**2D Image to 3D Model using Pytorch**

Lam Nguyen

**Due Date:** 06Dec2023

**Abstract**

Develop a method of converting a single 2D image into a 3d Point Cloud Model using a modified Autoencoder type of Neural Network. Using traditional Convolutional Network architecture for creating 3D models is computationally wasteful so a different method will be developed where only the relevant data on the surface of the object is taken into the network. We use 2D Convolutional Neural Networks to create a 3D structure and generate images from multiple viewpoints.

**1. Introduction**

The challenge we will be tackling is how to train a Deep Learning model to construct a 3D object from one RGB image. A lot of the problems being tackled today with deep learning and computer vision involve doing 2D imaging tasks such as image classification, object detection, semantic segmentation, instance segmentation, filtering, etc.… Now the next progression is to work with 3D models.

When projecting a 3D object onto a 2D plane, some image is inevitably going to be lost in the process of rendering. Therefore, in order to do the opposite which is to move from a 2D plane to 3D object, we have to have some way of making up for that lost information. Deep learning networks can be used to make up for that lost information.

We will be using an Autoencoder to feed 3D models in order for the Autoencoder to learn a compressed representation of the input. Autoencoders are a type of neural network architecture that extracts features out of images that allow the image to be compressed to a smaller size. These features can then be used to reconstruct the original image. This ability to get fundamental features that a neural network can use to recreate something with more data and information is a very useful mechanism for converting a 2D image to a 3D model where a 2D image acts essentially as a compressed feature map that can be decoded back into a 3D model.

As of 2018, and this is probably different now in the year 2023, converting a 2D image to a 3D model only seems to work by training one type of item at a time since 3D models are so computationally expensive.

Before going into the specific architecture of this network, it will be beneficial to have a quick discussion of how 2D images and 3D models are represented

**1.1 How 2D image Data is Represented**

2D image data is usually represented in two ways:

**1.1.1 Raster Image**

The first method is as a raster image. A Raster image is a matrix of pixels generally in a grid pattern. Each Pixel can be 1-4 channels. If it is one channel, the image will be gray-scale. If it is 3 channels, the image will have varying intensities of Red, Green and Blue. If it is 4 channels, the image will have varying levels of Red, Green, Blue and Alpha(transparency). The more pixels you have in an image, the more detail you can have. And each individual pixel can have its own value allowing for a lot of detail in an image depending on how many pixels are contained within an image.

**1.1.2 Vector Image**

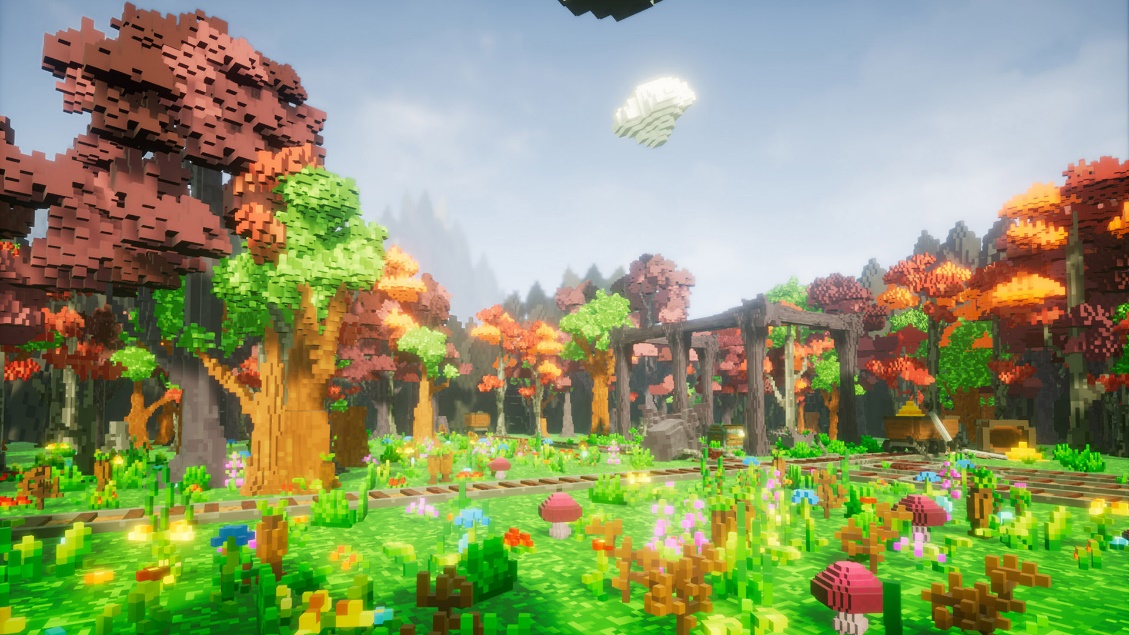
Another less common method for 2D image representation is in the form of vectors. Vectors images can be manipulated with programs like Adobe Illustrator. Vector images are created from geometric shapes defined on a cartesian plane, such as points, lines, curves and polygons. These geometric shapes can be defined with mathematical formulas which allow for a high level of precision. This precision is needed for domains such as architecture, engineering, land surveying, chip design, 3D rendering, etc. Another advantage of vector images is that since they are basically a collection of geometric formulas, vector images can be easily scaled up or down to any size and resolution without losing any detail or crispness in the image. This property is good for creating large banners, t-shirts, car wraps, etc… that can be very large, or vary small using the same vector image.

However, the downside of vector images is that they don’t allow for as much subtle detail, and expressiveness compared to rasterized graphics. They also can’t be put through a Neural Network for training.

**1.2 How 3D image Data is Represented**

3D image data can be represented in a greater variety of ways compared to 2D images. For our purposes, this choice of 3D data representation is crucial as it can mean the difference between not being able to use network training or being able to. Below will be given a quick overview of a few types of 3D image formats:

**1.2.1 Raster Form**

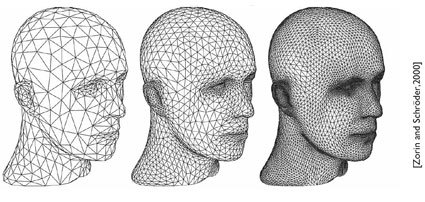


The 3D version of pixels are known as volumetric pixels or Voxels. 2D rasterized images are a collection of square pixels arranged spatially in a 2D space. 3D rasterized models are a collection of cube pixels arranged spatially in a 3D space. Each voxel has a predefined size. The location of each voxel as well as the properties inside each voxel define the unique structure of the model. Rasterized 3D models have the advantage of being able to be passed through a neural network for training in converting a 2D image to 3D model.

However, there some downsides. As you increase the resolution of the 3D model, the density of the voxels decreases. This is akin to scaling up a 2D image without increasing the number of pixels… You can see that an image becomes much more jagged. Another downside is that rasterized 3D models use a lot of computational resources compared to other methods of 3D model representation because each voxel cube has to be rendered individually.

The reasons are that the computer has to account for each and every single voxel and it’s movement through 3D space. Another reason is that voxel 3D models, have internal pixels that can’t be seen… But nevertheless, they have to be accounted for by the computer. This waste computational resources. To be more specific, say you want to represent a 3d model of a laptop computer. All you see are the outside case of the laptop, the screen, the keyboard, etc.… But you don’t see the motherboard, the CPU, the internal fans, etc… Since you don’t see these internal components when manipulating a 3d model, there is no reason to render them since this wastes a lot of resources. All you would need to do is to render the outer surface of the laptop when doing 3d animation. However, with voxels, you would have no choice. This is not a totally accurate metaphor, but hopefully it proves the point.

**1.2.2 Polygonal Mesh**

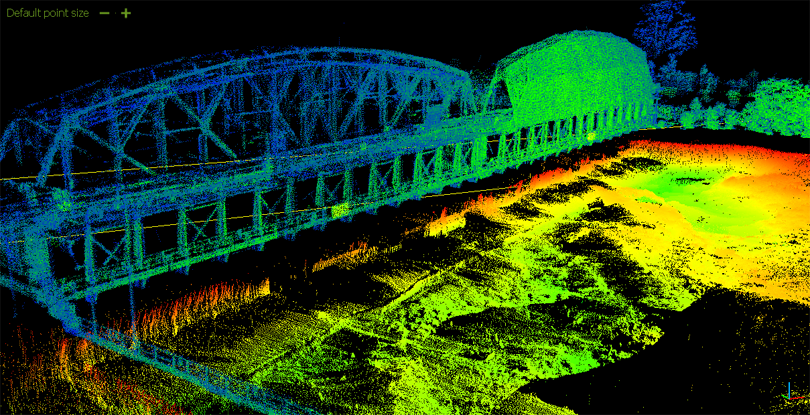


Polygonal meshes are similar to vector images for 2D image representations in that Polygonal meshes are a collection of vertices (points), edges and faces that define an object’s SURFACE in 3 dimensions. Meshes don’t calculate or store internal dimensions of an object unlike Voxels. Also, since pixels, edges and faces are geometric shapes, they can be mathematically represented meaning they can be compressed more easily. This means that Polygonal meshes are relatively compact while still being able to capture fine details.

Polygonal meshes nowadays normally use triangles or quadrilaterals to represent faces on an object. And these faces are made of edges. And the edges are made up of points. The more triangles you have in an object, the more detail you can have.

The downside of polygonal meshes is that they can’t be input into a neural network for training… So, we have to turn to a different solution.

**1.2.3 Point Clouds**



Point clouds are a collection of points in a 3D coordinate system. Together, these points form a cloud in the shape of the 3D object. The greater the number of points, the greater the detail of the object. A point in a point cloud and a voxel are very similar in that each point or voxel has to be placed in a 3D space. The difference is that a voxel has predetermined size. For example, a voxel has to be set to 1 inch. Whereas a point has no size. You can make a point cloud and then convert it to whatever unit size you want. This allows a lot of saving of computational resources. Also point clouds are conducive to certain technologies that scan real life 3D objects creating points along the surface. This allows point clouds to be focused only on the surface of an object which saves more resources than having to focus also on the invisible internal volume of an object.

Point clouds are used in scanning real life objects and converting them to 3D models due to their compact size and ability to be easily converted to other different types of 3D data. Some software that can be used to convert point clouds to other 3D objects are:

1. Euclideon
2. MeshLab
3. CloudCompare
4. Point Cloud Library

However, the downside of point clouds is that they cannot be applied directly to a neural network. However, the ability of point clouds to be easily converted to other forms will allow us to use them with some modifications to be input into a trainable neural network.

**2. Methodology on Converting 2D Image to 3D Model**

The approach we will use is to combine Point Cloud 3D representation with a 2D Autoencoder trained with a Convolutional Network. A general outline of the technique is described in the steps below:

1. Build an Autoencoder that can extract features from an input RGB image and returns multiple 2D Projections of a point cloud at predetermined viewpoints.
2. The next step is to take those multiple 2D point cloud projections and combine them together into a 3D point cloud image. This is possible because the viewpoints are consistently fixed and predetermined.
3. The Next step is to do a pseudo-render of the Point Cloud at different viewpoints.

When we combine steps 1-4 to get model that is able to generate a point-cloud representation from a single 2D image, using a 2D Autoencoder. The key for this model was to make the fusion of the 2D point cloud projections, and the pseudo-rendering steps differentiable and geometric.

Geometric means that there are no learnable parameters which decreases the size of the model and makes it easier to train. Differentiable means that we can perform back-propagation

In the following sections below, a more detailed discussion about each of the steps and the results will be given.

**2.1 Step 1: Build Autoencoder and Generate multiple 2D Point Cloud Projections**

The Autoencoder is a standard CNN architecture that takes in an RGB image and outputs multiple 2D projections of a point cloud. The 2D projection at a viewpoint is defined as:

**2D Projection == 3D coordinates(x,y,z) + binary mask (m)**

The architecture of the network is shown below:



**2.2 Step 2: Fuse multiple 2D Point Cloud Projections into 3D Point Cloud Image**

In this stage we take those Point Cloud Projections and fuse them together to create a 3D Point Cloud. Then the point cloud Then we compare the loss between the ground truth point cloud and the reconstructed point cloud and then back propagate to adjust the parameters.

**3. Experiments**

**3.1 Datasets**

The dataset used is a subsection of the Shapenet Dataset. The subset we will use are chair objects.

A unit in the dataset includes Input RGB images stored in the form of Numpy Array file (.npy) as well as the corresponding Point Cloud file stored in the form of a Matlab file (.mat).

The data is then split into a test set and a training set.

**3.2 Evaluation Metrics**

Training was divided into 2 stages which was to first build the Modified Autoencoder and then to build the Pseudorenderer. After each epoch, the training data loss and the test data loss are calculated and recorded in the logs files.

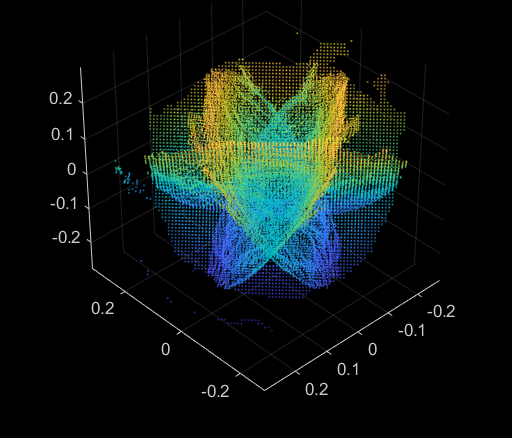
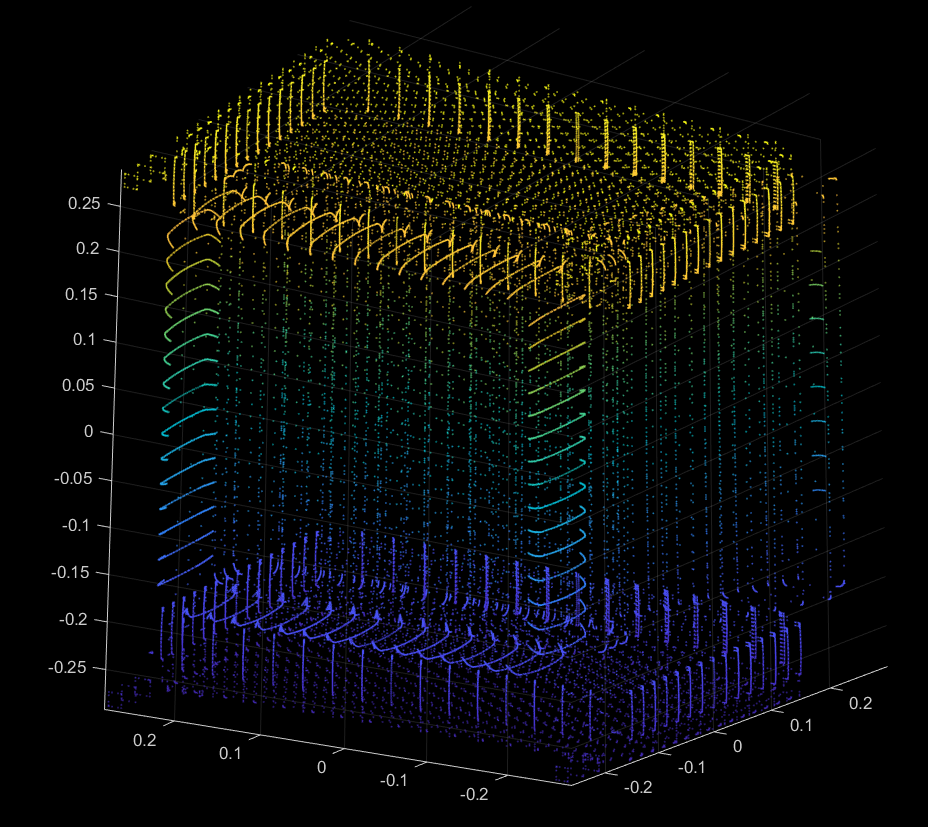
The final Metric is to take the Point Cloud Model generated from the Model and Compare it to the Ground Truth Point Cloud created from the ShapeNet Dataset. The loss will then be determined and recorded.

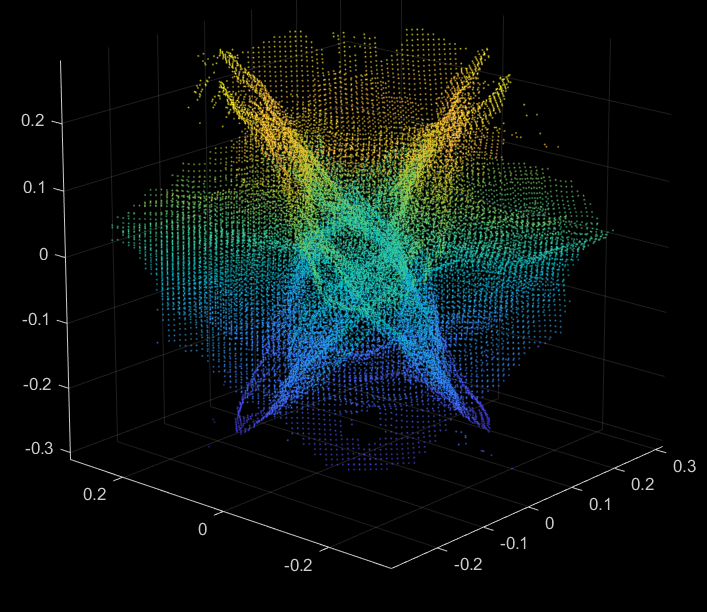
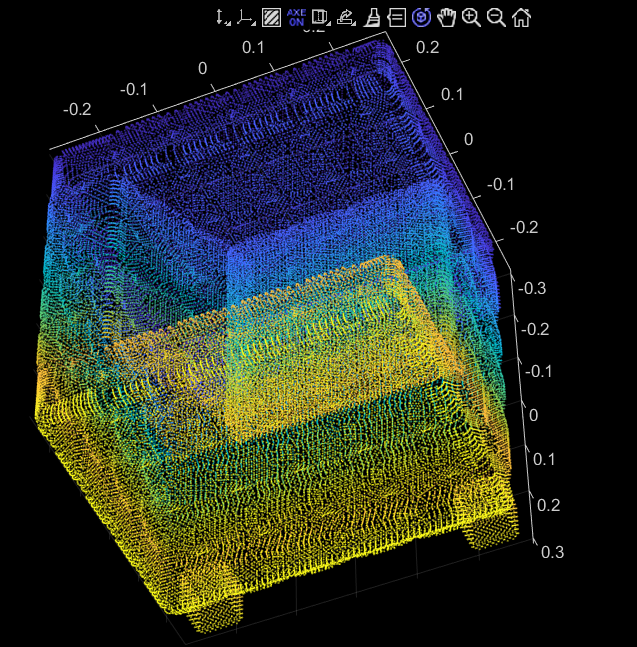
**3.3 Results of the Training**

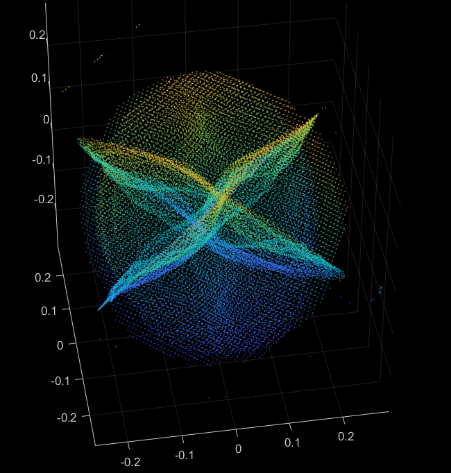
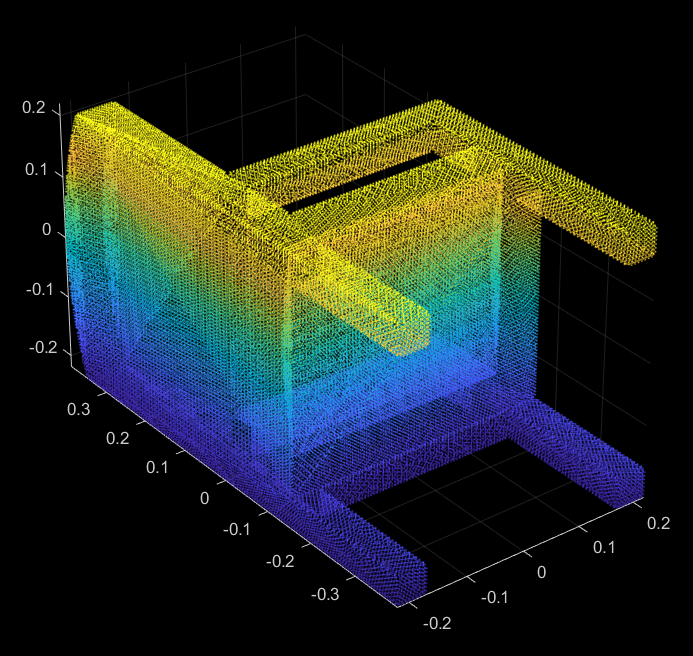
Stage 1 of training where the Autoencoder was created to produce 2D Projections of a 3D point cloud started with a training data loss score of .78 and validation loss score of .70. After 1000 epochs, the training data loss was brought down to .2 and the validation data loss was brought down to .20.

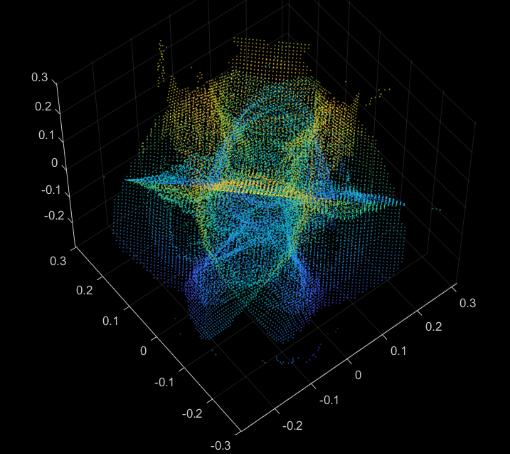
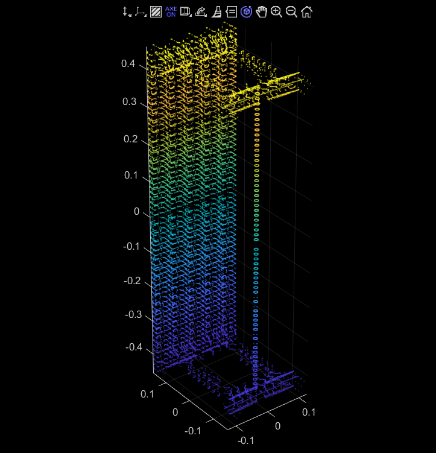
Stage

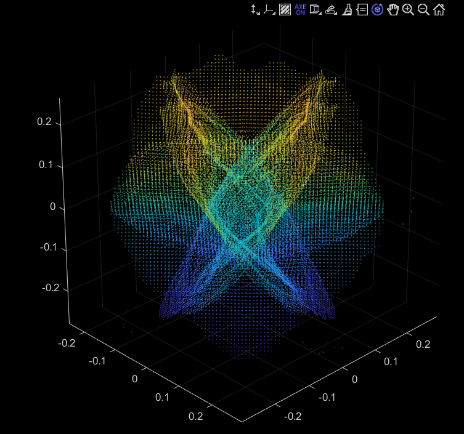
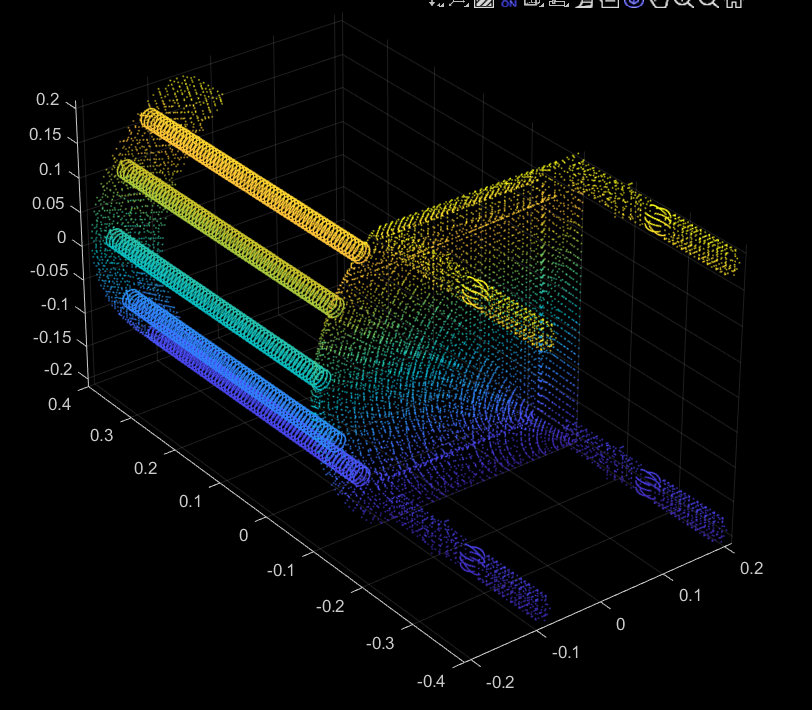
So, the results of the training were not good. Here is a ground truth Point Cloud Image on the left with the reconstructed image on the right:











**3.4 Analysis and Discussion**

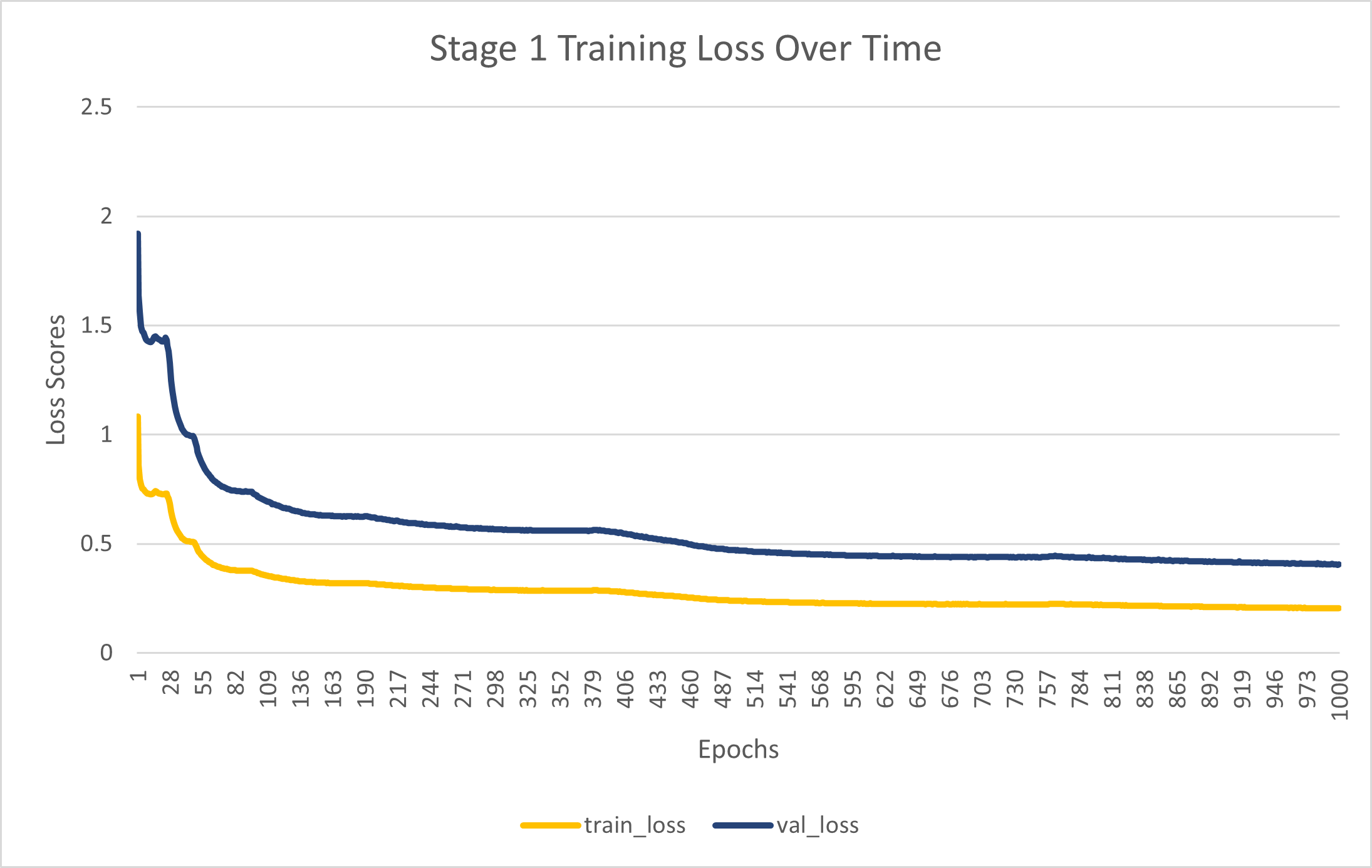
So based on the images alone, the training results were not very good.

There are some possible reasons for this. I might have to adjust the architecture of the Autoencoder. But I suspect the reason had something to do with the adjustments I made to the code that I borrowed from a paper written in 2018 that I sourced a lot of this material from. Since the code was written 5-6 years ago… A lot of the packages have changed and previous modules had been deprecated or updated. Also, the GPU drivers have changed and been updated. This required a lot of alteration in the code and how matrices were dealt with. In doing this, I adjusted a lot of the hyper parameters and the way matrices were manipulated. This could have caused the poor results.

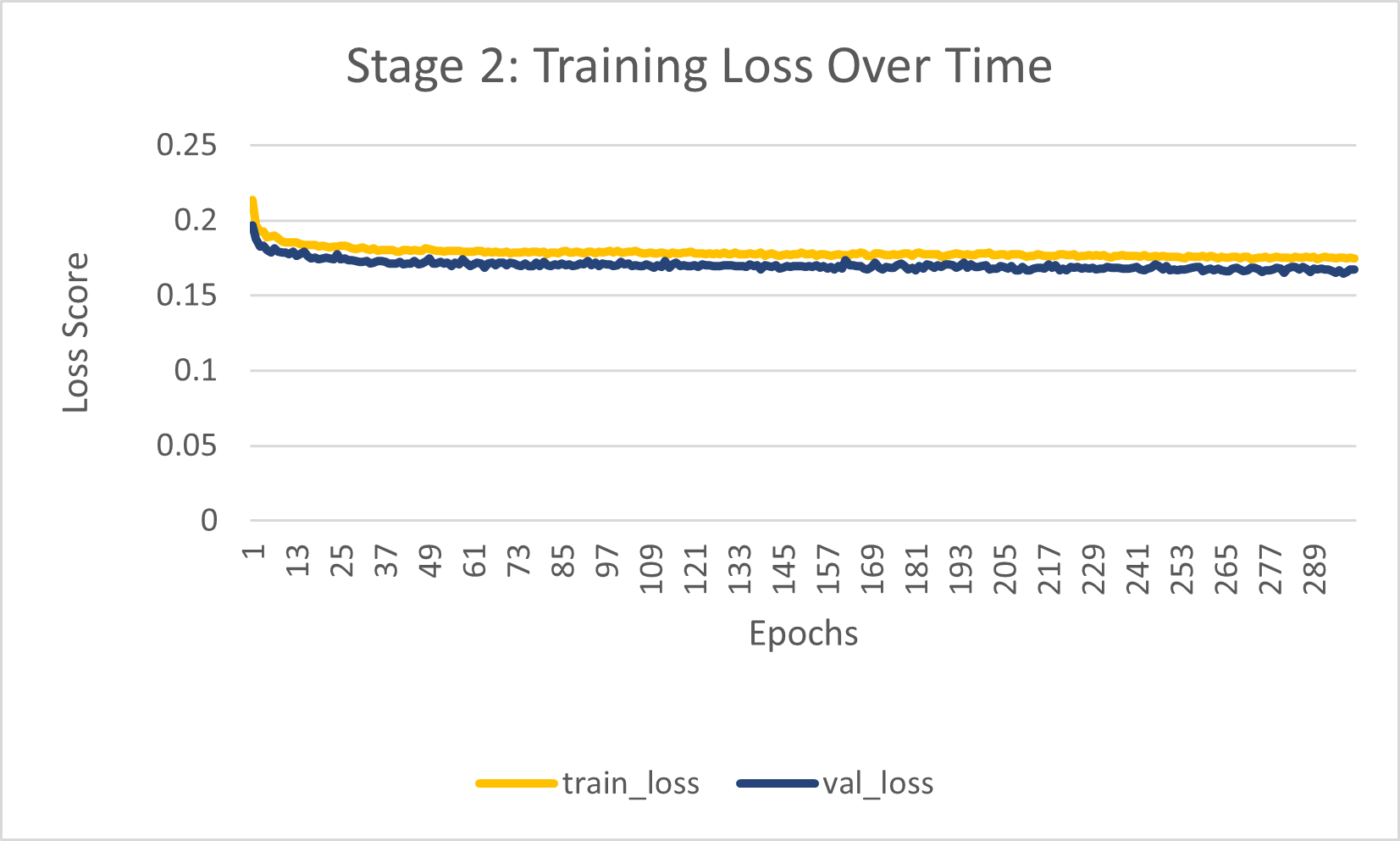
Another reason could have been that the Stage 2 Training of the Pseudo renderer was not done for enough epochs. The second stage of the model was trained for 300 epochs due to time and computational power limitations.

More adjustments can be made in the future but limits on time and computational power have not made this feasible.

**Loss for Stage 1**

****

**Loss for Stage 2**

****

**4. Conclusion**

In conclusion, the strategy of using an autoencoder to take a 2D RGB image, convert it to 2D projections of a point cloud, fuse the projections of the point cloud together and then do a pseudo-render did work.

However, the results were not good. This is due to the fact that more adjustments have to be made in the specific architecture of the network. However, due to time and computational power limitations, I was not able to do that. However given more time, I would be able to adjust the hyper parameters and fine tune the code to get much better results.