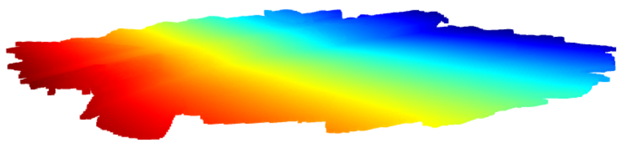
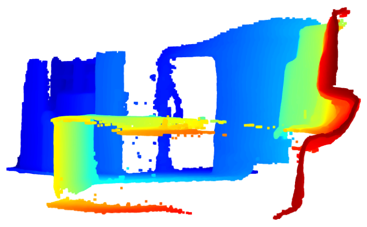
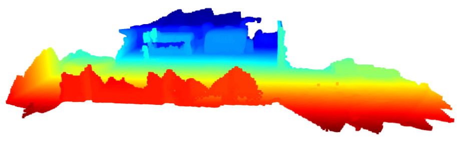
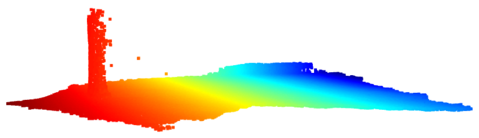
Current Results

Can't get the model to consistently and accurately produce inferences on NEW lidar scans without complete collapse of the model. I'll include photos to give further context to the struggles but simply it's either a severe case of overfitting or the model returns points randomly scattered to try and minimize the loss functions by cheating essentially. I've done some research on attempting to prevent the model from collapsing via curriculum learning, adjusting loss functions/adding new ones to discourage that behavior, and minimizing initial training sample size. I'm sure I'm missing one or two other attempts but those were the attempts that I spent the most time attempting to implement. Not incredibly unsurprisingly, none worked which is why I'm reaching out to you!

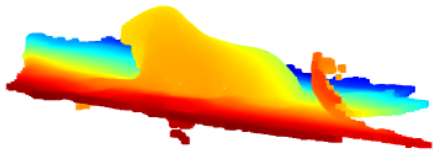
Inference inputs (used for all models_x). I tried using a variety of different scans not used in training to hopefully yield a different result when the model created the inpainting inferences, however, the outputs were MOSTLY consistent across each inference sadly:

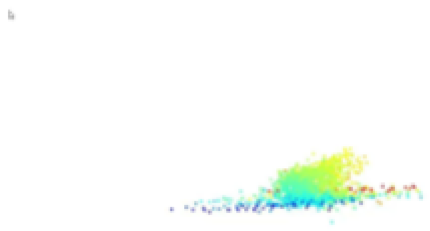
ATTEMPTS!:

Model 1 was only trained on one input scan to ensure the model could be trained in the first place. We were hoping for severe overfitting and we got it so we know that in theory this could work!

Scans model_1 was trained on:

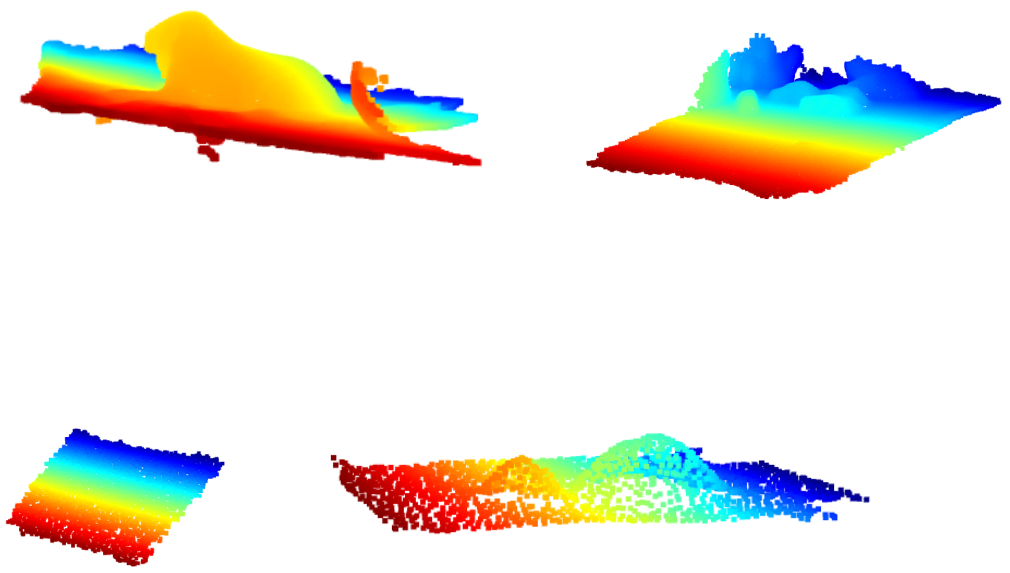


Model_1 inference outputs (singular because they're all the same via overfitting and the other model outputs will have similar videos):
Double click on the thumbnail directly below this. It will pull up the video. Google doc has no way to import videos straight into the document :(

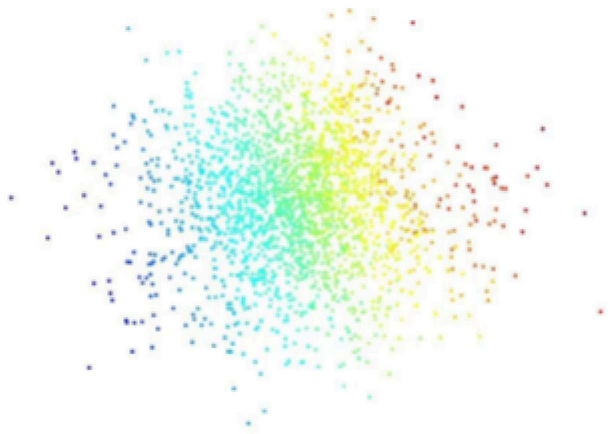


Model 2 used all the same variables and whatnot of model_1 however the input list grew to four simple/moderate scans. This was enough to cause complete collapse of the model though :(

Scans model_2 was trained on:



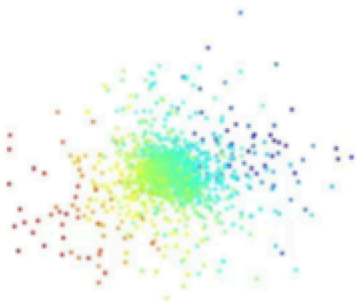
Model_2 output:



Model_3 was trained on all 80 homemade scans ranging from the floor to an entire living room (you'll see later!). Expectedly collapsed.

Scans model_3 was trained on: lidar_scans

Model_3 output:



Model_4:

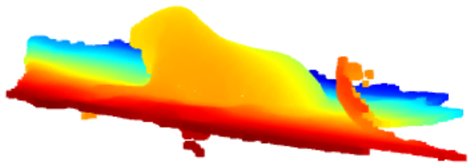
- Used mostly the same inputs as Model_2, but included one additional complex scan to test its effect because I was curious!
- Instead of training on all 5 scans at once, the model was trained incrementally:
    1. Train on scan 001 only.
    2. Restart training (still building off previous model), but now with scans 001 + 002.
    3. Restart again with 001 + 002 + 003.
    4. Continue until all 5 scans are included.

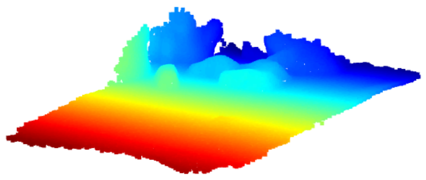
Why I thought it could work:

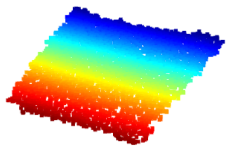
- The goal was to intentionally overfit on a single scan at first, giving the model a strong baseline.
- By gradually adding new scans, the model is encouraged to backtrack and adjust its learned representation to accommodate the new scans.

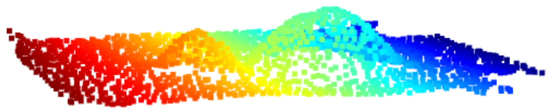
- This strategy definitely helped prevent collapse, since the model doesn't start from scratch on all scans simultaneously but instead builds progressively on a foundation.

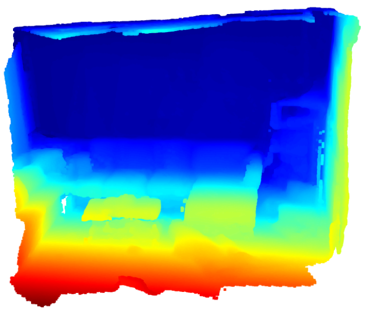
Scans:





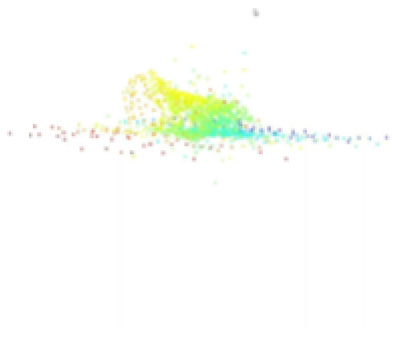




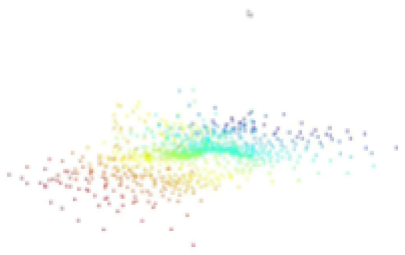


Model_4 output (all videos):

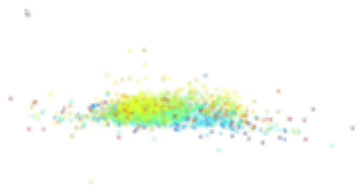
Training_1: 000001.xyz



Training_2: 000001, 000002.xyz

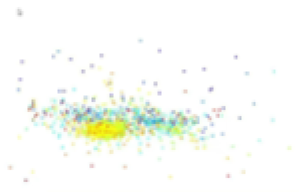


Training_3: 000001, 000002, 000003.xyz



Training_4: 000001, 000002, 000003, 000004.xyz (almost identical to training_3)

Training_5: 000001, 000002, 000003, 000004, 000009.xyz



This training method definitely yielded the best results. However, you can see that it begins to pool points in the center, and if we were to continue with this methodology, I have no doubt it would eventually collapse again. Another idea I've considered is grouping the scans based on apparent structure (simple floors together, walls together, bumpy surfaces, and complex rooms) so that I could introduce the simpler structures early on and gradually work up the "complexity scale." Unfortunately, I've lost much of my hope with this project, because I'm simply not knowledgeable enough in this area of science.

Documentation of the Code

Original Scans

The ground_truth directory contains the original set of LiDAR scans collected entirely on my iPhone 14 Pro. A total of 80 scans were taken across a variety of environments, including indoor and outdoor settings. The majority of these scans are flat or mildly undulating surfaces, with a few capturing more complex environments such as full detailed rooms and intricate surfaces like bookshelves.

For the purposes of this project, these raw scans are treated as the ground truth (GT). While in practice LiDAR (especially mobile LiDAR) is imperfect and cannot provide a true GT, these scans represent the closest approximation I had at my disposal. In an ideal workflow, one would have access to a real GT, then evaluate how closely a model can reconstruct it from a noisy real scan. Since a perfect GT is unattainable, instead I'm using the following approach:

1. Use original scans as GT
    a. The original scans in this folder are the reference data.

2. Generate synthetic scans with noise/errors
    a. Synthetic point clouds are generated to replicate typical failures of mobile LiDAR systems.

3. Train a model to inpaint / reconstruct scans
    a. By training on pairs of synthetic (noisy) scans and their corresponding original (GT) scans, the model learns to correct errors (or rather ideally would but certainly isn't right now!).

b. The eventual goal is for the model to take a real-world scan (which itself contains errors) and generate an improved version that is closer to the ideal GT.

This folder is the foundation of the dataset pipeline where original LiDAR scans act as the "ground truth" while synthetic variations provide the imperfect inputs used for training and evaluation.

Creating Synthetic Scans

The synthetic_scans directory contains modified versions of the original ground truth LiDAR scans, generated by intentionally introducing realistic degradations. These synthetic variations include local holes, global dropout, occlusion planes, and Gaussian noise. Each corruption technique is designed to mimic common errors in mobile LiDAR captures such as missing regions, partial occlusions, and measurement inaccuracies. The creation process is handled by individual loss functions and orchestrated by synthetic_file_creation.py which apply several transformations and save the results as new scan files. By simulating these imperfections, the synthetic scans serve as corrupted training inputs while the originals remain as reference outputs, enabling the model to learn effective inpainting and reconstruction strategies.

Created from these synthetic scans are masked files. These are files that document which points were kept and removed (kept = true, removed = false). This helps the model during training better calculate loss by focusing the model on points that need to be inpainted (masked[idx] = false).

So, in summary, each GT has 15ish synthetic files generated from its point cloud and each synthetic file has exactly 1 mask file to keep documentation on what exactly has been adjusted by the corruption functions.

What we are creating:

Synthetic scans are modified versions of the original point clouds that intentionally introduce corruptions, including:

- Local holes: Small spherical regions randomly removed to mimic occluded or missing parts.
- Global dropout: Randomly dropping a percentage of points across the entire scan to simulate sensor sparsity or dropouts.

- Occlusion planes: Removing points on one side of a random plane to imitate partial views caused by occluders.
- Noise addition: Adding Gaussian jitter noise to all points to represent measurement inaccuracies.

How we create synthetic scans:

Using functions implemented in the functions.py and synthetic_file_creation.py modules, each synthetic scan is generated by applying one or more of the following procedures:

- random_local_hole(points, radius, num_holes) - Selects random centers within the point cloud and removes all points inside spheres of specified radius, creating local holes.
- random_global_dropout(points, dropout_ratio) - Randomly drops a fraction of points uniformly across the scan.
- occlusion_plane(points, plane_normal, plane_point) - Defines a plane (either random or specified) and removes all points on one side of that plane, simulating occlusions.
- add_noise(points, noise_std) - Adds Gaussian noise with given standard deviation to all points to mimic sensor jitter.

Each function operates on the original point cloud coordinates, outputs a modified version, and these transformations can be composed or applied independently to generate a diverse set of synthetic scans.

Why are we creating synthetic scans? To reiterate:

- Simulate Real-World Variations: Synthetic degradations approximate the variability and imperfections common in real sensor data, such as occlusions by objects, sensor noise, and missing measurements because we cannot view those via the original scans without losing the GT.
- Improve Model Robustness: Training on these synthetic examples helps the model learn to reconstruct point clouds even when the input data is incomplete or noisy, increasing performance on real-world scans, the overall purpose behind this Project.
- Proof of Concept Progression: This step is essential before scaling to larger datasets, ensuring the training pipeline and model architecture can handle corrupted data scenarios successfully.

How the functions and synthetic_file_creation.py actually work:

Random_local_hole:

1. Create local holes (missing regions) by removing points inside random circles.

2. Create a copy in “points” to not overwrite data.
3. For each hole:
    a. Randomly select point in data and set it as the center of hole
    b. Compute Euclidean distance from each point to the center point
    c. Keep only points outside (removing all points on the inside)
        i. In the code this is saved in the variable “masking” which is a general term for creating a Boolean array where true is keep/seen and false means throw away or haven’t seen.
    d. Return points

Random_global_dropout:

4. Simulate global sparsity or dropout by randomly removing a percentage of points anywhere in the point cloud.
5. Calculate how many points to keep based on the dropout ratio in keep_num.
6. Indices then randomly selects keep_num amount of points and return the new reduced point cloud.

Occlusion_plane:

7. Simulate partial view occlusions by removing points on one side of a plane
    a. Occlusion means that some points are hidden from view because something else is in front of them. For example, if you take a picture of a chair from the front then the back legs are occluded by the seat and front legs. This is a VERY big deal and issue in LiDAR scans

8. First it checks if no plane normal vector is provided and if not then generates a random 3d vector and normalizes it.
    a. If no plane point is provided it chooses a random point for the plane to lie on.
9. Calculates the distance from every point to the plane and if it’s positive or negative (positive if it’s on the “right” side of the plane).
10. Return all points on the positive side essentially cutting the cloud in half (sometimes more sometimes less).

Add_noise:

11. Add random jitter noise to all points to mimic sensor measurement errors.
    a. Noise means unwanted, random and slightly inaccurate variations in the data set caused by sensor errors.

12. This function simply uses the random.random function to create gaussian noise with mean 0 and standard deviation noise_std for every coordinate of every point then add the noise to the points and return it.

Synthetic_file_creation:

13. Loads all original scans and applies the four different corruption techniques described in the functions.py file and saves each result as a new xyz file in the synthetic scans directory to increase dataset size and make training harder and more realistic of real-world LiDAR scans.
14. The save_points_as_xys saves the corrupted data to the synthetic scans directory

15. Generate_synthetic_scans:
    a. Goes through each of the 4 techniques and creates x amount
    b. For i in range of the desired number of files to create(currently 3-4):
    c. Created the new file using the functions.py and save it. That's it.
16. Process_all_scans:
    a. Setting up the generate synthetic scans function by initializing the original scans that will be edited inside a for loop

---

Training Model

Config.py:

Most if not all files in the training model directory require some constants that can be easily maintained in a configuration file which currently only holds two (thought I'd require more when I created this file initially):

- K = 32 (K nearest neighbors)
- NUM_POINTS = 2048 (how many points are included in the inputs and outputs of the model)
- LEARNING_RATE = 1e-3 after each backward pass, weights are updated by 0.001 × gradient.
    - If it's too high (1e-1), the optimizer may overshoot minima, making loss unstable or diverging (collapse).
    - If it's too low (e.g., 1e-6), training becomes very slow and might get stuck in poor local minima.

Model (GNN_autoencoder_model.py in “model\\”):

This file implements a Graph Neural Network (GNN) autoencoder designed to inpaint and reconstruct LiDAR point clouds. The autoencoder consists of a GNN-based encoder that extracts meaningful features from the input point cloud, and a decoder that generates a reconstructed point cloud from the encoded representation. The architecture uses EdgeConv layers to capture local geometric relationships between points. I ultimately decided to use the GNN because it is quite well-suited for 3D LiDAR data, which is inherently sparse and irregular given our current technologies!

Components:

GNNEncoder (nn.Module):

Purpose: Converts/Encodes a point cloud, represented as a graph, into a compact latent embedding vector as the Torch library depends on that formatting.

Input: A torch_geometric.data.Data object containing:

- x: Node feature matrix ([num_nodes, num_node_features])
  - Num_node_features can represent any information that the node requires for proper context. In our instance we only store xyz coordinates but if we wanted to we could also store RGB information.
- edge_index: Graph connectivity information
- batch: Graph membership for nodes in a batch

EdgeConv Layers:

- Perform convolution over edges in the graph, aggregating local neighborhood features rather than using a fixed grid like standard image convolutions.
  - Convolution is a way to combine information from a point and its neighbors using a learned function, summarizing local patterns. In our case this means aggregating features from each point and its connected neighbors to capture local geometric relationships. Similar to sliding a filter over a group of a few pixels in images.
- Capture local geometric relationships by measuring differences between each point and its k-nearest neighbors.
- Layer progression:
  - Conv1: Input features 6 (self + neighbor xyz), outputs 64 channels
  - Conv2: 64 → 128 channels
  - Conv3: 128 → 256 channels

- Channels: Each channel represents a different learned measurement of point geometry or neighborhood relationships, increasing representational capacity as layers deepen.

- Global Max Pooling: compiles point-level features to a fixed-length vector per graph, handling varying point cloud sizes.

- Fully Connected Layer: Compresses the pooled 256-dimensional features to a 128-dimensional embedding for efficiency, regularization, and overfitting reduction.

GNNDecoder (nn.Module):

Purpose: Decodes the 128-dimensional latent embedding back into a reconstructed point cloud.

Structure: Fully connected layers that expand the embedding to [NUM_POINTS * 3] (xyz coordinates for each point), reshaped to [batch_size, NUM_POINTS, 3].

Output: Reconstructed point cloud with a fixed number of points (NUM_POINTS), ensuring consistency regardless of input cloud size.

GNNAutoencoder (nn.Module):

Purpose: Combines the encoder and decoder into a complete autoencoding pipeline.

Forward pass:

- Input graph is encoded into a latent vector z.
- Latent vector is decoded to reconstruct the full point cloud.

Intended to take corrupted or synthetic scans and produces repaired point clouds approximating the ground truth.

Train_loader.py:

This file defines the custom PyTorch Dataset used for training the GNN autoencoder on LiDAR point clouds. Its purpose is to provide aligned batches of input synthetic scans, corresponding ground truth point clouds, and masks indicating corrupted or missing regions. This dataset also incorporates curriculum masking which gradually increases the fraction of masked points during training. The purpose is to progressively challenge the model by forcing it to learn to inpaint missing or corrupted regions rather than rely on shortcuts (uniform or collapsed clouds). By controlling the difficulty of masked regions over time, curriculum masking is supposed to encourage robust reconstruction strategies and helps prevent the model from collapsing to uniform outputs, ultimately improving its ability to generalize to new, unseen LiDAR scans. That is the goal at least… hasn't worked thus far and the model still CERTAINLY outputs collapsed outputs when the training model uses more than 2 GT to train at a time :(

Class: PointCloudDataset (torch.utils.data.Dataset

Initialization (__init__):

- Inputs:
    - ground_points_list: List of original (ground truth) point clouds.
    - synthetic_points_list: List of synthetically corrupted scans.
    - mask_points_list: Boolean masks indicating corrupted/missing points in synthetic scans.
    - mask_fraction_schedule: Optional function that takes the current epoch and returns a fraction of masked points to keep (used for curriculum learning).
- Purpose: Stores the input lists and provides a reference to the current epoch for dynamic masking.

Method: set_epoch(epoch):

- Updates the current epoch externally from the training loop.
- Allows the dataset to adjust the number of points kept in masked regions according to a curriculum schedule.

Method: __len__():

- Returns the number of ground truth point clouds in the dataset.

Method: __getitem__(idx):

- Retrieves a single training sample corresponding to index 'idx'.
- Steps:
    - Converts the synthetic scan and mask to torch.Tensor.
    - Applies curriculum masking if a schedule is provided:
        - Determines the fraction of masked points to retain based on the current epoch.
        - Randomly selects which masked points to keep, progressively increasing difficulty over training.

    - Downsamples points: Uses fixed_size_points_with_mask_torch to ensure the input cloud has a consistent number of points, adjusting the mask accordingly.
    - Builds the graph: Converts the downsampled point cloud into a graph structure using create_graph_from_point_cloud. Assigns a default batch index (all zeros for single graph per batch).
    - Prepares target ground truth points: Converts to fixed-size tensor using fixed_size_points.
- Returns: Tuple (input_graph, target_points, mask) for training the GNN autoencoder.

Fetch_Points.py

This file provides utility functions for downsampling or padding point clouds to a fixed size (NUM_POINTS from config file) suitable for our GNN model processing. It ensures that every point cloud input to the model (GT, synthetic, or masked) has a consistent number of points, which is critical for batching and stable training.

Functions:

fixed_size_points(points: torch.Tensor):

- Takes a point cloud tensor of variable size and either downsamples or pads it to exactly NUM_POINTS.
- If the point cloud has more than NUM_POINTS, it randomly selects a subset of points.
- If fewer, it pads by repeating the last point. (model will train to ignore repeated points)
- Ensures that all downstream GNN inputs have a consistent number of points.

fixed_size_points_with_mask_torch(points: torch.Tensor, mask: torch.Tensor):

- Similar to fixed_size_points but handles a boolean mask indicating corrupted or missing points.
- Keeps all masked points first, then samples unmasked points to fill up to NUM_POINTS.
- Pads if necessary and shuffles the points to remove ordering bias.
- Returns both the downsampled/padded point cloud and an aligned mask of size NUM_POINTS.
- Use Case: Maintains alignment between points and their mask, which is essential for computing masked losses during training.

Create_graphs.py

This file provides a helper function to convert point clouds into graph structures for use with graph neural networks. GNNs (Graph Neural Networks) expectedly require edges between nodes not just points floating in a cloud, and this function generates that structure efficiently using k-nearest neighbors. By creating these graph structures from point clouds, this function allows the encoder to apply EdgeConv layers, which aggregate local neighborhood differences. This enables the model to capture both fine local geometry and larger-scale structural patterns, essential for accurate inpainting of missing or corrupted LiDAR data.

Function:

create_graph_from_point_cloud(points, k=K)

- Converts a point cloud tensor [N,3] into a PyTorch Geometric Data object.
- Constructs a k-nearest neighbors graph where each point is connected to its k closest neighbors.
- Stores point coordinates as node features (x) and graph connectivity as edge_index.
- Returns the PyG Data object, ready for input to the GNN autoencoder.

Fetching_files.py

This module handles loading, organizing, and aligning point cloud data for the training pipeline (not straight forward because there are about 15ish synthetic/mask files per GT so we needed to use a map). It connects ground truth scans, synthetic scans, and masks into consistent lists for downstream dataset and training use. By standardizing data retrieval and checking alignment between files, it ensures that training batches contain properly paired samples.

Functions:

load_xyz_file(filepath):

Purpose: Loads a .xyz point cloud file into a NumPy array.

Inputs: filepath *(str)* – Path to the .xyz file.

Outputs: NumPy array of shape (N, 3) representing the point cloud.

---

build_pointcloud_lists():

Purpose: Builds aligned lists of ground truth point clouds, synthetic scans, and corresponding masks.

Process:

- Creates a dictionary of ground truth point clouds, indexed by filename (all files begin with xxxxxx. e.g. 000001.xyz is the GT and 000001_localhole_1.xyz is the synthetic).
- Iterates through synthetic scan files (sorted for consistency).
- Matches each synthetic scan with its ground truth counterpart.
- Loads the corresponding mask from .npy if available.
  - If mask dimensions don't match the point cloud, or if no mask exists, it defaults to an all-zero (unmasked) array.
- Appends aligned (ground, synthetic, mask) triplets to lists.

Outputs:

- ground_points_list – list of NumPy arrays (ground truth clouds).
- synthetic_points_list – list of NumPy arrays (synthetic clouds).
- mask_points_list – list of boolean NumPy arrays indicating masked points

---

Main.py

This file serves as the central training loop for the GNN autoencoder. It handles dataset construction, model initialization, checkpoint loading/resuming, training progression, and checkpoint saving. The refactor moves auxiliary functionality (loss tracking, checkpoint saving/loading) into helper modules under training_model/, keeping main.py clean!

Functions & Methods:

linear_mask_schedule(epoch, max_epoch, start_frac=0.1, end_frac=0.7):

- Defines a linear progression for curriculum masking.
- Starts masking a small fraction of points (start_frac) and gradually increases difficulty until end_frac.
- If epoch exceeds max_epoch, the fraction is capped at end_frac.

first_epoch_check(epoch, total_epochs):

- Ensures the loop continues until TOTAL_NEW_EPOCHS have passed.

second_epoch_check(epoch, total_epochs, has_saved):

- training continues up to 10 × total_epochs if no strong checkpoint has yet been saved which prevents runs from halting prematurely when progress is slow.

main()

- Initializes model (GNNAutoencoder) and optimizer.
- Loads previous best loss values via fetch_best_losses().
- If a checkpoint exists, prompts the user to resume training from it; otherwise, resets strategy-specific loss tracking.
- Adjusts the "strategy" threshold to encourage early updates after restarting, while progressively tightening over time.
- Iterates through epochs:
    - Updates the dataset epoch to apply curriculum masking.
    - Performs forward/backward passes with inpainting_loss.
    - Tracks average loss across batches.
    - Saves checkpoints and updates loss records when thresholds are surpassed.

Calculating Loss

The loss functions provide feedback to guide model training. Chamfer Distance ensures predicted points lie close to the ground truth, while Masked Chamfer emphasizes reconstruction in missing or corrupted regions. To improve surface quality, Laplacian Loss regularizes geometry for smoothness. An optional Approximate EMD enforces stricter point-to-point correspondence, though at higher computational cost (one my poor little pc cannot afford right now without GPU power which isn’t compatible yet sadly). Finally, the Inpainting Loss combines these components into a weighted training score.

Chamfer_distance:

Computes the Chamfer Distance (CD) between two point clouds. CD is a widely used loss for point cloud reconstruction because it measures how close each point in one set is to the other set. How it works:

- Showcase points1 and points2 so pairwise distances are computed.
- For each point in points1, find the nearest point in points2 (min_dist_x)
- For each point in points2, find the nearest point in points1 (min_dist_y)
- Loss is the mean of both directions encouraging mutual correlations.
- Output: Scalar loss (average CD across the batch)
    - A single number which is the average reconstruction error between the two point clouds.

Masked_Chamfer_Distance:

Computes Chamfer Distance but only on a subset of points, defined by a mask. This is useful for inpainting, where only corrupted or missing regions should contribute to the loss. How it works:

- Pairwise distance between predicted points and ground truth (GT) points.
- Same min-distance calculations as normal CD.
- Multiply distances by to zero out irrelevant points.
- Normalize by the number of valid points (mask.sum).
- Output: Scalar loss (focusing only on masked points).

Laplacian_loss:

Encourages smoothness of the reconstructed point cloud. Used to avoid noisy or jagged reconstructions by penalizing points that deviate from their local neighborhood. How it works:

- For each point in the batch
    - Compute pairwise distances.
    - Find the k nearest neighbors (K-NN) for each point (established in our config file).
    - Compute each point's nearest neighbor mean.
    - Penalize the squared difference between each point and its local mean.
- Average across batch
- Output: Scalar loss (small values mean points lie close their local surface)

Approximate_emd:

Computes an approximate Earth Mover's Distance (EMD) between two point clouds. EMD gives a more accurate similarity measure than Chamfer but is computationally heavier. How it works:

- For each batch element:
    - Compute pairwise distances between predicted and GT

    - Use Hungarian algo (via linear_sum_assignment) for optimal one-to-one matching.
        - The Hungarian algorithm is a way to pair up items from two groups so that the total “cost” of all the pairs is as small as possible. It looks at all the possible pairings and finds the best overall matching.
    - Average matched distances across points.
- Aggregate across batch.
- Output: Scalar EMD loss (lower is better)

Inpainting_loss:

Inpainting_loss is the main loss function that combines geometric fidelity (Chamfer/EMD) with surface smoothness (Laplacian). If mask is provided for given synthetic file use masked_chamfer distance (it should always be included and CD should be a redundant function if all is well), find laplacian loss, calculate EMD loss (optional and disabled by default because it’s VERY slow even with our configuration data (K and NUM_POINTS) being very small!), and return the loss. Weighted sum of losses are the sums of the adjustable weights of each loss function (lambda) and the loss itself.

Inferencing

Now we’re running an inference with the trained GNN autoencoder on LiDAR scans. It loads a trained checkpoint, processes raw .xyz scans into graph-structured input, performs inpainting/reconstruction, and saves the resulting reconstructions.

Functions:

load_checkpoint(path, device):

Purpose: Loads a trained model checkpoint into a fresh GNNAutoencoder instance.
Inputs:

- path *(str)* – path to the saved checkpoint (.pth).
- device *(torch.device)* – CPU or CUDA.

Outputs: A PyTorch GNNAutoencoder in evaluation mode.

---

load_points_np(path):

Purpose: Loads a point cloud from an .xyz file.
Outputs: NumPy array of shape (N, 3) (3D coordinates).

---

fixed_size_points_np(points, num_points=NUM_POINTS):

Purpose: Normalizes point clouds to a fixed size for model input.
Behavior:

- If N == num_points: returns unchanged copy.
- If N > num_points: randomly downsamples without replacement.
- If N < num_points: pads with duplicates of the last point.

Outputs: NumPy array (num_points, 3).

---

make_data_from_np(points_np):

Purpose: Converts raw points into a graph structure suitable for the GNN.

Steps:

- Chooses an effective k-nearest-neighbors (k_effective) for graph edges.
- Calls create_graph_from_point_cloud to construct graph connectivity.
- Assigns a batch index (all points belong to batch 0).

Outputs: PyTorch Geometric Data object with x, edge_index, etc.

---

save_points_np(path, points_np):

Purpose: Saves a point cloud to .xyz format.
Outputs: Writes file with shape (N, 3) coordinates.

---

run_inference_on_file(model, path, device):

Purpose: Core inference function for a single .xyz file.

Steps:

- Loads raw .xyz file.
- Ensures fixed-size input.

- Converts to graph (make_data_from_np).
- Runs forward pass through model (model(data)).
- Saves reconstructed point cloud with _recon.xyz suffix in model_inferencing/inferences.

Outputs: Reconstructed .xyz file.

Execution Flow (Main Block)

1. Loads the trained checkpoint (model/strategy.pth).
2. Iterates over files in model_inferencing/cleaned_scans/.
3. For each scan, runs run_inference_on_file() to generate reconstructed LiDAR point clouds.
4. Saves results in model_inferencing/inferences/.

Sources for GNN:

GNN background information:

- https://distill.pub/2021/gnn-intro
- https://www.sciencedirect.com/science/article/pii/S2666651021000012

GNN and EdgeConv:

Wang et al., 2019 – *Dynamic Graph CNN for Learning on Point Clouds (DGCNN)*

- https://arxiv.org/abs/1801.07829

Kipf & Welling, 2017 – *Semi-Supervised Classification with Graph Convolutional Networks*

- https://arxiv.org/pdf/1609.02907

autoencoder:

Yang et al., 2018 – *FoldingNet: Point Cloud Auto-Encoder via Deep Grid Deformation*

- https://arxiv.org/pdf/1712.07262

Loss/inpainting::

Fan et al., 2017 – *A Point Set Generation Network for 3D Object Reconstruction from a Single Image*

- https://arxiv.org/pdf/1612.00603

Liu et al., 2020 – *Morphing and Sampling Network for Dense Point Cloud Completion*

- https://arxiv.org/pdf/1912.00280

PyTorch Geometric Documentation

- https://pytorch-geometric.readthedocs.io/en/latest
- https://pytorch-geometric.readthedocs.io/en/latest/cheatsheet/gnn_cheatsheet.html
- https://pytorch-geometric.readthedocs.io/en/latest/modules/nn.html#

And of course ChatGPT:

OpenAI. (2025). *ChatGPT* [Large language model]. https://chat.openai.com/

Used OpenAI's ChatGPT (GPT-4 and GPT-5, 2025) to generate explanations and initial code drafts during model development