COMPSCI 574: Intelligent Visual Computing Spring 23

Prachi Jain

Assignment 4: Neural Surface reconstruction from point clouds

1 Deep SDF Reconstruction

Results after training on best model for each of these objects:

1.1 Sphere

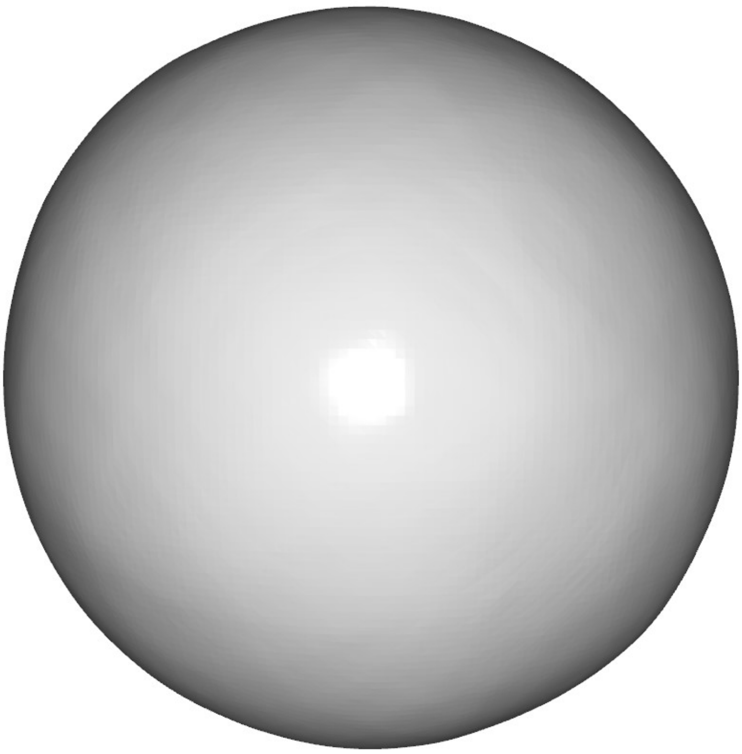

Figure 1: Deep SDF Reconstruction of Sphere

1.2 Bunny with 500 Points

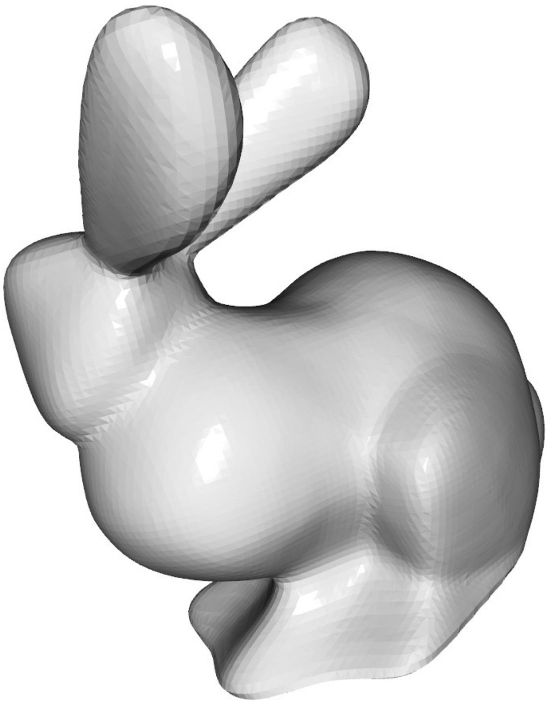

Figure 2: Deep SDF Reconstruction of Bunny from 500 points

1.3 Bunny with 1000 Points

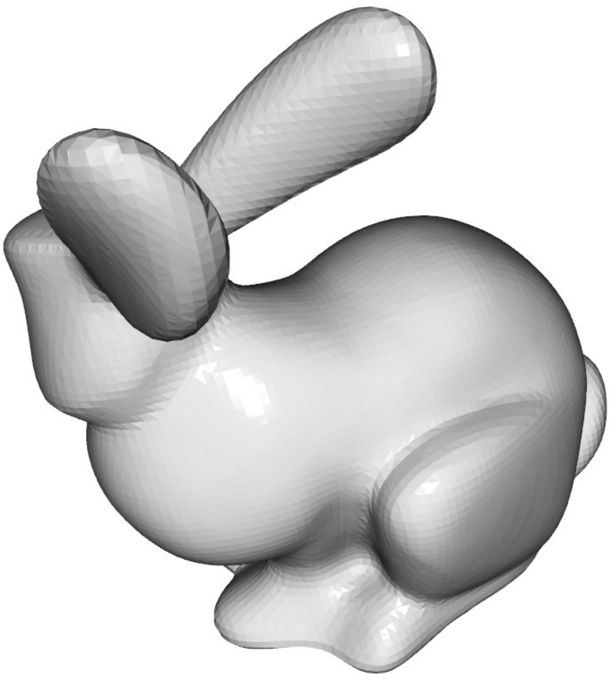

Figure 3: Deep SDF Reconstruction of Bunny from 1000 points

2 Implementation

Please find attached code below to replicate the above results.

a. ***model.py***

```
import torch.nn as nn
import torch
import torch.nn.functional as F


class Decoder(nn.Module):
    def __init__(
        self,
        args,
        dropout_prob=0.1,
    ):
        super(Decoder, self).__init__()

        # **** YOU SHOULD IMPLEMENT THE MODEL ARCHITECTURE HERE ****
        # Define the network architecture based on the figure shown in the assignment page.
        # Read the instruction carefully for layer details.
        # Pay attention that your implementation should include FC layers, weight_norm layers,
        # PReLU layers, Dropout layers and a tanh layer.
        # ***********************************************************************
        self.dropout_prob = dropout_prob

        self.fc1  = nn.utils.weight_norm(nn.Linear(3, 512))
        self.fc2  = nn.utils.weight_norm(nn.Linear(512,512))
        self.fc3  = nn.utils.weight_norm(nn.Linear(512,512))
        self.fc4  = nn.utils.weight_norm(nn.Linear(512,509))
        self.fc5  = nn.utils.weight_norm(nn.Linear(512,512))
        self.fc6  = nn.utils.weight_norm(nn.Linear(512,512))
        self.fc7  = nn.utils.weight_norm(nn.Linear(512,512))
        self.fc8  = nn.Linear(512,1)

        self.prelu = nn.PReLU()
        self.drop = nn.Dropout(dropout_prob)
        self.th = nn.Tanh()

    # input: N x 3
    def forward(self, input):

        # **** YOU SHOULD IMPLEMENT THE FORWARD PASS HERE ****
        # Based on the architecture defined above, implement the feed forward procedure
        # ***********************************************************************

        x_copy = input

        x1 = self.drop(self.prelu(self.fc1(input)))
        x2 = self.drop(self.prelu(self.fc2(x1)))
        x3 = self.drop(self.prelu(self.fc3(x2)))
```

```python
        x4 = self.drop(self.prelu(self.fc4(x3)))

        x4 = torch.cat((x4, x_copy), dim = 1)

        x5 = self.drop(self.prelu(self.fc5(x4)))
        x6 = self.drop(self.prelu(self.fc6(x5)))
        x7 = self.drop(self.prelu(self.fc7(x6)))

        x8 = self.fc8(x7)
        out = self.th(x8)
        return out
```

b. *train.py*

```python
import os
import shutil
import argparse
import numpy as np
import torch
import torch.backends.cudnn as cudnn
from model import Decoder
from utils import normalize_pts, normalize_normals, SdfDataset, mkdir_p, isdir, showMeshReconstruction

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

# function to save a checkpoint during training, including the best model so far
def save_checkpoint(state, is_best, checkpoint_folder='checkpoints/', filename='checkpoint.pth.tar'):
    checkpoint_file = os.path.join(checkpoint_folder, 'checkpoint_{}.pth.tar'.format(state['epoch']))
    torch.save(state, checkpoint_file)
    if is_best:
        shutil.copyfile(checkpoint_file, os.path.join(checkpoint_folder, 'model_best.pth.tar'))


def train(dataset, model, optimizer, args):
    model.train()  # switch to train mode
    loss_sum = 0.0
    loss_count = 0.0
    num_batch = len(dataset)
    for i in range(num_batch):
        data = dataset[i]  # a dict

        # **** YOU SHOULD ADD TRAINING CODE HERE, CURRENTLY IT IS INCORRECT ****

        optimizer.zero_grad()

        #Load XYZ tensor
        xyz_tensor = data['xyz'].to(device)
        xyzTensorShape = xyz_tensor.shape[0]

        #Load Ground Truth and Model Predictions Tensors
        gt_sdf_tensor = data['gt_sdf'].to(device)
        pred_sdf_tensor = model(xyz_tensor)

        #Clamping Groundtruth and Predictions
```

```
        c = args.clamping_distance
        loss = torch.abs(torch.clamp(pred_sdf_tensor, -c, c) - torch.clamp(gt_sdf_tensor, -c ,c))

        #Loss Computation
        loss = torch.sum(loss)
        loss.backward()
        optimizer.step()

        #Update Loss Sum and Counts
        loss_sum += loss.detach() * xyzTensorShape
        loss_count += xyzTensorShape

        # ***********************************************************************

    return loss_sum / loss_count


# validation function
def val(dataset, model, optimizer, args):
    model.eval()  # switch to test mode
    loss_sum = 0.0
    loss_count = 0.0
    num_batch = len(dataset)
    for i in range(num_batch):
        data = dataset[i]  # a dict

        # **** YOU SHOULD ADD TRAINING CODE HERE, CURRENTLY IT IS INCORRECT ****
        with torch.no_grad():
            xyz_tensor = data['xyz'].to(device)
            xyzTensorShape = xyz_tensor.shape[0]

            #Load Ground Truth and Model Predictions Tensors
            gt_sdf_tensor = data['gt_sdf'].to(device)
            pred_sdf_tensor = model(xyz_tensor)

            #Clamping Groundtruth and Predictions
            c = args.clamping_distance
            loss = torch.abs(torch.clamp(pred_sdf_tensor, -c, c) - torch.clamp(gt_sdf_tensor, -c ,c))

            #Loss Computation; Update Loss Sum and Counts
            loss = torch.sum(loss)
            loss_sum += loss.detach() * xyzTensorShape
            loss_count += xyzTensorShape

            # ***********************************************************************

    return loss_sum / loss_count


# testing function
def test(dataset, model, args):
    model.eval()  # switch to test mode
    num_batch = len(dataset)
    number_samples = dataset.number_samples
```

```
    grid_shape = dataset.grid_shape
    IF = np.zeros((number_samples, ))
    start_idx = 0
    for i in range(num_batch):
        data = dataset[i]  # a dict
        xyz_tensor = data['xyz'].to(device)
        this_bs = xyz_tensor.shape[0]
        end_idx = start_idx + this_bs
        with torch.no_grad():
            pred_sdf_tensor = model(xyz_tensor)
            pred_sdf_tensor = torch.clamp(pred_sdf_tensor, -args.clamping_distance, args.clamping_distance)
        pred_sdf = pred_sdf_tensor.cpu().squeeze().numpy()
        IF[start_idx:end_idx] = pred_sdf
        start_idx = end_idx
    IF = np.reshape(IF, grid_shape)

    verts, triangles = showMeshReconstruction(IF)
    with open('test.obj', 'w') as outfile:
        for v in verts:
            outfile.write( "v " + str(v[0]) + " " + str(v[1]) + " " + str(v[2]) + "\n" )
        for f in triangles:
            outfile.write( "f " + str(f[0]+1) + " " + str(f[1]+1) + " " + str(f[2]+1) + "\n" )
    outfile.close()
    return

def main(args):
    best_loss = 2e10
    best_epoch = -1

    # create checkpoint folder
    if not isdir(args.checkpoint_folder):
        print("Creating new checkpoint folder " + args.checkpoint_folder)
        mkdir_p(args.checkpoint_folder)

    #default architecture in DeepSDF
    model = Decoder(args)


    model.to(device)
    print("=> Will use the (" + device.type + ") device.")

    # cudnn will optimize execution for our network
    cudnn.benchmark = True

    if args.evaluate:
        print("\nEvaluation only")
        path_to_resume_file = os.path.join(args.checkpoint_folder, args.resume_file)
        print("=> Loading training checkpoint '{}'".format(path_to_resume_file))
        checkpoint = torch.load(path_to_resume_file)
        model.load_state_dict(checkpoint['state_dict'])
        test_dataset = SdfDataset(phase='test', args=args)
        test(test_dataset, model, args)
        return

```

```
    optimizer = torch.optim.AdamW(filter(lambda p: p.requires_grad, model.parameters()), lr=args.lr, weight_decay=args.weight_decay)
    print("=> Total params: %.2fM" % (sum(p.numel() for p in model.parameters()) / 1000000.0))

    # create dataset
    input_point_cloud = np.loadtxt(args.input_pts)
    training_points = normalize_pts(input_point_cloud[:, :3])
    training_normals = normalize_normals(input_point_cloud[:, 3:])
    n_points = training_points.shape[0]
    print("=> Number of points in input point cloud: %d" % n_points)

    # split dataset into train and validation set by args.train_split_ratio
    n_points_train = int(args.train_split_ratio * n_points)
    full_indices = np.arange(n_points)
    np.random.shuffle(full_indices)
    train_indices = full_indices[:n_points_train]
    val_indices = full_indices[n_points_train:]
    train_dataset = SdfDataset(points=training_points[train_indices], normals=training_normals[train_indices], args=args)
    val_dataset = SdfDataset(points=training_points[val_indices], normals=training_normals[val_indices], phase='val', args=args)

    # perform training!
    scheduler = torch.optim.lr_scheduler.MultiStepLR(optimizer, args.schedule, gamma=args.gamma)

    for epoch in range(args.start_epoch, args.epochs):
        train_loss = train(train_dataset, model, optimizer, args)
        val_loss = val(val_dataset, model, optimizer, args)
        scheduler.step()
        is_best = val_loss < best_loss
        if is_best:
            best_loss = val_loss
            best_epoch = epoch
        save_checkpoint({"epoch": epoch + 1, "state_dict": model.state_dict(), "best_loss": best_loss, "optimizer": optimizer.state_dict()},
                        is_best, checkpoint_folder=args.checkpoint_folder)
        print(f"Epoch {epoch+1:d}. train_loss: {train_loss:.8f}. val_loss: {val_loss:.8f}. Best Epoch: {best_epoch+1:d}. Best val loss: {best_loss:.8f}.")


if __name__ == "__main__":
    parser = argparse.ArgumentParser(description='DeepSDF')

    parser.add_argument("-e", "--evaluate", action="store_true", help="Activate test mode - Evaluate model on val/test set (no training)")

    # paths you may want to adjust
    parser.add_argument("--input_pts", default="data/sphere.pts", type=str, help="Input point cloud")
    parser.add_argument("--checkpoint_folder", default="checkpoints/", type=str, help="Folder to save checkpoints")
    parser.add_argument("--resume_file", default="model_best.pth.tar", type=str, help="Path to retrieve latest checkpoint file relative to checkpoint folder")

```

```
    # hyperameters of network/options for training
    parser.add_argument("--weight_decay", default=1e-4, type=float, help="Weight decay/L2 regularization on weights")
    parser.add_argument("--lr", default=1e-4, type=float, help="Initial learning rate")
    parser.add_argument("--schedule", type=int, nargs="+", default=[40, 50], help="Decrease learning rate at these milestone epochs.")
    parser.add_argument("--gamma", default=0.1, type=float, help="Decays the learning rate of each parameter group by gamma once the number of epoch reaches one of the milestone epochs")
    parser.add_argument("--start_epoch", default=0, type=int, help="Start from specified epoch number")
    parser.add_argument("--epochs", default=100, type=int, help="Number of epochs to train (when loading a previous model, it will train for an extra number of epochs)")
    parser.add_argument("--train_batch", default=512, type=int, help="Batch size for training")
    parser.add_argument("--train_split_ratio", default=0.8, type=float, help="ratio of training split")
    parser.add_argument("--N_samples", default=100.0, type=float, help="for each input point, N samples are used for training or validation")
    parser.add_argument("--sample_std", default=0.05, type=float, help="we perturb each surface point along normal direction with mean-zero Gaussian noise with the given standard deviation")
    parser.add_argument("--clamping_distance", default=0.1, type=float, help="clamping distance for sdf")


    # various options for testing and evaluation
    parser.add_argument("--test_batch", default=2048, type=int, help="Batch size for testing")
    parser.add_argument("--grid_N", default=128, type=int, help="construct a 3D NxNxN grid containing the point cloud")
    parser.add_argument("--max_xyz", default=1.0, type=float, help="largest xyz coordinates")

    print(parser.parse_args())
    main(parser.parse_args())
```

C. *utils.py*

```
import torch.utils.data as data
import numpy as np
import math
import torch
import os
import errno
import open3d as o3d;
from skimage import measure

def mkdir_p(dir_path):
    try:
        os.makedirs(dir_path)
    except OSError as e:
        if e.errno != errno.EEXIST:
            raise


```

```python
def isdir(dirname):
    return os.path.isdir(dirname)


def normalize_pts(input_pts):
    center_point = np.mean(input_pts, axis=0)
    center_point = center_point[np.newaxis, :]
    centered_pts = input_pts - center_point

    largest_radius = np.amax(np.sqrt(np.sum(centered_pts ** 2, axis=1)))
    normalized_pts = centered_pts / largest_radius   # / 1.03  if we follow DeepSDF completely

    return normalized_pts


def normalize_normals(input_normals):
    normals_magnitude = np.sqrt(np.sum(input_normals ** 2, axis=1))
    normals_magnitude = normals_magnitude[:, np.newaxis]

    normalized_normals = input_normals / normals_magnitude

    return normalized_normals

def showMeshReconstruction(IF):
    """
    calls marching cubes on the input implicit function sampled in the 3D grid
    and shows the reconstruction mesh
    Args:
        IF    : implicit function sampled at the grid points
  Returns:
    verts, triangles: vertices and triangles of the polygon mesh after iso-surfacing it at level 0
    """
    verts, triangles, normals, values = measure.marching_cubes(IF, 0)

    # Create an empty triangle mesh
    mesh = o3d.geometry.TriangleMesh()
    # Use mesh.vertex to access the vertices' attributes
    mesh.vertices = o3d.utility.Vector3dVector(verts)
    # Use mesh.triangle to access the triangles' attributes
    mesh.triangles = o3d.utility.Vector3iVector(triangles.astype(np.int32))
    mesh.compute_vertex_normals()
    o3d.visualization.draw_geometries([mesh])
    return verts, triangles


class SdfDataset(data.Dataset):
    def __init__(self, points=None, normals=None, phase='train', args=None):
        self.phase = phase

        if self.phase == 'test':
            self.bs = args.test_batch
            max_dimensions = np.ones((3, )) * args.max_xyz
            min_dimensions = -np.ones((3, )) * args.max_xyz

            bounding_box_dimensions = max_dimensions - min_dimensions  # compute the bounding box dimensions of the point cloud
```

```python
            grid_spacing = max(bounding_box_dimensions) / (args.grid_N - 9)  # each cell in the grid will have the same size
            X, Y, Z = np.meshgrid(list(
                np.arange(min_dimensions[0] - grid_spacing * 4, max_dimensions[0] + grid_spacing * 4, grid_spacing)),
                                  list(np.arange(min_dimensions[1] - grid_spacing * 4,
                                                 max_dimensions[1] + grid_spacing * 4,
                                                 grid_spacing)),
                                  list(np.arange(min_dimensions[2] - grid_spacing * 4,
                                                 max_dimensions[2] + grid_spacing * 4,
                                                 grid_spacing)))  # N x N x N
            self.grid_shape = X.shape
            self.samples_xyz = np.array([X.reshape(-1), Y.reshape(-1), Z.reshape(-1)]).transpose()
            self.number_samples = self.samples_xyz.shape[0]
            self.number_batches = math.ceil(self.number_samples * 1.0 / self.bs)

        else:
            self.points = points
            self.normals = normals
            self.sample_std = args.sample_std
            self.bs = args.train_batch
            self.number_points = self.points.shape[0]
            self.number_samples = int(self.number_points * args.N_samples)
            self.number_batches = math.ceil(self.number_samples * 1.0 / self.bs)

            if phase == 'val':
                # **** YOU SHOULD ADD TRAINING CODE HERE, CURRENTLY IT IS INCORRECT ****
                # Sample random points around surface point along the normal direction based on
                # a Gaussian distribution described in the assignment page.
                # For validation set, just do this sampling process for one time.
                # For training set, do this sampling process per each iteration (see code in __getitem__).

                self.samples_sdf = np.random.normal(0, self.sample_std, size=(self.number_samples, 1))
                self._points  = np.repeat(self.points, args.N_samples, axis = 0)
                self._normals = np.repeat(self.normals, args.N_samples, axis = 0)

                self.samples_xyz = self._points + self.samples_sdf * self._normals
                # ***********************************************************************

    def __len__(self):
        return self.number_batches

    def __getitem__(self, idx):
        start_idx = idx * self.bs
        end_idx = min(start_idx + self.bs, self.number_samples)  # exclusive
```

```
        if self.phase == 'val':
            xyz = self.samples_xyz[start_idx:end_idx, :]
            gt_sdf = self.samples_sdf[start_idx:end_idx, :]
        elif self.phase == 'train':  # sample points on the fly
            this_bs = end_idx - start_idx
            # **** YOU SHOULD ADD TRAINING CODE HERE, CURRENTLY IT IS INCORRECT ****
            # Sample random points around surface point along the normal direction based on
            # a Gaussian distribution described in the assignment page.
            # For training set, do this sampling process per each iteration.
            gt_sdf = np.random.normal(0, self.sample_std, size=(this_bs, 1))
            select = np.random.randint(self.points.shape[0], size = this_bs)
            xyz = self.points[select] + gt_sdf * self.normals[select]

            # ***********************************************************************

        else:
            assert self.phase == 'test'
            xyz = self.samples_xyz[start_idx:end_idx, :]

        if self.phase == 'test':
            return {'xyz': torch.FloatTensor(xyz)}
        else:
            return {'xyz': torch.FloatTensor(xyz), 'gt_sdf': torch.FloatTensor(gt_sdf)}
```