PointNet architecture for 3D object classification.

1. \*\*Imports and Setup\*\*:

- Import necessary libraries like Torch, NumPy, etc.

import numpy as np

import math

import random

import os

import torch

import scipy.spatial.distance

from torch.utils.data import Dataset, DataLoader

from torchvision import transforms, utils

import plotly.graph\_objects as go

import plotly.express as px

- Mount Google Drive for access to files.

from google.colab import drive

drive.mount('/content/drive')

- Install required packages.

!pip install path.py;

from path import Path

random.seed = 42

- Download the dataset (ModelNet10) containing 3D object models.

!wget http://3dvision.princeton.edu/projects/2014/3DShapeNets/ModelNet10.zip

2. \*\*Dataset Preparation\*\*:

- Unzip the downloaded dataset.

!unzip -q ModelNet10.zip;

path = Path("ModelNet10")

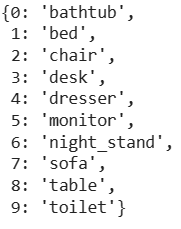
- Organize the classes and their corresponding numerical labels.

folders = [dir for dir in sorted(os.listdir(path)) if os.path.isdir(path/dir)]

classes = {folder: i for i, folder in enumerate(folders)};

classes

inv\_classes



3. \*\*Read and Visualize Meshes\*\*:

- Define functions to read .off files (containing mesh data) and visualize 3D meshes and point clouds.

def read\_off(file):

if 'OFF' != file.readline().strip():

raise('Not a valid OFF header')

n\_verts, n\_faces, \_\_ = tuple([int(s) for s in file.readline().strip().split(' ')])

verts = [[float(s) for s in file.readline().strip().split(' ')] for i\_vert in range(n\_verts)]

faces = [[int(s) for s in file.readline().strip().split(' ')][1:] for i\_face in range(n\_faces)]

return verts, faces

- Visualize a bed mesh and its vertices, emphasizing the need for clearer point clouds.

with open(path/"bed/train/bed\_0001.off", 'r') as f:

verts, faces = read\_off(f)

len(x)

def visualize\_rotate(data):

x\_eye, y\_eye, z\_eye = 1.25, 1.25, 0.8

frames=[]

def rotate\_z(x, y, z, theta):

w = x+1j\*y

return np.real(np.exp(1j\*theta)\*w), np.imag(np.exp(1j\*theta)\*w), z

for t in np.arange(0, 10.26, 0.1):

xe, ye, ze = rotate\_z(x\_eye, y\_eye, z\_eye, -t)

frames.append(dict(layout=dict(scene=dict(camera=dict(eye=dict(x=xe, y=ye, z=ze))))))

fig = go.Figure(data=data,

layout=go.Layout(

updatemenus=[dict(type='buttons',

showactive=False,

y=1,

x=0.8,

xanchor='left',

yanchor='bottom',

pad=dict(t=45, r=10),

buttons=[dict(label='Play',

method='animate',

args=[None, dict(frame=dict(duration=50, redraw=True),

transition=dict(duration=0),

fromcurrent=True,

mode='immediate'

)]

)

]

)

]

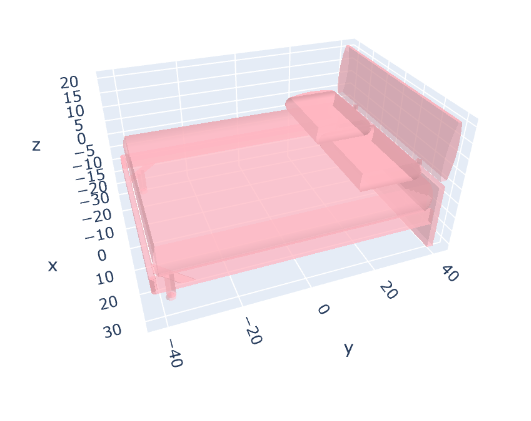
),

frames=frames

)

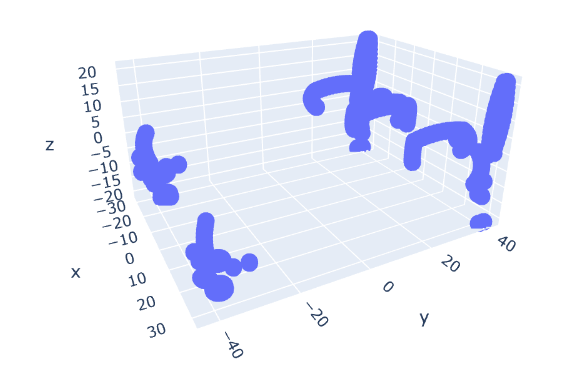
return fig

visualize\_rotate([go.Mesh3d(x=x, y=y, z=z, color='lightpink', opacity=0.50, i=i,j=j,k=k)]).show()



visualize\_rotate([go.Scatter3d(x=x, y=y, z=z,

mode='markers')]).show()



def pcshow(xs,ys,zs):

data=[go.Scatter3d(x=xs, y=ys, z=zs,

mode='markers')]

fig = visualize\_rotate(data)

fig.update\_traces(marker=dict(size=2,

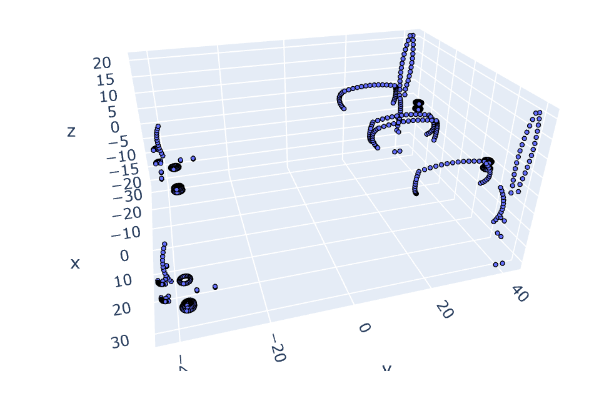
line=dict(width=2,

color='DarkSlateGrey')),

selector=dict(mode='markers'))

fig.show()

pcshow(x,y,z)



4. \*\*Point Sampling and Normalization\*\*:

- Implement a point sampler to generate point clouds from meshes.

class PointSampler(object):

def \_\_init\_\_(self, output\_size):

assert isinstance(output\_size, int)

self.output\_size = output\_size

def triangle\_area(self, pt1, pt2, pt3):

side\_a = np.linalg.norm(pt1 - pt2)

side\_b = np.linalg.norm(pt2 - pt3)

side\_c = np.linalg.norm(pt3 - pt1)

s = 0.5 \* ( side\_a + side\_b + side\_c)

return max(s \* (s - side\_a) \* (s - side\_b) \* (s - side\_c), 0)\*\*0.5

def sample\_point(self, pt1, pt2, pt3):

# barycentric coordinates on a triangle

# https://mathworld.wolfram.com/BarycentricCoordinates.html

s, t = sorted([random.random(), random.random()])

f = lambda i: s \* pt1[i] + (t-s)\*pt2[i] + (1-t)\*pt3[i]

return (f(0), f(1), f(2))

def \_\_call\_\_(self, mesh):

verts, faces = mesh

verts = np.array(verts)

areas = np.zeros((len(faces)))

for i in range(len(areas)):

areas[i] = (self.triangle\_area(verts[faces[i][0]],

verts[faces[i][1]],

verts[faces[i][2]]))

sampled\_faces = (random.choices(faces,

weights=areas,

cum\_weights=None,

k=self.output\_size))

sampled\_points = np.zeros((self.output\_size, 3))

for i in range(len(sampled\_faces)):

sampled\_points[i] = (self.sample\_point(verts[sampled\_faces[i][0]],

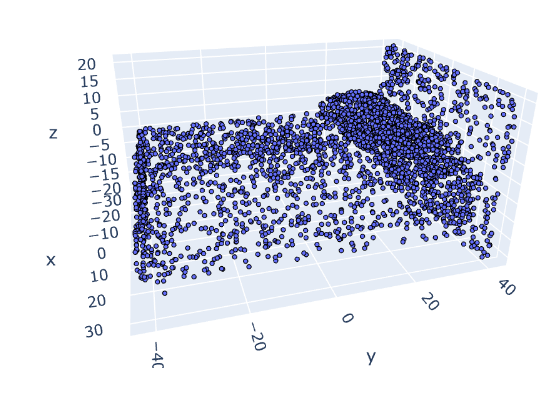
verts[sampled\_faces[i][1]],

verts[sampled\_faces[i][2]]))

return sampled\_points

pointcloud = PointSampler(3000)((verts, faces))

pcshow(\*pointcloud.T)



- Normalize the point clouds to fit within a unit sphere.

class Normalize(object):

def \_\_call\_\_(self, pointcloud):

assert len(pointcloud.shape)==2

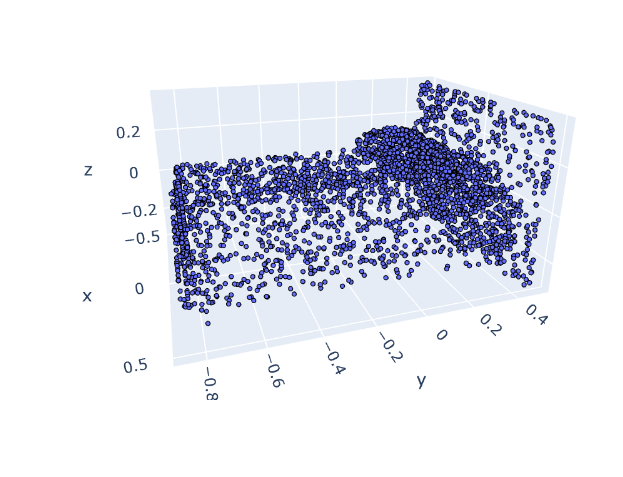
norm\_pointcloud = pointcloud - np.mean(pointcloud, axis=0)

norm\_pointcloud /= np.max(np.linalg.norm(norm\_pointcloud, axis=1))

return norm\_pointcloud

norm\_pointcloud = Normalize()(pointcloud)

pcshow(\*norm\_pointcloud.T)



5. \*\*Augmentations\*\*:

- Define methods for random rotation and adding noise to the point clouds.

class RandRotation\_z(object):

def \_\_call\_\_(self, pointcloud):

assert len(pointcloud.shape)==2

theta = random.random() \* 2. \* math.pi

rot\_matrix = np.array([[ math.cos(theta), -math.sin(theta), 0],

[ math.sin(theta), math.cos(theta), 0],

[0, 0, 1]])

rot\_pointcloud = rot\_matrix.dot(pointcloud.T).T

return rot\_pointcloud

class RandomNoise(object):

def \_\_call\_\_(self, pointcloud):

assert len(pointcloud.shape)==2

noise = np.random.normal(0, 0.02, (pointcloud.shape))

noisy\_pointcloud = pointcloud + noise

return noisy\_pointcloud

rot\_pointcloud = RandRotation\_z()(norm\_pointcloud)

noisy\_rot\_pointcloud = RandomNoise()(rot\_pointcloud)

6. \*\*Transformations and Dataset Creation\*\*:

- Implement transformations to convert data into tensors and create a custom dataset class for the PointNet model.

class ToTensor(object):

def \_\_call\_\_(self, pointcloud):

assert len(pointcloud.shape)==2

return torch.from\_numpy(pointcloud)

def default\_transforms():

return transforms.Compose([

PointSampler(1024),

Normalize(),

ToTensor()

])

class PointCloudData(Dataset):

def \_\_init\_\_(self, root\_dir, valid=False, folder="train", transform=default\_transforms()):

self.root\_dir = root\_dir

folders = [dir for dir in sorted(os.listdir(root\_dir)) if os.path.isdir(root\_dir/dir)]

self.classes = {folder: i for i, folder in enumerate(folders)}

self.transforms = transform if not valid else default\_transforms()

self.valid = valid

self.files = []

for category in self.classes.keys():

new\_dir = root\_dir/Path(category)/folder

for file in os.listdir(new\_dir):

if file.endswith('.off'):

sample = {}

sample['pcd\_path'] = new\_dir/file

sample['category'] = category

self.files.append(sample)

def \_\_len\_\_(self):

return len(self.files)

def \_\_preproc\_\_(self, file):

verts, faces = read\_off(file)

if self.transforms:

pointcloud = self.transforms((verts, faces))

return pointcloud

def \_\_getitem\_\_(self, idx):

pcd\_path = self.files[idx]['pcd\_path']

category = self.files[idx]['category']

with open(pcd\_path, 'r') as f:

pointcloud = self.\_\_preproc\_\_(f)

return {'pointcloud': pointcloud,

'category': self.classes[category]}

train\_transforms = transforms.Compose([

PointSampler(1024),

Normalize(),

RandRotation\_z(),

RandomNoise(),

ToTensor()

])

train\_ds = PointCloudData(path, transform=train\_transforms)

valid\_ds = PointCloudData(path, valid=True, folder='test', transform=train\_transforms)

inv\_classes = {i: cat for cat, i in train\_ds.classes.items()};

Inv\_classes

{0: 'bathtub',

1: 'bed',

2: 'chair',

3: 'desk',

4: 'dresser',

5: 'monitor',

6: 'night\_stand',

7: 'sofa',

8: 'table',

9: 'toilet'}

print('Train dataset size: ', len(train\_ds))

print('Valid dataset size: ', len(valid\_ds))

print('Number of classes: ', len(train\_ds.classes))

print('Sample pointcloud shape: ', train\_ds[0]['pointcloud'].size())

print('Class: ', inv\_classes[train\_ds[0]['category']])

Train dataset size: 3991

Valid dataset size: 908

Number of classes: 10

Sample pointcloud shape: torch.Size([1024, 3])

Class: bathtub

train\_loader = DataLoader(dataset=train\_ds, batch\_size=32, shuffle=True)

valid\_loader = DataLoader(dataset=valid\_ds, batch\_size=64)

7. \*\*Model Definition\*\*:

- Define the PointNet architecture for 3D object classification.

import torch

import torch.nn as nn

import numpy as np

import torch.nn.functional as F

class Tnet(nn.Module):

def \_\_init\_\_(self, k=3):

super().\_\_init\_\_()

self.k=k

self.conv1 = nn.Conv1d(k,64,1)

self.conv2 = nn.Conv1d(64,128,1)

self.conv3 = nn.Conv1d(128,1024,1)

self.fc1 = nn.Linear(1024,512)

self.fc2 = nn.Linear(512,256)

self.fc3 = nn.Linear(256,k\*k)

self.bn1 = nn.BatchNorm1d(64)

self.bn2 = nn.BatchNorm1d(128)

self.bn3 = nn.BatchNorm1d(1024)

self.bn4 = nn.BatchNorm1d(512)

self.bn5 = nn.BatchNorm1d(256)

class Transform(nn.Module):

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.input\_transform = Tnet(k=3)

self.feature\_transform = Tnet(k=64)

self.conv1 = nn.Conv1d(3,64,1)

self.conv2 = nn.Conv1d(64,128,1)

self.conv3 = nn.Conv1d(128,1024,1)

self.bn1 = nn.BatchNorm1d(64)

self.bn2 = nn.BatchNorm1d(128)

self.bn3 = nn.BatchNorm1d(1024)

def forward(self, input):

matrix3x3 = self.input\_transform(input)

# batch matrix multiplication

xb = torch.bmm(torch.transpose(input,1,2), matrix3x3).transpose(1,2)

xb = F.relu(self.bn1(self.conv1(xb)))

matrix64x64 = self.feature\_transform(xb)

xb = torch.bmm(torch.transpose(xb,1,2), matrix64x64).transpose(1,2)

xb = F.relu(self.bn2(self.conv2(xb)))

xb = self.bn3(self.conv3(xb))

xb = nn.MaxPool1d(xb.size(-1))(xb)

output = nn.Flatten(1)(xb)

return output, matrix3x3, matrix64x64

class PointNet(nn.Module):

def \_\_init\_\_(self, classes = 10):

super().\_\_init\_\_()

self.transform = Transform()

self.fc1 = nn.Linear(1024, 512)

self.fc2 = nn.Linear(512, 256)

self.fc3 = nn.Linear(256, classes)

self.bn1 = nn.BatchNorm1d(512)

self.bn2 = nn.BatchNorm1d(256)

self.dropout = nn.Dropout(p=0.3)

self.logsoftmax = nn.LogSoftmax(dim=1)

8. \*\*Training Loop\*\*:

- Set up the training loop that includes:

- Loading data batches.

device = torch.device("cuda:0" if torch.cuda.is\_available() else "cpu")

print(device)

pointnet = PointNet()

pointnet.to(device);

optimizer = torch.optim.Adam(pointnet.parameters(), lr=0.001)

def train(model, train\_loader, val\_loader=None, epochs=15, save=True):

for epoch in range(epochs):

pointnet.train()

running\_loss = 0.0

for i, data in enumerate(train\_loader, 0):

inputs, labels = data['pointcloud'].to(device).float(), data['category'].to(device)

optimizer.zero\_grad()

outputs, m3x3, m64x64 = pointnet(inputs.transpose(1,2))

loss = pointnetloss(outputs, labels, m3x3, m64x64)

loss.backward()

optimizer.step()

# print statistics

running\_loss += loss.item()

if i % 10 == 9: # print every 10 mini-batches

print('[Epoch: %d, Batch: %4d / %4d], loss: %.3f' %

(epoch + 1, i + 1, len(train\_loader), running\_loss / 10))

running\_loss = 0.0

train(pointnet, train\_loader, valid\_loader, save=False)

- Forward pass through the network.

def forward(self, input):

# input.shape == (bs,n,3)

bs = input.size(0)

xb = F.relu(self.bn1(self.conv1(input)))

xb = F.relu(self.bn2(self.conv2(xb)))

xb = F.relu(self.bn3(self.conv3(xb)))

pool = nn.MaxPool1d(xb.size(-1))(xb)

flat = nn.Flatten(1)(pool)

xb = F.relu(self.bn4(self.fc1(flat)))

xb = F.relu(self.bn5(self.fc2(xb)))

#initialize as identity

init = torch.eye(self.k, requires\_grad=True).repeat(bs,1,1)

if xb.is\_cuda:

init=init.cuda()

matrix = self.fc3(xb).view(-1,self.k,self.k) + init

return matrix

- Calculating and minimizing the loss.

def pointnetloss(outputs, labels, m3x3, m64x64, alpha = 0.0001):

criterion = torch.nn.NLLLoss()

bs=outputs.size(0)

id3x3 = torch.eye(3, requires\_grad=True).repeat(bs,1,1)

id64x64 = torch.eye(64, requires\_grad=True).repeat(bs,1,1)

if outputs.is\_cuda:

id3x3=id3x3.cuda()

id64x64=id64x64.cuda()

diff3x3 = id3x3-torch.bmm(m3x3,m3x3.transpose(1,2))

diff64x64 = id64x64-torch.bmm(m64x64,m64x64.transpose(1,2))

return criterion(outputs, labels) + alpha \* (torch.norm(diff3x3)+torch.norm(diff64x64)) / float(bs)

- Performing Testing and validation (if validation data is available).

pointnet.eval()

correct = total = 0

# validation

if val\_loader:

with torch.no\_grad():

for data in val\_loader:

inputs, labels = data['pointcloud'].to(device).float(), data['category'].to(device)

outputs, \_\_, \_\_ = pointnet(inputs.transpose(1,2))

\_, predicted = torch.max(outputs.data, 1)

total += labels.size(0)

correct += (predicted == labels).sum().item()

val\_acc = 100. \* correct / total

print('Valid accuracy: %d %%' % val\_acc)

# save the model

if save:

torch.save(pointnet.state\_dict(), "save\_"+str(epoch)+".pth")

9. Performance Meteics Analysis:

from sklearn.metrics import confusion\_matrix

pointnet = PointNet()

pointnet.load\_state\_dict(torch.load('/save.pth'))

pointnet.eval();

all\_preds = []

all\_labels = []

with torch.no\_grad():

for i, data in enumerate(valid\_loader):

print('Batch [%4d / %4d]' % (i+1, len(valid\_loader)))

inputs, labels = data['pointcloud'].float(), data['category']

outputs, \_\_, \_\_ = pointnet(inputs.transpose(1,2))

\_, preds = torch.max(outputs.data, 1)

all\_preds += list(preds.numpy())

all\_labels += list(labels.numpy())

Batch [ 1 / 15]

Batch [ 2 / 15]

Batch [ 3 / 15]

Batch [ 4 / 15]

Batch [ 5 / 15]

Batch [ 6 / 15]

Batch [ 7 / 15]

Batch [ 8 / 15]

Batch [ 9 / 15]

Batch [ 10 / 15]

Batch [ 11 / 15]

Batch [ 12 / 15]

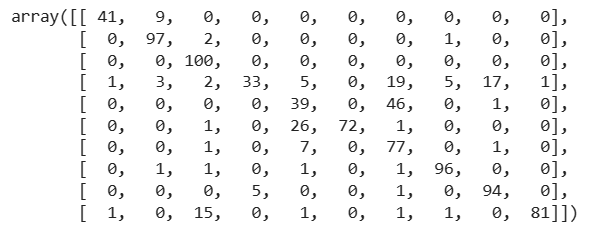
Batch [ 13 / 15]

Batch [ 14 / 15]

Batch [ 15 / 15]

cm = confusion\_matrix(all\_labels, all\_preds);

cm



import itertools

import numpy as np

import matplotlib.pyplot as plt

# function from https://deeplizard.com/learn/video/0LhiS6yu2qQ

def plot\_confusion\_matrix(cm, classes, normalize=False, title='Confusion matrix', cmap=plt.cm.Blues):

if normalize:

cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]

print("Normalized confusion matrix")

else:

print('Confusion matrix, without normalization')

plt.imshow(cm, interpolation='nearest', cmap=cmap)

plt.title(title)

plt.colorbar()

tick\_marks = np.arange(len(classes))

plt.xticks(tick\_marks, classes, rotation=45)

plt.yticks(tick\_marks, classes)

fmt = '.2f' if normalize else 'd'

thresh = cm.max() / 2.

for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):

plt.text(j, i, format(cm[i, j], fmt), horizontalalignment="center", color="white" if cm[i, j] > thresh else "black")

plt.tight\_layout()

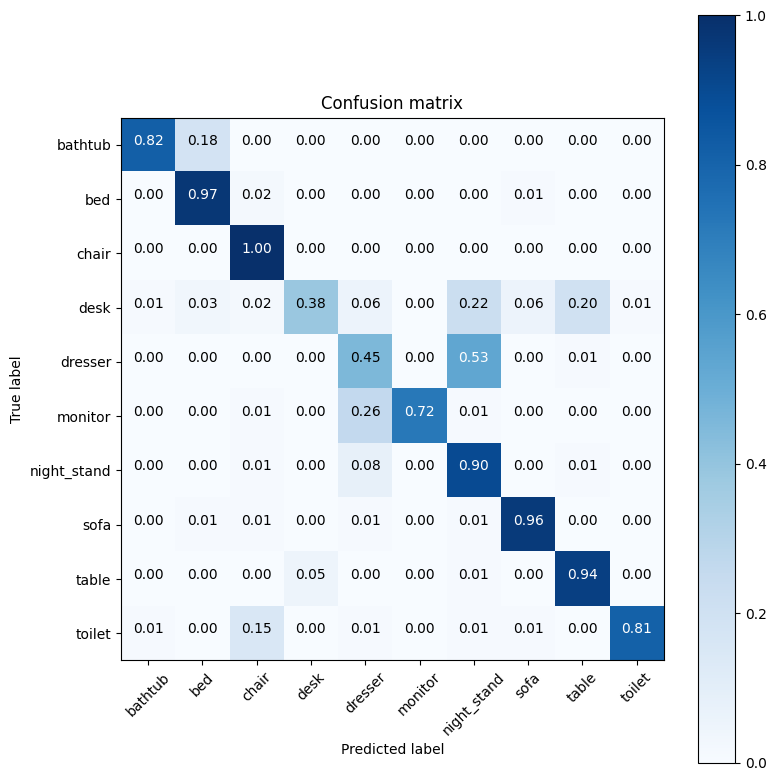
plt.ylabel('True label')

plt.xlabel('Predicted label')

plt.figure(figsize=(8,8))

plot\_confusion\_matrix(cm, list(classes.keys()), normalize=True)

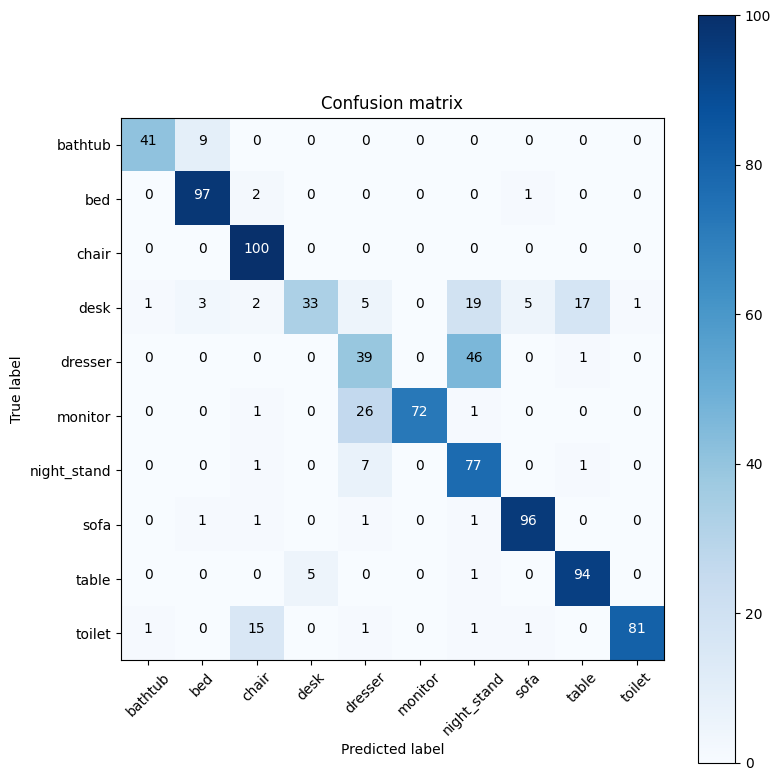
**Normalized confusion matrix**



plt.figure(figsize=(8,8))

plot\_confusion\_matrix(cm, list(classes.keys()), normalize=False)

**Confusion matrix, without normalization**



from sklearn.metrics import accuracy\_score

accuracy = accuracy\_score(all\_labels, all\_preds)

print(f'Accuracy: {accuracy \* 100:.2f}%')

Accuracy: 80.40%

from sklearn.metrics import precision\_score

precision = precision\_score(all\_labels, all\_preds, average='weighted')

print(f'Precision: {precision \* 100:.2f}%')

Precision: 83.20%

from sklearn.metrics import accuracy\_score, precision\_score, recall\_score, f1\_score, confusion\_matrix

y\_true = all\_labels

y\_pred = all\_preds

# Compute accuracy

accuracy = accuracy\_score(y\_true, y\_pred)

print(f"Accuracy: {accuracy:.4f}")

# Compute precision

precision = precision\_score(y\_true, y\_pred, average='weighted') # 'weighted' for multi-class

print(f"Precision: {precision:.4f}")

# Compute recall

recall = recall\_score(y\_true, y\_pred, average='weighted') # 'weighted' for multi-class

print(f"Recall: {recall:.4f}")

# Compute F1 score

f1 = f1\_score(y\_true, y\_pred, average='weighted') # 'weighted' for multi-class

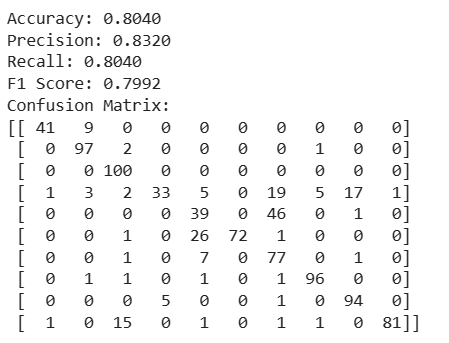
print(f"F1 Score: {f1:.4f}")

# Compute confusion matrix

conf\_matrix = confusion\_matrix(y\_true, y\_pred)

print("Confusion Matrix:")

print(conf\_matrix)



objects\_mapping = {

'object\_1': ['bathtub'],

'object\_2': ['bed'],

'object\_3': ['chair'],

'object\_4': ['desk'],

'object\_5': ['dresser'],

'object\_6': ['monitor'],

'object\_7': ['night\_stand'],

'object\_8': ['sofa'],

'object\_9': ['table'],

'object\_10': ['toilet']

}

object\_wise\_data = {obj: {'true\_labels': [], 'predicted\_labels': []} for obj in objects\_mapping}

for obj, classes in objects\_mapping.items():

obj\_classes\_indices = [classes.index(cls) for cls in classes]

# Extract rows and columns corresponding to classes of the object from the confusion matrix

obj\_cm = cm[np.ix\_(obj\_classes\_indices, obj\_classes\_indices)]

obj\_true\_labels = np.sum(obj\_cm, axis=1)

obj\_predicted\_labels = np.sum(obj\_cm, axis=0)

object\_wise\_data[obj]['true\_labels'] = obj\_true\_labels

object\_wise\_data[obj]['predicted\_labels'] = obj\_predicted\_labels

# Now, object\_wise\_data contains true labels and predicted labels for each object.

from sklearn.metrics import accuracy\_score, precision\_score, recall\_score, f1\_score

# Initialize dictionaries to store metrics for each object

object\_wise\_accuracy = {}

object\_wise\_precision = {}

object\_wise\_recall = {}

object\_wise\_f1 = {}

# Iterate through objects and calculate metrics

for obj in object\_wise\_preds.keys():

preds = object\_wise\_preds[obj]

labels = object\_wise\_labels[obj]

# Compute accuracy

accuracy = accuracy\_score(labels, preds)

object\_wise\_accuracy[obj] = accuracy

# Compute precision

precision = precision\_score(labels, preds, average='weighted', zero\_division=0)

object\_wise\_precision[obj] = precision

# Compute recall

recall = recall\_score(labels, preds, average='weighted', zero\_division=0)

object\_wise\_recall[obj] = recall

# Compute F1 score

f1 = f1\_score(labels, preds, average='weighted', zero\_division=0)

object\_wise\_f1[obj] = f1

# Print metrics for each object

for obj in object\_wise\_preds.keys():

print(f"Object {objects\_mapping[obj][0]}:")

print(f" Accuracy: {object\_wise\_accuracy[obj]:.4f}")

print(f" Precision: {object\_wise\_precision[obj]:.4f}")

print(f" Recall: {object\_wise\_recall[obj]:.4f}")

print(f" F1 Score: {object\_wise\_f1[obj]:.4f}")

print()

Object bathtub:

Accuracy: 0.8200

Precision: 1.0000

Recall: 0.8200

F1 Score: 0.9011

Object bed:

Accuracy: 0.9700

Precision: 1.0000

Recall: 0.9700

F1 Score: 0.9848

Object chair:

Accuracy: 1.0000

Precision: 1.0000

Recall: 1.0000

F1 Score: 1.0000

Object desk:

Accuracy: 0.3837

Precision: 1.0000

Recall: 0.3837

F1 Score: 0.5546

Object dresser:

Accuracy: 0.5000

Precision: 1.0000

Recall: 0.5000

F1 Score: 0.6667

Object monitor:

Accuracy: 0.7100

Precision: 1.0000

Recall: 0.7100

F1 Score: 0.8304

Object night\_stand:

Accuracy: 0.9186

Precision: 1.0000

Recall: 0.9186

F1 Score: 0.9576

Object sofa:

Accuracy: 0.9600

Precision: 1.0000

Recall: 0.9600

F1 Score: 0.9796

Object table:

Accuracy: 0.9300

Precision: 1.0000

Recall: 0.9300

F1 Score: 0.9637

Object toilet:

Accuracy: 0.8400

Precision: 1.0000

Recall: 0.8400

F1 Score: 0.9130

MEASURES TO IMPROVE THE PERFORMANCE OF THE POINTNET MODEL :

"desk":

Low accuracy (38.37%) Low recall (38.37%) Low F1 Score (55.46%) This suggests the model has difficulty correctly identifying instances of the "desk" class, potentially misclassifying them as other classes.

"dresser":

Moderate accuracy (50.00%) Low recall (50.00%) Moderate F1 Score (66.67%) Similar to "desk," the model might have challenges in correctly identifying "dresser" instances. These lower-performing classes might have imbalanced data or might be visually similar to other classes, leading to confusion for the model in distinguishing them accurately.

To improve performance for these classes, you could consider strategies like:

Data Augmentation: Increase the diversity of "desk" and "dresser" instances in the dataset to help the model learn better representations. Model Tuning: Adjust model architecture or hyperparameters specifically addressing the characteristics of these challenging classes. Class Balancing: If the dataset is highly imbalanced, consider techniques like oversampling or undersampling to balance the representation of each class. These steps might help improve the model's ability to recognize and classify instances of these challenging classes more accurately.