loss at iter:  1  -  44315.94921875
     IoU sum at iter:  1  -  6.936493873596191
     SSIM sum at iter:  1  -  8.617851257324219
     Average reconstruction loss per object:  4496.131640625
     Average IoU per object:  0.7062487840652466
     Average SSIM per object:  0.8531678199768067
```

Figure 40:Base conditional model results

The model seems to do pretty good in reconstructing lamps and cars and has the same SSIM for chairs of the non-conditional model.

2.6.10 Multiview 3D-VAE-GAN

The Multiview models use the same generator and discriminator, but employ a different encoder with the same base structure that is applied to every available image to generate different latent vectors, and a specialized layer combines all representations into a single vector.

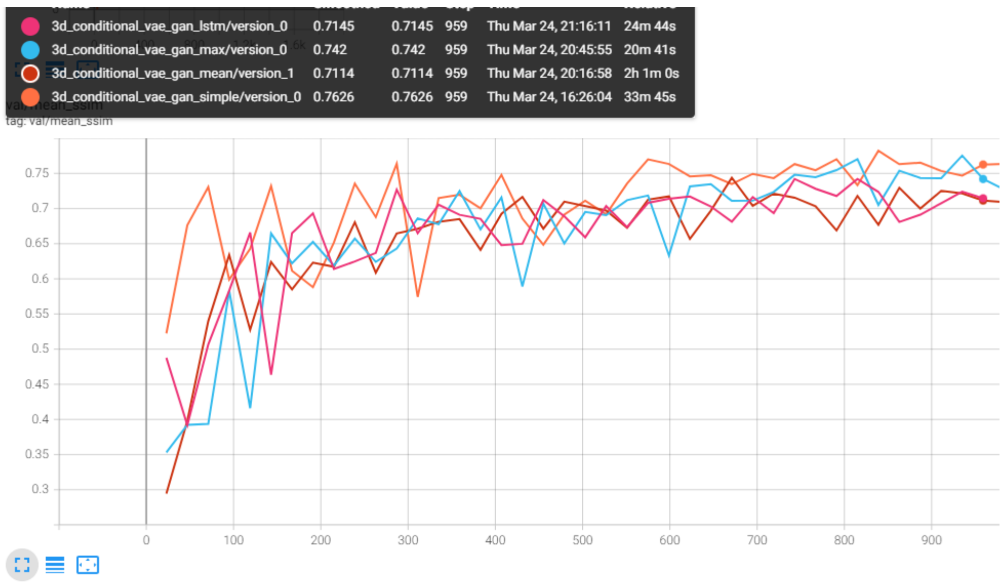
By using the same layers with the same weights to elaborate each image reduces the computational requirements for training and inference, but also allows us to obtain a **“view-independent”** representation that focuses in extracting relevant key points from each image: thus, overfitting is reduced and this representation can be used independently to estimate latent vectors from single images.

The methods tested to combine representations were the **average** operator (Multiview Mean), the **max** operator (Multiview Max) and a LSTM network (Multiview LSTM).

2.6.10.1 Conditional Multiview models

The Multiview models with conditional VAE-GAN were trained with images from 5 points of view for each object and on the following classes: chair, lamp and cars. All models are voxelized in a $64\ x\ 64\ x\ 64$ grid.

After 40 epochs, there seemed to be few differences on the SSIM values on the validation data.



●	3d_conditional_vae_gan_lstm/version_0	0.7145	0.7145	959	Thu Mar 24, 21:16:11	24m 44s
●	3d_conditional_vae_gan_max/version_0	0.742	0.742	959	Thu Mar 24, 20:45:55	20m 41s
●	3d_conditional_vae_gan_mean/version_1	0.7114	0.7114	959	Thu Mar 24, 20:16:58	2h 1m 0s
●	3d_conditional_vae_gan_simple/version_0	0.7626	0.7626	959	Thu Mar 24, 16:26:04	33m 45s

Figure 41:SSIM validation data Multiview conditional models

We hypothesize that, given the presence of lamps and chairs whose corresponding **voxelized models** are **simple** and **symmetric**, the network does not particularly use the added information from multiple points of view to learn more useful features: this information might be beneficial to reconstruct chairs, since they are a bit more complicated.

2.6.11 Multiview models for chairs

To verify the effectiveness of Multiview encoders, three non-conditional models were trained for 160 epochs with the three different types of Multiview encoders, all on voxelized $64\ x\ 64\ x\ 64$ chairs. The three models were then compared to the one with the base encoder.
The training graphs seem pretty similar between the four models, except the model with the max operator (corresponding to the red graph) for which the values seem to improve more slowly.

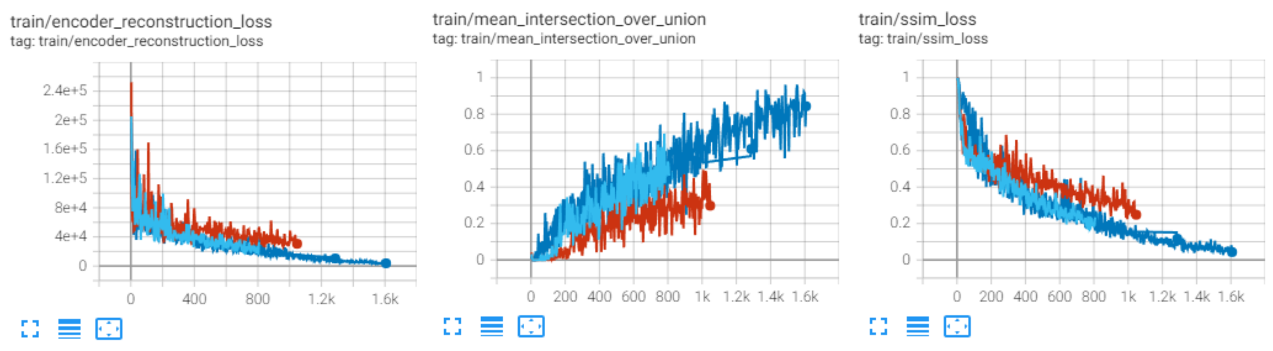
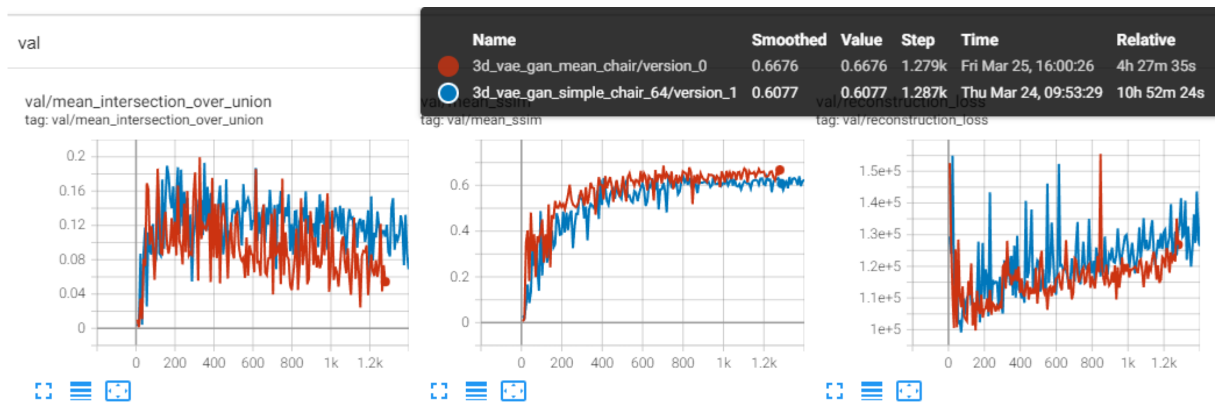


Figure 42:Training graphs comparison: different encoders

The results on validation data were slightly different for each model: those differences highlighted the pros and cons of choosing a particular encoder. Each model will then be compared with the model with the base encoder.
Multiview Mean comparison
The model with the encoder with the average operator seemed to obtain the best results, keeping a higher SSIM value and a lower reconstruction loss.



	Name	Smoothed	Value	Step	Time	Relative
🟥	3d_vae_gan_mean_chair/version_0	0.6676	0.6676	1.279k	Fri Mar 25, 16:00:26	4h 27m 35s
🟦	3d_vae_gan_simple_chair_64/version_1	0.6077	0.6077	1.287k	Thu Mar 24, 09:53:29	10h 52m 24s

Figure 43:Validation data comparison for Multiview Mean

The IoU values are lower, implying that the generated reconstructions are more sparse and less equal pointwise.

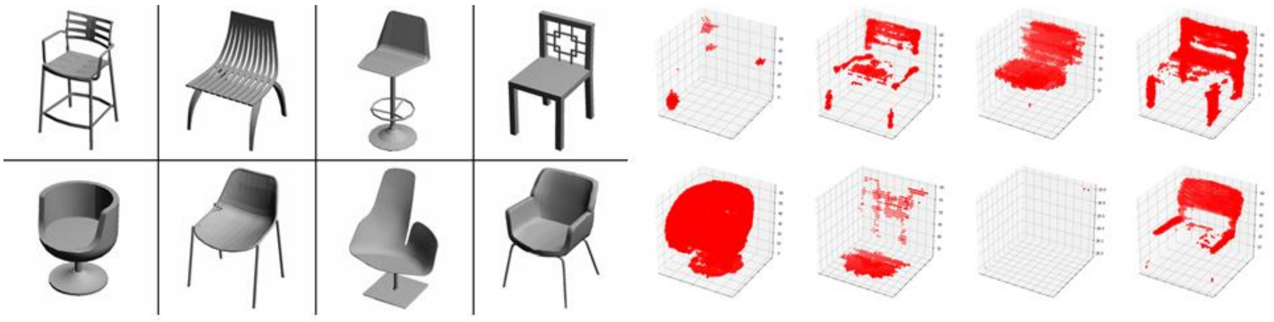


Figure 44:Validation set reconstruction base encoder (967 iterations)

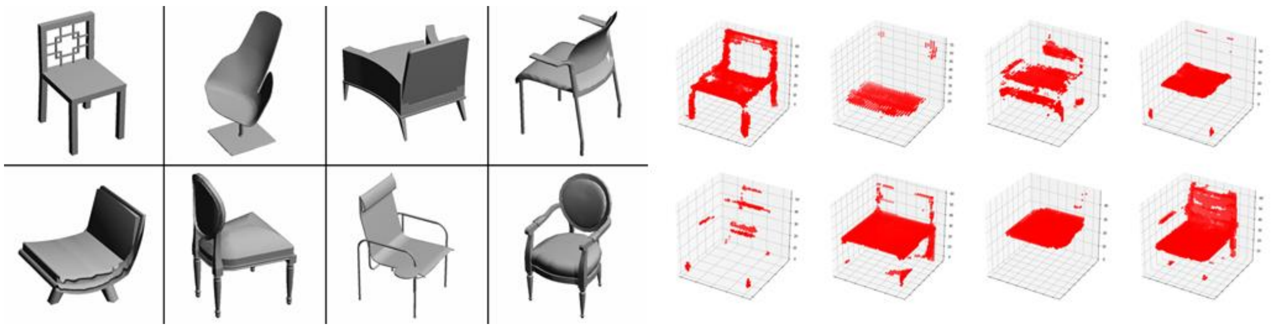
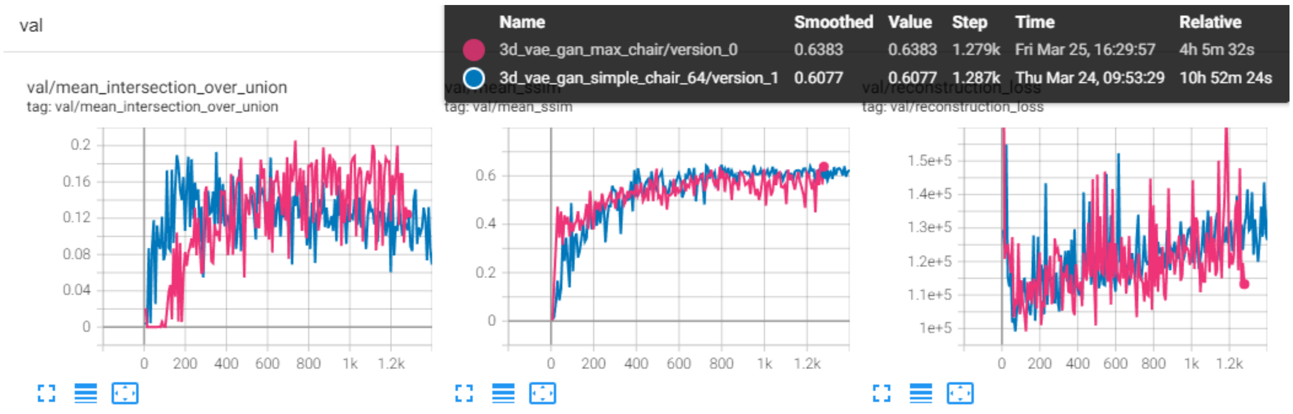


Figure 45:Vvalidation set reconstruction Multiview Mean (967 iterations)

Sparsity of certain three-dimensional models persists, even though the Multiview Mean model seems to make fewer mistakes: the average of latent representations from single images allows the model to compensate the lack of information in certain vectors with information from other vectors.

The average operator to combine vectors is handy to compensate ambiguity in interpreting details from different points of view.

Multiview Max comparison



	Name	Smoothed	Value	Step	Time	Relative
●	3d_vae_gan_max_chair/version_0	0.6383	0.6383	1.279k	Fri Mar 25, 16:29:57	4h 5m 32s
●	3d_vae_gan_simple_chair_64/version_1	0.6077	0.6077	1.287k	Thu Mar 24, 09:53:29	10h 52m 24s

Figure 46:Validation data comparison for Multiview Max

The model with the encoder with the max operator had no apparent improvements: SSIM values are similar, the reconstruction loss oscillates even more, while IoU values tend to increase.

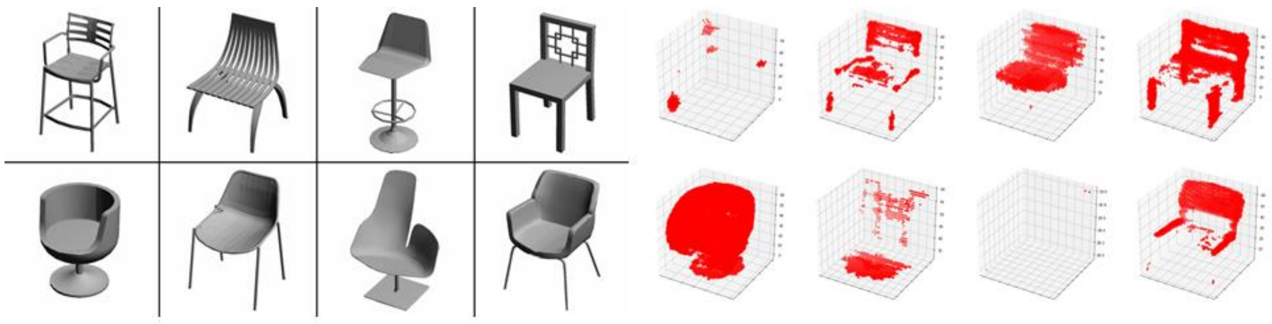


Figure 47:Validation set reconstruction with base encoder (967 iterations)

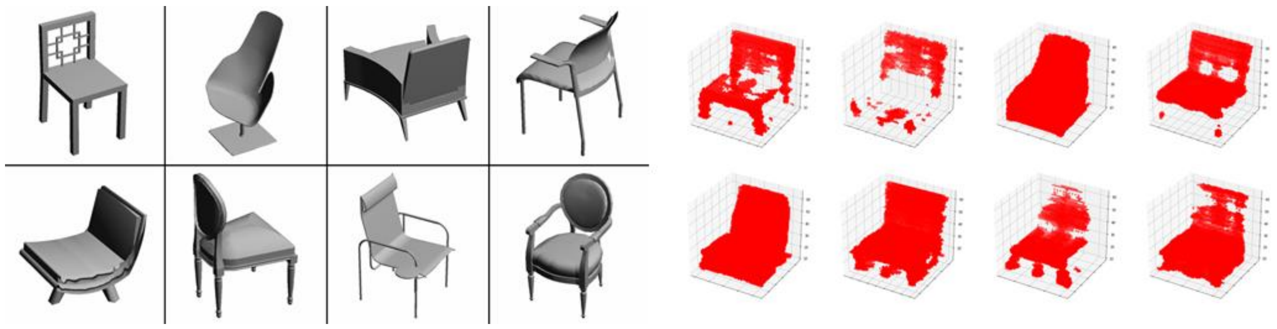
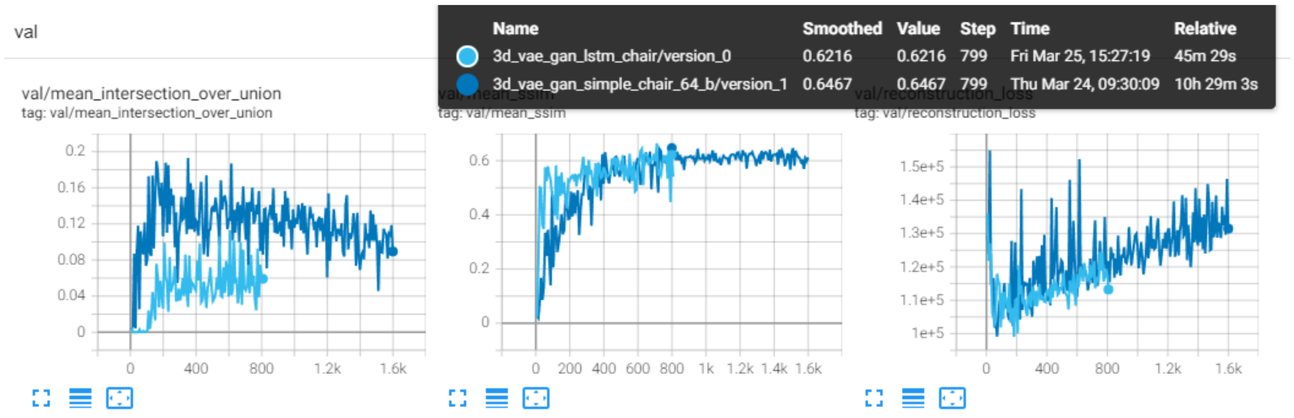


Figure 48:Validation set reconstruction with Multiview Max (967 iterations)

An incentive of using a Multiview Max model is to have latent vectors with higher values and, consequently, **denser models**.
It is also noteworthy that, with bigger datasets of more complicated objects, computing the maximum component-wise can help piece together relevant information from each point of view: the average might attenuate important values in each latent vector and the corresponding detail might be less present in the final three-dimensional model.

Multiview LSTM comparison



	Name	Smoothed	Value	Step	Time	Relative
●	3d_vae_gan_lstm_chair/version_0	0.6216	0.6216	799	Fri Mar 25, 15:27:19	45m 29s
●	3d_vae_gan_simple_chair_64_b/version_1	0.6467	0.6467	799	Thu Mar 24, 09:30:09	10h 29m 3s

Figure 49:Validation data comparison for Multiview LSTM

By using a LSTM to combine latent representations sequentially, the SSIM converges more quickly, though the final value remains roughly the same.
The reconstruction loss oscillates less and the IoU values are lower.

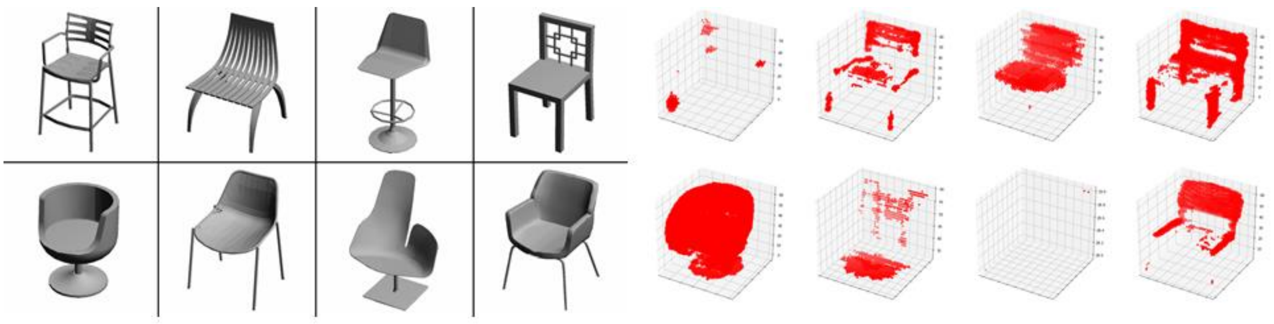


Figure 50:Validation set reconstruction with base encoder (967 iterations)

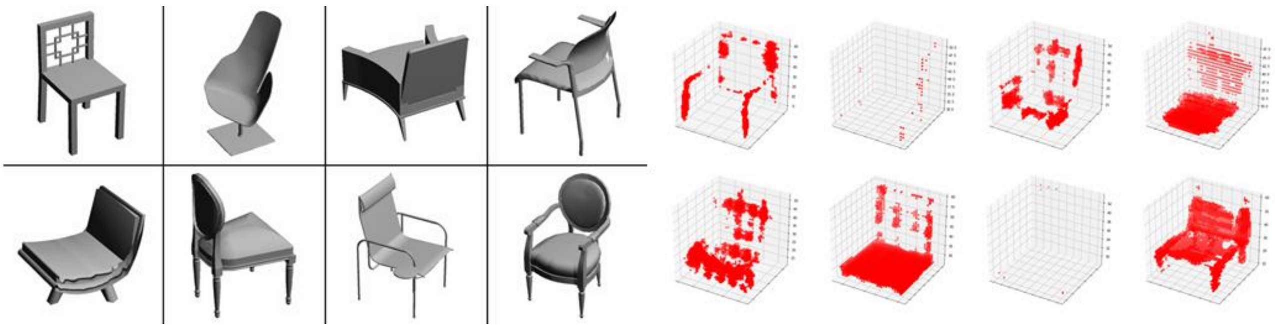
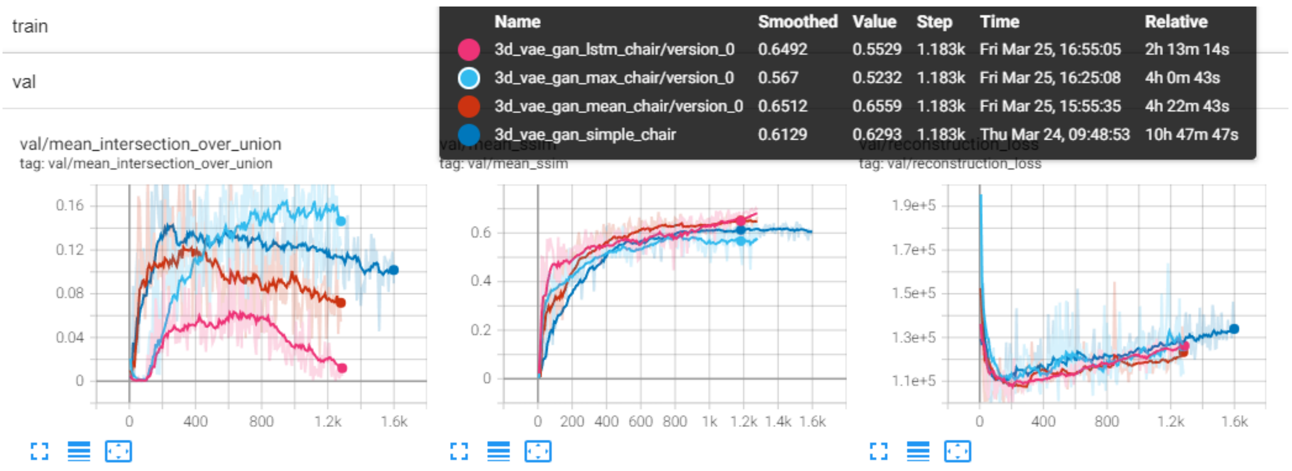


Figure 51:Validation set reconstruction with Multiview LSTM (816 iterations)

The Multiview LSTM model gave the sparsest models from the validation data. The sequential processing of the latent vectors of each image resulted in weaker final vectors and so a less generalizable encoder.

A sequential architecture might be handy in case of a lot of images and if those images are in the **right sequence**: in photogrammetry, images are first ordered to match key points.



	Name	Smoothed	Value	Step	Time	Relative
●	3d_vae_gan_lstm_chair/version_0	0.6492	0.5529	1.183k	Fri Mar 25, 16:55:05	2h 13m 14s
●	3d_vae_gan_max_chair/version_0	0.567	0.5232	1.183k	Fri Mar 25, 16:25:08	4h 0m 43s
●	3d_vae_gan_mean_chair/version_0	0.6512	0.6559	1.183k	Fri Mar 25, 15:55:35	4h 22m 43s
●	3d_vae_gan_simple_chair	0.6129	0.6293	1.183k	Thu Mar 24, 09:48:53	10h 47m 47s

Figure 52: Final comparison between the four models (smoothed graphs)

3. DIB-R

3.1 Introduction

The 3D-GAN architecture and its derivatives try to learn object characteristics from the training dataset to build an approximate shape inside a limited size grid. More elaborate architectures should be able to learn how to predict other components such as texture maps and illuminations, so they should be built by taking **rendering** into account. Rendering is a process in which bidimensional images are created from 3D models that interact with light.
NVIDIA's DIB-R [6] consists in a **differentiable renderer** that can compute gradients from all pixels of the image: this can then be used along with neural networks to predict three-dimensional shapes, textures, lightning from a single image.

3.2 The DIB-R renderer

DIB-R is a renderer based on **rasterization** which computes background pixels by aggregating global information and foreground pixels by interpolating vertex attributes.

A typical rendering process consists in at least three shaders: vertex shader, rasterization shader and fragment shader.

The **vertex shader** projects vertices on the bidimensional image plane and therefore is a directly differentiable operation; the **fragment shader** puts together local properties to establish pixel colours, usually by differentiable operations.

The **rasterization shader** is not usually differentiable, so the DIB-R renderer applies a z-buffering test for each foreground pixel to find the closest covering face. The I_i value for a certain pixel is computed with a barycentric interpolation of that face's vertex attributes. Considering three vertices v_0, v_1, v_2 with three attributes u_0, u_1, u_2, the pixel value is computed as such:

$$I_i = w_0 u_0 + w_1 u_1 + w_2 u_2$$

In which the weights w_0, w_1, w_2 are computed based on the vertex and pixel position with differentiable functions.

$$w_k = \Omega_k(v_0, v_1, v_2, p_i)$$

However, with just these operations background pixels not covered by faces will have a null gradient and will not be able to propagate useful information. To solve this problem, a **stochastic sampling of faces** is done by assigning them with respect to this probability:

$$A_{i'}^{j} = e^{-\frac{d(p_{i'}, f_j)}{\delta}}$$

In which $d(p_{i'}, f_j)$ is the distance of pixel $p_{i'}$ from face f_j in the bidimensional projected space and δ is an hyperparameter.

The probabilities are then combined to obtain the total influence of all n faces on a certain pixel:

$$A_{i'} = 1 - \prod_{j=1}^{n}\left(1 - A_{i'}^{j}\right)$$

3.3 3D model prediction

The neural architecture associated with DIB-R makes it possible to obtain both the three-dimensional shape of the object and a **texture map** that can be applied to it.

This architecture uses different loss functions for different objectives. The loss functions for geometry and colour are the IoU loss, an L1 loss on the image, a smoothing loss and a Laplacian loss.

The **IoU loss** is needed to predict the silhouette $\tilde{S}$:

$$L_{IOU}(\theta) = \mathbb{E}_{I\sim p_{data}(I)}\left[1 - \frac{\left\|S * \tilde{S}\right\|_1}{\left\|S + \tilde{S} - S * \tilde{S}\right\|_1}\right]$$

In which S contains the alpha channel values of the image I and the "$*$" operator consists in a pointwise product.

The **image loss** for colour is:

$$L_{col}(\theta) = \mathbb{E}_{I\sim p_{data}(I)}\left[\left\|I - \tilde{I}\right\|_1\right]$$

The **smoothing loss** is:

$$L_{sm} = \sum_{e_i \in E} (\cos(\theta_i) + 1)^2$$

In which E is the set of all edges and θ_i is the angle between two faces that share edge e_i. This loss function makes it possible to have similar normals between adjacent faces.

The **Laplacian loss** is:

$$L_{lap} = \left(\delta_v - \frac{1}{\|N(v)\|}\sum_{v' \in N(v)} \delta_{v'}\right)^2$$

In which $N(v)$ is the set of vertices closest to the vertex v and δ_v is the predicted movement of vertex v. This loss function, if optimized, makes close vertices move all at once, thus avoiding dispersion and resulting in more consistent shapes.

The final loss L_1 for colour and shape is a linear combination:

$$L_1 = L_{IOU} + \lambda_{col}L_{col} + \lambda_{sm}L_{sm} + \lambda_{lap}L_{lap}$$

To avoid overfitting, during optimization, other than comparing the original image with the predicted, rendered image, a comparison with a render from a random point of view is made.

Losses for the adversarial framework are added to the L_1 loss, in particular those for the **Wasserstein GAN** framework [7] with gradient penalty: the adversarial loss L_{adv}, the gradient penalty L_{gp} and a perceptive loss L_{percep}.

The adversarial loss is defined as the difference between the values from the discriminator D (often called "critic" in the WGAN-GP framework) on the ground truth image and the predicted, rendered images.

$$L_{adv}(\theta, \phi) = \mathbb{E}_{I \sim p_{data}(I)}[D(I;\phi) - D(\tilde{I};\phi)]$$

The gradient penalty is defined as:

$$L_{gp}(\phi) = \mathbb{E}_{I \sim p_{data}(I)}\left[\left(\left\|\nabla_{\tilde{I}} D(\tilde{I};\phi)\right\|_2 - 1\right)^2\right]$$

The perceptive loss is defined as:

$$L_{percep}(\phi) = \mathbb{E}_{I \sim p_{data}(I)}\left[\sum_{i=1}^{M_V}\left(\frac{1}{N_i^V}\left\|V^i(I) - V^i(\tilde{I})\right\|_1\right) + \sum_{i=1}^{M_D}\left(\frac{1}{N_i^D}\left\|D^i(I;\phi) - D^i(\tilde{I};\phi)\right\|_1\right)\right]$$

In which V^i is the i-th layer of a VGG network V pre-trained with N_i^V elements, D^i the i-th layer of the discriminator D with N_i^D elements; M_V is the number of layers in the network V and M_D the number of layers of the discriminator D.

The final objective function is:

$$\theta^*, \phi^* = \arg_\theta \min\left(\arg_\phi \max\left(\lambda_{adv} L_{adv} - \lambda_{gp} L_{gp}\right) + \lambda_{percep} L_{percep} + L_1\right)$$

3.4 Model and texture prediction

The obtained network after training is capable of obtaining realistic shapes, though the textures are not of significant quality. To mitigate the problem, a **second discriminator** D_2 for texture maps is introduced and the textures from the original network are used as ground truth.

The overall network consists in a **shape generator** G_1, a **texture generator** G_2, an **image discriminator** D_1 and a **texture discriminator** D_2.

In the original network, G_1 takes the image as input, while in this extended network it takes a random noise vector of length 128 as input, which goes through 8 fully connected layers to generate predictions for the vertex positions; the fifth layer activations are then used to predict textures and they are concatenated with more random noise of size 128, and then passed to the texture generator G_2 as input.

The texture generator G_2 is made of 7 convolutional layers and returns a texture map with $256x256$ resolution.

The discriminators D_1 and D_2 have 4 convolutional layers, but the image discriminator D_1 takes $64x64$ images as input, while the texture one D_2 takes $256x256$ images as input.

3.5 Demo

DIB-R and the associated machine learning tools are available in the Kaolin NVIDIA package for Python. Various other tools are available on the Omniverse application to create training records, though they are not usable with Google Colab because of the lack of graphical interface.

Each model is represented with different points of view and for each point of view there are three files: a PNG image, a JSON file for metadata and an NPY file for semantics.

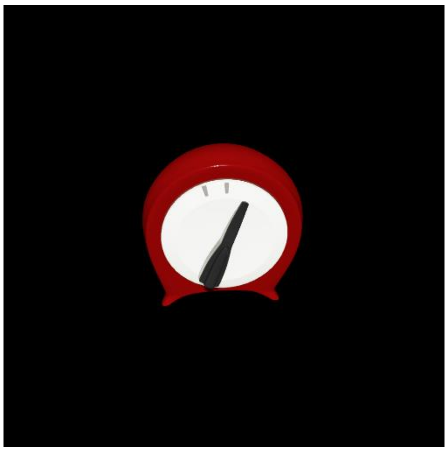


Figure 53:Model example

Meshes and vertices are transformed into tensors and during training various Kaolin functions are called, such as the DIB-R renderer and a function for texture mapping; finally, loss functions are computed for optimization.

Training the models with the example data given by NVIDIA for 40 epochs gives the following results:

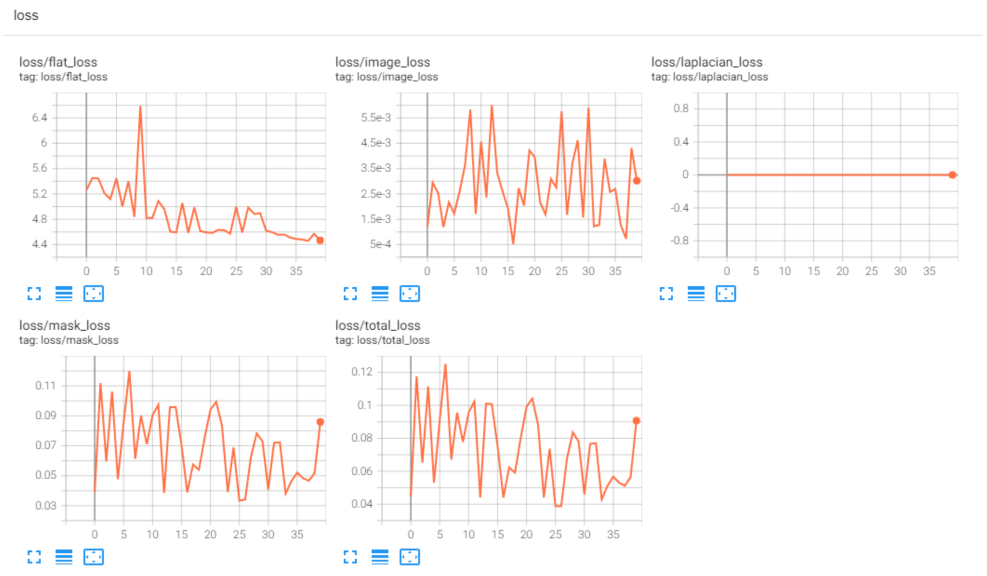


Figure 54:Loss graphs DIB-R

All losses seem to oscillate between small values, except the flat loss decreasing and the null Laplacian loss, probably because the model is already pre-trained.

With the help of Kaolin functions, it is also possible to obtain a visualization with matplotlib of the DIB-R renders and the obtained texture maps.

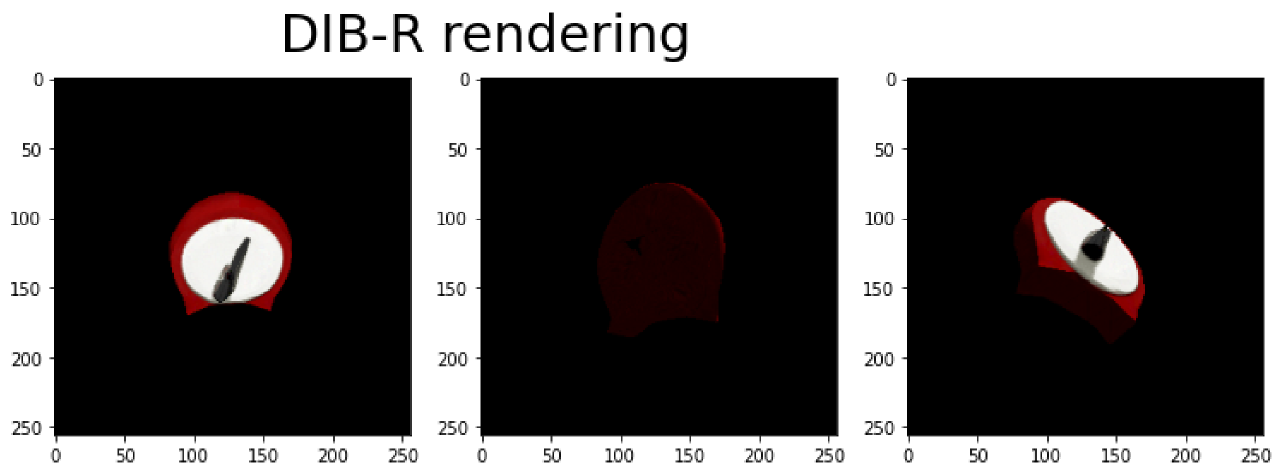


Figure 55:Visualization example Kaolin DIB-R

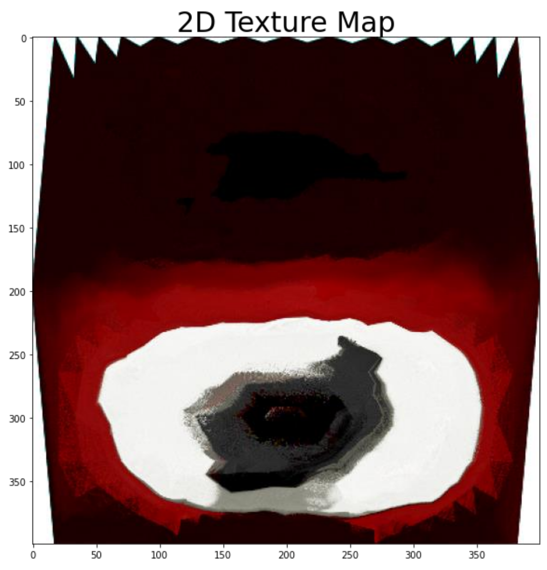


Figure 56:Texture map visualization example

4. PIFuHD

4.1 Introduction

Facebook's PIFuHD [8] is an architecture that can create a 3D model of a person with clothes from a single, high-resolution image. PIFuHD is not GAN-based, though there are GAN extensions for textures.
The low quality of models generated by the previous architectures was caused by the low resolution of input images: PIFuHD mitigates the problem with a multi-level architecture.

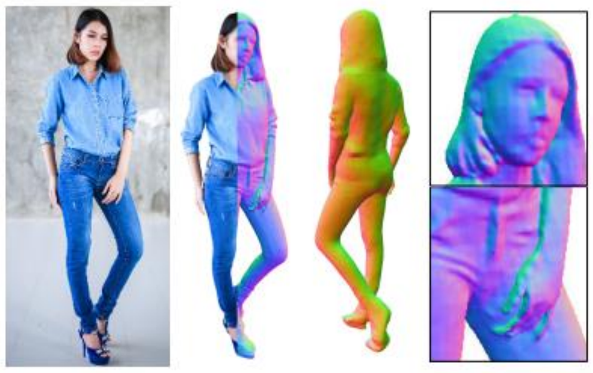


Figure 57 PIFuHD example inference 1

4.2 Pixel-Aligned Implicit Function

PIFuHD is based on the concept of PIFu, Pixel-Aligned Implicit Function [9]. A PIFu is an efficient **representation** for **three-dimensional surfaces**: by representing a surface as a level set of a function, there is no need to memorize it explicitly. A PIFu function consists in an encoder g composed of convolutional layers and a continuous implicit function f represented by a multi-layer perceptron; the surface is defined as a level set of:
$f(F(x), z(X)) = s: s \in \mathbb{R}$
For a three-dimensional point X, $x = \pi(X)$ is its bidimensional projection, $z(X)$ is the probability value in the camera coordinate system and $F(x) = g\big(I(x)\big)$ is the image features in x.
In particular, we want to model a function like this one:
$f(X, I) = \begin{cases} 1\ if\ X\ is\ inside\ the\ mesh\ surface \\ \qquad\qquad 0 \end{cases}$
In which X is a position in the three-dimensional camera space and I is an RGB image.
This approach allows us to save memory by not explicitly memorizing 3D volumes.
The function is modelled through a neural network, which consists of an embedding of a feature space $\Phi(x, I)$ of the image x obtained through a projection of the model with a function $\pi(X)$ and a function g that takes the features and the depth Z through the projection ray as input.
$f(X, I) = g(\Phi(x, I), Z)$
During inference, the three-dimensional space is uniformly sampled to establish the occupancy of the model and the final iso-surface is extracted with the marching cubes algorithm.

4.3 Multi-level PIFu

The PIFuHD model consists in **two PIFu levels**: a coarse level that takes the downscaled $512x512$ image as input and produces $128x128$ features as output, and a fine level that adds details by taking the original, $1024x1024$ image and producing $512x512$ features.

Both the coarse and the fine level have the same, base PIFu structure, though they also take the predicted **normal maps** of front F and back B at the corresponding resolutions:

$f^L(X) = g^L(\Phi^L(x_L, I_L, F_L, B_L), Z)$

$f^H(X) = g^H(\Phi^{\mathrm{H}}(x_H, I_H, F_H, B_H), \Omega(X))$

The fine level also has other differences compared to the coarse level: the receptive field does not cover the whole image, but a **random sliding window** is used; there is also an **embedding** function Ω, that takes intermediate activations from the coarse network as input.

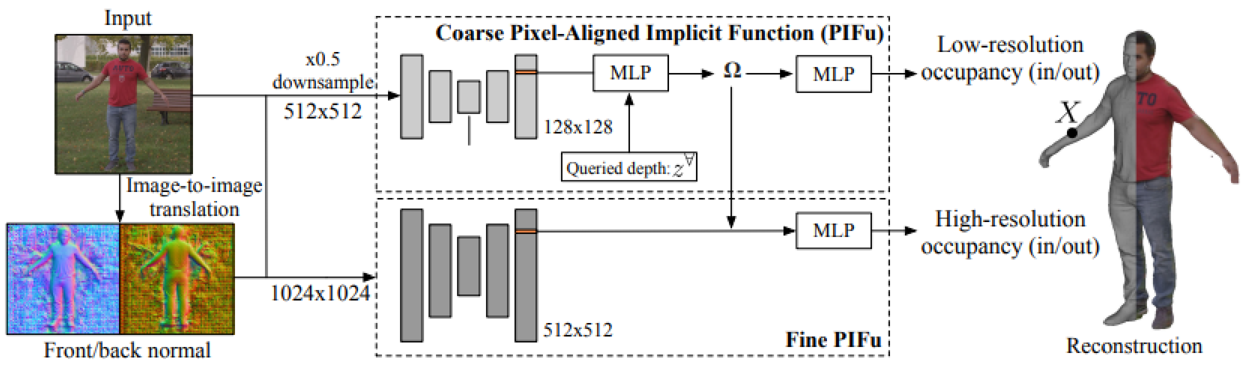

Input
x0.5
downsample
512x512
Coarse Pixel-Aligned Implicit Function (PIFu)
MLP
Ω
MLP
Low-resolution occupancy (in/out)
X
128x128
Queried depth: z
Image-to-image translation
MLP
High-resolution occupancy (in/out)
1024x1024
512x512
Fine PIFu
Front/back normal
Reconstruction

Figure 58: PIFuHD Architecture

4.4 Back inference

The input photos given to the network usually represent a person's front: generating the back with a lack of information is not trivial.

To solve this issue, normal map of both front and back are predicted with a pix2pixHD network [10] and they are given as input to both the PIFuHD coarse and fine levels.

4.5 Loss functions

The main loss function used is the Binary Cross Entropy loss:

$$L_o = \sum_{X \in S} \lambda f^*(X) \log f^{\{L,H\}}(X) + (1-\lambda)(1 - f^*(X)) \log\left(1 - f^{\{L,H\}}(X)\right)$$

In which S represents the samples of the current batch, λ indicates the number of points outside the surface in S, the function $f^*(X)$ indicates the actual occupation in the point X and $f^{\{L,H\}}$ indicates the two PIFu functions.

The three-dimensional points sampling of the training models is done with both uniform sampling and **importance sampling** with respect to the surface, with gaussian perturbations on the surface points taken uniformly.

The loss function to predict both front and back normal is the following:

$$L_N = L_{VGG} + \lambda_{l1} L_{L1}$$

In which L_{VGG} is the perceptive loss previously seen for DIB-R, while L_{L1} is an $L1$ distance between predicted and ground truth normals.

4.6 Demo

The AI Facebook team published the code on GitHub along with a demo that can be used with Google Colab. The demo allows us to download the model, upload a photo, pre-process it and pass it through the network.
The visualization can be done with utility functions present in the repository.

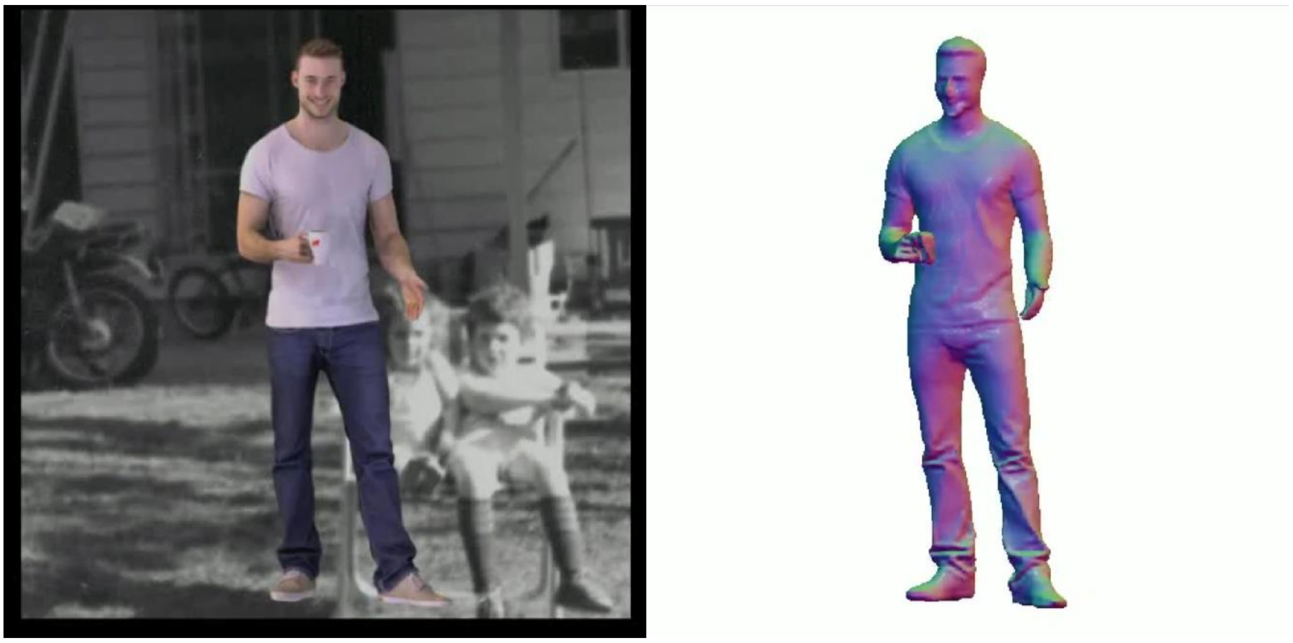


Figure 59: PIFuHD example inference 2

5. Conclusions

3D model generation and building 3D models from images are complicated tasks: a big enough network is needed to implicitly learn object structure and a lot of data is needed.
The **3D-GAN** architecture and its extension **3D-VAE-GAN** can mainly be used in case of relaxed level of detail requirements, since they use models represented by big **grids** that can lead to high time and memory requirements.
The preliminary model **voxelization** can be a long process and a manual check for quality might be necessary, to avoid **outliers** that complicate the training process and give wrong information to the generator.
Regarding the task of 3D model construction from images, the resulting model quality depends on the object type, on the number of available images and the applied encoding.
The encoder needs to be able to capture important characteristics from each image and put them together in a single representation.
Combining vectors with an **average** has led to the best results, because the mistakes in the representation of certain vectors were **compensated** with values from other vectors.
Doing just an average might not be sufficient in case of more complicated objects, with lots of details that need to be captured from different points of view: the max operator, in this case, might allow the encoder to select important characteristics from each vector. It also reduces the sparsity of the resulting three-dimensional models, thus often guaranteeing a consistent output.
In photogrammetry, the features found in an image are compared with those of images close to it in terms of position and angle: given the right image order, an **LSTM-based** encoder might put together more important features sequentially. In case of a lack of sorting, though, it can easily lead to overfitting and sparse representations for models not seen in the training step.
To obtain the **best possible encoder**, one should probably **put together** all the encoders implemented. Position and angle of an image might be used as training data or one might try to predict it with a separate network, to then sort images and combine them appropriately.
In any case, the 3D-VAE-GAN architecture is not **state of the art**: in recent years different, more elaborate frameworks were developed that can overcome its cons and make it possible to construct other important elements of 3D models, such as normal maps and texture maps.
The **DIB-R** architecture from NVIDIA allows us to obtain three-dimensional models of higher quality, thanks to the differentiable renderer and the numerous, specialized loss functions for each task, like shape, colour and vertex positions with respect to their neighbourhood.
DIB-R still need model voxelization with its cons. Facebook’s **PIFuHD** completely resolves the issue by representing the surface with **implicit functions**: thanks to a coarse and a fine level, the network can easily represent shape and detail.

References

[1] J. Wu, Z. Chengkai, X. Tianfan, W. Freeman and J. Tenenbaum, “Learning a Probabilistic Latent Space of Object Shapes via 3D Generative-Adversarial Modeling,” 2016.

[2] H. Su, S. Maji, E. Kalogerakis and E. Learned-Miller, “Multi-view Convolutional Neural Networks for 3D Shape Recognition,” 2015.

[3] P. Min, “Binvox 3D mesh voxelizer”.

[4] Dimatura, “Binvox-rw-py,” [Online]. Available: https://github.com/dimatura/binvox-rw-py.

[5] M. Mirza and S. Osindero, “Conditional Generative Adversarial Nets,” 2014.

[6] W. Chen, J. Gao, H. Ling, E. Smith, J. Lehtinen, A. Jacobson and S. Fidler, “Learning to Predict 3D Objects with an Interpolation-based Differentiable Renderer,” 2019.

[7] M. Arjovsky, S. Chintala and L. Bottou, “Wasserstein GAN,” 2017.

[8] S. Saito, T. Simon, J. Saragih and H. Joo, “PIFuHD: Multi-Level Pixel-Aligned Implicit Function for High-Resolution 3D Human Digitization,” 2020.

[9] S. Saito, Z. Huang, R. Natsume, S. Morishima, A. Kanazawa and H. Li, “PIFu: Pixel-Aligned Implicit Function for High-Resolution Clothed Human Digitization,” 2019.

[10] T.-C. Wang, M.-Y. Liu, J.-Y. Zhu, A. Tao, J. Kautz and B. Catanzaro, “High-resolution image synthesis and semantic Manipulation with Conditional GANs,” 2018.

[11] I. Goodfellow, J. Pouget-Abadie and M. Mirza, “Generative Adversarial Nets,” 2014.