# Original Scans

Extremely small sample of LiDAR scans taken on my iPhone 14 pro of only 7 scans. They were taken inside, outside, and mostly of flat surfaces or mildly undulating surfaces, but there was also one scan of a complex living room. This small and relatively similar set of data points led to excessive overfitting and does NOT work well with point clouds outside of the original scans. This is a nonissue for the time being as this experiment was merely proof of concept to ensure that the model and training worked as intended and I feel confident that it does and I’m ready to move on in the process of this research.

# Creating Synthetic Scans

To improve the number of training data we had access to we generated synthetic scans by applying various transformations and corruptions to the original LiDAR scans. These synthetic scans simulate real-world imperfections such as occlusions, missing data, and sensor noise that naturally occur during 3D scanning. This gives us the ability to synthetically simulate poor scans and train them against their former versions which are now the ground truth. In the future, by training the model on these augmented data, we aim to reduce overfitting to the limited original dataset and prepare the model to better handle diverse and noisy input point clouds. However, ironically, for right now it almost certainly was the direct cause of our overfitting. We created 10 slightly different point clouds for each function meaning with 4 functions we created 40 synthetic data sets for each original data set. By having so many synthetic data sets tailored to the same ground truth it was bound to create excessive overfitting.

What we are creating:

Synthetic scans are modified versions of the original point clouds that intentionally introduce realistic degradations, 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 we’re creating synthetic scans:

- 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.
- 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 research.
- 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.py and synthetic_file_creation.py actually work:

1. Functions.py
    a. Random_local_hole:
        i. Big picture:
            1. Create local holes (missing regions) by removing points inside random circles
        ii. Create a copy in “points” to not overwrite data
        iii. For each hole:
            1. Randomly select point in data and set it as the center of hole
            2. Compute Euclidean distance from each point to the center point
            3. Keep only points outside (removing all points on the inside)
                a. 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 (bit masking are fun problems on CSES!)
            4. Return points
    b. Random_global_dropout
        i. Simulate global sparsity or dropout by randomly removing a percentage of points anywhere in the point cloud
        ii. Calculate how many points to keep based on the dropout ratio in keep_num
        iii. Indices then randomly selects keep_num amount of points and return the new reduced point cloud
    c. Occlusion_plane:
        i. Simulate partial view occlusions by removing points on one side of a plane
            1. 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

ii. First it checks if no plane normal vector is provided and if not then generates a random 3d vector and normalizes it
iii. Then if no plane point is provided it choses a random point for the plane to lie on
iv. Creates another copy of the points
v. 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)
vi. Return all points on the positive side essentially cutting the cloud in half (sometimes more sometimes less)

d. Add_noise:
    i. Add random jitter noise to all points to mimic sensor measurement errors
        1. Noise means unwanted, random and slightly inaccurate variations in the data set caused by sensor errors
    ii. Simply uses the random.random function to created 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

2. Synthetic_file_creation:
    a. 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.
    b. The save_points_as_xys simply saves the corrupted data to the synthetic scans directory
    c. Generate_synthetic_scans:
        i. Goes through each of the 4 techniques and creates x amount
        ii. For i in range of the desired number of files to create(currently 10 but with more real scans we can hopefully continue to reduce this to 3-4):
            1. Created the new file using the functions.py and save it. That's it

d. Process_all_scans:
    i. Setting up the generate synthetic scans function by initializing the original scans that will be edited inside a for loop

# Training Model

1. Chamfer_Distance
    a. Simply a way to measure similarity between two point clouds
    b. There are other ways to measure loss but this was the easiest implementation and could change or be added to in the future for sure
    c. DATA INFORMATION
        i. Points1 and points2 are both tensors shaped [B, N, 3]
            1. Where B is batch size, N is NUM_POINTS, 3 = xyz coordinates
        ii. Unsqueeze x/y to prepare for pairwise distance subtraction calculations
            1. X reshaped to have size 1 dimension and y is similar but along a different axis
        iii. Torch.norm calculates the Euclidean distance for each pair
        iv. Then we find the average nearest neighbor distances in each direction with min_distance variables combined into a loss variable
    d. Simple metrics like MSE don’t work because pcd’s aren’t aligned nor ordered
    e. More simply put the Chamfer algo does:
        i. For each point in set A find the closest point in set B and measure distance
        ii. For each point in set B find the closest point in A and measure distance
        iii. Average those distances to symmetric similarity measure
    f. Adapted from classical geometry and shape matching algos and very commonly used as a loss function in other deep learning models
    g. Discovered as a practical and mathematically sound way to compared unordered point sets

2. Config
    a. Basic configuration file that holds global variables and information across an entire directory
    b. Only holds two variables as of now but there's always a chance we increase it's use in the future

3. Create_graphs
    a. This file does the conversion from a raw pcd (input as [N, 3] where N is NUM_POINTS and 3 is 'xyz' format) file to a graph data structure compatible with a graph neural network (GNN)
    b. Build edges using k-nearest neighbors (k-NN) with knn_graph which returns those k-nearest edges in edge_index
        i. K-NN algo
            1. For a given point P compute the distance from P to EVERY OTHER point in dataset and sort at the end
            2. Select top K smallest distances
    c. Each node is represented by a feature vector which is a fancy way of saying information describing it's location in the graph. For our use each node is described by a feature vector of 3 numbers being xyz. A different graph could be 6 characters of xyz rgb if it included color as well!
    d. Lastly create data object from PyTorch Geometric (PyG) which bundles node features and the graph structure into single object the GNN model requires
    e. Big picture
        i. Raw point clouds are just sets of points without any connectivity. Many GNN architectures need graphs where nodes are connected by edges and this connectivity captures local geometric relationships!
        ii. Turns a raw cloud of points into a graph structure, letting our GNN learn from local point neighborhoods instead of isolated points, enabling it to understand 3D shapes better.

4. Fetch_points
    a. Ensuring that no matter how many points are given by an object Points it will always return exactly NUM_POINTS in the output

(1024 currently but subject to increase once we have GPU access)

b. Points is again the tensor shaped [N, 3] where N is number of points and 3 is feature vector of 3 ‘xyz’ and NUM_POINTS is the global variable of desired number of points from config file the model expects in the input
c. If current object has MORE than NUM_POINTS it’s randomly downsampled by literally randomizing the order of the points object then selecting the first NUM_POINTS feature vectors given is newly ordered list
d. If object has LESS than NUM_POINTS we simply copy the last element until we get to NUM_POINTS. This feels like an awful strategy but it’s used occasionally because models USUALLY learn to ignore these duplicates if there’s enough real data in the res of the pcd. There are certainly more extravagant approaches that include random point duplication similar to our downsampling, zero-padding with a  mask, and adding some random small noise to added padding so they’re less obvious. Those aren’t required by our model because we typically have 100x the points required by NUM_POINTS and I highly doubt we come even CLOSE to having less than the required NUM_POINTS so the duplicating is merely a safety measure.

5. Fetching_files
    a. Simply a file to collect the pcd from original_scans for the ground truth (GT) and the synthetic files which were created from the GT’s and altered in a structured random way to replicate real LiDAR occlusions, noise, etc

6. GNN_autoencoder_model.py
    a. Core architecture of our model. It defines our GNN autoencoder which is the neural network that learns to compress and reconstruct point clouds
    b. Class GNNEncoder(nn.module):
        i. Takes point cloud GRAPH (after transformation in create_graphs) converts it to a compact embedding (vector representation)
        ii. Edge Convolution:

1. Convolution in general is the mathematical operation that aggregates information from a local neighborhood into a new value
2. Special convolution from Dynamic Graph Convolution Neural Network (DGCNN)
    a. Normal 2D Convolution NN (like for images) convolution slides a filter over a grid and collects information from spatially adjacent pixels
    b. Graphs have no fixed grid so EdgeConv or Edge Convolution is called “special” because it performs convolution over edges in a graph not fixed pixels so it learns based on relative differences between connected points rather than absolute differences
3. Works by looking at each point and k nearest neighbors
4. Measure differences between neighbors (local geometric relationships)
5. Updates point features to capture more abstract geometry

iii. Layer sizes:
    1. Conv1: input is 6 because each edge contains self and neighbors features where each is 3 ‘xyz’
    2. After conv1: 64
    3. After conv2: 128
    4. After conv3: 256
        a. Convolution1 (conv1) takes each point and processes those features (3 ‘xyz’) and outputs a new representation where each point has 64 channels
        b. A channel represents a different way the model has learned to describe the same pixel/point. So 64 channels means the model can describe a single point 64 different ways.
            i. For example, input features [x,y,z,r,g,b] and Conv1 learns 64 filters where each filter ouputs ONE number per point and now there are 64 channels describing

geometry, color patters, or relationships to neighbors per point.

ii. What's actually inside a channel?
    1. Each channel is one learned measurement across all points so Channel 5 might respond strongly if a point is part of flat surface, 12 might activate for sharp corners, 37 might capture color if you're including rgb, and channel 103 might capture some combination of geometry and texture

c. Conv2 takes the 64 features from Conv1 and outputs 128 feature per point. Etc
    i. The reason for increasing channels is early layers are generally capturing simple, local patterns (this point is near an edge) and later layers are capturing more abstract, high level patterns (this point is part of a chair leg) and more channels=more representational capacity (how much and how complex the model can store/express about the data it's processing)

iv. Global pooling:
    1. Number of points can vary so we do a global max pool to make a fixed-length representation per graph
    2. Pcd will obviously have immensely varying number of points and after fetch_points its stationary at 1024 NUM_POINTS so why do we reshape it to a fixed-length vector? Because when we compute inferences in the future after the model has been trained those pcd will most CERTAINLY have too many points
    3. We then embed conv3 back into 128 to shrink a fully connected layer (fc). This is done for compression, Efficiency, and Regularization (fewer dimensions) which can help with over fitting. Not technically necessary but pretty standard choice for embedding layers

c. Class GNNDecoder(nn.module):

i. The decoder takes the 128D embedding and attempts to regenerate the full point cloud (1024 NUM_POINTS)
ii. Fully connected layers (fc):
1. Output size is NUM_POINTS*3 for xyz coordinates for each point which reshapes to [batch_size, NUM_POINTS, 3]
2. This forces the model to predict exactly NUM_POINTS regardless of what the input had which is why we created the fetch_points file and the fixed_size_points() function specifically

d. Class GNNAutoencoder(nn.Module):
i. Combines the encoder and decoder
ii. Forward():
1. Data is a torch_geometric.data.Data object
a. X is the Node feature matrix with a shape of [num_nodes, num_node_features]
b. Edge_index is the graph connectivity list
c. Batch tells PyG which nodes belong to which graph when processing multiple pcd in the same batch
2. Given a graph is encoded into a compressed vector Z
3. And decodes that back into a reconstructed set points

e. Big picture:
i. The GNN encoder learns to capture local and global geometry via the EdgeConvolutions
ii. The decoder tries to rebuild the original point cloud
iii. Chamfer Distance is used as the loss function to compare reconstruction to original
iv. This model was designed to take a corrupted/incomplete synthetic scan and produce a repaired version that should match the GT

7. Train_loader
a. Creates a custom PyTorch Dataset class that loads paris of point clouds (the synthetic and original scans) to be compatible with PyTorch DataLoader to our training loop can iterate over it
i. Process each point into a graph representation for the synthetic points (in create_graph)

ii. Fixed-size point tensor for the original points so all samples in a batch are compatible

b. Class PointCloudDataset:
   i. Subclassing torch.utils.data.Dataset to PyT can call __len__ and __getitem__ when batching.
   ii. __init__
      1. Stores the given lists of point clouds in memory (separating GT and synthetic scans) along with K (number of Nearest Neighbors that is in our global config)
      2. It prepares raw data for later graph conversion and fixed-size processing
   iii. __len__
      1. Return the number of samples in the dataset and PyTorch uses this to know when to stop iterating ensuring our DataLoader can loop through the full data set safely
   iv. __getitem__
      1. This is the bridge from our raw point cloud data to train-ready tensors and graphs!
      2. Load synthetic points (input) for index and convert them to torch.float
      3. Normalize size with fixed_size_points (either randomly removing or duplicating last element)
      4. Convert to graph with create_graph
      5. Load original points (original_scan and GT) and apply the same fixing re-labeling it to target_points
      6. Add batch vector to the graph which is needed by PyT Geometric when combining multiple graphs into a batch
      7. Return input_graph (model input) and target points (GT for loss calculations)

8. Training_loop
   a. Big picture
      i. This runs the autoencoder training process by feeding batched graphs through the network and calculating the reconstruction error with chamfer loss. Then updates the

model weights and prints progress via average loss over epochs.

b. Model initializes the GNN and moves it to GPU or CPU and the optimizer updates our models' weights based on gradients where the learning rater (lr) controls the step size of weight updates
c. Training loop (real for loop)
    i. Runs 100 epochs (one full pass over our training dataset)
        1. Epoch literally just refers to one full pass over the training dataset it's not short for anything or any more complicated than that so if you have 10 scans and run 100 epochs then you visit each of the 10 scans 100 times.
    ii. Model.train() sets the model in training mode which is important for dropout, batch norm layers, etc
    iii. Total_loss will accumulate the loss for averaging later on which will determine if the model gets updated or not
    iv. Looping through training batches and target points for loop
        1. Train_loader (previous file) gives batches of input_graphs and target_points and moves them both to GPU/CPU
        2. Forward pass/Loss Computatoin
            a. Optimizer.zero resets gradients before backpropagation
                i. Backpropagation is the process the model uses to learn from mistakes. After computing the loss, it calculates gradients for every parameter (weight) in the model. Those gradients tell the model how to change each weight to reduce the loss. The gradients flow backward from the output through the layers which is how it was given the name backpropagation
            b. Output_points sends graphs through the model autoencoder and outputs reconstructed point cloud
            c. Chamfer_distance measures how close two point clouds are and act as our loss measurement
        3. Backpropagation

a. Loss.backward computs gradients (backpropagation)
b. Omptimizer updates model weights based on the gradients calculated in loss.backward (if the gradient says the weight should be increased this step will increase it by some amount controlled by the learning rate) (also how the model changes/improves)
c. Loss.item gets the scalar loss value. This allows us to print the progress of the model and allows us the opportunity to update the model when the loss has improved over time. For example, you'll see later that we literally update the model in our file when the average loss is less than the previous best loss value that way our model only ever gets more accurate (according to chamfer distance that is)

9. Main
    a. The file that compiles everything else. This is the training script for our GNN autoencoder model that handles loading the data, setting up the model/optimizer, optionally resuming from a saved checkpoint, running training over multiple epochs, computing loss, and saving the best model
    b. Simple functions of fetch_best_loss, save_best_loss, and load_checkpoint help keep the loss from previous models so if the user wanted to continue the training from a previous model with new training data they could ensure the model won't be updated/affected until it's improved upon it's previous versions!
    c. Main():
        i. Instantiates our GNN autoencoder model and optimizer, loads the best loss from file, sets an initial threshold of 0.1 (will dynamically adjust later on) for saving new model if loss improves sufficiently
        ii. Asks the user if the want to resume training from last checkpoint. If not then best loss resets and training starts from stratch
        iii. Runs the epoch loop

iv. Epoch loop:
    1. Within each loop we set the model to train mode and loop over each batch in train loader
    2. For input/target in train_loader:
        a. Moves data to device
        b. Clears previous gradients (optimizer.zero_grad)
        c. Feeds input graph into the model to get reconstructed output points
        d. Computes chamfer distance loss between output and target points
        e. Calls loss.backward to compute gradients via backpropagation
        f. Calls optimizer.step to update model parameters
        g. Accumulates loss for this specific epoch
        h. After all batches, calculates epoch loss and prints it
        i. Saving the best model:
            i. If the average loss improves beyond the currest best by at least loss_threshold it updates the best loss, saves new best loss to best_loss file, dynamically updates the loss_threshold to 10% of current average loss, and saves the model checkpoint

# Model Inferencing

10. Inferencing.py
    a. Big picture
        i. Loads trained GNN autoencoder from saved checkpoint in the ptf file
        ii. Loads file to be put through the model to create the inpaintings
        iii. Saves the reconstructed point clouds after inpainting
        iv. This file is supposed to show off what the model has learned in training

b. Load_checkpoint:
    i. Load the saved model from device and move it to CPU or GPU
    ii. Set .eval mode so no dropout/batchnorm randomness happen
c. Load_points_np:
    i. Converts point cloud to numpy float32 for PyTorch
d. Fixed_size_points_np:
    i. Ensures all point clouds are same size for batch processing
    ii. Same as function before (if too many points downsample if too few create dplicates of last element)
e. Make_data_from_np:
    i. Determines number of neighbors (k_effective) for graph edges
    ii. Converts NumPy to a PyTorch tensor
    iii. Calls create_graph_from_point_cloud to create GNN graph
    iv. Adds .batch vector for PyTorch geometric
f. Save_points_np:
    i. Simply saves the points to device as text
g. Run_inference_on_file:
    i. Loads and preprocesses the point cloud to be inpainted
    ii. Runs it through the trained model without gradient tracking
        1. Torch.no_grad to make it faster and use less memory
    iii. Converts model output to NumPy
    iv. Saves reconstructed cloud in the inpainting directory

# Git Ignore

Lastly, to keep this short and sweet, the only things not included in the github repo are the synthetic scans and reconstructed point clouds after inferences simply because there’s too many files that are too large. It would take a substantial amount of time to upload those files to github as well as when a new user is cloning the repository. The last thing excluded from the project is the gnn_autoencoder.ptf file which is the model itself. That file is also very large unfortunately, however, all original scans are included so any user can run the training themselves and create their own model and use the repo the same way I can!