Generative AI

Assignment 1 - Report

Ioannis Bouritis - 2021030173

Πολυτεχνείο Κρήτης
Σχολή ΗΜΜΥ
May 22, 2025

Overview

In this assignment you will implement two methods that generate 3D shape surfaces conditioned on sparse point clouds. This assignment counts for 25 points (out of 100) towards your final grade.

The problem can be stated as follows:

Given a set of points $P = p_1, p_2, ..., p_n$ in a point cloud, we will define an implicit function $f(x, y, z)$ that measures the signed distance to the surface approximated by these points. The surface is extracted at $f(x, y, z) = 0$ using the marching cubes algorithm. All you need to implement are two implicit functions that measure distance based on the following methods: (a) a geometric method that calculates the signed distance to tangent plane of the surface point nearest to each point (x,y,z) of the grid storing the implicit function (b) a neural network trained to fit an implicit function $f(x, y, z)$ representing the signed distance to the surface approximated by these points.

The scikit-image package already provides an implementation of marching cubes. Thus, for part (a), you need to fill the code in the script *"naiveReconstruction.py"* to implement the geometric method, and for part (b), you need to fill *"model.py"* and *"neuralNetReconstruction.py"* to implement the neural network architecture and training respectively. Both implicit methods rely on surface normal information per input surface point. In the provided test data files, surface normals are included (the format of the point-cloud file is: 'point_x_coordinate' 'point_y_coordinate' 'point_z_coordinate' 'point_normal_x' 'point_normal_y' 'point_normal_z' [newline]). There are three point clouds (bunny-500.pts, bunny-1000.pts, and sphere.pts) that you will experiment with.

Your first step is to install python 3.13, if you haven't done so already. Download the starter code below and study it carefully. You will need to also install the following packages:

```
pip install numpy
pip install scikit-image
pip install scikit-learn
pip install trimesh
pip install pyglet<2
pip install torch
```

Tasks

Naive Reconstruction

Instructions

[5 points] One way to estimate the signed distance of any point $p = \{x, y, z\}$ of the 3D grid to the sampled surface points p_i is to compute the distance of p to the tangent plane of the surface point p_i that is nearest to p. In this case, your signed distance function is:

$$f(p) = n_j \cdot (p - p_j) \; with \; j = argmin_i\{||p - p_i||\}$$

Your task: Implement this distance function in the naiveReconstruction function of the script naiveReconstruction.py. Show screenshots of the reconstructed bunny (500 and 1000 points) and sphere in your report.

Implementation

To implement the distance function, the 3D grid coordinates (X, Y, Z) are first converted into a 2D array Q of shape (N, 3), where each row represents a 3D point in the grid. A KD-tree is then constructed from the input point cloud to enable efficient nearest-neighbor searches. For each grid point in Q, the KD-tree identifies the index of the closest point in the point cloud and returns this index:

$$j = argmin_i\{||p - p_i||\}$$

These indices are flattened into a 1D array and used to extract both the closest points and their corresponding normals from the original data. Next, a vector is computed for each grid point, representing the displacement from its closest point in the point cloud to the grid point itself:

$$p - p_j$$

The signed distance from each grid point to the tangent plane at its nearest point is calculated as the dot product of this vector and the associated normal:

$$f(p) = n_j \cdot (p - p_j)$$

Finally, the array of signed distances is reshaped to match the original 3D grid structure, forming the implicit function over the grid.

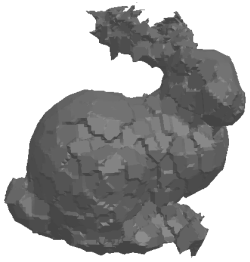

Figure 1: Bunny 500 points with Naive Recontsruction

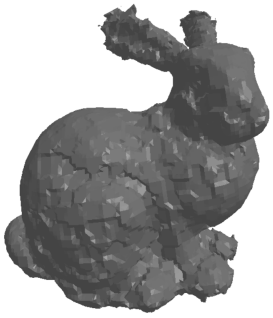

Figure 2: Bunny 1000 points with Naive Recontsruction

Figure 3: Sphere with Naive Recontsruction

Neural Network Approximation

Instructions

[10 points] Implement the architecture and forward pass in *"model.py"* for the neural network approximation. More specifically, given a set of 3D point samples **P'** and their SDF values $\mathbf{S} = \{s_1, s_2, ..., s_n\}$, we train the parameters θ of a multi-layer fully-connected neural network f_θ on (**P'**, **S**) to make f_θ an effective SDF approximator:

$$s_i = f_\theta(pi'), \quad pi' \in P', \quad s_i \in S$$

The network architecture is shown in the following figure. It takes as input the sample point coordinate $p_i' = (x, y, z)$ and passes it through 8 fully-connected layers (FC in figure). The size of the vector representaton in each hidden layer has size 512 as shown in the figure. Note that after the fourth FC layer, the layer outputs the hidden vector with 509-dim, which is concatenated with the original 3-dim point coordinate (x,y,z) to form the final 512-dim vector (see blue dashed line in the figure). In the first seven FC layers (i.e., except the last), a weight normalization layer is appended (nn.utils.weight_norm), then a leaky ReLu non-linear activation layer with a common learnable slope for all channels and a dropout layer (dropout rate=0.1) are added. Finally, the predicted SDF value is obtained with the last FC layer which transforms the vector dimension from 512 to 1 and another

tanh activation layer (denoted as TH in the figure).

Implementation

In the first task, the constructor of the Decoder class is designed. After setting input_dim as 3 (X, Y, Z), using nn.Sequential() the sequential layers are constructed as follows:

- **Layer 1:** Linear($3 \rightarrow 512$) expands the input to a high-dimensional latent space.
- **Layers 2–4:** Three consecutive Linear($512 \rightarrow 512$) layers refine features while maintaining dimensionality.
- **Layer 4:** A deliberate bottleneck Linear($512 \rightarrow 509$) prepares for skip connections by reducing dimensions to 509.
- **Layers 5–7:** Three Linear($512 \rightarrow 512$) layers further transform the concatenated features, enabling hierarchical reasoning about shape topology.
- **Layer 8:** A Linear($512 \rightarrow 1$) layer collapses features to a scalar value, representing occupancy probability or signed distance.

Next, the forward pass is implemented as follows:

1. **Layers 1–4:**
   (a) **Process**: The input 3D coordinates $\mathbf{p} = (x, y, z)$ pass through four sequential fully connected (FC) layers:

$$\mathbf{h}_1 = W_1\mathbf{p} + \mathbf{b}_1 \quad (\text{Linear: } \mathbb{R}^3 \rightarrow \mathbb{R}^{512})$$

$$\mathbf{h}_i = W_i\phi(\mathbf{h}i - 1) + \mathbf{b}_i \quad \text{for } i \in 2, 3, 4$$

   (b) **Activation**: Parametric LeakyReLU with slope $\alpha = 0.2$ is applied after each FC layer:

$$\phi(x) = \begin{cases} x & \text{if } x > 0 \\ 0.2x & \text{otherwise} \end{cases}$$

   (c) **Regularization**: Dropout ($p = 0.1$) is applied post-activation to prevent overfitting:

$$\mathbf{h}_i^{\text{drop}} = \text{Dropout}(\mathbf{h}_i)$$

2. **Skip Connection Integration:**

- **Process**: Concatenates the Layer 4 output $\mathbf{h}_4 \in \mathbb{R}^{509}$ with the original input $\mathbf{p} \in \mathbb{R}^3$.

3. **Layers 5–7:**

   (a) **Process**: Processes the concatenated features through three FC layers:

   $$\mathbf{h}_i = W_i\phi(\mathbf{h}i-1) + \mathbf{b}_i \quad \text{for } i \in 5, 6, 7$$

   (b) **Activation**: Parametric LeakyReLU with slope $\alpha = 0.2$ is applied after each FC layer:

   $$\phi(x) = \begin{cases} x & \text{if } x > 0 \\ 0.2x & \text{otherwise} \end{cases}$$

   (c) **Regularization**: Dropout ($p = 0.1$) is applied post-activation to prevent overfitting:

   $$\mathbf{h}_i^{\text{drop}} = \text{Dropout}(\mathbf{h}_i)$$

4. **Final Output Layer:**

   (a) **Process**: Projects features to a scalar value using a final FC layer:

   $$f(\mathbf{p}) = W_8\mathbf{h}_7 + \mathbf{b}_8 \quad (\text{Linear: } \mathbb{R}^3 \to \mathbb{R}^{512})$$

   (b) **Activation**: Tanh activation bounds outputs to $[-1, 1]$:

   $$f_{\text{final}}(\mathbf{p}) = \tanh(f(\mathbf{p}))$$

Dimension Evolution:

$$\mathbb{R}^3 \xrightarrow{Layer\ 1} \mathbb{R}^{512} \xrightarrow{Layer\ 4} \mathbb{R}^{509} \xrightarrow{Skip} \mathbb{R}^{512} \xrightarrow{Layer\ 8} \mathbb{R}$$

Neural Network Training

Instructions

[10 points] Complete the code in *"neuralNetReconstruction.py"* that implements the training procedure. The training is done by minimizing the sum over losses between the predicted and real SDF values of sample points under the clamped *L1* loss function:

$$L(f_\theta(p_i'), s_i) = |clamp(f_\theta(pi), \sigma) - clamp(s_i, \sigma)|$$

where *clamp(t, σ)* clamps *t* value within the limits (-*σ*,*σ*), where *σ=0.1* in this implementation.

The sampling of training points and their SDF values is already implemented for you in *"utils.py"*. The training 3D points are sampled around the given surface points of the point cloud along their normal directions: *pi' = pi*+ε *ni* where ε is randomly sampled from a Gaussian distribution N(0, 0.052). For each point *pi*, 100 different samples {*pi'}* are created for the training and validation set.

The neural network weights are optimized by the AdamW optimizer, as already specified in the starter code. The default learning rate is set to 0.0001, weight decay is set to 0.0001. The network is trained for 100 epochs with batch size=512. The loss for training and validation set are reported per epoch. The starter code saves the best model with the lowest loss on the validation set. Detailed training hyper-parameters can be found in the training code argument parser. After training the model, to evaluate the model and see the reconstructed mesh after executing marching cubes on the neural implicit, use the "-e" command line argument.

Implementation

In the implementation of the **train()** function in *"neuralNetReconstruction.py"*, the following additions have been made:

1. **L1 Loss Computation:** Uses Mean Absolute Error (MAE) between predicted and ground truth SDF values:

$$\mathcal{L} = \frac{1}{N} \sum_{i=1}^{N} |y_{predicted} - y_{true}|$$

2. **Gradient Propagation:** Explicit *loss.backward()* call enables proper backpropagation through the decoder network.

3. **Loss Normalization:** Accumulates loss as *loss_sum += loss.item() * batch_size* to handle variable batch sizes correctly.

In the implementation of the **val()** function, the following additions have been made:

1. **Value Clamping:** Restricts SDF predictions and ground truth to *[-clamping_distance, clamping_distance]*

2. **Evaluation Mode:** Uses *torch.no_grad()* context manager to disable gradient computation, in order to reduce GPU memory usage during validation.

Figure 4: Bunny 500 point with Neural Network Reconstruction

Figure 5: Bunny 1000 point with Neural Network Reconstruction

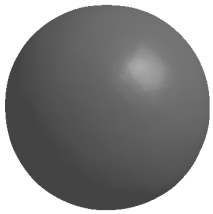

Figure 6: Sphere with Neural Network Reconstruction