Weight-Encoded neural implicit 3D shapes
Alexander Breeze 101 143 291

I am in COMP4900D, however I chose a custom topic, so I have included the proposal here and then my report after it.
This is a resubmission on December 14th with minor improvements, if that is too late then please refer to my previous submission on December 7th which is nearly identical.

PROPOSAL:

My project is to use a machine learning neural network to learn a 3D mesh, thus encoding an approximation of that mesh into the weights of the network. The purpose of this is to significantly reduce the size of the mesh without significant loss of quality. I chose this topic because I have a strong interest in machine learning, and it seemed like a fascinating intersection of machine learning and geometry processing.

My method will be to sample the same points that the marching cubes function would use at a fixed resolution to make a voxel grid, then train a neural net on that 3D voxel matrix to compress it into an occupancy function. This should result in an insane level of data compression without any loss in reconstruction quality compared to standard marching cubes on the original mesh.

This paper https://arxiv.org/pdf/1812.03828.pdf is like what I want to do, but instead of predicting the occupancy network of a fixed resolution voxel grid it instead samples random points to learn a general function of the mesh and then reconstructs with Multiresolution IsoSurface Extraction (MISE) and an octree, which is essentially marching cubes where it recursively increases the resolution of voxels the the surface of the mesh. This results in a reconstruction of higher quality, but takes more time and space to do it since the model must be trained for that level of resolution and the reconstruction is higher quality. Essentially, the advantage of this paper is that it allows a variable resolution at the cost of time and space. I consider implementing MISE to be an optional enhancement to current project, requiring alteration to the geomproc marching cubes function, which I will do if I have time (I did not have time).

A more circumstantial work is this paper: https://arxiv.org/pdf/2009.09808.pdf, which uses the signed distance field (SDF) as its target for the model to learn and samples points based on their distance to the mesh. The paper uses SDF because it gives more information in the labels for the model to train on (distance as well as sign), and because SDF is more useful for graphics and geometry processing. However as I am only focused on reconstruction, and since computing the distance from a point to the nearest triangle so I know its SDF value is time intensive, and learning the SDF is more difficult than learning a simple occupancy function, I believe an occupancy-function-only method would be faster and smaller.

2. Methodology.

First I will need to use some code from marching cubes to know what points will be sampled for reconstruction, making a 3D grid of points. Then for each point I will need to determine if it is inside the mesh, using raycasting, resulting in a 3D grid of booleans. This will introduce issues with meshes that contain holes, as the raycasting may fail. Then I need to learn the 3D grid using machine learning (multilayer feedforward with ReLu activations, binary classification output), with a preference for model architectures that occupy minimal memory, possibly optimizing the hyperparameters of the model as I encode to minimize space. Finally I must make that ML model interface with the geomproc marching cubes function to facilitate reconstruction.

3. Implementation.

The main components I will implement are:
fixed-resolution voxel sampling
occupancy function for given point, with redundancy for incomplete meshes
ML model to learn binary classification task
hyperparameter optimiser for ML model (optional)
ML model interface for geomproc marching cubes function

I intend to use python with the geomproc library, tensorflow for ML models, kerastuner or hardcoded loops to optimize the model hyperparameters, and numpy for arrays.

I will use the provided example meshes, and I will look online for significantly larger and fully connected ones to better demonstrate the size reduction of the method. I will evaluate the method based on the compression achieved for a given resolution of marching cubes, compared to other formats such as the binary matrix and the original mesh.

REPORT:

Weight-Encoded neural implicit 3D shapes
Alexander Breeze 101 143 291

Introduction:

The project I chose to implement was a custom one regarding machine learning, where I would train a machine learning model on a mesh to encode the mesh as a neural implicit function.

To accomplish this I used the geomproc mesh and pointcloud objects, as well as the marching cubes function, specifically the copy optimized by one of my peers (discord@Frostflight). I also used the tensorflow library for the ML model and kerastuner for automatically optimizing hyperparameters and architecture to encode arbitrary meshes at arbitrary resolutions. For code I made use of chatgpt for tensorflow/numpy syntax and the Möller–Trumbore intersection algorithm for raycasting between a point and a mesh. I don't have the exact query history, but all the other code is mine, and I explain it all in depth in this report. Finally my approach was loosely based on the paper https://arxiv.org/pdf/1812.03828.pdf.

The reason I chose this project is because I am very interested in AI, in fact it is my current university stream, and I had the idea of encoding a mesh into a neural implicit independently before learning it was a common topic in geometry, thus I was very interested in trying to do it for real.

Method:

I separated both the code and the output file structure into nicely segmented parts. The marchingCubes.py is FrostFlight's alteration that computes the 3D occupancy function binary matrix before generating the mesh, which isn't strictly necessary now that my code can determine if a point is inside the mesh quite quickly. The project.py is all my code.

ProjectPart2.py was a secondary failed attempt to train a second model on only the points on the edge of the mesh to simulate MISE (adaptive marching cubes) in preparation for a full implementation, so feel free to ignore it.

In project.py there are various helper functions:

intersectsTriangle determines if a ray from a point intersects a triangle, using the Möller–Trumbore algorithm

raycast chooses a ray from a given point and sees how many triangles in the mesh are intersected, determining the point's winding number aka its occupancy function (whether or not it is in the mesh)

```
raycast(point, mesh):
    ray=chooseRay()
    windingNumber=0
    for triangle in mesh:
        if intersectsTriangle(point, triangle, ray):
            windingNumber+=1
    return windingNumber%2 == 1      # ==1 makes it a boolean instead of an int
```

genPoints generates a list of all the points that marchingCubes will use for reconstruction at the given quality level

genPoints3D was for the projectPart2.py, it is an equivalent function that makes a 3D spacial matrix of points instead of a simple list. It can be ignored

At the bottom is a main function which takes arguments, specifically the mesh to encode and the quality, e.g.
python project.py bunny 32
If no quality is given it will default to 16, recommended quality be a power of 2. Mesh names do not require .obj and will be searched for in the meshes folder.
main calls 2 functions, first encodeToModel() which encodes the mesh into a ML model and generates various files, then reconstructFromModel() which reconstructs the mesh from the ML model files. I separated them so it would be clear that the only data used in the reconstruction is the ML model files loaded by reconstructFromModel().

The output folder gets one folder for every mesh+quality pair, all files generated are stored there.

encodeToModel() does many things:

First it imports and normalizes the target mesh, then uses genPoints() to get a 1D list of coordinates for every point that marchingCubes will use. It then maps points through the raycast function to create a second list, binary, which represents the occupancy function. These two lists are linked, so points[x] has occupancy of binary[x]. This function takes a lot of time because each point must raycast with each triangle in the mesh, with ~36 000 points for quality=32 and around 1000 faces in a given mesh, (36 million raycasts), so it tries to load in a pre-saved binary and if none exists then it saves one after computing it. This significantly reduces runtime on repeated attempts.

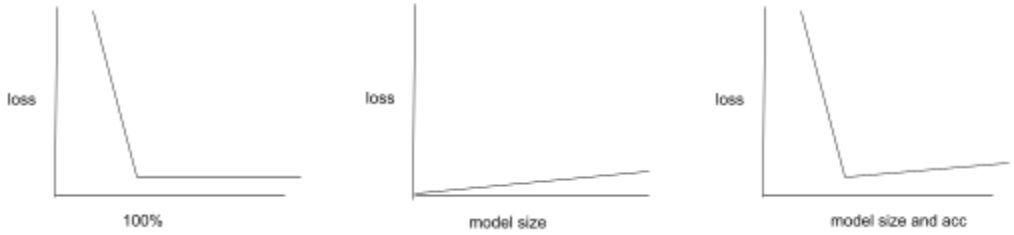
As the first output we create a point cloud of every point in points, colored by binary (occupancy), so you can load that and turn on point colors to see the occupancy function representation of the original mesh (move your mouse in a little circle to add movement for better viewing, as a static image is not very informative).

Next we run marching cubes on the occupancy function points/binary, by making a wrapper function which takes a point, finds its index in points, and returns the corresponding value in binary. Note that marchingCubes handles occupancy via sign, e.g. all points with values <0 are inside the mesh. Thus all mask functions for calling marchingCubes do a mapping from {1: -1, 0: 1} via the equation (bool*-2)+1. This naive reconstruction is saved, along with a copy of the original mesh, both for reference to compare to the actual model.

After this we begin the main part of the code, training the ML model on points and binary to perform binary classification with an attempt to maximize the accuracy on the train set. There is no validation set since the goal is to overfit on the given dataset. We first define a bestModelContainer to store the best model, then make a hypermodel object which has a build function to build a neural network and a fit function to fit the network on data and return the custom objective value. Previously this was to optimize hyperparameters, but now I hardcode them and tweak it by hand. In the fit function we set the weights of the labels inversely to their frequency, so that if there are more points inside the mesh than outside (or vice versa) the model won't learn to only predict the more common label. We fit a few epochs with Adam optimizer, which automatically sets the learning rate, then we set a fixed very small learning rate and use every data point for every batch (so one step per epoch) to finish overfitting until the loss doesn't decrease for 20 epochs. This takes a few minutes up to a few hours. At the end we used to compute the objective with the function

objective = 10000000 * (1 - currAcc) + model_size
And then save the model to the bestModelContainer if it is the new best to return the best model found after searching through hyperparameters.

The objective function was designed to find the smallest model that could achieve 100% accuracy, by adding 2 terms:
10000000 * (1 - currAcc) + model_size = objective

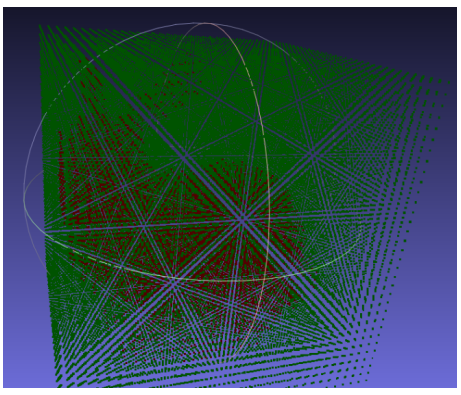


The first term dominates when accuracy<100%, forcing the search to use a larger model. Once accuracy=100% the first term goes to 0, at which point the second term forces it to find the smallest model that still achieves 100% accuracy. However while this worked for precision of 4 and even 8, once we got to 16 or 32 there were so many points that achieving 100% accuracy was infeasible, hence why the hyperparameter optimization for a minimum size network was commented out in favor of a network which could achieve 98-99% on 32^3 point sets.
Finally I save the model weights and architecture.

reconstructFromModel() first loads in the model architecture and weights, then puts it inside a masking function that takes a point and uses the ML model to return the predicted occupancy. This function is fed to marchingCubes for reconstruction, and the final reconstruction is saved in the output folder.

Results:
Input is the original mesh, which is converted into an occupancy function and reconstructed via marching cubes (naive). Then an ML model learns the occupancy function and marching cubes uses the ML model to reconstruct again.

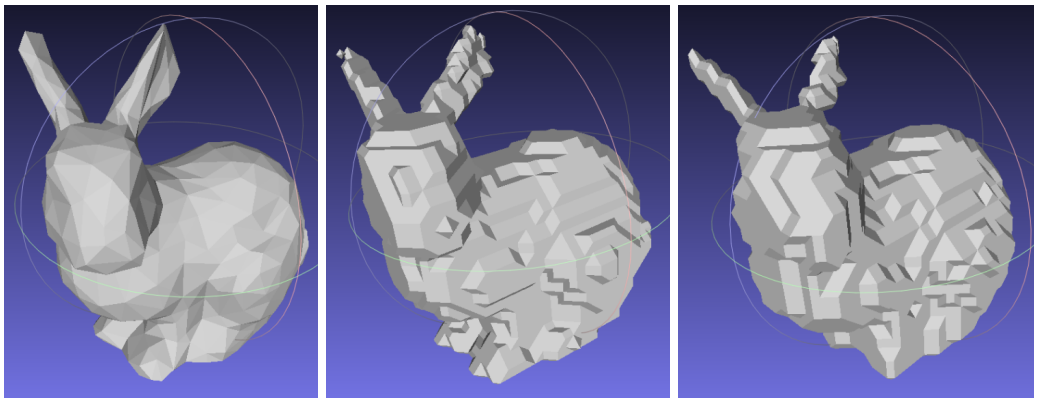
The input is a mesh, which is labeled here as “original”. It is converted to an occupancy function point cloud.



Bunny as occupancy function point cloud in marching cubes format, quality=32
It is recommended to load one into meshlab and spin around it slowly to see it more clearly, as motion gives you a better view of the red (inside mesh) through the green (outside mesh). You may need to explicitly turn on point colors

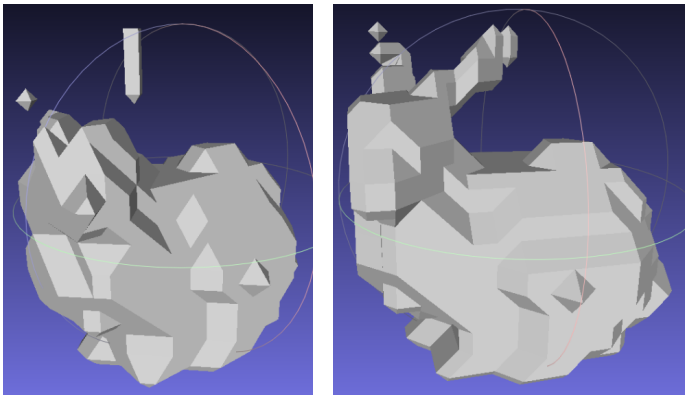
This variant of the occupancy function is used to reconstruct the original mesh via marching cubes, and is labeled “naive”. Then the point cloud is encoded into a neural network and reconstruction happens once more on that trained model. Each step is a lossy conversion, so quality decreases as the process continues (Naive is worse than original, machine learning is worse than naive).

ORIGINAL NAIVE MACHINE LEARNING



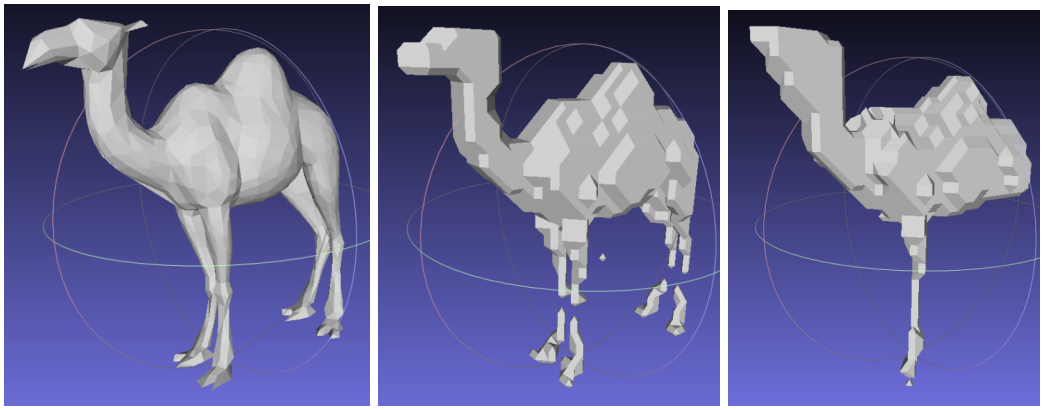
First we see the bunny, encoded to a quality of 32. This gave very good results as it does not have many protrusions for the model to learn. The model aims for high accuracy, and since the marching cubes points are evenly distributed this leads the model to prioritize parts with larger volumes relative to their complexity. This is why it does good on the bunny body and head, but struggles a bit on the ears since they are relatively thin and thus have less volume, so less importance. However we note that, aside from the ears being slightly smaller and the front paws being further in, there is negligible loss in quality in the encoded model.
Encoding for maximum accuracy can take as little as a few minutes for lucky starts, up to a few hours for unlucky ones, as the closer the Adam optimizer gets to the optimal the less time we must spend optimizing with a fixed small learning rate. Encoding the mesh via occupancy function tends to take around half an hour, as does reconstruction, because there are many points in the marching cubes and many triangles in the mesh. Also both of those algorithms are written in python, and thus are slow. The ML model training is tensorflow meaning it is optimized in C and thus runs as fast as it can for this model type.

ML reconstruction Naive



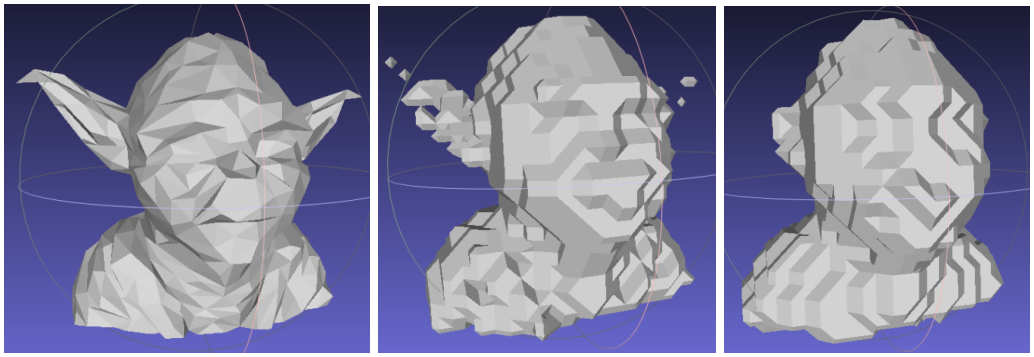
Here we can see the bunny with quality=16, and an ML encoding with 99% accuracy. As before it gets the torso well but struggles with the thin ears. This also demonstrates the code’s ability to handle variable quality levels fluidly.

ORIGINAL NAIVE
MACHINE LEARNING



The camel has many protrusions in the form of legs, which are so narrow that even the naive reconstruction has holes in them due to them being between points in the marching cubes. The naive reconstruction also misses the hooked beak of the camel. Note that I used quadric decimation first to reduce the complexity of the camel, as well as filling its holes to make it manifold, so that the raycast would work correctly and quickly. That is why the original model has a hooked beak.
The ML reconstruction handles the body well, even getting the hump perfectly, but ignores most of the legs. Note that once again the model prioritizes volume, so perfecting the surface of the body is just as important as adding a leg of equal volume. the head is also higher and cut off at the top, this may be a model issue or it may be an unrelated reconstruction issue.

ORIGINAL NAIVE MACHINE LEARNING



Finally we added a new model, the jedi master Yoda. We can see marching cubes made parts of his ears separate from the main mesh due to being too skinny to be caught by the pointcloud. The model encoding ignores the ears since so few points intersect them, but handles the body very well. We can also see at the bottom of the reconstruction (Around the shoulders) there are concentric rings in the ML reconstruction akin to layer lines from 3D printing, which are indicative of this being a function approximation rather than a direct reduction of an arbitrary model.

Discussion:

The main part of the method was fully implemented, the optional extra of implementing Adaptive Marching Cubes was not as I ran out of time before I could get it working. The largest challenge was achieving a high level of accuracy with the neural net encoding, forcing me to pivot away from model compression and towards general occupancy function encoding.

Training neural nets is normally more difficult and time-consuming than every other part of the project combined, and this case was no exception. I even implemented automatic hyperparameter optimization, which is currently pseudo-disabled but can be re-enabled trivially by uncommenting the code to make the model using hp hyperparameters and increasing the number of trials past 1.

If I had also implemented Adaptive Marching Cubes to recurse on the cubes on the surface of the mesh, then the majority of points would be on the surface of the mesh. This would theoretically have made the model prioritize surface area over volume, and reduced the total number of points to train on meaning faster raycasting, training and reconstruction, however this is theoretical as I could not get that code to work in time.

Also note that the reconstructions were sometimes not manifold since the marching cubes worked on a [-1,-1,-1] to [1,1,1] cube, which is exactly the same dimensions as the normalized mesh, resulting in the mesh sometimes touching the edge of the pointcloud where no faces would be put on it. To fix this one would need to slightly increase the cube size for marching cubes. This was not implemented as it would render all data generated so far incompatible with the new code.

The results I achieved looked good, and corroborated the expected outcome of the model prioritizing volume over surface area. My results were similar to the paper they were based on. I am glad I used only the points marching cubes cares about rather than random ones, since raycasting is so slow. The papers use around 1-10 million random points, which I think is only feasible time-wise if the ray casting is implemented in C.

I put a lot more time into this than it appears due to ML models requiring so much trial and error, meaning most of my efforts are deleted lines rather than implemented bulk code. I also had good separation of code chunks and a good output file format, which I am quite proud of.

Overall I enjoyed the project and the course very much, thank you for teaching it!