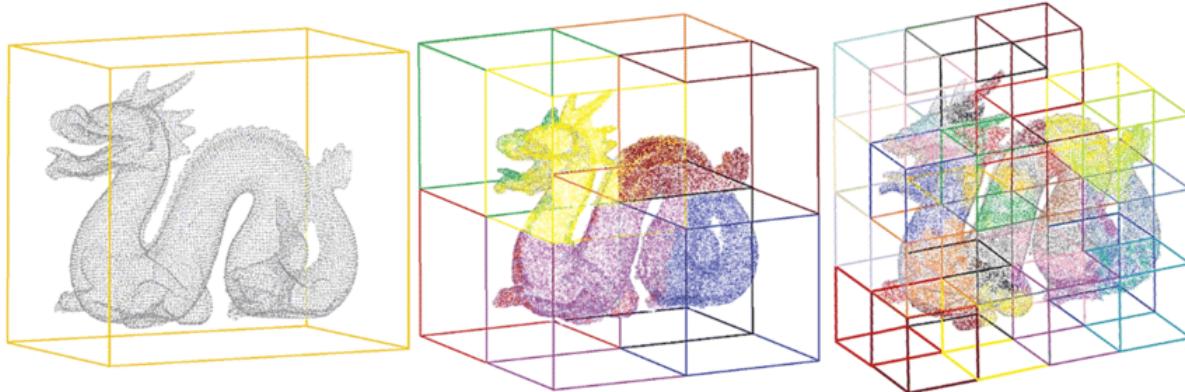


# Point Cloud Techniques and Applications

Tuan Dang and Phuc Pham

Ngày 23 tháng 4 năm 2024



## 1 Introduction to Point Cloud and basic techniques to process

Trong phần này chúng ta sẽ được giới thiệu về một số thao tác cơ bản trên 3D Point Clouds data bằng thư viện Open3D.

### 1.1 Cài đặt môi trường

Để tránh bị xung đột về thư viện các bạn hãy cài đặt môi trường bằng các câu lệnh sau đây:

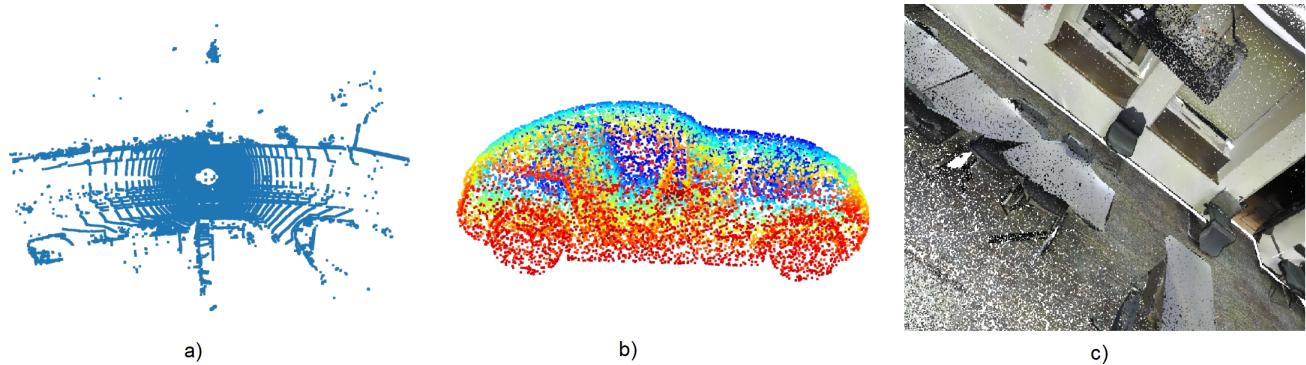
```
1 conda create -n 3d_pc python=3.8
2 conda activate 3d_pc
3 pip install open3d==0.18.0
4 pip install seaborn
5 pip install opencv-python
```

### 1.2 Load a point cloud

Có các loại file lưu trữ một đám mây điểm 3D thông dụng đó là file có đuôi .bin, .ply và .txt. Đây chính là code mẫu để load data với ".bin" file từ bộ data KITTI, ".ply" từ bộ PCN và ".txt" từ bộ S3DIS.

```
1 def load_bin(path):
2     pcd_arr = np.fromfile(path, dtype=np.float32).reshape(-1, 4) # x,y,x,r
3     return pcd_arr[:, :3], pcd_arr[:, 3:]
4
5 def load_ply(path):
6     pcd      = o3d.io.read_point_cloud(path)
7     pcd_arr = np.asarray(pcd.points)
8     pcd_color = np.asarray(pcd.colors)
9     return pcd_arr, pcd_color
10
11 def load_txt(path):
12     pcd_arr = np.loadtxt(path)
13     return pcd_arr[:, :3], pcd_arr[:, 3:]
```

Sau đó chúng ta sẽ sử dụng hàm draw\_point\_clouds để visualize data.



Hình 1: Hình biểu diễn 3D point clouds sau khi load data. Point cloud từ bộ a) KITTI, b) PCN, c) S3DIS.

```

1 def draw_point_clouds(array, color = None, logits_color = False, name = "Open3D"):
2     """
3         Visualize 3D point clouds from an 3D point cloud array and its corresponding color
4         array.
5     Args:
6         array (np.array): a point clouds (N,3)
7         color (np.array, optional): color of the points. Defaults to None.
8         logits_color (bool, optional): flags if it is True, size of input color must
9             be (N,3). Otherwise, size of color must be (N,1). Defaults to False.
10        name (str, optional): name of display window. Defaults to "Open3D".
11    """
12    pcd = o3d.geometry.PointCloud()
13    pcd.points = o3d.utility.Vector3dVector(array)
14    if color is not None:
15        if logits_color == False:
16            colorLength = abs(np.max(color)) + 1
17            colorPalette = sns.color_palette("Paired", int(colorLength))
18            colorArray = np.array(colorPalette)
19            color = colorArray[color]
20            pcd.colors = o3d.utility.Vector3dVector(color)
21        else:
22            if np.max(color) > 1:
23                color = color/255.0
24            pcd.colors = o3d.utility.Vector3dVector(color)
25
26    o3d.visualization.draw_geometries([pcd], window_name = name)

```

Kết quả sau được trực quan hóa như hình 1.

### 1.3 Downsampling

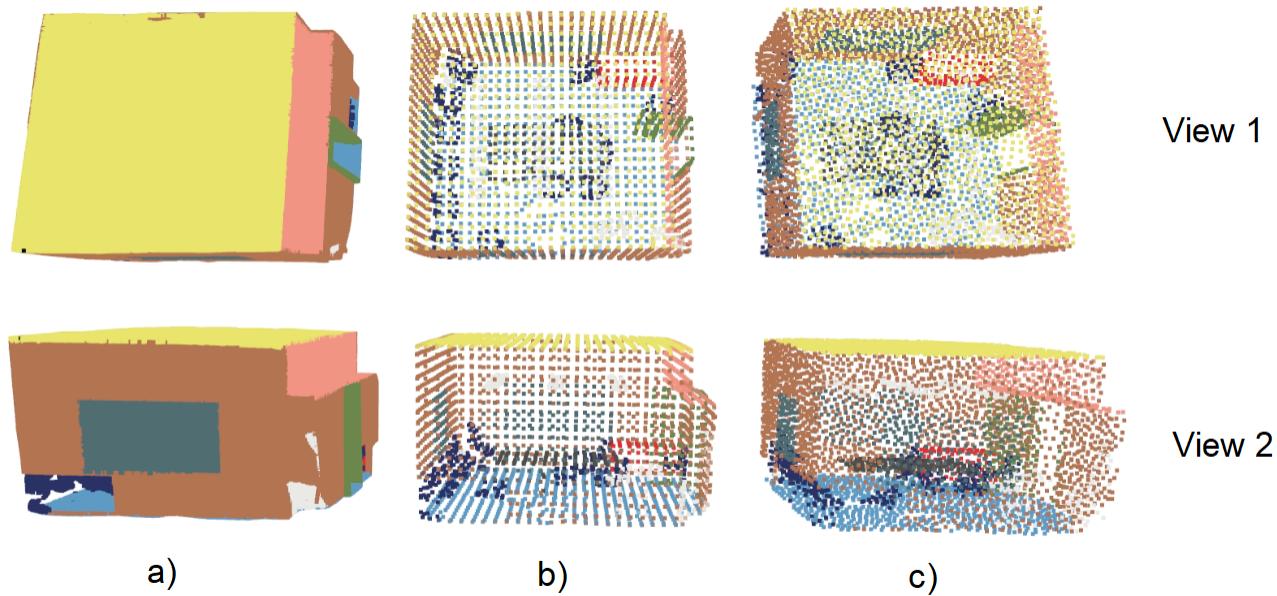
Dữ liệu 3D point cloud cung cấp thông tin chi tiết về hình dạng và kích thước của các đối tượng trong không gian 3D. Tuy nhiên, lượng dữ liệu lớn này có thể gây tốn kém về tính toán và lưu trữ. Giảm mẫu (Downsampling) là kỹ thuật then chốt giúp giữ lại thông tin quan trọng giảm thiểu kích thước của dữ liệu điểm 3D mà vẫn bảo toàn các đặc trưng cần thiết. Có 2 phương pháp phổ biến: voxel\_grid\_sampling và farthest\_point\_sampling.

#### Phân ô voxel (Voxel Grid Sampling):

Ý tưởng: Chia không gian 3D thành các ô nhỏ có kích thước bằng nhau, gọi là voxel.

Giảm mẫu:

Chọn một điểm đại diện (ví dụ như trọng tâm) cho mỗi voxel. Các điểm nằm trong cùng voxel sẽ được



Hình 2: Hình so sánh point cloud qua các phương pháp sampling a) original point cloud, b) voxel grid sampling và c) farthest point sampling.

thay thế bằng điểm đại diện này.

Ưu điểm:

- Giữ được thông tin về mật độ điểm của dữ liệu gốc.
- Kiểm soát được mức độ giảm mẫu chính xác.
- Dễ dàng thực hiện và hiệu quả về tính toán.

Nhược điểm:

- Có thể làm mất các chi tiết quan trọng, đặc biệt là các chi tiết ở ranh giới giữa các voxel.
- Hiệu quả giảm sút với dữ liệu điểm phân bố không đều.

#### Lấy mẫu xa nhất (Farthest Point Sampling - FPS):

Ý tưởng: Lặp lại việc chọn điểm xa nhất so với các điểm đã chọn trước đó.

Giảm mẫu: Bắt đầu với một điểm ngẫu nhiên, sau đó chọn điểm có khoảng cách lớn nhất đến tất cả các điểm đã chọn trước đó. Lặp lại quá trình này cho đến khi đạt được số lượng điểm mong muốn.

Ưu điểm:

- Giữ được phân bố tương đối đều của các điểm trên đám mây 3D, bảo toàn tốt các chi tiết.
- Hiệu quả với cả dữ liệu điểm phân bố đều và không đều.

Nhược điểm:

- Không trực tiếp kiểm soát được mức độ giảm mẫu.
- Có thể tính toán tốn thời gian hơn so với phân ô voxel.

Kết quả của hai phương pháp sampling được biểu diễn qua hình 2.

```

1 def voxel_grid_sampling(pcd, voxel_size = 0.05):
2     downpcd = pcd.voxel_down_sample(voxel_size=voxel_size)
3     return downpcd
4
5 def farthest_point_sampling(pcd, num_points = 100):
6     downpcd = pcd.farthest_point_down_sample(num_samples=num_points)
7     return downpcd

```

## 1.4 Noise removal

Dữ liệu 3D point clouds đóng vai trò quan trọng trong nhiều lĩnh vực như khảo sát, lập bản đồ, robot, v.v. Tuy nhiên, dữ liệu này thường bị ảnh hưởng bởi nhiễu, gây ra sai sót trong việc xử lý và phân tích. Lọc nhiễu (noise removal) là bước quan trọng để làm sạch dữ liệu, giúp tăng độ chính xác và khai thác tối đa tiềm năng của data.

### Loại bỏ nhiễu thống kê (Statistic Noise Removal):

Ý tưởng: Sử dụng các tính chất thống kê của dữ liệu điểm đám 3D để phân biệt điểm nhiễu với các điểm thực tế.

Thực hiện:

1. Tính toán các thống kê của các điểm xung quanh một điểm, chẳng hạn như giá trị trung bình, độ lệch chuẩn, mật độ điểm.
2. So sánh các giá trị thống kê của điểm đang xét với các điểm xung quanh.
3. Xác định điểm là nhiễu nếu các giá trị thống kê của nó quá khác biệt so với các điểm xung quanh.

Ưu điểm:

- Có thể loại bỏ nhiễu đa dạng, không phụ thuộc vào dạng hình học của nhiễu.
- Hiệu quả với cả nhiễu Gauss (nhiễu theo phân phối chuẩn) và nhiễu xung (nhiễu đột biến).

Nhược điểm:

- Cần lựa chọn các tham số ngưỡng phù hợp để phân biệt điểm nhiễu.
- Có thể loại bỏ nhầm các điểm nằm trên vùng biên của đối tượng.

### Loại bỏ nhiễu bán kính (Radius Noise Removal):

Ý tưởng: Loại bỏ các điểm nằm xa hơn một khoảng cách nhất định (bán kính) so với các điểm lân cận của chúng.

Thực hiện:

1. Xác định một giá trị bán kính nhất định.
2. Tìm kiếm các điểm lân cận của mỗi điểm trong đám mây điểm.
3. Loại bỏ các điểm có số lượng điểm lân cận trong bán kính quy định nhỏ hơn một giá trị ngưỡng.

Ưu điểm:

- Đơn giản dễ hiểu và dễ dàng thực hiện.
- Hiệu quả với các nhiễu dạng điểm đơn lẻ hoặc cụm nhiễu nhỏ.

Nhược điểm:

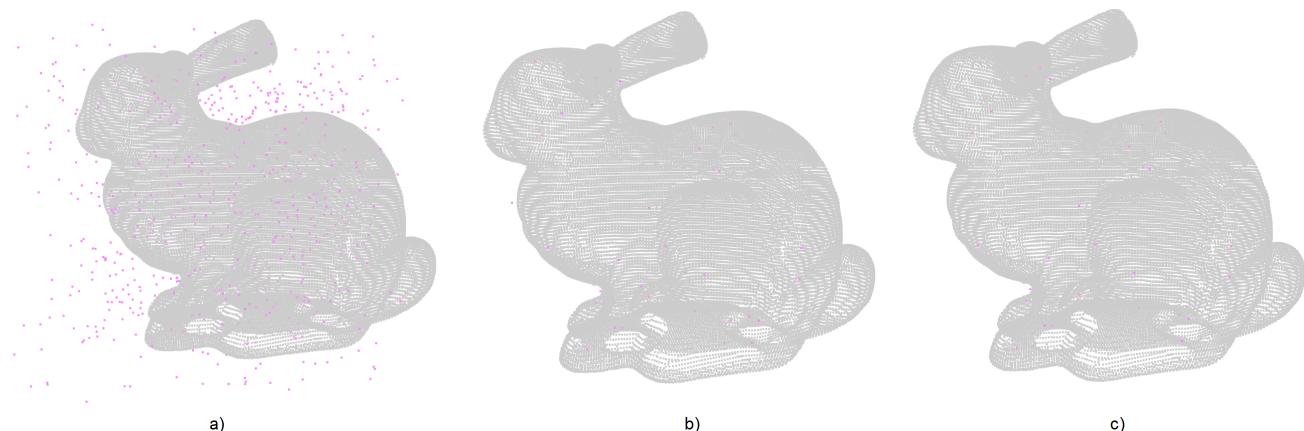
- Khó lựa chọn bán kính phù hợp cho các trường hợp nhiễu khác nhau.
- Có thể loại bỏ nhầm các điểm nằm trên chi tiết nhỏ của đối tượng.

Hình ảnh 3 mô tả hiệu quả của noise removal trên một đám point cloud bị nhiễu.

```

1 def statistic_outlier_removal(pcd, nb_neighbors=10, std_ratio=1.0):
2     _, ind = pcd.remove_statistical_outlier(nb_neighbors=nb_neighbors,
3                                             std_ratio=std_ratio)
4
5     display_inlier_outlier(pcd, ind)
6     inlier_cloud = pcd.select_by_index(ind)
7     o3d.visualization.draw_geometries([inlier_cloud], "Statistical oulier removal")
8     return inlier_cloud
9
10
11 def radius_outlier_removal(pcd, nb_points=16, radius=0.05):
12     _, ind = pcd.remove_radius_outlier(nb_points=nb_points, radius=radius)
13     display_inlier_outlier(pcd, ind)
14     inlier_cloud = pcd.select_by_index(ind)
15     o3d.visualization.draw_geometries([inlier_cloud], "Radius oulier removal")
16     return inlier_cloud

```



Hình 3: Hình ảnh mô tả các phương pháp khử nhiễu trên point cloud. a) Noise, b) Radius noise removal với radius = 0.5, c) Statistical noise removal với standard ratio = 0.5 và number of neighbor = 10.

## 1.5 Nearest neighbor search

Dám mây điểm có cấu trúc không đều (irregular). Trong khi đó, vùng lân cận cục bộ của các pixel trong ảnh 2D có thể dễ dàng được xác định bằng cách tạo ra một lưới xung quanh pixel, thì dám mây điểm không có biểu diễn dựa trên lưới tự nhiên và việc xây dựng lưới là không đơn giản. Thay vào đó, tìm kiếm nearest neighbor (NN) đóng vai trò là yếu tố cơ bản để xây dựng các vùng lân cận cục bộ cho các điểm trong dám mây điểm. Tìm kiếm NN được sử dụng trong các tác vụ loại bỏ nhiễu bán kính (radius\_outlier\_removal), loại bỏ nhiễu thống kê (statistic\_outlier\_removal) như được mô tả trong Phần 1.4, để tính toán các đặc điểm cục bộ cho mỗi điểm với vùng lân cận cục bộ của nó.

Trong lĩnh vực xử lý dữ liệu điểm 3D, hai cấu trúc dữ liệu quan trọng thường được sử dụng để tìm kiếm lân cận (Search Neighbor) hiệu quả là cây KD (KD-Tree) và Octree.

### Cây KD (KD-Tree)

**Khái niệm:** Cây KD là một cây tìm kiếm nhị phân đa chiều được sử dụng để lưu trữ dữ liệu điểm trong không gian đa chiều (ví dụ như không gian 3D). Nó phân chia lặp lại không gian thành các vùng con dựa trên các trục tọa độ (X, Y, Z).

**Hoạt động:** Khi tìm kiếm lân cận của một điểm truy vấn, cây KD sẽ hướng dẫn tìm kiếm theo các trục tọa độ, loại bỏ hiệu quả các vùng không gian không thể chứa điểm lân cận.

**Ưu điểm:**

- Tìm kiếm lân cận nhanh chóng, đặc biệt hiệu quả với dữ liệu điểm phân bố đều.
- Sử dụng bộ nhớ hiệu quả.

**Nhược điểm:**

- Hiệu quả tìm kiếm có thể giảm với dữ liệu điểm phân bố không đều.
- Cấu trúc cây có thể phức tạp đối với dữ liệu chiều cao.

### Octree

**Khái niệm:** Octree là một cây tìm kiếm nhị phân được sử dụng để lưu trữ dữ liệu điểm trong không gian 3D. Nó phân chia lặp lại không gian thành các khối lập phương (octant) bằng nhau.

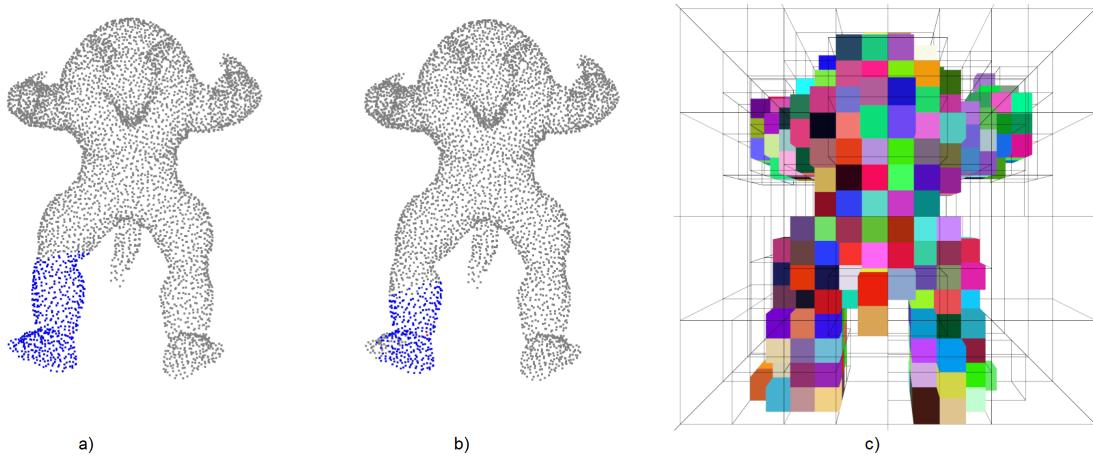
**Hoạt động:** Tương tự như cây KD, Octree hướng dẫn tìm kiếm lân cận bằng cách loại bỏ các octant không thể chứa điểm lân cận.

**Ưu điểm:**

- Hiệu quả tốt với dữ liệu điểm phân bố không đều.
- Cấu trúc đơn giản và dễ dàng truy cập dữ liệu theo vùng.

**Nhược điểm:**

- Có thể sử dụng nhiều bộ nhớ hơn so với cây KD cho cùng một lượng dữ liệu điểm.



Hình 4: Nearest neighbor search bằng KD-Tree và Octree. a) KD-Tree KNN với  $k = 200$ , b) KD-Tree Radius với  $\text{radius} = 0.02$ , c) Octree với  $\text{max depth} = 4$ .

- Không hiệu quả bằng cây KD với dữ liệu điểm phân bố đều.

```

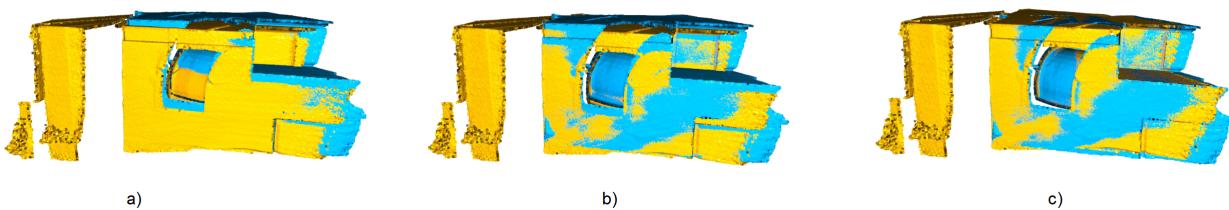
1 def kd_tree_nearest_neighbor_knn(pcd, point_index = 5, number_of_neighbor = 200):
2     pcd.paint_uniform_color([0.5, 0.5, 0.5])
3     pcd_tree = o3d.geometry.KDTreeFlann(pcd)
4     pcd.colors[point_index] = [1, 0, 0]
5     [k, idx, _] = pcd_tree.search_knn_vector_3d(pcd.points[point_index],
6         number_of_neighbor)
6     np.asarray(pcd.colors)[idx[1:], :] = [0, 0, 1]
7     o3d.visualization.draw_geometries([pcd], "Kd_tree_nearest_neighbor_knn")
8
9
10 def kd_tree_nearest_neighbor_radius(pcd, point_index = 5, radius = 0.02):
11     pcd.paint_uniform_color([0.5, 0.5, 0.5])
12     pcd_tree = o3d.geometry.KDTreeFlann(pcd)
13     pcd.colors[point_index] = [1, 0, 0]
14     [k, idx, _] = pcd_tree.search_radius_vector_3d(pcd.points[point_index], radius)
15     np.asarray(pcd.colors)[idx[1:], :] = [0, 0, 1]
16     o3d.visualization.draw_geometries([pcd], "Kd_tree_radius_knn")
17
18
19 def octree_nearest_neighbor(pcd, point_index = 5, max_depth = 4, size_expand=0.01):
20     pcd.colors = o3d.utility.Vector3dVector(np.random.uniform(0, 1, size=(np.asarray(
21         pcd.points).shape[0], 3)))
22     octree = o3d.geometry.Octree(max_depth=max_depth)
23     octree.convert_from_point_cloud(pcd, size_expand=size_expand)
24     o3d.visualization.draw_geometries([octree], f"Octree_depth_{max_depth}")
25     octree.traverse(f_traverse)
26     octree.locate_leaf_node(pcd.points[point_index])

```

Kết quả sau khi chạy code trên với các tham số như hình 4.

## 1.6 Registration

Point cloud registration là một kỹ thuật liên quan đến việc sắp xếp nhiều tập điểm 3D, được gọi là đám mây điểm, vào một hệ tọa độ duy nhất. Hãy tưởng tượng bạn có hai bản quét khác nhau của cùng một căn phòng từ các góc nhìn hơi khác nhau. Đăng ký đám mây điểm giúp bạn chồng chéo các bản quét



Hình 5: Hình ảnh so sánh kết quả registration của 2 đám mây điểm. a) Original (inlier\_rmse = 1.177e-02), b) Point-to-point ICP (inlier\_rmse = 7.76e-03), c) Point-to-plane ICP (inlier\_rmse = 6.58e-03)

này để tạo ra mô hình 3D hoàn chỉnh và chính xác của căn phòng.

**Point-to-Point ICP:** Đây là cách tiếp cận đơn giản và truyền thống hơn. Nó tập trung vào việc tìm điểm gần nhất trong đám mây điểm đích cho mỗi điểm trong đám mây điểm nguồn. Khoảng cách giữa các điểm gần nhất này sau đó được thu nhỏ để tinh chỉnh lặp lại phép biến đổi (quay và dịch chuyển) căn chỉnh hai đám mây.

**Point-to-plane ICP:** Phương pháp này xem xét thông tin bề mặt của đám mây điểm đích. Nó tính toán vectơ pháp tuyến (vuông góc với bề mặt) cho mỗi điểm trong đám mây đích. Thay vì chỉ đơn giản tìm điểm gần nhất, nó chiếu vectơ chênh lệch (giữa điểm nguồn và láng giềng gần nhất của nó) lên vectơ pháp tuyến của điểm đích. Điều này hiệu quả làm giảm khoảng cách vuông góc giữa điểm nguồn và bề mặt đích, dẫn đến sự căn chỉnh chính xác hơn cho các bề mặt cong.

```

1 def point_to_point_ICP(source, target, threshold, trans_init):
2     reg_p2p = o3d.pipelines.registration.registration_icp(
3         source, target, threshold, trans_init,
4         o3d.pipelines.registration.TransformationEstimationPointToPoint())
5     draw_registration_result(source, target, reg_p2p.transformation, "Point to point
6     ICP")
7     return reg_p2p.transformation
8
9 def point_to_plane_ICP(source, target, threshold, trans_init):
10    reg_p21 = o3d.pipelines.registration.registration_icp(
11        source, target, threshold, trans_init,
12        o3d.pipelines.registration.TransformationEstimationPointToPlane())
13    draw_registration_result(source, target, reg_p21.transformation, "Point to plane
14    ICP")
15    return reg_p21.transformation

```

Sau khi registration, hai đám point clouds đạt được độ chồng chéo cao hơn đáng kể so với trước đó. Điều này thể hiện rõ ràng qua việc giảm dần giá trị inlier\_rmse từ bản gốc đến Point-to-Point ICP và đạt mức thấp nhất với Point-to-Plane ICP hình 6.

## 1.7 RGB-D Reconstruction

RGB-D Reconstruction (Tái tạo 3D từ RGB-D) là kỹ thuật tạo ra mô hình 3D của một cảnh vật sử dụng dữ liệu thu thập từ camera RGB-D.

Camera RGB-D là loại camera đặc biệt thu thập đồng thời hai loại dữ liệu:

- Ảnh RGB: Ảnh màu thông thường cung cấp thông tin thị giác về cảnh vật.
- Ảnh độ sâu: Ảnh này mã hóa khoảng cách đến từng điểm trong cảnh vật từ vị trí camera.

Quá trình tái tạo: Ảnh RGB cung cấp chi tiết như màu sắc và kết cấu. Ảnh độ sâu cung cấp thông tin

về cấu trúc 3D của cảnh vật. Kết hợp hai nguồn dữ liệu này, các thuật toán có thể ước tính hình học 3D của cảnh vật.

Lợi ích của RGB-D Reconstruction:

- Cung cấp thông tin 3D phong phú: Mang lại hình ảnh toàn diện hơn so với chỉ sử dụng ảnh RGB hoặc ảnh độ sâu riêng lẻ.
- Ứng dụng rộng rãi: Được sử dụng trong các lĩnh vực như robot (điều hướng, thao tác), thực tế tăng cường (AR), thực tế ảo (VR) và mô hình 3D.

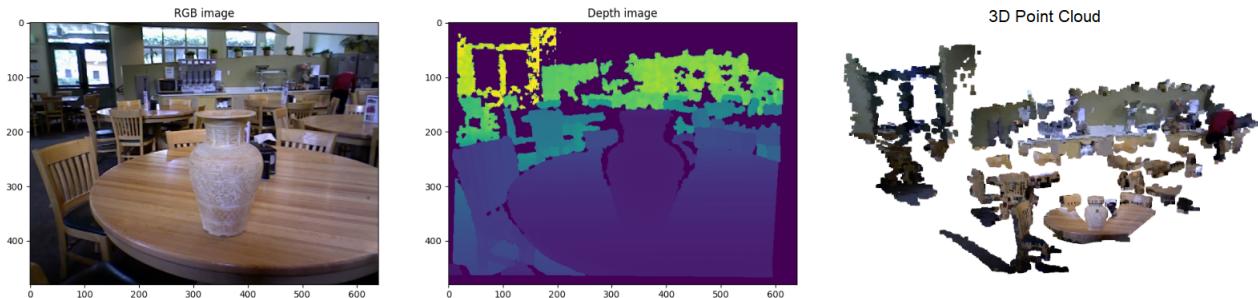
Thách thức của RGB-D Reconstruction:

- Hạn chế của cảm biến: Cảm biến độ sâu có thể bị nhiễu, đặc biệt đối với các bề mặt phản xạ hoặc trong suối.
- Độ phức tạp tính toán: Các thuật toán cần kết hợp và giải thích hiệu quả dữ liệu RGB và độ sâu.

```

1 def get_xyz_from_pts(u, v, depth, cx=319.5, cy=239.5, fx=525.0, fy=525.0):
2     d = depth[int(v), int(u)] # height, width in depth image
3     x = ((u - cx) / fx) * d
4     y = ((v - cy) / fy) * d
5     return np.array([x, y, d]).transpose()
6
7 def show_rgbd(rgb, depth):
8     plt.subplot(1, 2, 1)
9     plt.title('Grayscale image')
10    plt.imshow(rgb)
11    plt.subplot(1, 2, 2)
12    plt.title('Depth image')
13    plt.imshow(depth)
14    plt.show()
15
16 def np_to_pc(points, colors):
17     pcd = o3d.geometry.PointCloud()
18     pcd.points = o3d.utility.Vector3dVector(points)
19     pcd.colors = o3d.utility.Vector3dVector(colors)
20     return pcd
21
22 def get_pcl(name = "0"):
23     rgb = cv2.cvtColor(cv2.imread(f"./data/rgbd/{name}.jpg", cv2.IMREAD_UNCHANGED),
24     cv2.COLOR_BGR2RGB) #read rgb
25     depth = cv2.imread(f"./data/rgbd/{name}.png", cv2.IMREAD_UNCHANGED)
26     h, w = depth.shape
27     N = w*h
28     points = np.zeros((N,3))
29     colors = np.zeros((N,3))
30     index = 0
31     for u in range(w):
32         for v in range(h):
33             points[index,:] = get_xyz_from_pts(u, v, depth)
34             colors[index,:] = rgb[v,u]/255.0
35             index += 1
36     pcd = np_to_pc(points, colors)
37     pcd.transform([[1, 0, 0, 0], [0, -1, 0, 0], [0, 0, -1, 0], [0, 0, 0, 1]])
38     show_rgbd(rgb, depth)
39     o3d.visualization.draw_geometries([pcd], window_name = name)

```



Hình 6: Bên trái: Camera RGB-D thu được hai bức ảnh: ảnh màu sắc (RGB) và ảnh chiều sâu (Depth). Bên phải: Từ hai bức ảnh này, ta có thể tái tạo đám mây điểm 3D, mô tả hình dạng và vị trí của các vật thể trong không gian một cách chi tiết.

## 2 Machine Learning with Point Cloud

Trong phần này, chúng ta sẽ đi tìm hiểu về bài toán point clouds classification. Chúng ta sẽ sử dụng PointNet làm model để phân loại các vật thể 3D trong tập data ShapeNet.

### 2.1 Data preparation

Đầu tiên chúng ta sẽ tải bộ Shapenet bằng câu lệnh sau:

```

1 dataset_url = "https://git.io/JiY4i"
2
3 dataset_path = keras.utils.get_file(
4     fname="shapenet.zip",
5     origin=dataset_url,
6     cache_subdir="datasets",
7     hash_algorithm="auto",
8     extract=True,
9     archive_format="auto",
10    cache_dir="datasets",
11 )

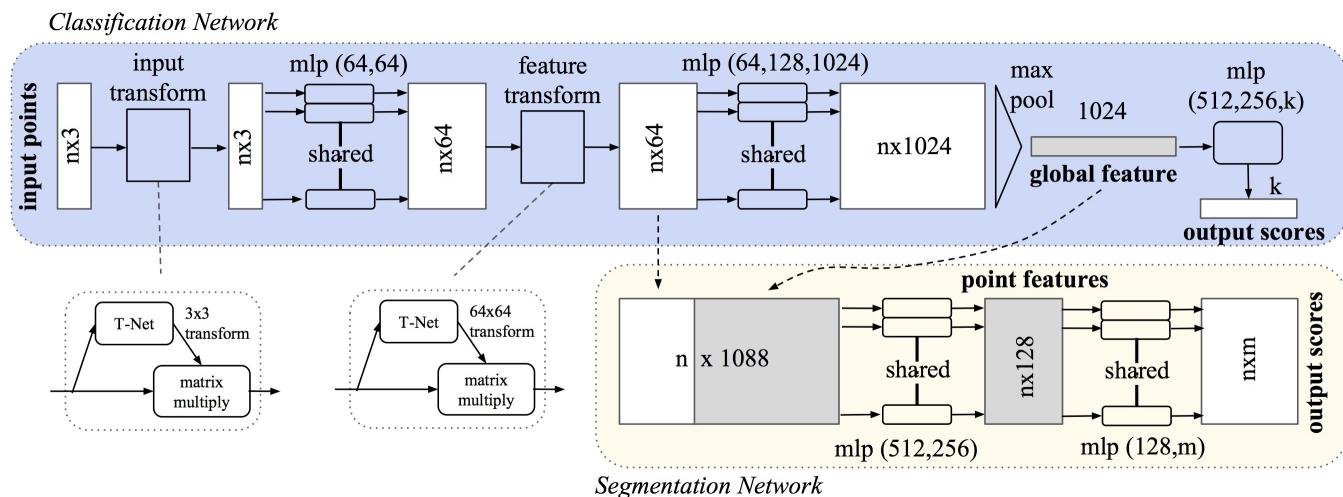
```

Trong thí nghiệm này để kiểm chứng kết quả thì ta chỉ lấy 3 classes trong tập Shapenet và mỗi class lấy tối đa 500 vật:

```

1 point_clouds      = []
2 dense_labels     = []
3 code_index_mapping = {}
4 for i, class_name in enumerate(dict_mapping):
5     point_files = glob(f'/tmp/.keras/datasets/PartAnnotation/{class_name}/points/*')
6     num_object = 0
7     for file in point_files:
8         point_cloud = np.loadtxt(file)
9         if point_cloud.shape[0] < NUM_SAMPLE_POINTS:
10             continue
11         if num_object == 500:
12             break
13         shuffle = np.random.choice(point_cloud.shape[0], NUM_SAMPLE_POINTS, replace =
14             False)
15         point_cloud = point_cloud[shuffle]
16         point_clouds.append(point_cloud)
17         dense_labels.append(i)
18         code_index_mapping[i] = dict_mapping[class_name]
19         num_object += 1

```



Hình 7: Mô hình thiết kế của kiến trúc Pointnet

Tạo dataloader:

```

1 point_clouds = np.stack(point_clouds)
2 dense_labels = np.stack(dense_labels)
3 # label_clouds = keras.utils.to_categorical(dense_labels, num_classes=3)
4 label_clouds = dense_labels
5 class PointCloudData(Dataset):
6     def __init__(self, point_clouds, label_clouds):
7         self.point_clouds = point_clouds
8         self.label_clouds = label_clouds
9     def __len__(self):
10        return point_clouds.shape[0]
11    def __getitem__(self, idx):
12        return {'pointcloud': torch.from_numpy(point_clouds[idx]),
13                'category': torch.from_numpy(np.asarray(label_clouds[idx]))}
14 X_train, X_val, y_train, y_val = train_test_split(point_clouds, label_clouds,
15 test_size=0.2, random_state=42)
16 train_loader = PointCloudData(X_train,y_train)
17 val_loader = PointCloudData(X_val,y_val)
18 train_loader = DataLoader(dataset=train_loader, batch_size=32, shuffle=True)
19 val_loader = DataLoader(dataset=val_loader, batch_size=32, shuffle=True)

```

## 2.2 Model

Trong phần này chúng ta sẽ hiện thực lại kiến trúc Pointnet như hình 7.

### 2.2.1 T-Net

Trong PointNet, T-net đóng vai trò quan trọng trong việc đạt được tính bất biến hướng của đám mây điểm. T-net là một mạng con bên trong PointNet, có chức năng học một ma trận biến đổi. Ma trận biến đổi này sau đó được áp dụng cho từng điểm trong đám mây điểm, về cơ bản là xoay các điểm. Bằng cách học phép biến đổi này, mạng có thể xử lý các đám mây điểm được trình bày từ các góc nhìn khác nhau.

Tính ổn định và khởi tạo:

Nếu T-net được khởi tạo với các giá trị bằng 0, thì phép biến đổi ban đầu sẽ là phép biến đổi null (không thay đổi). Điều này có thể dẫn đến mất ổn định khi huấn luyện, đặc biệt là trong giai đoạn đầu. Ma trận đơn vị biểu diễn phép biến đổi đồng nhất, về cơ bản giữ nguyên các điểm. Sử dụng ma trận

đơn vị để khởi tạo cung cấp một điểm bắt đầu ổn định cho T-net để học các phép biến đổi có ý nghĩa. Nó cho phép T-net dần dần lệch khỏi phép biến đổi đồng nhất khi học. Điều này giúp việc học diễn ra trơn tru hơn và tránh những thay đổi đột ngột trong quá trình huấn luyện.

```

1  class Tnet(nn.Module):
2      def __init__(self, k=3):
3          super().__init__()
4          self.k=k
5          self.conv1 = nn.Conv1d(k,64,1)
6          self.conv2 = nn.Conv1d(64,128,1)
7          self.conv3 = nn.Conv1d(128,1024,1)
8          self.fc1 = nn.Linear(1024,512)
9          self.fc2 = nn.Linear(512,256)
10         self.fc3 = nn.Linear(256,k*k)
11
12         self.bn1 = nn.BatchNorm1d(64)
13         self.bn2 = nn.BatchNorm1d(128)
14         self.bn3 = nn.BatchNorm1d(1024)
15         self.bn4 = nn.BatchNorm1d(512)
16         self.bn5 = nn.BatchNorm1d(256)
17
18
19     def forward(self, input):
20         # input.shape == (bs,n,3)
21         bs = input.size(0)
22         xb = F.relu(self.bn1(self.conv1(input)))
23         xb = F.relu(self.bn2(self.conv2(xb)))
24         xb = F.relu(self.bn3(self.conv3(xb)))
25         pool = nn.MaxPool1d(xb.size(-1))(xb)
26         flat = nn.Flatten(1)(pool)
27         xb = F.relu(self.bn4(self.fc1(flat)))
28         xb = F.relu(self.bn5(self.fc2(xb)))
29
30         #initialize as identity
31         init = torch.eye(self.k, requires_grad=True).repeat(bs,1,1)
32         if xb.is_cuda:
33             init=init.cuda()
34         matrix = self.fc3(xb).view(-1,self.k,self.k) + init
35         return matrix

```

## 2.2.2 Transform

Module Transform thực hiện thao tác biến đổi bằng T-Net 2.2.1 hai lần. Lần thứ nhất trong không gian 3 chiều (coordinate) và lần thứ hai trong không gian 64 chiều (feature).

```

1  class Transform(nn.Module):
2      def __init__(self):
3          super().__init__()
4          self.input_transform = Tnet(k=3)
5          self.feature_transform = Tnet(k=64)
6          self.conv1 = nn.Conv1d(3,64,1)
7
8          self.conv2 = nn.Conv1d(64,128,1)
9          self.conv3 = nn.Conv1d(128,1024,1)
10
11          self.bn1 = nn.BatchNorm1d(64)
12          self.bn2 = nn.BatchNorm1d(128)
13          self.bn3 = nn.BatchNorm1d(1024)
14
15

```

```

16     def forward(self, input):
17         matrix3x3 = self.input_transform(input)
18         # batch matrix multiplication
19         xb = torch.bmm(torch.transpose(input, 1, 2), matrix3x3).transpose(1, 2)
20
21         xb = F.relu(self.bn1(self.conv1(xb)))
22
23         matrix64x64 = self.feature_transform(xb)
24         xb = torch.bmm(torch.transpose(xb, 1, 2), matrix64x64).transpose(1, 2)
25
26         xb = F.relu(self.bn2(self.conv2(xb)))
27         xb = self.bn3(self.conv3(xb))
28         xb = nn.MaxPool1d(xb.size(-1))(xb)
29         output = nn.Flatten(1)(xb)
30         return output, matrix3x3, matrix64x64

```

### 2.2.3 PointNet

Mạng PointNet sử dụng module Transform 2.2.2 và các lớp fully connected layer để tổng hợp thông tin và đưa ra dự đoán dựa vào hàm softmax.

```

1  class PointNet(nn.Module):
2      def __init__(self, classes = 3):
3          super().__init__()
4          self.transform = Transform()
5          self.fc1 = nn.Linear(1024, 512)
6          self.fc2 = nn.Linear(512, 256)
7          self.fc3 = nn.Linear(256, classes)
8
9          self.bn1 = nn.BatchNorm1d(512)
10         self.bn2 = nn.BatchNorm1d(256)
11         self.dropout = nn.Dropout(p=0.3)
12         self.logsoftmax = nn.LogSoftmax(dim=1)
13
14     def forward(self, input):
15         xb, matrix3x3, matrix64x64 = self.transform(input)
16         xb = F.relu(self.bn1(self.fc1(xb)))
17         xb = F.relu(self.bn2(self.dropout(self.fc2(xb))))
18         output = self.fc3(xb)
19
20         return self.logsoftmax(output), matrix3x3, matrix64x64

```

## 2.3 Hàm loss

Trong PointNet, việc áp dụng ràng buộc ma trận chuyển đổi học được bởi các mô-đun Tnet gần với ma trận trực giao ( $\|X * X^T - I\| \approx 0$ ) mang lại nhiều lợi ích quan trọng:

- Khi ma trận chuyển đổi gần với ma trận trực giao, chúng chủ yếu ảnh hưởng đến việc xoay các điểm, không phải khoảng cách. Điều này đảm bảo rằng các tính năng được mạng trích xuất nắm bắt các mối quan hệ hình học thực sự giữa các điểm, điều cần thiết cho việc nhận dạng đối tượng chính xác.
- Ma trận trực giao được điều chỉnh tốt, nghĩa là những thay đổi nhỏ trong đầu vào (đám mây điểm) không dẫn đến những thay đổi lớn trong đầu ra (đám mây điểm được biến đổi). Điều này cải thiện độ ổn định số của mạng trong quá trình đào tạo và giúp nó hội tụ nhanh hơn.

- Ma trận trực giao có một đặc tính đặc biệt: nghịch đảo của chúng đơn giản là chuyển vị của chúng. Điều này đơn giản hóa các phép tính trong mạng. Trong quá trình truyền tiếp, khi đám mây điểm được biến đổi được nhân với chuyển vị của ma trận chuyển đổi, nó trở thành phép nhân ma trận đơn thay vì một phép toán phức tạp hơn liên quan đến nghịch đảo. Điều này làm giảm chi phí tính toán cho việc xử lý đám mây điểm.

```

1 def pointnetloss(outputs, labels, m3x3, m64x64, alpha = 0.0001):
2     criterion = torch.nn.NLLLoss()
3     bs=outputs.size(0)
4     id3x3 = torch.eye(3, requires_grad=True).repeat(bs,1,1)
5     id64x64 = torch.eye(64, requires_grad=True).repeat(bs,1,1)
6     if outputs.is_cuda:
7         id3x3=id3x3.cuda()
8         id64x64=id64x64.cuda()
9     diff3x3 = id3x3-torch.bmm(m3x3,m3x3.transpose(1,2))
10    diff64x64 = id64x64-torch.bmm(m64x64,m64x64.transpose(1,2))
11    return criterion(outputs, labels) + alpha * (torch.norm(diff3x3)+torch.norm(
12        diff64x64)) / float(bs)

```

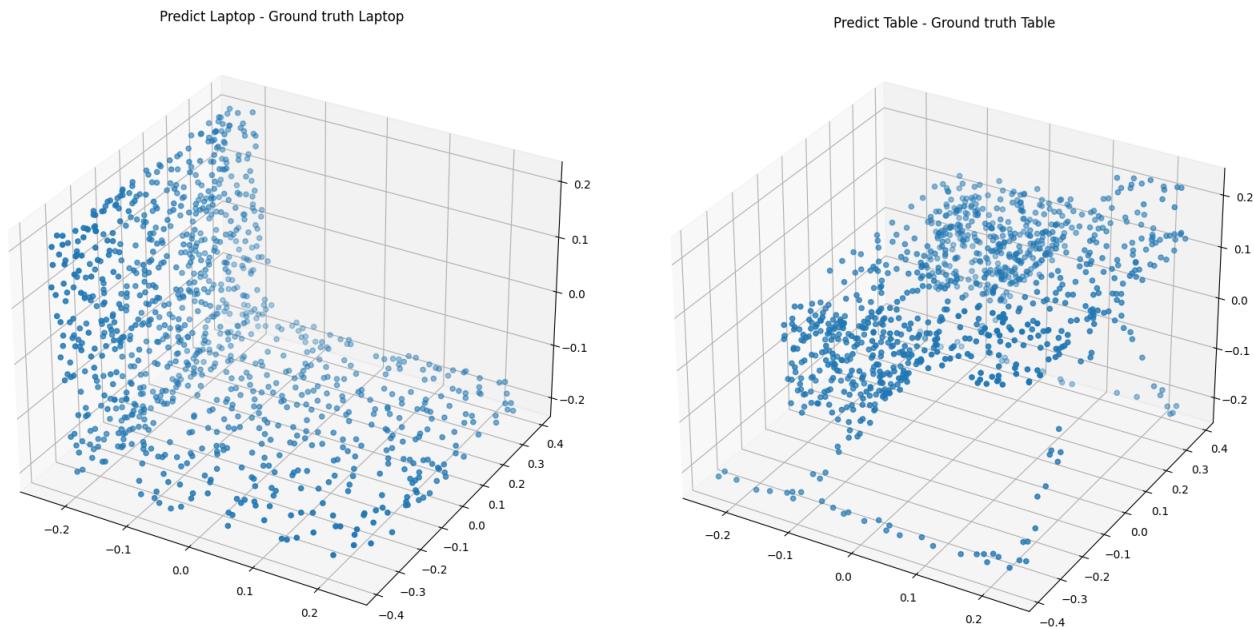
## 2.4 Training

Kết hợp dataloader và model để train:

```

1 pointnet = PointNet(classes = num_classes)
2 pointnet.to(device);
3 optimizer = torch.optim.Adam(pointnet.parameters(), lr=0.001)
4 def train(model, train_loader, val_loader=None, epochs=5, save=True):
5     for epoch in range(epochs):
6         pointnet.train()
7         running_loss = 0.0
8         for i, data in enumerate(train_loader, 0):
9             inputs, labels = data['pointcloud'].to(device).float(), data['category'].to(device)
10
11             optimizer.zero_grad()
12             outputs, m3x3, m64x64 = pointnet(inputs.transpose(1,2))
13
14             loss = pointnetloss(outputs, labels, m3x3, m64x64)
15             loss.backward()
16             optimizer.step()
17
18             # print statistics
19             running_loss += loss.item()
20             if i % 10 == 9:    # print every 10 mini-batches
21                 print('[Epoch: %d, Batch: %4d / %4d], loss: %.3f' %
22                       (epoch + 1, i + 1, len(train_loader), running_loss / 10))
23             running_loss = 0.0
24
25         pointnet.eval()
26         correct = total = 0
27
28         # validation
29         if val_loader:
30             with torch.no_grad():
31                 for data in val_loader:
32                     inputs, labels = data['pointcloud'].to(device).float(), data['category'].to(device)
33                     outputs, __, __ = pointnet(inputs.transpose(1,2))
34                     _, predicted = torch.max(outputs.data, 1)

```



Hình 8: Kết quả classification của PointNet trên Shapenet

```

35         total += labels.size(0)
36         correct += (predicted == labels).sum().item()
37     val_acc = 100. * correct / total
38     print('Valid accuracy: %d %%', % val_acc)
39
40     torch.save(pointnet.state_dict(), "save_"+str(epoch)+".pth")
41

```

## 2.5 Inference

Sau khi train model xong, inference ta được kết quả như hình 8.

## Phần II: Câu Hỏi

1. When loading a point cloud, what information might be included besides the 3D coordinates (XYZ)?  
(A). Color information (RGB).  
(B). Intensity values.  
(C). Normal vectors.  
(D). All of the above.
2. When dealing with 3D point cloud data, which data structure offers faster average-case search time complexity?  
(A). Kd-Tree.  
(B). Octree.  
(C). Both (a) and (b) have similar complexity.  
(D). The answer depends on the data distribution.
3. When dealing with point cloud data containing elongated or anisotropic features, which data structure might be less efficient for nearest neighbor search?  
(A). Kd-Tree.  
(B). Octree.  
(C). Both (a) and (b) can be affected.  
(D). Neither (a) nor (b).
4. When dealing with point clouds with smooth surfaces, which ICP variant might be more suitable?  
(A). Point-to-Point ICP.  
(B). Point-to-Plane ICP.  
(C). Both (a) and (b) are equally suitable.  
(D). Neither (a) nor (b) is suitable.
5. When dealing with point clouds with varying surface orientations (not necessarily smooth), which ICP variant might be more robust?  
(A). Point-to-Point ICP.  
(B). Point-to-Plane ICP.  
(C). Both (a) and (b) are equally robust.  
(D). Neither (a) nor (b) is suitable.
6. What is the main input data format for PointNet?  
(A). Images.  
(B). Meshes.  
(C). Point Clouds.  
(D). Voxels.
7. PointNet utilizes symmetrical functions for processing point clouds. What does this mean?  
(A). The order of points in the input cloud doesn't affect the output.  
(B). The network has the same number of layers in the encoder and decoder.  
(C). The network can process both 2D and 3D data.  
(D). The network requires pre-segmented point clouds.
8. PointNet++ is an extension of PointNet that addresses a limitation. What is that limitation?  
(A). Inability to handle large point clouds.  
(B). Sensitivity to point cloud order.  
(C). Difficulty capturing local features.  
(D). Limited number of learnable parameters.

**9. How does Voxel Net process point cloud data compare to PointNet?**

- (A). It directly operates on the point cloud coordinates.
- (B). It converts the point cloud into a 3D volumetric grid (voxel grid).
- (C). It requires pre-segmentation of the point cloud.
- (D). It is computationally less expensive than PointNet.

**10. What is a potential drawback of using Voxel Net?**

- (A). It can be computationally expensive for high-resolution voxel grids.
- (B). It requires color information for each point to be effective.
- (C). It is sensitive to the chosen voxel size.
- (D). Both (A) and (C).