A Different Approach to Surface Re-Construction from Point cloud Data using Self Trained Neural Networks

Rakesh Reddy Theegala **P.R.S Rakesh Gupta** **Rohith Chodavarapu**
Indian Institute of Technology Guwahati
{rakes170101071, sai170101050, rohit170101017}@iitg.ac.in

1 Introduction

Construction of 3D surfaces from point clouds is a widely discussed problem in computer graphics domain. Currently, there are a variety of mathematical and Deep Learning based pre trained models to do this. But they come with their own short comings. In case of mathematical models such as ball rolling algorithm or Poisson's reconstruction, they usually required a huge number of points with high precision to produce surface accurately.Coming to deep learning based methods, all pre trained models suffer with a common problem of being biased towards training data set.

We suggest an alternative to this using a self-training deep learning based approach, where given a point cloud data with limited number of points we train a neural network to predict whether an independent point is inside the actual figure or outside. This trained model serves as an implicit function for our 3d figure. When the given input point cloud is very low, we perform Delaunay triangulation and sample more points and classify them using ray algorithm. Once training is completed, we can use the model as the implicit surface function to build the surface of the network. The entire pipeline of the algorithm we have used is explained in Section 2. In section3, we will talk about the results and conclusion and in section4, we will conclude with the limitations and possible future works.

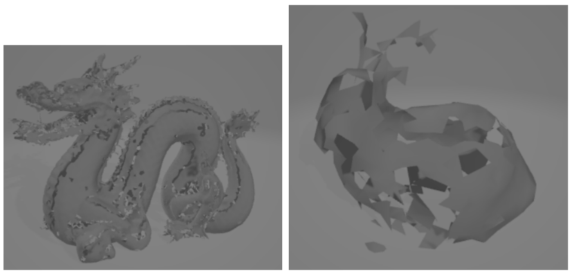

Figure 1: Surface Reconstruction using open3d tool. Left figure is the output of 50K points and right figure is the output of 500 points

2 Method

2.1 Triangulation

To preserve every minute detail of the points data, we will try to perform a delaunay triangulation to the given point cloud data. The example we used was with the polybunny with 500 points.

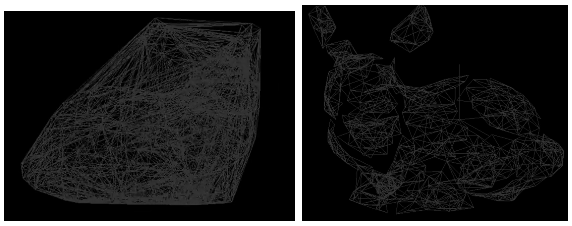

Figure 2: Left side is triangles with just using CGAL and right is with considering edges only inside same cluster

We start with using the CGAL tool to perform the required delaunaay triangulation. But we have observed that out of the 7700 edges we get from this there are lot of noise and unnecesarry extra edges which shouldnt be there for the polybunny [Figure 2]. So in order to combat this problem we should have a way to decide which edges are accepted and which ones are not. This we have performed by clustering the points into groups based on distances with each other using k-means algorithm. After this using 20 clusters we got around 5000 points and observed that even though all the

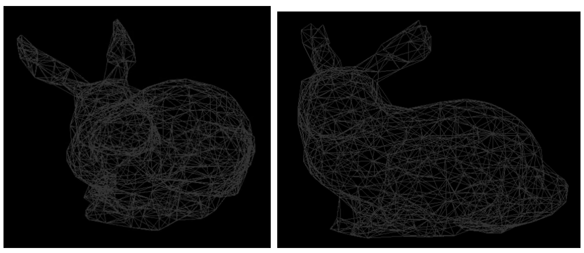

Figure 3: Left figure is when only considering the edges between close clusters and right figure is when normals of each point are considered

noise edges were removed a lot of correct and useful edges were also removed making the complete figure look as if it is in patches [Figure 2]. So to further refine our edges rejection method, we have allowed edges only inside the same cluster or between clusters which are close by. By experimenting with different values, we have decided that 2 clusters are close by if the distance between their corresponding centroids is less than or equal to 3× third closest cluster distance(measured between centroids). With this we have retained around 7000 edges [Figure 3]. But to improve this even further we have used a tool open3d, and were able to compute the normals of each point from the unorganized points data. Using the normals we have only allowed edges making close to 90 degrees with the normals. Finally we got got around 5800 edges which even though has some noise gave almost clean triangulation [Figure 3].

2.2 Dataset Creation

Deluany triangles are used for data set building in our model.we used those triangles to form data points. we pick random points in the triangle and displaced with vector along the normal for a distance on the precision we need(Figure 4) and we pushed all these data points in a list. now Data point is formed, we have to classify these points. We use the delaunay traingles to classify a point into a) lies inside the poly bunny(-1) b) on the boundary of poly bunny(0) c) lies outside the poly bunny(1)

Specifically,To say a given point P which is inside the poly bunny or not,we selected a random direction D using rand(3)[gives three random values which represent the three values of direction].Now,we will pass a ray from this point P in the direction D and then for each triangle we calculate whether the given ray R intersects the triangle or not . and we keep a counter of how many these type of triangles are there in the mesh. From the visualisation we can say that if the count of those triangles are even then point lies outside the poly bunny else point lies inside the poly bunny(figure 5). And while calculating for a given whether a triangle intersects w.r.t a point p and direction r,we also calculated the distance between point and triangle if it is sufficent small then we are classifying this point as on the boundary point. There is one loophole in above algo that there might be case where ray pass through edge so it cut to triangles at a time so to handle these cases,we try to do the above algo of some x random direction and take the one which has high frequency(figure 3)

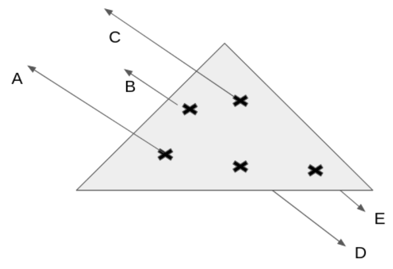

Figure 4: dataset formation

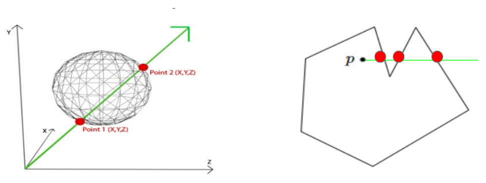

Figure 5: intersection of object with ray (outside point, inside point)

To calculate the intersection of triangle with a ray , we used Möller-Trumbore algorithm because using usuall numpy arrays in python to calculate normals,determent,distance taking too much time and giving us delay in classifying the points.But Möller-Trumbore algorithm gives the distance and classifying the point in best time as far as now.we used +1 0 -1 convention for classification and data set printed in (point,+1 0 -1) format.

2.3 Training neural network

Now that we have obtained a sufficiently large data set, we will start training a neural network to replicate the Implicit surface function of the poly bunny. Since, we have already classified the data into 3 categories(Inside, On, Outside of the boundary), we need to construct a viable neural network and train it. After going through a series of experiments with different architectures and their low accuracy(around 50%), in order to have a proof of concept to justify this method we have generated a rather simple surface and carried our experiments on it.

Using a unit sphere equation, we have created a data set of 64K points and classified each of them into required 3 categories. Then we did some experimentation with the selection of architectures con-

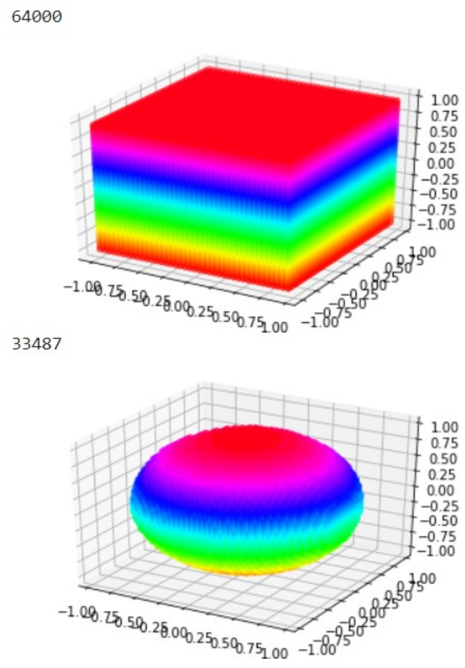

Figure 6: Original cube data set and the filtered sphere data points

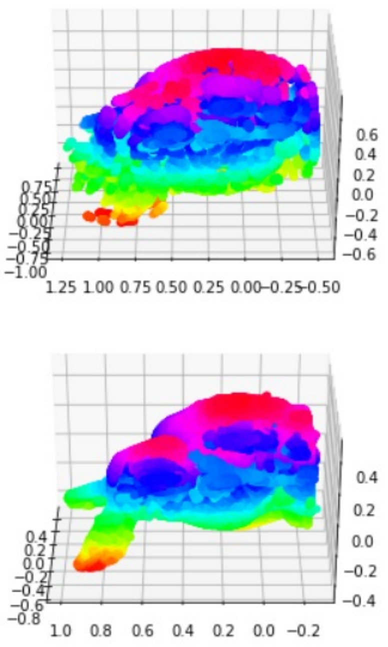
Figure 7: Original 1.5M data set and the filtered 1M data points of the poly bunny

sidering the complexity of the problem and tried various activation and loss functions.Finally, after some additional experimentation on optimizer methods, we have obtained the required model which gave 99.5% accuracy on the sphere data set. The neural network used for this purpose is a shallow network with 2 hidden layers each with 10 and 5 neurons. Tanh activation function is used in the hidden layers and sigmoid is used in the final layer. MSE(mean squared error) is taken as the loss function and since sigmoid only gives output between 0 and 1, this resulted in very low accuracy as output though the model is working perfectly. So to solve this misinterpretation, I have re-categorized the data into 2 categories(inside and outside) by merging on boundary data points into inside category. After training, we have predicted the output of the same training data set to obtain a sphere from a cube. Since, we are creating personalized models for each dataset, using same training and testing datasets will not create a problem if we are able to get a good surface using them.The initial cube data set with 64K points along with the filtered 33K points that generated the sphere are plotted and shown in figure 6.

A similar architecture was created to replicate poly bunny surface. Considering the increase in the complexity because of poly bunny structure, a more wider(around 64 to 512 neurons in each layer) and deeper architecture(11 hidden layers) is used keeping other parameters such as activation, loss functions and optimizer methods same to previous architecture. This model gave 98.5% accuracy on 20K poly bunny data set we have generated in section 2.2. To get a better picture at the resultant surface using our generated implicit function which is in the form of a neural network, we predicted the output of around 1.5 million points and plotted the around 1million points which resulted as either inside or on the boundary of the poly bunny. The initial 1.5 Million data set along with the filtered data set are shown in figure 7.

3 Results and Conclusion

To get a better picture of our re constructed surface, we rendered an approximate surface by using 1 million points by converting them into 1D voxels. For each point, we generated an extra point by adding 0.01 to its x coordinate and rendered a line segment between these two points. In this way, we have used 1D voxels to get an approximate surface of our implicit surface function which is in the form of a neural network. The obtained surface can be seen in the below figure. As we can see in the figure 8, our neural network has successfully obtained key features in poly bunny and gave better outputs as compared to tools like open3d.

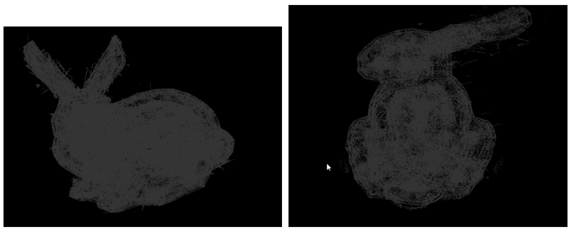

Figure 8: Final resulted output

Coming to statistical results, Initially we took a poly bunny point cloud with around 500 points and we sampled around 20K points using our ray algorithm. This took around 90minutes on google colab with 2gb ram. The training of neural network on this 20K data set took over 10minutes time on a 2gb ram google colab and resulted in 98.5% accuracy. It took another 5min to sample 1.5 Million points and predict them using our trained model. Though it is costly in time as compared to other models like open3d but it compensated time by producing better results in terms of accuracy and precision. In this project, we have explored a way of using a neural network as an optimization tool to obtain an implicit surface function along with estimating unknown features. We also improved delaunay traingulation output obtained from cgal tool and implemented ray algorithm in an efficient manner.

4 limitation and Future Work

In [Figure 3] even though we see an almost clean surface, actually we still got a lot of triangles on top of one another which made training the NN harder as a set points can be inside according to one triangle and outside according to the other.So there is still a lot of room for improvement and as expected when we used our Delaunay to create train dataset for our NN; we only got around 70% accuracy compared to 98-99% accuracy when using the correct Delaunay triangulation.

While building the datasets to train the neural network from the Delaunay triangulation, we picked the points at a small fixed distance from each triangle in the normal direction and decided where these picked points position's would be with respect to the actual 3d figure. But by doing this we have created 3 layers of data, one above the surface, one on the surface and one inside the surface. So if we see the output we also see a second layer inside the surface caused by this. To improve this, instead of using fixed distance from the triangles we have to change the precision for the outside and inside layer where the outside layer will have less precision and the inside will have high precision. This we will be able to achieve if we are able to decide which side of each triangle will be outside and inside with respect to the actual 3d figure. One simpler approach which we did not implement due to lack of time is that we can change the precisions randomly while picking the points.

And while creating the neural network architecture, though we have experimented different architectures and activation functions, we can see in the final output that there is a bit of noise at the edges. This is the result of using a over fitted model to predict output of new data points. We can solve this problem by using normalization layers in between hidden layers to control the level of over fitting. Also, we can try various architectures and can formulate a theory to design the complexity of the architecture, activation functions and other hyper parameters based on the complexity of the required surface.

While creating the test data set which will be used to generate the actual output 3d figure, as our model is trained to overfit, the more closer it is to the actual expected output the better shape we get. And as out model will be acting as an implicit function if we collected very huge number of points we can get a very smooth figure, for example here in our output we have used approximately 15 lakh points but if we change that nnumber to 1 crore or so we would definitely get a bettr output. And also as this doesnt affect the model in any way we can decide the number of points just based on how much resolution we want out output to be. Also as our projects main focus was to explore the idea of creating an alternate method to use a trained model to act as an implicit surface, we did not pay attention while showing the output. So we have showed it in small 1 dimensional edges for each point instead voxels which is enough for understanding the shape and smoothness of the output but can be improved very further like with the use of normal voxels.