3DV2022: tutorial

310551083 許嘉倫

1 Tutorial 1: Fit mesh

In this tutorial, we need to deform a source mesh (e.g. sphere) to form a target mesh using 3D loss functions.
We will cover:

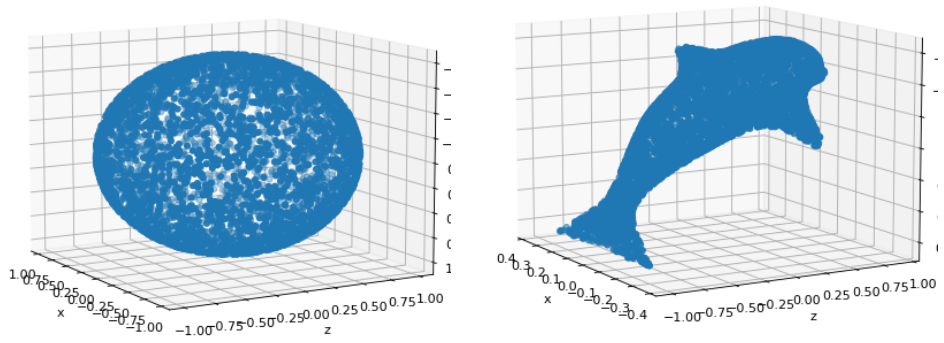
- How to load a mesh from an .obj file
- How to use the PyTorch3D Meshes data structure
- How to use 4 different PyTorch3D mesh loss functions
- How to set up an optimization loop

1.1 Load the dolphin mesh

Using load_obj to read 3D model. Scale normalize and center the target mesh to fit in a sphere of radius 1 centered at (0, 0, 0). Normalizing the target mesh can speed up the optimization. Finally, construct a meshes structure for the target mesh. Figure 1 shows the source and target mesh.

```
# Load the dolphin mesh.
trg_obj = os.path.join('dolphin.obj')
# We read the target 3D model using load_obj
verts, faces, aux = load_obj(trg_obj)

# verts is a FloatTensor of shape (V, 3) where V is the number of vertices in the mesh
# faces is an object which contains the following LongTensors: verts_idx, normals_idx and
# textures_idx
# For this tutorial, normals and textures are ignored.
faces_idx = faces.verts_idx.to(device)
verts = verts.to(device)

# We scale normalize and center the target mesh to fit in a sphere of radius 1 centered at
# (0,0,0).
# (scale, center) will be used to bring the predicted mesh to its original center and scale
# Note that normalizing the target mesh, speeds up the optimization but is not necessary!
center = verts.mean(0)
verts = verts - center
scale = max(verts.abs().max(0)[0])
verts = verts / scale

# We construct a Meshes structure for the target mesh
trg_mesh = Meshes(verts=[verts], faces=[faces_idx])
```

```
# We initialize the source shape to be a sphere of radius 1
src_mesh = ico_sphere(4, device)
```



(a) Source mesh: sphere (b) Target mesh: dolphin

Figure 1: Deform a source mesh to form a target mesh

1.2 Optimization loop

We will learn to deform the source mesh by offsetting its vertices. The shape of the deform parameters is equal to the total number of vertices in src_mesh.
We will use 4 different PyTorch3D mesh loss function:

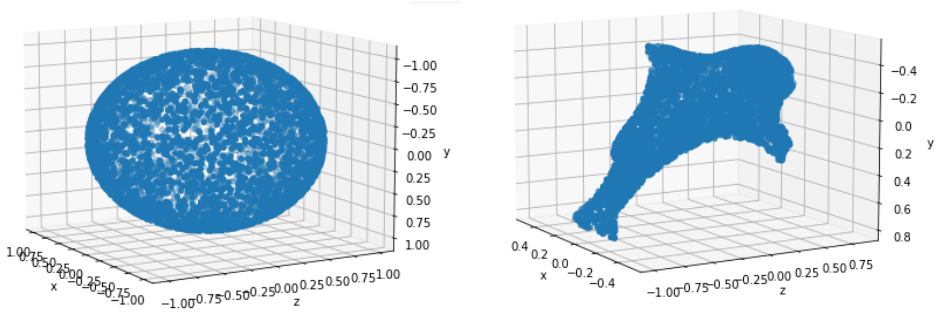
- chamfer loss: chamfer distance between predicted and target point cloud.
- edge loss: edge length of the predicted mesh
- normal loss: normal consistency
- laplacian loss: laplacian smoothing

Figure 2 shows the predicted mesh of different iteration. And figure 3 shows the training loss.

```
# We will learn to deform the source mesh by offsetting its vertices
# The shape of the deform parameters is equal to the total number of vertices in
# src_mesh
deform_verts = torch.full(src_mesh.verts_packed().shape, 0.0, device=device, requires_grad=True)
# The optimizer
optimizer = torch.optim.SGD([deform_verts], lr=1.0, momentum=0.9)
# Number of optimization steps
Niter = 2000
# Weight for the chamfer loss
w_chamfer = 1.0
# Weight for mesh edge loss
w_edge = 1.0
# Weight for mesh normal consistency
w_normal = 0.01
# Weight for mesh laplacian smoothing
w_laplacian = 0.1
# Plot period for the losses
plot_period = 250
```

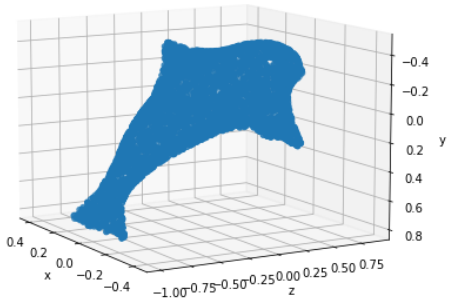
```
loop = tqdm(range(Niter))

chamfer_losses = []
laplacian_losses = []
edge_losses = []
normal_losses = []

%matplotlib inline

for i in loop:
    # Initialize optimizer
    optimizer.zero_grad()

    # Deform the mesh
    new_src_mesh = src_mesh.offset_verts(deform_verts)

    # We sample 5k points from the surface of each mesh
    sample_trg = sample_points_from_meshes(trg_mesh, 5000)
    sample_src = sample_points_from_meshes(new_src_mesh, 5000)

    # We compare the two sets of pointclouds by computing (a) the chamfer loss
    loss_chamfer, _ = chamfer_distance(sample_trg, sample_src)

    # and (b) the edge length of the predicted mesh
    loss_edge = mesh_edge_loss(new_src_mesh)

    # mesh normal consistency
    loss_normal = mesh_normal_consistency(new_src_mesh)

    # mesh laplacian smoothing
    loss_laplacian = mesh_laplacian_smoothing(new_src_mesh, method="uniform")

    # Weighted sum of the losses
    loss = loss_chamfer * w_chamfer + loss_edge * w_edge + loss_normal * w_normal + loss_laplaci

    # Print the losses
    loop.set_description('total_loss = %.6f' % loss)

    # Save the losses for plotting
    chamfer_losses.append(float(loss_chamfer.detach().cpu()))
    edge_losses.append(float(loss_edge.detach().cpu()))
    normal_losses.append(float(loss_normal.detach().cpu()))
    laplacian_losses.append(float(loss_laplacian.detach().cpu()))

    # Plot mesh
    if i % plot_period == 0:
        plot_pointcloud(new_src_mesh, title="iter: %d" % i)

    # Optimization step
```

```
    loss.backward()
    optimizer.step()
```

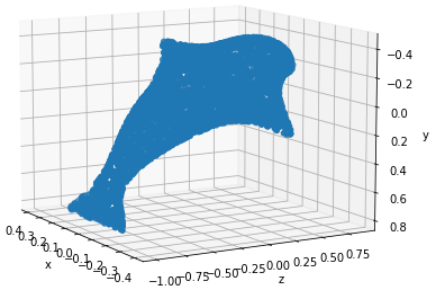


(a) Predicted mesh of iter 0

(b) Predicted mesh of iter 500



(c) Predicted mesh of iter 1000



(d) Predicted mesh of iter 1500

Figure 2: Predicted mesh of different iteration

Figure 3: Training loss of each iteration

2 Tutorial 2: Render textured meshes

2.1 Load a mesh and texture file

Load an .obj file and its associated .mtl file and create a textures and meshes object. Figure 4 shows the mesh texture of cow.

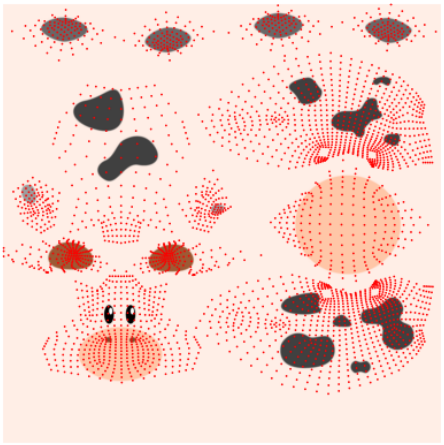


Figure 4: Mesh texture: cow

```
# Set paths
DATA_DIR = "./data"
obj_filename = os.path.join(DATA_DIR, "cow_mesh/cow.obj")

# Load obj file
mesh = load_objs_as_meshes([obj_filename], device=device)
plt.figure(figsize=(7,7))
texturesuv_image_matplotlib(mesh.textures, subsample=None)
plt.axis("off");
```

2.2 Create a renderer

In this example we will first create a **renderer** which uses a **perspective camera**, a **point light** and applies **Phong shading**. Then we learn how to vary different components using the modular API. Here we set the output size of image as 512x512.

```
# Initialize a camera.
# With world coordinates +Y up, +X left and +Z in, the front of the cow is
# facing the -Z direction.
# So we move the camera by 180 in the azimuth direction so it is facing the
# front of the cow.
R, T = look_at_view_transform(2.7, 0, 180)
cameras = FoVPerspectiveCameras(device=device, R=R, T=T)

# Define the settings for rasterization and shading. Here we set the output
# image to be of size 512x512.
# As we are rendering images for visualization purposes only we will set
# faces_per_pixel=1 and blur_radius=0.0.
# We also set bin_size and max_faces_per_bin to None which ensure that
# the faster coarse-to-fine rasterization method is used.
raster_settings = RasterizationSettings(
    image_size=512,
    blur_radius=0.0,
```

```
    faces_per_pixel=1,
)

# Place a point light in front of the object. As mentioned above, the front
# of the cow is facing the -z direction.
lights = PointLights(device=device, location=[[0.0, 0.0, -3.0]])

# Create a Phong renderer by composing a rasterizer and a shader. The textured
# Phong shader will interpolate the texture uv coordinates for each vertex,
# sample from a texture image and apply the Phong lighting model
renderer = MeshRenderer(
    rasterizer=MeshRasterizer(
        cameras=cameras,
        raster_settings=raster_settings
    ),
    shader=SoftPhongShader(
        device=device,
        cameras=cameras,
        lights=lights
    )
)
```

2.3 Render the mesh

In figure 5.a, the light is in front of the object so it is bright and the image has specular highlights. Also, we can simply update the location of the lights and pass them into the call to the renderer. In figure 5.b, the image is now dark as there is only ambient lighting, and there are no specular highlights.

```
images = renderer(mesh)
plt.figure(figsize=(10, 10))
plt.imshow(images[0, ..., :3].cpu().numpy())
plt.axis("off");
```

(a) The light point is in front of the cow

(b) The light point is behind the cow

Figure 5: Render the mesh

2.4 Batched rendering

One of the core design choices of the PyTorch3D API is to support batched inputs for all components. The renderer and associated components can take batched inputs and render a batch of output images in one forward pass. We will now use this feature to render the mesh from many different viewpoints. Figure 6 shows the result.

```
# Set batch size - this is the number of different viewpoints from which we
# want to render the mesh.
batch_size = 20

# Create a batch of meshes by repeating the cow mesh and associated textures.
# Meshes has a useful `extend` method which allows us do this very easily.
# This also extends the textures.
meshes = mesh.extend(batch_size)

# Get a batch of viewing angles.
elev = torch.linspace(0, 180, batch_size)
azim = torch.linspace(-180, 180, batch_size)

# All the cameras helper methods support mixed type inputs and broadcasting. So we can
# view the camera from the same distance and specify dist=2.7 as a float,
# and then specify elevation and azimuth angles for each viewpoint as tensors.
R, T = look_at_view_transform(dist=2.7, elev=elev, azim=azim)
cameras = FoVPerspectiveCameras(device=device, R=R, T=T)

# Move the light back in front of the cow which is facing the -z direction.
lights.location = torch.tensor([[0.0, 0.0, -3.0]], device=device)
# We can pass arbitrary keyword arguments to the rasterizer/shader via the renderer
# so the renderer does not need to be reinitialized if any of the settings change.
images = renderer(meshes, cameras=cameras, lights=lights)
image_grid(images.cpu().numpy(), rows=4, cols=5, rgb=True)
```

3 Tutorial 3: Render color pointclouds

3.1 Load a point cloud and corresponding colors

Pointclouds is a unique data structure provided in PyTorch3D for working with batches of point clouds of different sizes.

```
# Set paths
DATA_DIR = "./data"
obj_filename = os.path.join(DATA_DIR, "PittsburghBridge/pointcloud.npz")
# Load point cloud
pointcloud = np.load(obj_filename)
verts = torch.Tensor(pointcloud['verts']).to(device)
rgb = torch.Tensor(pointcloud['rgb']).to(device)
point_cloud = Pointclouds(points=[verts], features=[rgb])
```

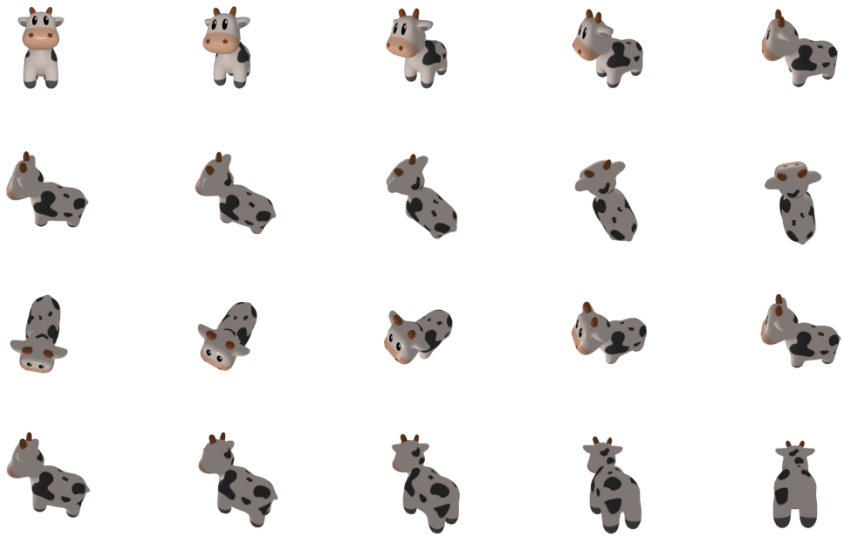


Figure 6: Batched rendering: cow

3.2 Create a renderer

Same as section 2.2, we will first create a renderer which uses an orthographic camera, and applies alpha compositing. Then we learn how to vary different components using the modular API. Figure 7 shows the rendered result of point cloud.

```
# Initialize a camera.
R, T = look_at_view_transform(20, 10, 0)
cameras = FoVOrthographicCameras(device=device, R=R, T=T, znear=0.01)

# Define the settings for rasterization and shading. Here we set the output
# image to be of size 512x512.
# As we are rendering images for visualization purposes only we will set
# faces_per_pixel=1 and blur_radius=0.0. Refer to raster_points.py for
# explanations of these parameters.
raster_settings = PointsRasterizationSettings(
    image_size=512,
    radius = 0.003,
    points_per_pixel = 10
)
# Create a points renderer by compositing points using an alpha compositor
# (nearer points are weighted more heavily). See [1] for an explanation.
rasterizer = PointsRasterizer(cameras=cameras, raster_settings=raster_settings)
renderer = PointsRenderer(
    rasterizer=rasterizer,
    compositor=AlphaCompositor()
)
images = renderer(point_cloud)
plt.figure(figsize=(10, 10))
plt.imshow(images[0, ..., :3].cpu().numpy())
plt.axis("off");
```

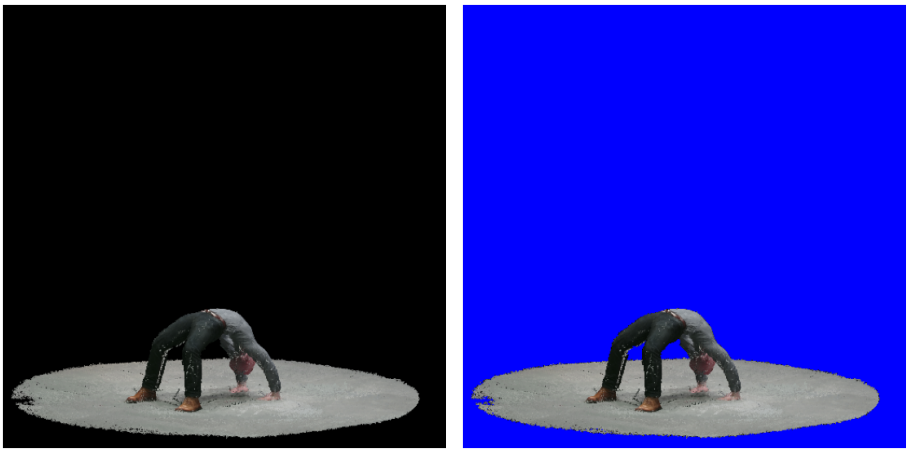


Figure 7: Render the point cloud

3.3 Change background color

It also can change different background color as figure 7 shows.

```
renderer = PointsRenderer(
    rasterizer=rasterizer,
    # Pass in background_color to the alpha compositor, setting the background color
    # to the 3 item tuple, representing rgb on a scale of 0 -> 1, in this case blue
    compositor=AlphaCompositor(background_color=(0, 0, 1))
)
images = renderer(point_cloud)

plt.figure(figsize=(10, 10))
plt.imshow(images[0, ..., :3].cpu().numpy())
plt.axis("off");
```

4 Tutorial 4: Fit a mesh via rendering

4.1 Load and normalize data

Same as section 1.1, we scale normalize and center the target mesh to fit in a sphere of radius 1 centered at (0,0,0). (scale, center) will be used to bring the predicted mesh to its original center and scale. Normalizing the target mesh can speed up the optimization.

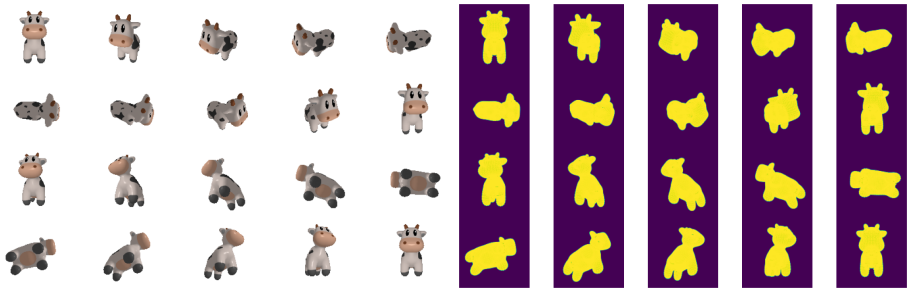
4.2 Dataset creation

We sample different camera positions that encode multiple viewpoints of the cow. We create a renderer with a shader that performs texture map interpolation. We render a synthetic dataset of images of the textured cow mesh from multiple viewpoints. Figure 8 shows the cow dataset.

```
# the number of different viewpoints from which we want to render the mesh.
num_views = 20

# Get a batch of viewing angles.
elev = torch.linspace(0, 360, num_views)
azim = torch.linspace(-180, 180, num_views)

# Place a point light in front of the object. As mentioned above, the front of
# the cow is facing the -z direction.
lights = PointLights(device=device, location=[[0.0, 0.0, -3.0]])

# Initialize an OpenGL perspective camera that represents a batch of different
# viewing angles. All the cameras helper methods support mixed type inputs and
# broadcasting. So we can view the camera from the a distance of dist=2.7, and
# then specify elevation and azimuth angles for each viewpoint as tensors.
R, T = look_at_view_transform(dist=2.7, elev=elev, azim=azim)
cameras = OpenGLPerspectiveCameras(device=device, R=R, T=T)

# We arbitrarily choose one particular view that will be used to visualize
# results
camera = OpenGLPerspectiveCameras(device=device, R=R[None, 1, ...],
                                  T=T[None, 1, ...])

# Define the settings for rasterization and shading. Here we set the output
# image to be of size 128X128. As we are rendering images for visualization
# purposes only we will set faces_per_pixel=1 and blur_radius=0.0. Refer to
# rasterize_meshes.py for explanations of these parameters.  We also leave
# bin_size and max_faces_per_bin to their default values of None, which sets
# their values using heuristics and ensures that the faster coarse-to-fine
# rasterization method is used.  Refer to docs/notes/renderer.md for an
# explanation of the difference between naive and coarse-to-fine rasterization.
raster_settings = RasterizationSettings(
    image_size=128,
    blur_radius=0.0,
    faces_per_pixel=1,
)

# Create a Phong renderer by composing a rasterizer and a shader. The textured
# Phong shader will interpolate the texture uv coordinates for each vertex,
# sample from a texture image and apply the Phong lighting model
renderer = MeshRenderer(
    rasterizer=MeshRasterizer(
        cameras=camera,
        raster_settings=raster_settings
    ),
    shader=SoftPhongShader(
        device=device,
        cameras=camera,
        lights=lights
```

```
    )
)

# Create a batch of meshes by repeating the cow mesh and associated textures.
# Meshes has a useful `extend` method which allows us do this very easily.
# This also extends the textures.
meshes = mesh.extend(num_views)

# Render the cow mesh from each viewing angle
target_images = renderer(meshes, cameras=cameras, lights=lights)

# Our multi-view cow dataset will be represented by these 2 lists of tensors,
# each of length num_views.
target_rgb = [target_images[i, ..., :3] for i in range(num_views)]
target_cameras = [OpenGLPerspectiveCameras(device=device, R=R[None, i, ...],
                                           T=T[None, i, ...]) for i in range(num_views)]
# RGB images
image_grid(target_images.cpu().numpy(), rows=4, cols=5, rgb=True)
plt.show()
```

And then, we also create cow silhouette dataset. we will render a dataset of silhouette images as figure 8 shows. Most shaders in PyTorch3D will output an alpha channel along with the RGB image as a 4th channel in an RGBA image. The alpha channel encodes the probability that each pixel belongs to the foreground of the object. We construct a soft silhouette shader to render this alpha channel.



(a) RGB dataset (b) Silhouette dataset

Figure 8: Dataset creation

4.3 Mesh prediction via silhouette rendering

In this section, we predict a mesh by observing those target images without any knowledge of the ground truth cow mesh. We assume we know the position of the cameras and lighting. We use 4 different loss function to optimize.

- Silhouette loss: squared L2 distance between the predicted and the target silhouette
- edge loss: edge length of the predicted mesh
- normal loss: normal consistency
- laplacian loss: laplacian smoothing

Figure 9 shows the training loss of silhouette dataset.

Figure 9: Silhouette training loss

4.4 Mesh and texture prediction via textured rendering

We can predict both the mesh and its texture if we add an additional loss based on the comparing a predicted rendered RGB image to the target image. As before, we start with a sphere mesh. We learn both translational offsets and RGB texture colors for each vertex in the sphere mesh. Since our loss is based on rendered RGB pixel values instead of just the silhouette, we use a **SoftPhongShader** instead of a **SoftSilhouetteShader**.
We use 5 different loss function to optimize:

- rgb loss: squared L2 distance between the predicted rgb image and the target image
- silhouette loss: squared L2 distance between the predicted and the target silhouette
- edge loss: edge length of the predicted mesh
- normal loss: normal consistency
- laplacian loss: laplacian smoothing

I found some issue of this section when optimizing. The rgb loss is NaN, this will result in total loss is also NaN. To solve this problem, we need to modify provided code. We could fix the issue by setting **perspective_correct=False** in the RasterizationSettings as below.

```
raster_settings_soft = RasterizationSettings(
    image_size=128,
    blur_radius=np.log(1. / 1e-4 - 1.)*sigma,
    faces_per_pixel=50,
    perspective_correct=False
)
```

Figure 10 shows the predicted and target mesh and figure 11 shows the training loss.

5 Tutorial 5: Fit a volume via raymarching

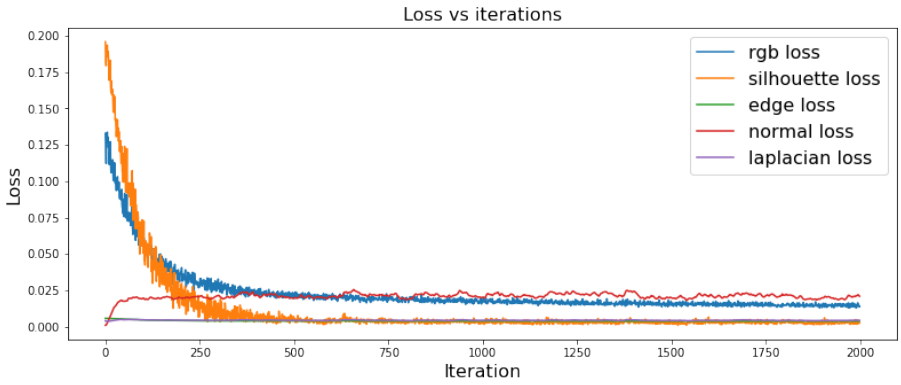
This tutorial shows how to fit a volume given a set of views of a scene using differentiable volumetric rendering.

5.1 Generate images of the scene and masks

Generate a batch of image and silhouette tensors that are produced by the cow mesh renderer. And a set of cameras corresponding to each render.

(a) Predicted mesh (b) Target mesh

Figure 10: Predicted and target mesh



(a) Predicted mesh

Figure 11: Predicted and target mesh

5.2 Initialize the volumetric renderer

Initialize a volumetric renderer that emits a ray from each pixel of a target image and samples a set of uniformly-spaced points along the ray. At each ray-point, the corresponding density and color value is obtained by querying the corresponding location in the volumetric model of the scene.

```
# render_size describes the size of both sides of the
# rendered images in pixels. We set this to the same size
# as the target images. I.e. we render at the same
# size as the ground truth images.
render_size = target_images.shape[1]

# Our rendered scene is centered around (0,0,0)
# and is enclosed inside a bounding box
# whose side is roughly equal to 3.0 (world units).
volume_extent_world = 3.0

# 1) Instantiate the raysampler.
# Here, NDCMultinomialRaysampler generates a rectangular image
# grid of rays whose coordinates follow the PyTorch3D
# coordinate conventions.
# Since we use a volume of size 128^3, we sample n_pts_per_ray=150,
```

```
# which roughly corresponds to a one ray-point per voxel.
# We further set the min_depth=0.1 since there is no surface within
# 0.1 units of any camera plane.
raysampler = NDCMultinomialRaysampler(
    image_width=render_size,
    image_height=render_size,
    n_pts_per_ray=150,
    min_depth=0.1,
    max_depth=volume_extent_world,
)


# 2) Instantiate the raymarcher.
# Here, we use the standard EmissionAbsorptionRaymarcher
# which marches along each ray in order to render
# each ray into a single 3D color vector
# and an opacity scalar.
raymarcher = EmissionAbsorptionRaymarcher()

# Finally, instantiate the volumetric render
# with the raysampler and raymarcher objects.
renderer = VolumeRenderer(
    raysampler=raysampler, raymarcher=raymarcher,
)
```

5.3 Initialize the volumetric model

We instantiate a volumetric model of the scene. This quantizes the 3D space to cubical voxels, where each voxel is described with a 3D vector representing the voxel's RGB color and a density scalar which describes the opacity of the voxel (ranging between [0-1], the higher the more opaque).

```
class VolumeModel(torch.nn.Module):
    def __init__(self, renderer, volume_size=[64] * 3, voxel_size=0.1):
        super().__init__()
        # After evaluating torch.sigmoid(self.log_colors), we get
        # densities close to zero.
        self.log_densities = torch.nn.Parameter(-4.0 * torch.ones(1, *volume_size))
        # After evaluating torch.sigmoid(self.log_colors), we get
        # a neutral gray color everywhere.
        self.log_colors = torch.nn.Parameter(torch.zeros(3, *volume_size))
        self._voxel_size = voxel_size
        # Store the renderer module as well.
        self._renderer = renderer

    def forward(self, cameras):
        batch_size = cameras.R.shape[0]

        # Convert the log-space values to the densities/colors
        densities = torch.sigmoid(self.log_densities)
        colors = torch.sigmoid(self.log_colors)
```

```
        # Instantiate the Volumes object, making sure
        # the densities and colors are correctly
        # expanded batch_size-times.
        volumes = Volumes(
            densities = densities[None].expand(
                batch_size, *self.log_densities.shape),
            features = colors[None].expand(
                batch_size, *self.log_colors.shape),
            voxel_size=self._voxel_size,
        )

        # Given cameras and volumes, run the renderer
        # and return only the first output value
        # (the 2nd output is a representation of the sampled
        # rays which can be omitted for our purpose).
        return self._renderer(cameras=cameras, volumes=volumes)[0]

# A helper function for evaluating the smooth L1 (huber) loss
# between the rendered silhouettes and colors.
def huber(x, y, scaling=0.1):
    diff_sq = (x - y) ** 2
    loss = ((1 + diff_sq / (scaling**2)).clamp(1e-4).sqrt() - 1) * float(scaling)
    return loss
```

5.4 Fit the volume

In order to fit the volume, we render it from the viewpoints of the target cameras and compare the resulting renders with the observed target images and target silhouettes.

The comparison is done by evaluating the mean huber (smooth-l1) error between corresponding pairs of target images/rendered images and target silhouettes/rendered silhouettes. Figure 12 shows the predicted volume from different viewpoints.

```
# First move all relevant variables to the correct device.
target_cameras = target_cameras.to(device)
target_images = target_images.to(device)
target_silhouettes = target_silhouettes.to(device)

# Instantiate the volumetric model.
# We use a cubical volume with the size of
# one side = 128. The size of each voxel of the volume
# is set to volume_extent_world / volume_size s.t. the
# volume represents the space enclosed in a 3D bounding box
# centered at (0, 0, 0) with the size of each side equal to 3.
volume_size = 128
volume_model = VolumeModel(
    renderer,
    volume_size=[volume_size] * 3,
    voxel_size = volume_extent_world / volume_size,
).to(device)
```

```
# Instantiate the Adam optimizer. We set its master learning rate to 0.1.
lr = 0.1
optimizer = torch.optim.Adam(volume_model.parameters(), lr=lr)

# We do 300 Adam iterations and sample 10 random images in each minibatch.
batch_size = 10
n_iter = 300
for iteration in range(n_iter):

    # In case we reached the last 75% of iterations,
    # decrease the learning rate of the optimizer 10-fold.
    if iteration == round(n_iter * 0.75):
        print('Decreasing LR 10-fold ...')
        optimizer = torch.optim.Adam(
            volume_model.parameters(), lr=lr * 0.1
        )

    # Zero the optimizer gradient.
    optimizer.zero_grad()

    # Sample random batch indices.
    batch_idx = torch.randperm(len(target_cameras))[:batch_size]

    # Sample the minibatch of cameras.
    batch_cameras = FoVPerspectiveCameras(
        R = target_cameras.R[batch_idx],
        T = target_cameras.T[batch_idx],
        znear = target_cameras.znear[batch_idx],
        zfar = target_cameras.zfar[batch_idx],
        aspect_ratio = target_cameras.aspect_ratio[batch_idx],
        fov = target_cameras.fov[batch_idx],
        device = device,
    )

    # Evaluate the volumetric model.
    rendered_images, rendered_silhouettes = volume_model(
        batch_cameras
    ).split([3, 1], dim=-1)

    # Compute the silhouette error as the mean huber
    # loss between the predicted masks and the
    # target silhouettes.
    sil_err = huber(
        rendered_silhouettes[..., 0], target_silhouettes[batch_idx],
    ).abs().mean()

    # Compute the color error as the mean huber
    # loss between the rendered colors and the
```

```
    # target ground truth images.
    color_err = huber(
        rendered_images, target_images[batch_idx],
    ).abs().mean()

    # The optimization loss is a simple
    # sum of the color and silhouette errors.
    loss = color_err + sil_err

    # Print the current values of the losses.
    if iteration % 10 == 0:
        print(
            f'Iteration {iteration:05d}:'
            + f' color_err = {float(color_err):1.2e}'
            + f' mask_err = {float(sil_err):1.2e}'
        )

    # Take the optimization step.
    loss.backward()
    optimizer.step()

    # Visualize the renders every 40 iterations.
    if iteration % 40 == 0:
        # Visualize only a single randomly selected element of the batch.
        im_show_idx = int(torch.randint(low=0, high=batch_size, size=(1,)))
        fig, ax = plt.subplots(2, 2, figsize=(10, 10))
        ax = ax.ravel()
        clamp_and_detach = lambda x: x.clamp(0.0, 1.0).cpu().detach().numpy()
        ax[0].imshow(clamp_and_detach(rendered_images[im_show_idx]))
        ax[1].imshow(clamp_and_detach(target_images[batch_idx[im_show_idx], ..., :3]))
        ax[2].imshow(clamp_and_detach(rendered_silhouettes[im_show_idx, ..., 0]))
        ax[3].imshow(clamp_and_detach(target_silhouettes[batch_idx[im_show_idx]]))
        for ax_, title_ in zip(
            ax,
            ("rendered image", "target image", "rendered silhouette", "target silhouette")
        ):
            ax_.grid("off")
            ax_.axis("off")
            ax_.set_title(title_)
        fig.canvas.draw(); fig.show()
        display.clear_output(wait=True)
        display.display(fig)
```

6 Tutorial 6: Fit a simple Neural Radiance Field via raymarching

This tutorial shows how to fit Neural Radiance Field given a set of views of a scene using differentiable implicit function rendering.

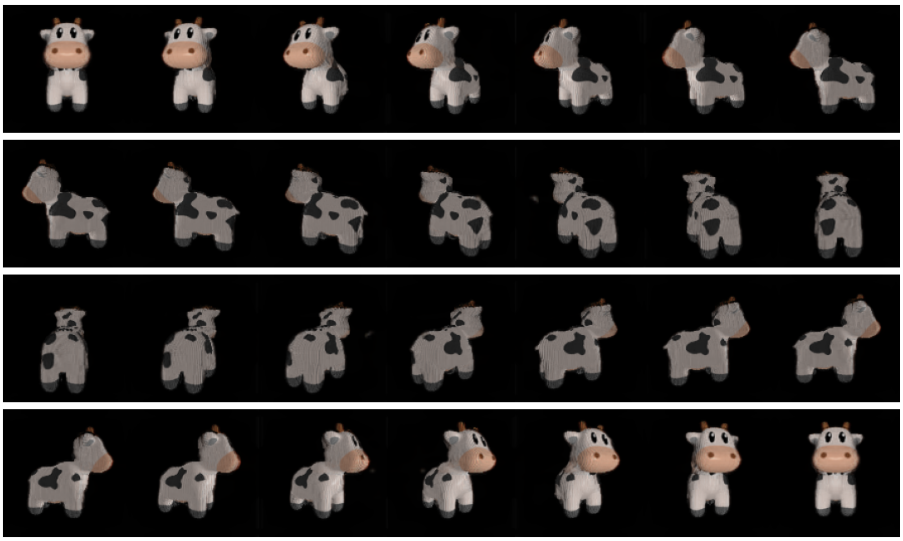


Figure 12: Predicted volume from different viewpoints

6.1 Generate images of the scene and masks

Same as section 5.1, generate a batch of image and silhouette tensors that are produced by the cow mesh renderer. And a set of cameras corresponding to each render.

6.2 Initialize the implicit renderer

Initializes an implicit renderer that emits a ray from each pixel of a target image and samples a set of uniformly-spaced points along the ray. At each ray-point, the corresponding density and color value is obtained by querying the corresponding location in the neural model of the scene.

```
# render_size describes the size of both sides of the
# rendered images in pixels. Since an advantage of
# Neural Radiance Fields are high quality renders
# with a significant amount of details, we render
# the implicit function at double the size of
# target images.
render_size = target_images.shape[1] * 2

# Our rendered scene is centered around (0,0,0)
# and is enclosed inside a bounding box
# whose side is roughly equal to 3.0 (world units).
volume_extent_world = 3.0

# 1) Instantiate the raysamplers.

# Here, NDCMultinomialRaysampler generates a rectangular image
# grid of rays whose coordinates follow the PyTorch3D
# coordinate conventions.
raysampler_grid = NDCMultinomialRaysampler(
    image_height=render_size,
    image_width=render_size,
```

```
    n_pts_per_ray=128,
    min_depth=0.1,
    max_depth=volume_extent_world,
)

# MonteCarloRaysampler generates a random subset
# of `n_rays_per_image` rays emitted from the image plane.
raysampler_mc = MonteCarloRaysampler(
    min_x = -1.0,
    max_x = 1.0,
    min_y = -1.0,
    max_y = 1.0,
    n_rays_per_image=750,
    n_pts_per_ray=128,
    min_depth=0.1,
    max_depth=volume_extent_world,
)

# 2) Instantiate the raymarcher.
# Here, we use the standard EmissionAbsorptionRaymarcher
# which marches along each ray in order to render
# the ray into a single 3D color vector
# and an opacity scalar.
raymarcher = EmissionAbsorptionRaymarcher()

# Finally, instantiate the implicit renders
# for both raysamplers.
renderer_grid = ImplicitRenderer(
    raysampler=raysampler_grid, raymarcher=raymarcher,
)
renderer_mc = ImplicitRenderer(
    raysampler=raysampler_mc, raymarcher=raymarcher,
)
```

6.3 Define the neural radiance field model

We define the Neural Radiance Field module, which specifies a continuous field of colors and opacities over the 3D domain of the scene.

The forward function of Neural Radiance Field (NeRF) receives as input a set of tensors that parametrize a bundle of rendering rays. The ray bundle is later converted to 3D ray points in the world coordinates of the scene. Each 3D point is then mapped to a harmonic representation using the HarmonicEmbedding layer (defined in the next cell). The harmonic embeddings then enter the color and opacity branches of the NeRF model in order to label each ray point with a 3D vector and a 1D scalar ranging in [0-1] which define the point's RGB color and opacity respectively.

Since NeRF has a large memory footprint, we also implement the NeuralRadiance-Field.forward_batched method. The method splits the input rays into batches and executes the forward function for each batch separately in a for loop. This lets us render a large set of rays without running out of GPU memory. Standardly, forward_batched would be used to render rays emitted from all pixels of an image in order to produce a full-sized render of a scene.

We can see the details of NeuralRadianceField module in **fit_simple_neural_radiance_field.ipynb**.

6.4 Fit the radiance field

In order to fit the radiance field, we render it from the viewpoints of the target cameras and compare the resulting renders with the observed target images and target silhouettes.

The comparison is done by evaluating the mean huber (smooth-l1) error between corresponding pairs of target images/rendered images and target silhouettes/rendered silhouettes. Figure 13 shows the predictd NeRF from different viewpoints.

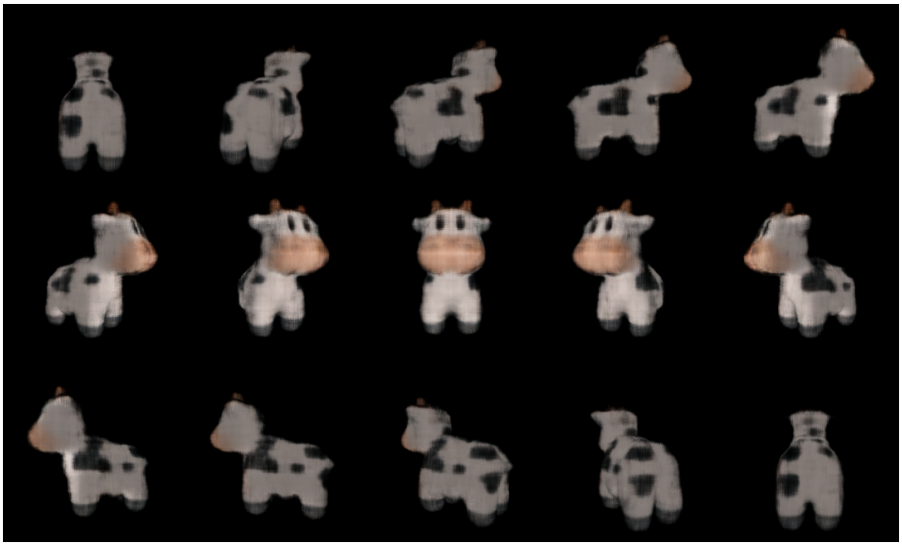


Figure 13: Predicted NeRF from different viewpoints