Implementing DeepSDF

Final Project, IFT6113 - Geometric Modeling and Shape Analysis

Sylvain Laporte - Fall 2020

Summary

- What is DeepSDF and why is it useful?
- Our implementation
- Results
- An idea of our extension

What is DeepSDF?

The paper

« DeepSDF: Learning Continuous Signed Distances Functions for Shape Representation »

DeepSDF: Learning Continuous Signed Distance Functions for Shape Representation

Jeong Joon Park[1,3†] Peter Florence [2,3†] Julian Straub[3] Richard Newcombe[3] Steven Lovegrove[3]

[1]University of Washington [2]Massachusetts Institute of Technology [3]Facebook Reality Labs

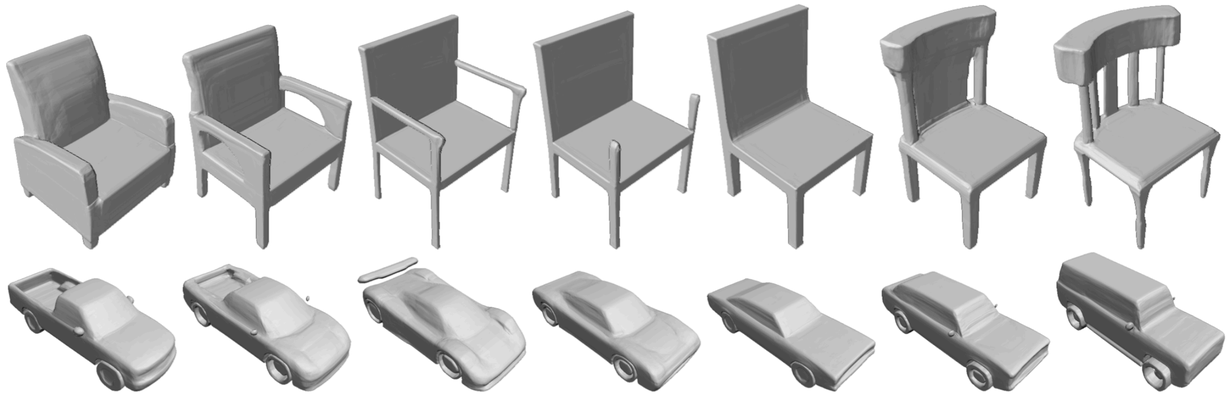


Figure 1: DeepSDF represents signed distance functions (SDFs) of shapes via latent code-conditioned feed-forward decoder networks. Above images are raycast renderings of DeepSDF interpolating between two shapes in the learned shape latent space. Best viewed digitally.

Abstract

Computer graphics, 3D computer vision and robotics communities have produced multiple approaches to representing 3D geometry for rendering and reconstruction.

1. Introduction

Deep convolutional networks which are a mainstay of image-based approaches grow quickly in space and time complexity when directly generalized to the 3rd spatial di-

The problem

How to represent 3D geometry efficiently for rendering and reconstruction?

- Computer graphics, 3D computer vision and robotics need a good way to represent 3D geometry:
  - quality
  - flexibility
  - fidelity
  - efficiency in time and memory

Previous work

Problems with existing representations

- Deep convolutional networks:
  - grow quickly in space and time complexity when generalized to the 3rd spatial dimension.
- Classical surface representations (triangle or quad meshes):
  - have an unknown number of vertices and an arbitrary topology; hence they are not well suited for training.

The solution

Implicit representation using SDF

- *SDF* = signed distance function
- A continuous function mapping each point in 3D space to its distance to the shape surface.
  - points inside the shape have negative SDF values
  - points outside the shape have positive SDF values
- SDF is represented as a volumetric field.
- The implicit surface is recovered from *SDF = 0*.

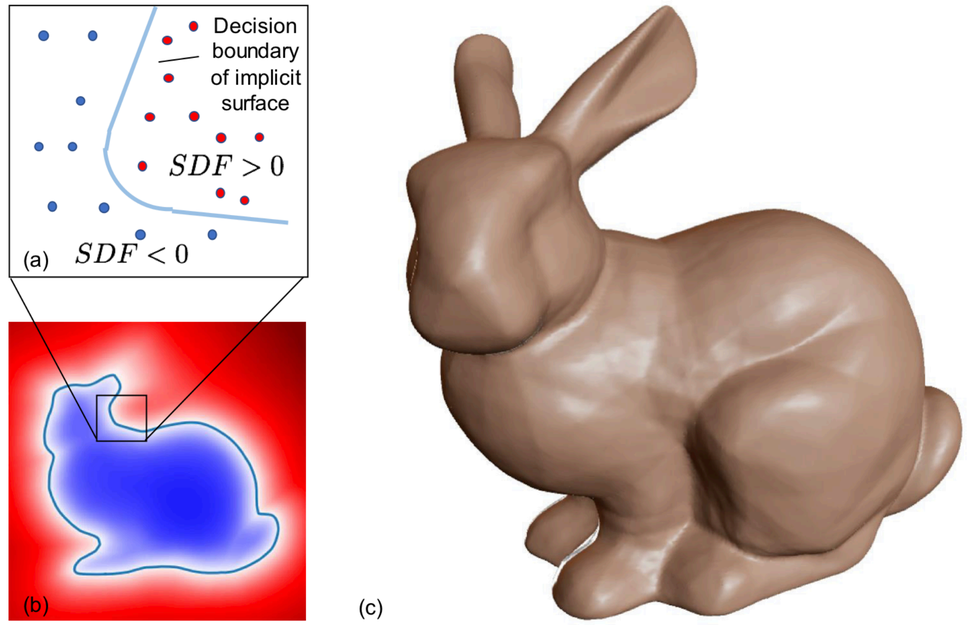
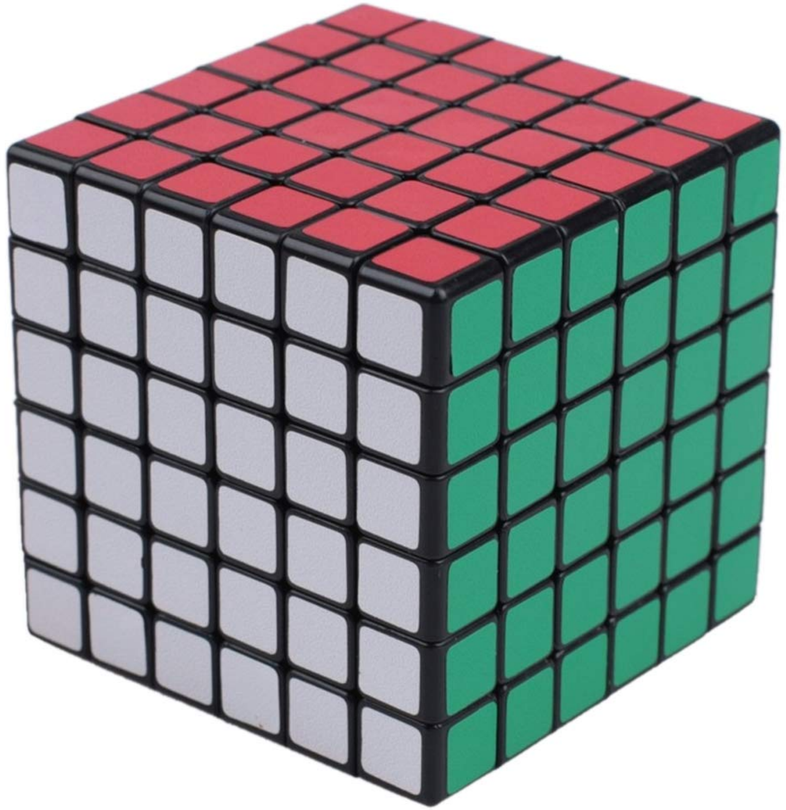


Figure 2: Our DeepSDF representation applied to the Stanford Bunny: (a) depiction of the underlying implicit surface $SDF = 0$ trained on sampled points inside $SDF < 0$ and outside $SDF > 0$ the surface, (b) 2D cross-section of the signed distance field, (c) rendered 3D surface recovered from $SDF = 0$. Note that (b) and (c) are recovered via DeepSDF.

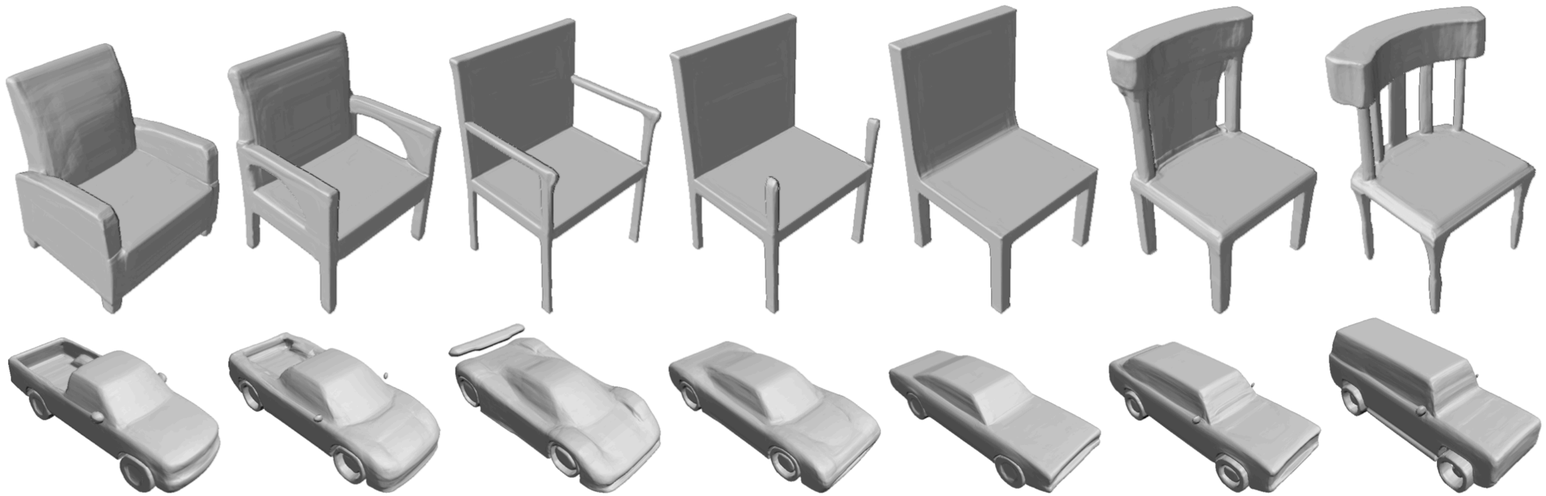


A volumetric field, made of voxels

The solution

DeepSDF - Learning a 3D generative model of SDFs

- Uses a deep neural network to learn the continuous SDF of a shape from spatial sample points and their known SDF value.
- Then, generate the mesh of that shape from the resulting SDF volumetric field with some technique such as Marching Cubes.
- Advantages:
  - Discretization happens only when the mesh needs to be generated.
  - Can be used to represent entire classes of shapes.

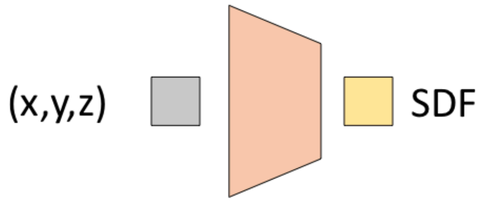


Classes of shapes

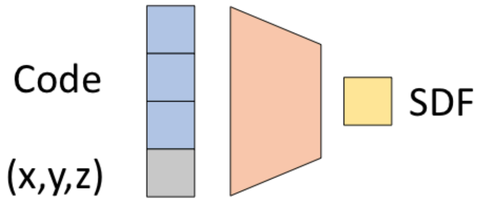
The solution

Two flavours of DeepSDF

- Single-shape DeepSDF:
  - Learns for one shape.
- Coded-shape DeepSDF:
  - Learns for a class of shapes.
  - Uses an auto-decoder only network.



(a) Single Shape DeepSDF



(b) Coded Shape DeepSDF

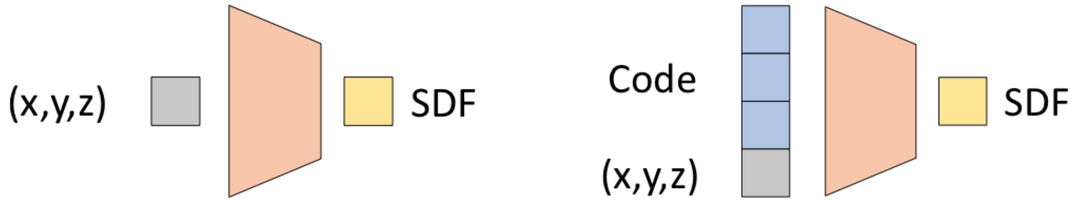
Figure 3: In the single-shape DeepSDF instantiation, the shape information is contained in the network itself whereas the coded-shape DeepSDF, the shape information is contained in a code vector that is concatenated with the 3D sample location. In both cases, DeepSDF produces the SDF value at the 3D query location,

The solution

Two flavours of DeepSDF

- Single-shape DeepSDF:
  - Learns for one shape.
- Coded-shape DeepSDF:
  - Learns for a class of shapes.
  - Uses an auto-decoder only network.

Our focus



(a) Single Shape DeepSDF **(b)** Coded Shape DeepSDF

Figure 3: In the single-shape DeepSDF instantiation, the shape information is contained in the network itself whereas the coded-shape DeepSDF, the shape information is contained in a code vector that is concatenated with the 3D sample location. In both cases, DeepSDF produces the SDF value at the 3D query location,

Our implementation

The steps

Let the journey begin…

A. Prepare the data

B. Build and train the neural network

C. Generate a mesh

The tools

What we used to code this project

- Python
- PyTorch — for building and training the neural network
- Google Colab — because Apple 😡 CUDA
- trimesh — for importing meshes
- skimage — for marching cubes
- third-party code (mesh-to-sdf) for data preprocessing — because mesh 😡 my code

Preparing the data

Which data?

- The network needs a list of tuples, each tuple containing:
  - a spatial sample point
  - its associated true computed SDF value

$$X_i = \{(\boldsymbol{x}_j, s_j) : s_j = SDF^i(\boldsymbol{x}_j)\}\,. \tag{6}$$

Preparing the data

How to prepare?

- We are provided complete 3D shape meshes.
- First, we normalize each mesh to a unit sphere.
  - The farthest point must be at a distance <= 1 from the mesh’s center.
  - In practice, we add a small margin, making the sphere’s radius = 1/1.03.
- Then, we sample around 250,000 points randomly and compute their SDF.
  - We sample more aggressively near the shape’s surface.

Build and train the neural network

Neural network architecture

- We train a multi-layer fully-connected feed-forward neural network:
  - 8 fully-connected layers, each applied with dropout.
  - Internal layers are 512-dimensional and have ReLU non-linearities.
  - Output layer has tanh non-linearities.
  - Instead of batch-normalization, we use weight-normalization.
- For single-shape DeepSDF, the network takes a spatial sample point as input.
- For coded-shape DeepSDF, we also add a latent vector as input which represents the desired shape from the specific class to train for.

The DeepSDF network

```python
class DeepSDF_single(nn.Module):
    """Multi-layer fully-connected feed-forward neural network with 8 layers, each applied with dropout"""
    def __init__(self, in_size, hidden_size, out_size):
        super().__init__()
        # Hidden layer 1
        self.linear1 = nn.Linear(in_size, hidden_size)
        # Hidden layer 2
        self.linear2 = nn.Linear(hidden_size, hidden_size)
        # Hidden layer 3
        self.linear3 = nn.Linear(hidden_size, hidden_size)
        # Hidden layer 4
        self.linear4 = nn.Linear(hidden_size, hidden_size)
        # Hidden layer 5
        self.linear5 = nn.Linear(hidden_size, hidden_size)
        # Hidden layer 6
        self.linear6 = nn.Linear(hidden_size, hidden_size)
        # Hidden layer 7
        self.linear7 = nn.Linear(hidden_size, hidden_size)
        # Output layer
        self.linear8 = nn.Linear(hidden_size, out_size)

    def forward(self, xb):
        out = self.linear1(xb)
        out = F.relu(out)
        out = self.linear2(out)
        out = F.relu(out)
        out = self.linear3(out)
        out = F.relu(out)
        out = self.linear4(out)
        out = F.relu(out)
        out = self.linear5(out)
        out = F.relu(out)
        out = self.linear6(out)
        out = F.relu(out)
        out = self.linear7(out)
        out = F.relu(out)
        out = self.linear8(out)
        return out

    def training_step(self, batch):
        point_samples, sdf_values = batch
        out = self(point_samples)       # generate predictions
        loss = loss_function(out, sdf_values)    # compute loss
        return loss

    def validation_step(self, batch):
        point_samples, sdf_values = batch
        out = self(point_samples)           # generate predictions
        loss = loss_function(out, sdf_values)        # compute loss
        acc = accuracy(out, sdf_values)     # compute accuracy
        return {'val_loss': loss, 'val_acc': acc}

    def validation_epoch_end(self, outputs):
        batch_losses = [x['val_loss'] for x in outputs]
        epoch_loss = torch.stack(batch_losses).mean()   # combine losses
        return {'val_loss': epoch_loss.item()}

        # batch_accs = [x['val_acc'] for x in outputs]
        # epoch_acc = torch.stack(batch_accs).mean()      # combine accuracies
        # return {'val_loss': epoch_loss.item(), 'val_acc': epoch_acc.item()}

    def epoch_end(self, epoch, result):
        print("Epoch [{}], val_loss: {:.4f}".format(epoch, result['val_loss'],))
        # print("Epoch [{}], val_loss: {:.4f}, val_acc: {:.4f}".format(epoch, result['val_loss'], result['val_acc']))
```

Building and training the neural network

Training the network

- We use an L1 loss function.

$$\mathcal{L}(f_\theta(\boldsymbol{x}), s) = |\,\mathrm{clamp}(f_\theta(\boldsymbol{x}), \delta) - \mathrm{clamp}(s, \delta)\,|, \quad (4)$$

  - where $\mathrm{clamp}(x, \delta) := \min(\delta, \max(-\delta, x))$

- We train the network for 1,000 epochs.
- We use the Adam optimizer.
- We try and tune a learning rate.

```python
def evaluate(model, val_loader):
    outputs = [model.validation_step(batch) for batch in val_loader]
    return model.validation_epoch_end(outputs)

def fit(epochs, lr, model, train_loader, val_loader, opt_func=torch.optim.Adam):
    history = []
    optimizer = opt_func(model.parameters(), lr=lr)
    for epoch in range(epochs):
        # Training
        for batch in train_loader:
            loss = model.training_step(batch)
            loss.backward()
            optimizer.step()
            optimizer.zero_grad()
        # Validation
        result = evaluate(model, val_loader)
        model.epoch_end(epoch, result)
        history.append(result)
    return history
```

Functions to implement the training

Generating a mesh

Voxels and marching cubes

- The trained model approximates the SDF for a specific mesh or a class of meshes.
- We can generate the volumetric field by providing the model with a N x N x N grid of points to evaluate and return their SDF values.
  - N is the resolution of the grid, we used N = 100.
- We can then give the volumetric field as input to a marching cube algorithm which will compute vertices, faces and normals based on the voxels where SDF = 0.

```python
# author: Sylvain Laporte
# program: reconstruct_mesh_from_model.py
# date: 2020-12-08
# object: Reconstructs a mesh from the trained DeepSDF model.

import torch
import deepSDF_model
import trimesh
from skimage import measure

# Load the model

PATH = "trained_models/horse-1-model-10.pt"

input_size = 3
hidden_size = 512
out_size = 1

model = deepSDF_model.DeepSDF_single(input_size, hidden_size=hidden_size, out_size=out_size)
model.load_state_dict(torch.load(PATH, map_location=torch.device('cpu')))
model.eval()

# Generate voxels

RESOLUTION = 100
voxels = torch.zeros(RESOLUTION, RESOLUTION, RESOLUTION)

intervals = [x / RESOLUTION for x in range(RESOLUTION)]
points = [[x, y, z] for x in intervals for y in intervals for z in intervals]
points = torch.tensor(points)

sdfs = model(points)

voxels = sdfs.view(RESOLUTION, RESOLUTION, RESOLUTION).detach().numpy()

# Generate mesh using marching cubes

vertices, faces, normals, _ = measure.marching_cubes(voxels, 0)
mesh = trimesh.Trimesh(vertices, faces, normals)
trimesh.repair.fix_normals(mesh)

# Show the resulting mesh

mesh.show()
```

Mesh generating script

Hurdles

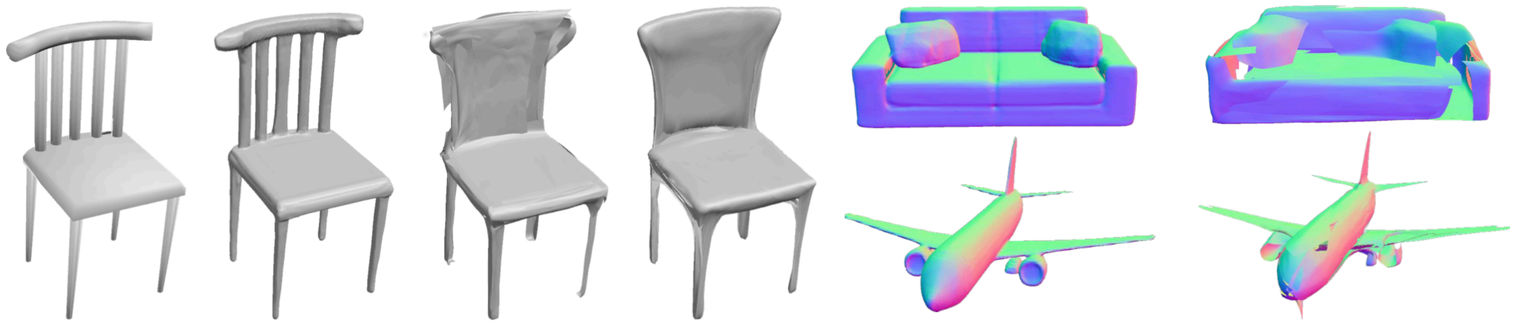
Lots of new stuff

- Getting to know PyTorch beyond MNIST
- Jupyter notebook « eating » my work
- Also, getting to know Google Colab beyond MNIST
  - render context, custom modules, getting things onto the GPU
- What kind of voxels does a marching cubes eats?

Results

Expected results

From the paper — shape reconstruction

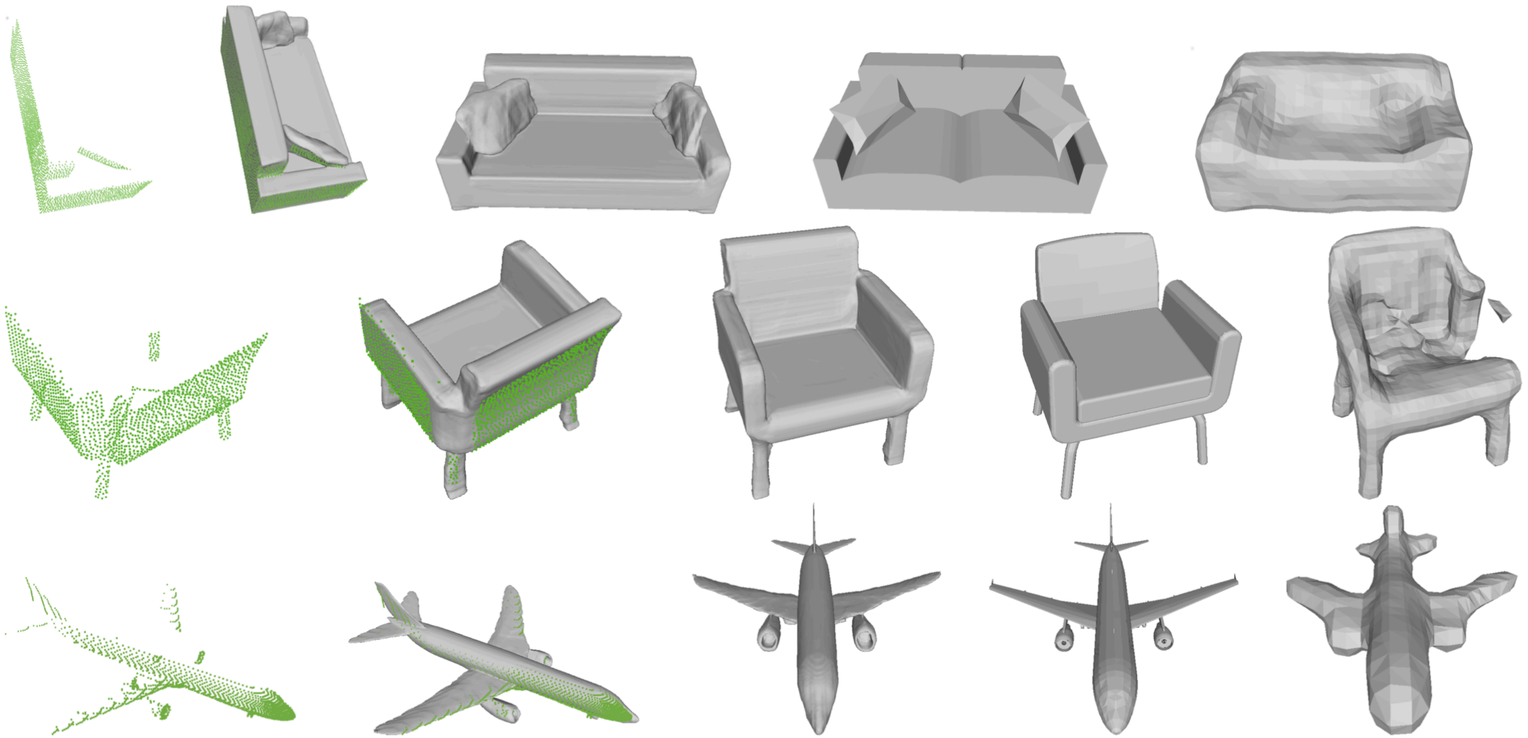


(a) Ground-truth **(b)** Our Result **(c)** [22]-25 patch **(d)** [22]-sphere **(e)** Our Result **(f)** [22]-25 patch

Figure 6: Reconstruction comparison between DeepSDF and AtlasNet [22] (with 25-plane and sphere parameterization) for test shapes. Note that AtlasNet fails to capture the fine details of the chair, and that (f) shows holes on the surface of sofa and the plane.

Expected results

From the paper — shape completion



(a) Input Depth **(b)** Completion (ours) **(c)** Second View (ours) **(d)** Ground truth **(e)** 3D-EPN

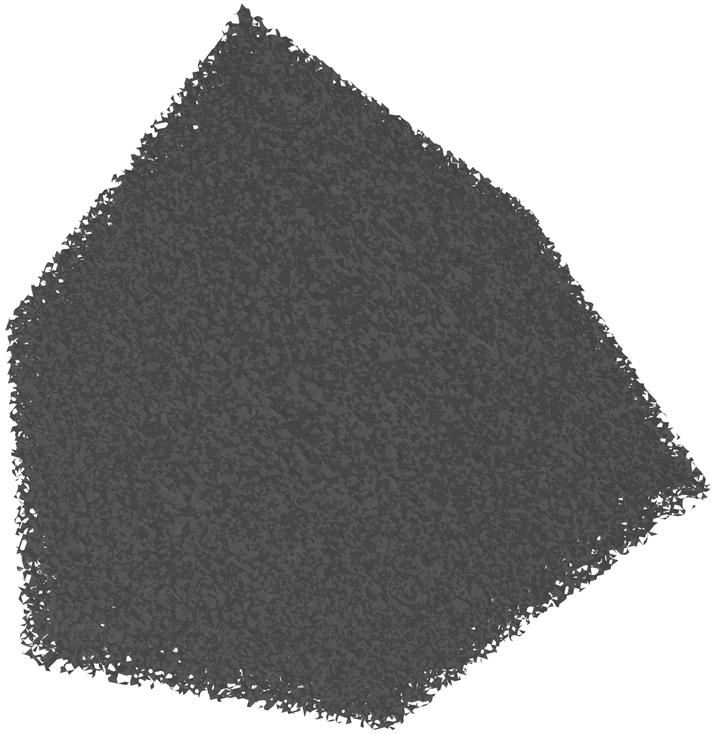
Figure 8: For a given depth image visualized as a green point cloud, we show a comparison of shape completions from our DeepSDF approach against the true shape and 3D-EPN.

Our results

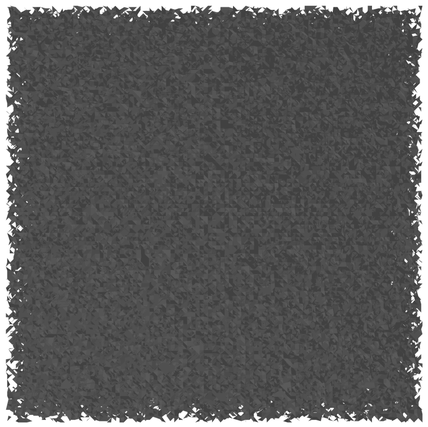
Drum roll…

Our results

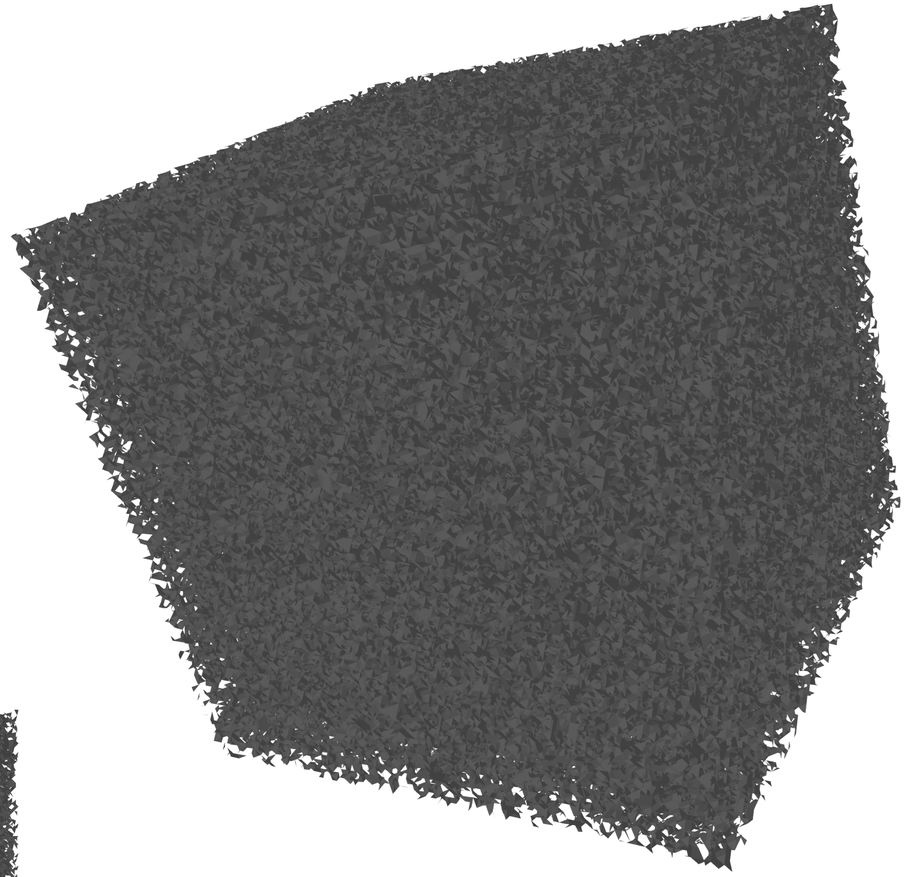
Not much… or maybe…



5 epochs, lr = 0.5



10 epochs, lr = 0.05



1000 epochs, lr = 0.005

Our results

Nope. This is an invisible horse.

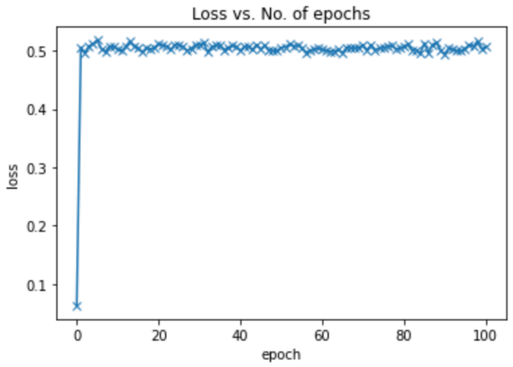
RuntimeError: No surface found at the given iso value.

10 epochs, lr = 0.005

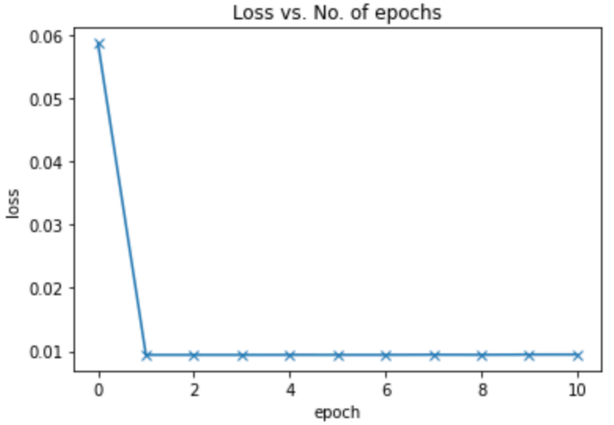
Known issues

Observations as of now

- For some reason, our loss does not seem to variate much
  - For the cube mesh, the loss varies between 0.48 and 0.52.
  - For the horse mesh, the loss just flatten at 0.01.



Training cube for 10 epochs

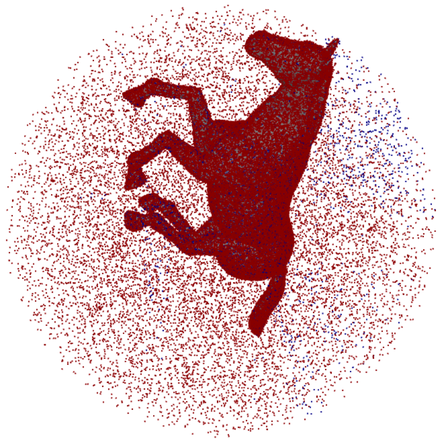


Training horse-1 for 10 epochs

Known issues

Observations as of now

- For the horse mesh, it seems that the SDF values for the spatial sample points are miscalculated.
- Some of the outer points are negative, whereas some inner points are positive.

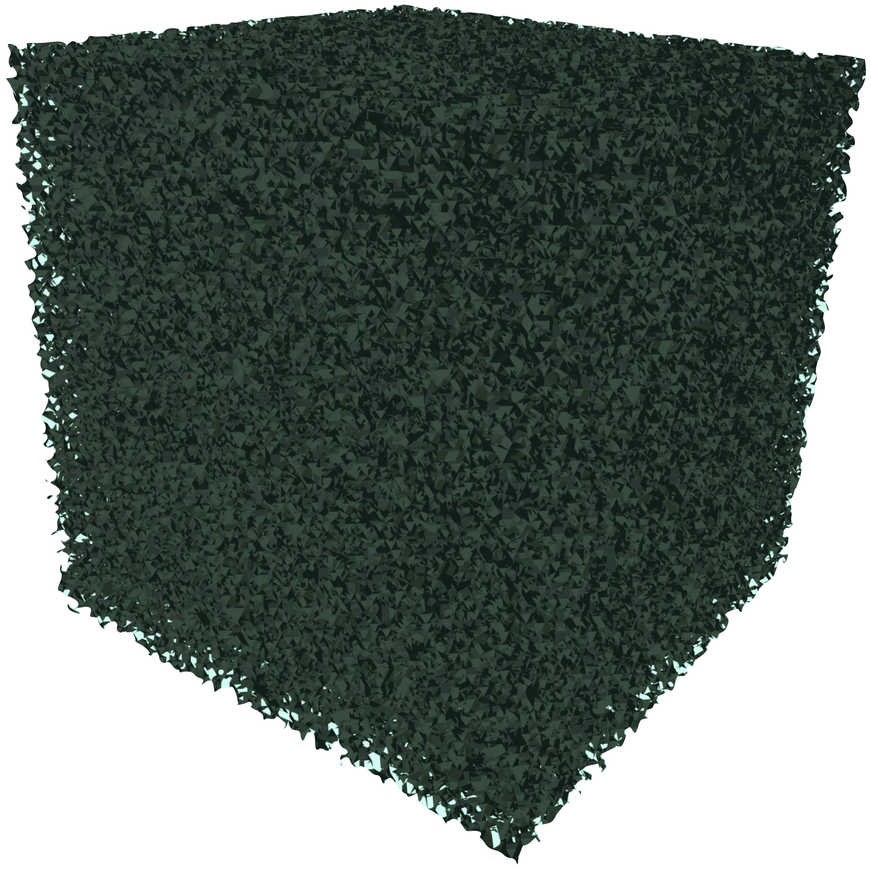


View of the spatial sample points for horse-1

An idea of our extension

Extension

- We did plan on experimenting with loss functions. Our goal was to maybe reduce training time or improve accuracy.
- However, much time has been spent debugging the existing loss function with still inconclusive results.
- Nevertheless, though beyond the scope of the original paper, it seems we may have found a slightly overengineered method to model some realistic cube bushes…



=>

Thank you!