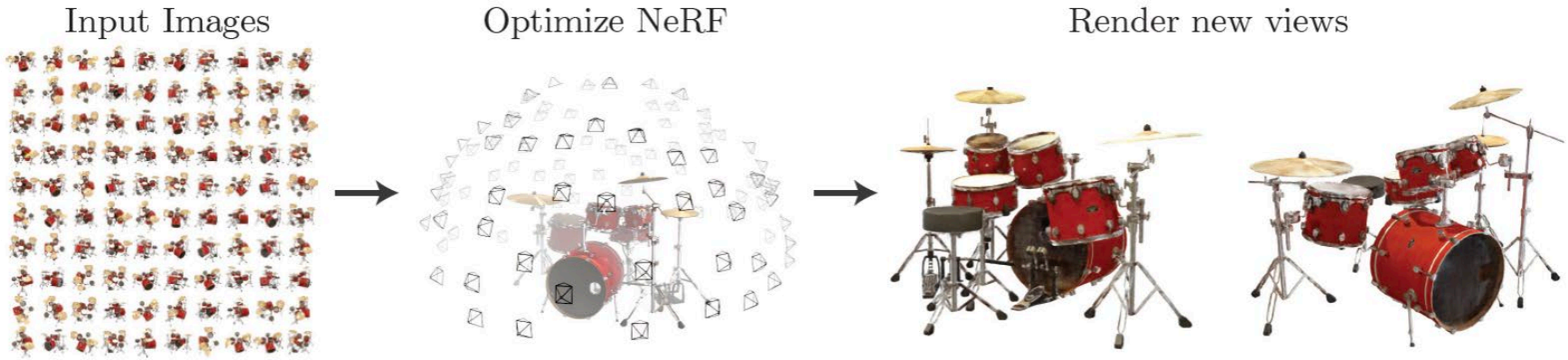
CS5182 Computer Graphics NeRF & 3D Gaussian Splatting Tutorial

2024/25 Semester A

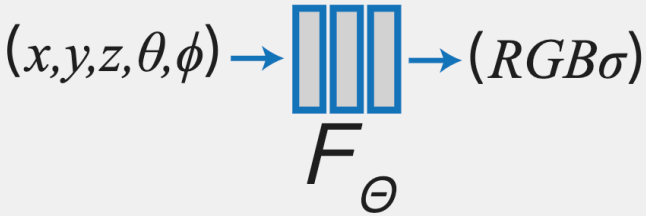
City University of Hong Kong (DG)

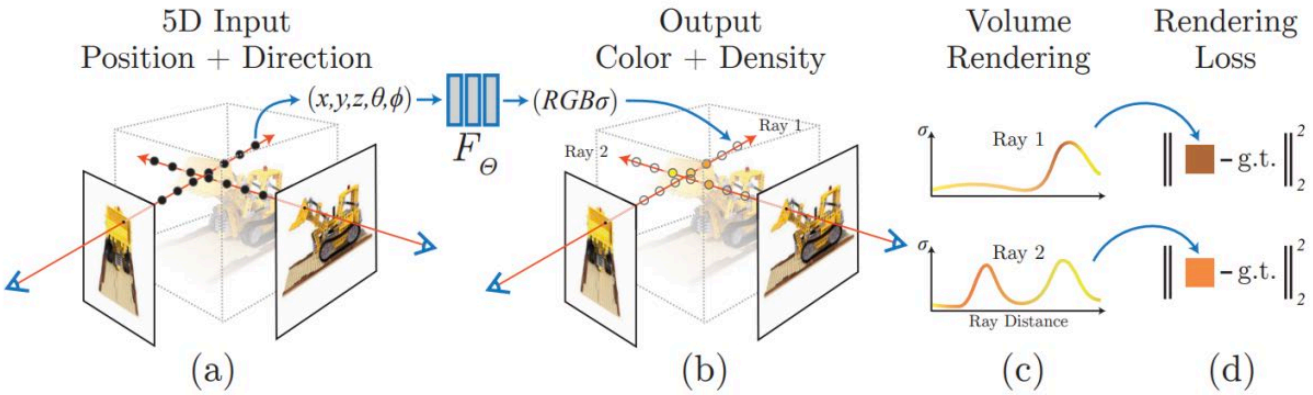
Introduction to NeRF



Ben Mildenhall, Pratul P Srinivasan, Matthew Tancik, Jonathan T Barron, Ravi Ramamoorthi, and Ren Ng. Nerf: Representing scenes as neural radiance fields for view synthesis. In *ECCV*, 2020.

How does NeRF work?





Volumetric Rendering

$C(\mathbf{r})$ of camera ray $\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$ with near and far bounds t_n and t_f is:

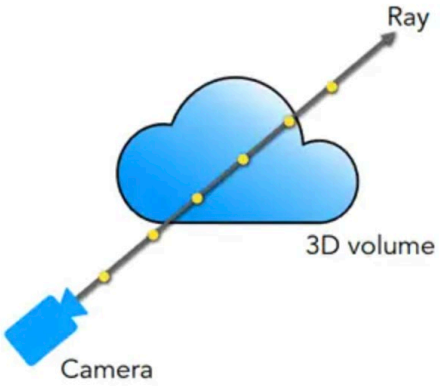
$$C(\mathbf{r}) = \int_{t_n}^{t_f} T(t)\sigma(\mathbf{r}(t))\mathbf{c}(\mathbf{r}(t), \mathbf{d})dt\,, \text{ where } T(t) = \exp\left(-\int_{t_n}^{t} \sigma(\mathbf{r}(s))ds\right).$$

transmittance

-- Implementation

$$t_i \sim \mathcal{U}\left[t_n + \frac{i-1}{N}(t_f - t_n),\ t_n + \frac{i}{N}(t_f - t_n)\right].$$

$$\hat{C}(\mathbf{r}) = \sum_{i=1}^{N} T_i(1 - \exp(-\sigma_i\delta_i))\mathbf{c}_i\,, \text{ where } T_i = \exp\left(-\sum_{j=1}^{i-1}\sigma_j\delta_j\right)$$



Optimization

-- Positional encoding for data representation

$$\gamma(p) = \left(\sin\left(2^0\pi p\right), \cos\left(2^0\pi p\right), \cdots, \sin\left(2^{L-1}\pi p\right), \cos\left(2^{L-1}\pi p\right)\right).$$

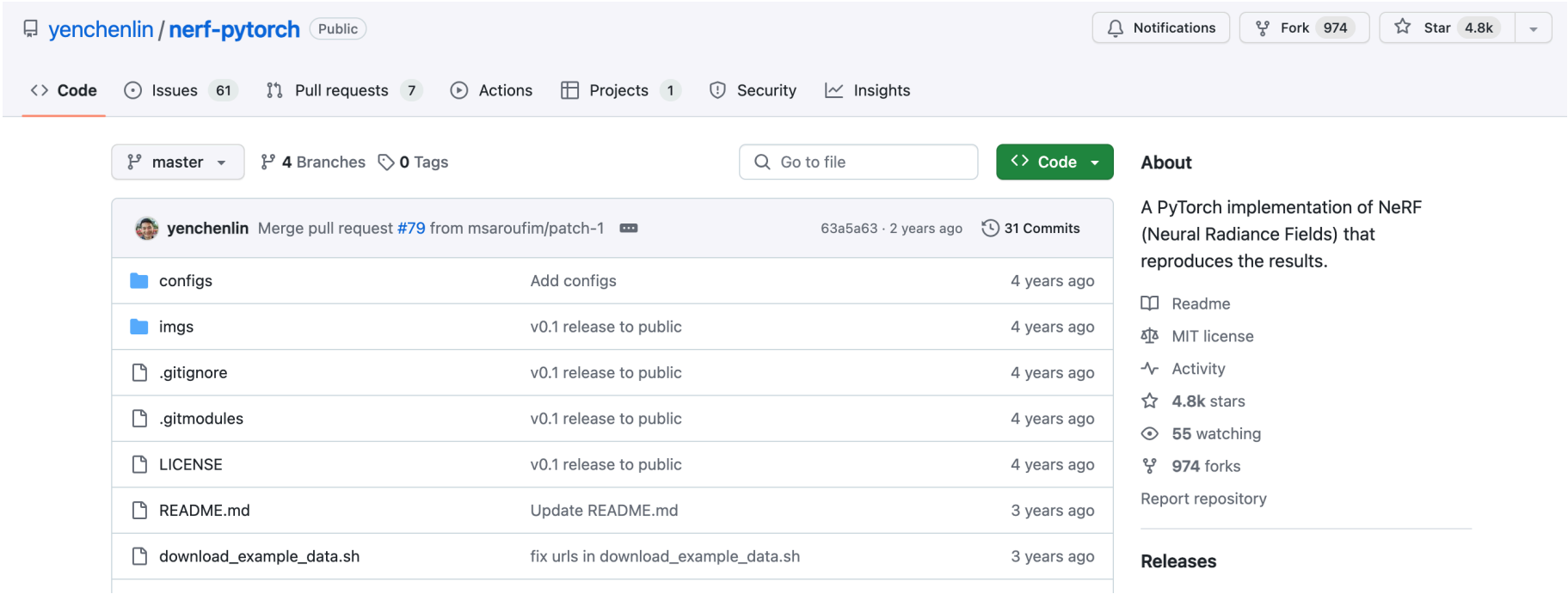
-- Hierarchical volume sampling strategy

- Coarse to fine
- Total $N_c + N_f$ samples

-- Loss function

$$\mathcal{L} = \sum_{\mathbf{r}\in\mathcal{R}}\left[\left\|\hat{C}_c(\mathbf{r}) - C(\mathbf{r})\right\|_2^2 + \left\|\hat{C}_f(\mathbf{r}) - C(\mathbf{r})\right\|_2^2\right]$$

NeRF-PyTorch



https://github.com/yenchenlin/nerf-pytorch

Installation

Installation

```
git clone https://github.com/yenchenlin/nerf-pytorch.git
cd nerf-pytorch
pip install -r requirements.txt
```

▼ Dependencies (click to expand)

Dependencies

- PyTorch 1.4
- matplotlib
- numpy
- imageio
- imageio-ffmpeg
- configargparse

The LLFF data loader requires ImageMagick.

You will also need the LLFF code (and COLMAP) set up to compute poses if you want to run on your own real data.

Codes

configs	Add configs	4 years ago
imgs	v0.1 release to public	
.gitignore	v0.1 release to public	
.gitmodules	v0.1 release to public	
LICENSE	v0.1 release to public	
README.md	Update README.md	
download_example_data.sh	fix urls in download_example_data.sh	
load_LINEMOD.py	Add load_LINEMOD.py	
load_blender.py	Update load_blender.py	
load_deepvoxels.py	v0.1 release to public	
load_llff.py	v0.1 release to public	
requirements.txt	Fix torch version to 1.11	
run_nerf.py	Fix with tricks present	
run_nerf_helpers.py	Update run_nerf_helpers.py	

Main running code

- func batchify
  - func ret
- func run_network
- func batchify_rays
- func render
- func render_path
- func create_nerf
- func raw2outputs
- func render_rays
- func config_parser
- func train

Codes

configs	Add configs
imgs	v0.1 release to public
.gitignore	v0.1 release to public
.gitmodules	v0.1 release to public
LICENSE	v0.1 release to public
README.md	Update README.md
download_example_data.sh	fix urls in download_example_data.sh
load_LINEMOD.py	Add load_LINEMOD.py
load_blender.py	Update load_blender.py
load_deepvoxels.py	v0.1 release to public
load_llff.py	v0.1 release to public
requirements.txt	Fix torch version to 1.11
run_nerf.py	Fix intrinsics problem
run_nerf_helpers.py	Update run_nerf_helpers.py

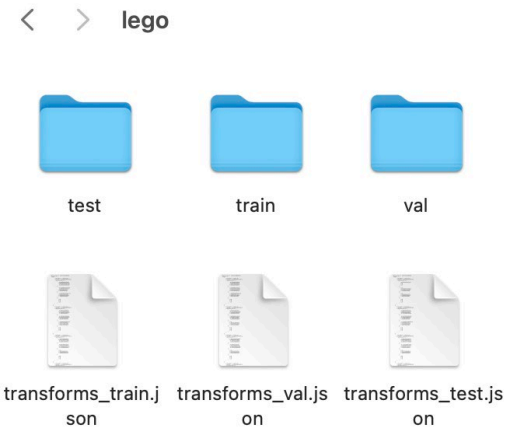
- class Embedder
  - func __init__
  - func create_embedding_fn
  - func embed
- func get_embedder
- class NeRF
  - func __init__
  - func forward
  - func load_weights_from_keras
- func get_rays
- func get_rays_np
- func ndc_rays
- func sample_pdf

Provide the basic functions will be needed in training and rendering

Codes

configs	Add configs	4 years ago
imgs	v0.1 release to public	4 years ago
.gitignore	v0.1 release to public	4 years ago
.gitmodules	v0.1 release to public	4 years ago
LICENSE	v0.1 release to public	4 years ago
README.md	Update README.md	3 years ago
download_example_data.sh	fix urls in download_example_data.sh	3 years ago
load_LINEMOD.py	Add load_LINEMOD.py	3 years ago
load_blender.py	Update load_blender.py	3 years ago
load_deepvoxels.py	v0.1 release to public	4 years ago
load_llff.py	v0.1 release to public	4 years ago
requirements.txt	Fix torch version to 1.11	2 years ago
run_nerf.py	Fix intrinsics problem	3 years ago
run_nerf_helpers.py	Update run_nerf_helpers.py	3 years ago

Dataloader



Dataloader

```python
def load_blender_data(basedir, half_res=False, testskip=1):
    splits = ['train', 'val', 'test']
    metas = {}
    for s in splits:
        with open(os.path.join(basedir, 'transforms_{}.json'.format(s)), 'r') as fp:
            metas[s] = json.load(fp)
    all_imgs = []
    all_poses = []
    counts = [0]
    for s in splits:
        meta = metas[s]
        imgs = []
        poses = []
        if s=='train' or testskip==0:
            skip = 1
        else:
            skip = testskip
        for frame in meta['frames'][::skip]:
            fname = os.path.join(basedir, frame['file_path'] + '.png')
            imgs.append(imageio.imread(fname))
            poses.append(np.array(frame['transform_matrix']))
        imgs = (np.array(imgs) / 255.).astype(np.float32) # keep all 4 channels (RGBA)
        poses = np.array(poses).astype(np.float32)
        counts.append(counts[-1] + imgs.shape[0])
        all_imgs.append(imgs)
        all_poses.append(poses)
    i_split = [np.arange(counts[i], counts[i+1]) for i in range(3)]
    imgs = np.concatenate(all_imgs, 0)
    poses = np.concatenate(all_poses, 0)
    H, W = imgs[0].shape[:2]
    camera_angle_x = float(meta['camera_angle_x'])
    focal = .5 * W / np.tan(.5 * camera_angle_x)
    render_poses = torch.stack([pose_spherical(angle, -30.0, 4.0) for angle in np.linspace(-180,180,40+1)[:-1]], 0)
    if half_res:
        H = H//2
        W = W//2
        focal = focal/2.
        imgs_half_res = np.zeros((imgs.shape[0], H, W, 4))
        for i, img in enumerate(imgs):
            imgs_half_res[i] = cv2.resize(img, (W, H), interpolation=cv2.INTER_AREA)
        imgs = imgs_half_res
    return imgs, poses, render_poses, [H, W, focal], i_split
```

Loops through each split, loading transformation metadata from JSON files corresponding to each split

For each frame in the current split, reads the image and pose matrix

Converts to NumPy array

Generates a series of camera poses for rendering, creating a spherical path around the scene

```
"camera_angle_x": 0.6911112070083618,
"frames": [
    {
        "file_path": "./train/r_0",
        "rotation": 0.012566370614359171,
        "transform_matrix": [
            [
                -0.9999021887779236,
                0.004192245192825794,
                -0.013345719315111637,
                -0.05379832163453102
            ],
            [
                -0.013988681137561798,
                -0.2996590733528137,
                0.95394366979599,
                3.845470428466797
            ],
            [
                -4.656612873077393e-10,
                0.9540371894836426,
                0.29968830943107605,
                1.2080823183059692
            ],
            [
                0.0,
                0.0,
                0.0,
                1.0
            ]
        ]
    },
```

Positional Encoding

```python
class Embedder:
    def __init__(self, **kwargs):
        self.kwargs = kwargs
        self.create_embedding_fn()

    def create_embedding_fn(self):
        embed_fns = []
        d = self.kwargs['input_dims']
        out_dim = 0
        if self.kwargs['include_input']:
            embed_fns.append(lambda x : x)
            out_dim += d

        max_freq = self.kwargs['max_freq_log2']
        N_freqs = self.kwargs['num_freqs']

        if self.kwargs['log_sampling']:
            freq_bands = 2.**torch.linspace(0., max_freq, steps=N_freqs)
        else:
            freq_bands = torch.linspace(2.**0., 2.**max_freq, steps=N_freqs)

        for freq in freq_bands:
            for p_fn in self.kwargs['periodic_fns']:
                embed_fns.append(lambda x, p_fn=p_fn, freq=freq : p_fn(x * freq))
                out_dim += d

        self.embed_fns = embed_fns
        self.out_dim = out_dim

    def embed(self, inputs):
        return torch.cat([fn(inputs) for fn in self.embed_fns], -1)
```

Iterates over each frequency band and each periodic function (sine and cosine)

Defines frequency bands based on the maximum frequency and the number of frequencies

Uses log sampling if specified, else linspace

NeRF Module

```python
class NeRF(nn.Module):
    def __init__(self, D=8, W=256, input_ch=3, input_ch_views=3, output_ch=4, skips=[4], use_viewdirs=False):
        super(NeRF, self).__init__()
        self.D = D
        self.W = W
        self.input_ch = input_ch
        self.input_ch_views = input_ch_views
        self.skips = skips
        self.use_viewdirs = use_viewdirs
        self.pts_linears = nn.ModuleList(
            [nn.Linear(input_ch, W)] + [nn.Linear(W, W) if i not in self.skips else nn.Linear(W + input_ch, W) for i in range(D-1)])
        self.views_linears = nn.ModuleList([nn.Linear(input_ch_views + W, W//2)])
        if use_viewdirs:
            self.feature_linear = nn.Linear(W, W)
            self.alpha_linear = nn.Linear(W, 1)
            self.rgb_linear = nn.Linear(W//2, 3)
        else:
            self.output_linear = nn.Linear(W, output_ch)

    def forward(self, x):
        input_pts, input_views = torch.split(x, [self.input_ch, self.input_ch_views], dim=-1)
        h = input_pts
        for i, l in enumerate(self.pts_linears):
            h = self.pts_linears[i](h)
            h = F.relu(h)
            if i in self.skips:
                h = torch.cat([input_pts, h], -1)

        if self.use_viewdirs:
            alpha = self.alpha_linear(h)
            feature = self.feature_linear(h)
            h = torch.cat([feature, input_views], -1)

            for i, l in enumerate(self.views_linears):
                h = self.views_linears[i](h)
                h = F.relu(h)

            rgb = self.rgb_linear(h)
            outputs = torch.cat([rgb, alpha], -1)
        else:
            outputs = self.output_linear(h)
        return outputs
```

Initializes class attributes based on the provided parameters

Creates a list of linear layers for processing 3D coordinates

Initializes a list of linear layers for processing viewing directions

Additional linear layers are created to compute features, alpha, and RGB values

Splits the input tensor into point coordinates and viewing directions

Passes the input points through the point processing layers
Apply ReLU activation after each layer
If the current layer index is in the skips, it concatenates the original input points to the output of the current layer

If using viewing directions, computes the alpha and features

Concatenates them with the input views

Processes them through the view layers to obtain RGB values

Render Rays

```
def render_rays(ray_batch,
                network_fn,
                network_query_fn,
                N_samples,
                retraw=False,
                lindisp=False,
                perturb=0.,
                N_importance=0,
                network_fine=None,
                white_bkgd=False,
                raw_noise_std=0.,
                verbose=False,
                pytest=False):
    N_rays = ray_batch.shape[0]
    rays_o, rays_d = ray_batch[:,0:3], ray_batch[:,3:6] # [N_rays, 3] each
    viewdirs = ray_batch[:,-3:] if ray_batch.shape[-1] > 8 else None
    bounds = torch.reshape(ray_batch[...,6:8], [-1,1,2])
    near, far = bounds[...,0], bounds[...,1] # [-1,1]
    t_vals = torch.linspace(0., 1., steps=N_samples)
    if not lindisp:
        z_vals = near * (1.-t_vals) + far * (t_vals)
    else:
        z_vals = 1./(1./near * (1.-t_vals) + 1./far * (t_vals))
    z_vals = z_vals.expand([N_rays, N_samples])

    if perturb > 0.:
        # get intervals between samples
        mids = .5 * (z_vals[...,1:] + z_vals[...,:-1])
        upper = torch.cat([mids, z_vals[...,-1:]], -1)
        lower = torch.cat([z_vals[...,:1], mids], -1)
        # stratified samples in those intervals
        t_rand = torch.rand(z_vals.shape)
        # Pytest, overwrite u with numpy's fixed random numbers
        if pytest:
            np.random.seed(0)
            t_rand = np.random.rand(*list(z_vals.shape))
            t_rand = torch.Tensor(t_rand)
        z_vals = lower + (upper - lower) * t_rand
    pts = rays_o[...,None,:] + rays_d[...,None,:] * z_vals[...,:,None] # [N_rays, N_samples, 3]
    raw = network_query_fn(pts, viewdirs, network_fn)
    rgb_map, disp_map, acc_map, weights, depth_map = raw2outputs(raw, z_vals, rays_d, raw_noise_std, white_bkgd, pytest=pytest)
```

Extracts the number of rays and their origins, directions, and bounds from the *ray_batch*

Sample points along rays

If perturbation is enabled, calculates random stratified samples between the computed z_*vals*

Computes the 3D points along each ray

Queries the neural network, processes to RGB colors, disparity, and accumulated opacity

Render Rays

```
if N_importance > 0:
    rgb_map_0, disp_map_0, acc_map_0 = rgb_map, disp_map, acc_map

    z_vals_mid = .5 * (z_vals[...,1:] + z_vals[...,:-1])
    z_samples = sample_pdf(z_vals_mid, weights[...,1:-1], N_importance, det=(perturb==0.), pytest=pytest)
    z_samples = z_samples.detach()

    z_vals, _ = torch.sort(torch.cat([z_vals, z_samples], -1), -1)
    pts = rays_o[...,None,:] + rays_d[...,None,:] * z_vals[...,:,None] # [N_rays, N_samples + N_importance, 3]

    run_fn = network_fn if network_fine is None else network_fine
    raw = network_query_fn(pts, viewdirs, run_fn)

    rgb_map, disp_map, acc_map, weights, depth_map = raw2outputs(raw, z_vals, rays_d, raw_noise_std, white_bkgd, pytest=pytes

ret = {'rgb_map' : rgb_map, 'disp_map' : disp_map, 'acc_map' : acc_map}
if retraw:
    ret['raw'] = raw
if N_importance > 0:
    ret['rgb0'] = rgb_map_0
    ret['disp0'] = disp_map_0
    ret['acc0'] = acc_map_0
    ret['z_std'] = torch.std(z_samples, dim=-1, unbiased=False)  # [N_rays]

for k in ret:
    if (torch.isnan(ret[k]).any() or torch.isinf(ret[k]).any()) and DEBUG:
        print(f"! [Numerical Error] {k} contains nan or inf.")

return ret
```

If using importance sampling, additional samples are generated based on the midpoint of the original *z_vals* and the weights computed from the first pass. These points are then queried again to refine the RGB and alpha estimates.

Constructs a dictionary to return the output

Train

Configuration & Data Loading

```
def train():

    parser = config_parser()
    args = parser.parse_args()

    K = None

    if args.dataset_type == 'blender':
        images, poses, render_poses, hwf, i_split = load_blender_data(args.datadir, args.half_res, args.testskip)
        print('Loaded blender', images.shape, render_poses.shape, hwf, args.datadir)
        i_train, i_val, i_test = i_split

        near = 2.
        far = 6.

        if args.white_bkgd:
            images = images[...,:3]*images[...,-1:] + (1.-images[...,-1:])
        else:
            images = images[...,:3]

    # Cast intrinsics to right types
    H, W, focal = hwf
    H, W = int(H), int(W)
    hwf = [H, W, focal]

    if K is None:
        K = np.array([
            [focal, 0, 0.5*W],
            [0, focal, 0.5*H],
            [0, 0, 1]
        ])
```

Depending on the specified dataset type, load data

Train

Model Initialization

```
# Create log dir and copy the config file
basedir = args.basedir
expname = args.expname
os.makedirs(os.path.join(basedir, expname), exist_ok=True)
f = os.path.join(basedir, expname, 'args.txt')
with open(f, 'w') as file:
    for arg in sorted(vars(args)):
        attr = getattr(args, arg)
        file.write('{} = {}\n'.format(arg, attr))
if args.config is not None:
    f = os.path.join(basedir, expname, 'config.txt')
    with open(f, 'w') as file:
        file.write(open(args.config, 'r').read())

# Create nerf model
render_kwargs_train, render_kwargs_test, start, grad_vars, optimizer = create_nerf(args)
global_step = start

bds_dict = {
    'near' : near,
    'far' : far,
}
render_kwargs_train.update(bds_dict)
render_kwargs_test.update(bds_dict)
```

Creates a directory for saving training logs and configurations

Initializes the NeRF model and optimizer

Sets the starting point for training

Train

Training Loop Preparation

```
# Prepare raybatch tensor if batching random rays
N_rand = args.N_rand
use_batching = not args.no_batching
if use_batching:
    # For random ray batching
    print('get rays')
    rays = np.stack([get_rays_np(H, W, K, p) for p in poses[:,:3,:4]], 0) # [N, ro+rd, H, W, 3]
    print('done, concats')
    rays_rgb = np.concatenate([rays, images[:,None]], 1) # [N, ro+rd+rgb, H, W, 3]
    rays_rgb = np.transpose(rays_rgb, [0,2,3,1,4]) # [N, H, W, ro+rd+rgb, 3]
    rays_rgb = np.stack([rays_rgb[i] for i in i_train], 0) # train images only
    rays_rgb = np.reshape(rays_rgb, [-1,3,3]) # [(N-1)*H*W, ro+rd+rgb, 3]
    rays_rgb = rays_rgb.astype(np.float32)
    print('shuffle rays')
    np.random.shuffle(rays_rgb)

    print('done')
    i_batch = 0

# Move testing data to GPU
render_poses = torch.Tensor(render_poses).to(device)
# Move training data to GPU
if use_batching:
    images = torch.Tensor(images).to(device)
poses = torch.Tensor(poses).to(device)
if use_batching:
    rays_rgb = torch.Tensor(rays_rgb).to(device)
```

Prepares batches of rays for training

Moves data to the GPU

Train

Training Process:

- **Ray Sampling**

```
start = start + 1
for i in trange(start, N_iters):
    time0 = time.time()

    # Sample random ray batch
    if use_batching:
        # Random over all images
        batch = rays_rgb[i_batch:i_batch+N_rand] # [B, 2+1, 3*?]
        batch = torch.transpose(batch, 0, 1)
        batch_rays, target_s = batch[:2], batch[2]

        i_batch += N_rand
        if i_batch >= rays_rgb.shape[0]:
            print("Shuffle data after an epoch!")
            rand_idx = torch.randperm(rays_rgb.shape[0])
            rays_rgb = rays_rgb[rand_idx]
            i_batch = 0
```

Sampling rays from the dataset

Train

Training Process:

- **Volume rendering**
- **Optimization**

```
#####  Core optimization loop  #####
rgb, disp, acc, extras = render(H, W, K, chunk=args.chunk, rays=batch_rays,
                                verbose=i < 10, retraw=True,
                                **render_kwargs_train)

optimizer.zero_grad()
img_loss = img2mse(rgb, target_s)
trans = extras['raw'][...,-1]
loss = img_loss
psnr = mse2psnr(img_loss)

if 'rgb0' in extras:
    img_loss0 = img2mse(extras['rgb0'], target_s)
    loss = loss + img_loss0
    psnr0 = mse2psnr(img_loss0)

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

Rendering RGB values and densities using the model

Calculating the loss

Performs backpropagation to update model weights

Train

Training Process:

- **Learning Rate Adjustment**
- **Logging and Checkpoints**

```
# NOTE: IMPORTANT!
###   update learning rate   ###
decay_rate = 0.1
decay_steps = args.lrate_decay * 1000
new_lrate = args.lrate * (decay_rate ** (global_step / decay_steps))
for param_group in optimizer.param_groups:
    param_group['lr'] = new_lrate
################################

dt = time.time()-time0
# print(f"Step: {global_step}, Loss: {loss}, Time: {dt}")
#####           end            #####

# Rest is logging
if i%args.i_weights==0:
    path = os.path.join(basedir, expname, '{:06d}.tar'.format(i))
    torch.save({
        'global_step': global_step,
        'network_fn_state_dict': render_kwargs_train['network_fn'].state_dict(),
        'network_fine_state_dict': render_kwargs_train['network_fine'].state_dict(),
        'optimizer_state_dict': optimizer.state_dict(),
    }, path)
    print('Saved checkpoints at', path)
```

Optionally adjusts the learning rate based on the current iteration number to make it follow the decay schedule

Logs the training progress, saves model checkpoints

Train

Evaluation & Rendering

```
    if i%args.i_video==0 and i > 0:
        # Turn on testing mode
        with torch.no_grad():
            rgbs, disps = render_path(render_poses, hwf, K, args.chunk, render_kwargs_test)
        print('Done, saving', rgbs.shape, disps.shape)
        moviebase = os.path.join(basedir, expname, '{}_spiral_{:06d}_'.format(expname, i))
        imageio.mimwrite(moviebase + 'rgb.mp4', to8b(rgbs), fps=30, quality=8)
        imageio.mimwrite(moviebase + 'disp.mp4', to8b(disps / np.max(disps)), fps=30, quality=8)

    if i%args.i_testset==0 and i > 0:
        testsavedir = os.path.join(basedir, expname, 'testset_{:06d}'.format(i))
        os.makedirs(testsavedir, exist_ok=True)
        print('test poses shape', poses[i_test].shape)
        with torch.no_grad():
            render_path(torch.Tensor(poses[i_test]).to(device), hwf, K, args.chunk, render_kwargs_test, gt_imgs=images[i_test], savedi
        print('Saved test set')
```

Renders images or videos from the test set or predefined poses

```
    if i%args.i_print==0:
        tqdm.write(f"[TRAIN] Iter: {i} Loss: {loss.item()}  PSNR: {psnr.item()}")
    """
```

Run

How To Run?

Quick Start

Download data for two example datasets: `lego` and `fern`

```
bash download_example_data.sh
```

To train a low-res `lego` NeRF:

```
python run_nerf.py --config configs/lego.txt
```

After training for 100k iterations (~4 hours on a single 2080 Ti), you can find the following video at `logs/lego_test/lego_test_spiral_100000_rgb.mp4`.

Introduction to 3D Gaussian Splatting

- Reconstructing the 3D world from images

Bernhard Kerbl, Georgios Kopanas, Thomas Leimkuhler, ¨ and George Drettakis. 3d gaussian splatting for real-time radiance field rendering. *ACM Transactions on Graphics*, 42 (4), 2023

Method

Gaussians are defined by a full 3D covariance matrix Σ defined in world space centered at point (mean) μ and is multiplied by α in blending process

3D Gaussian

- **Position (Mean μ)**: location (XYZ)
- **Covariance Matrix (Σ)**: rotation and scaling $\Sigma = RSS^T R^T$
- **Opacity (α)**: Transparency
- **Color (RGB)** or **Spherical Harmonics (SH) coefficients**

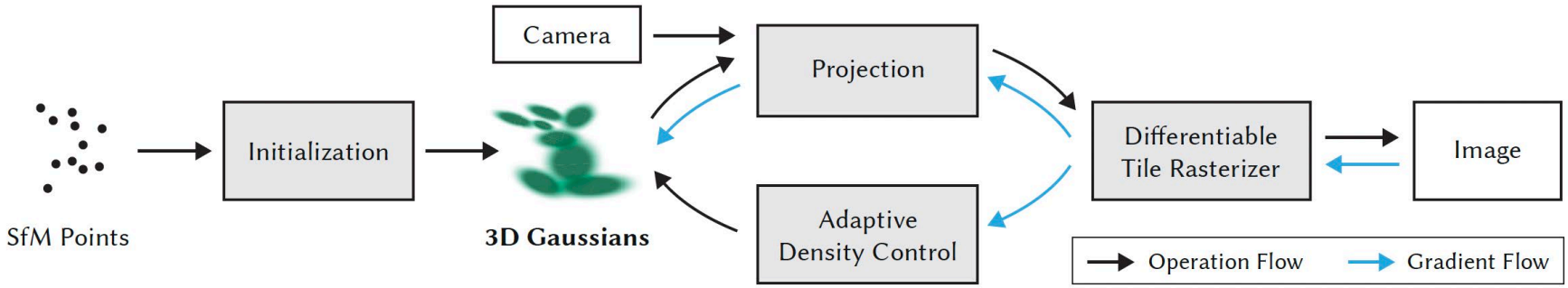


Fig. 2. Optimization starts with the sparse SfM point cloud and creates a set of 3D Gaussians. We then optimize and adaptively control the density of this set of Gaussians. During optimization we use our fast tile-based renderer, allowing competitive training times compared to SOTA fast radiance field methods. Once trained, our renderer allows real-time navigation for a wide variety of scenes.

Optimization

Adaptive Gaussian demsification

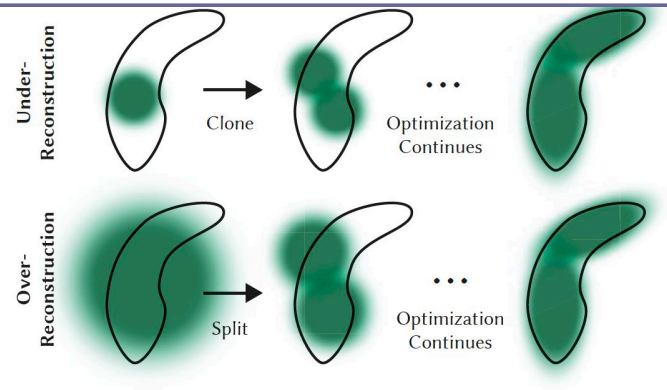


Fig. 4. Our adaptive Gaussian densification scheme. *Top row (under-reconstruction)*: When small-scale geometry (black outline) is insufficiently covered, we clone the respective Gaussian. *Bottom row (over-reconstruction)*: If small-scale geometry is represented by one large splat, we split it in two.

Forward process of 3D GS

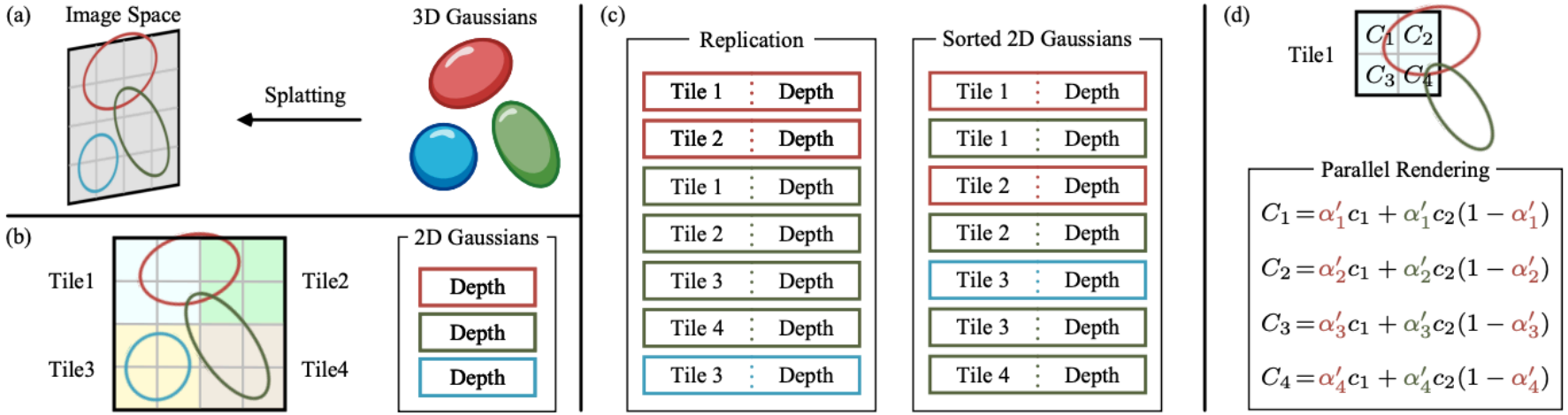
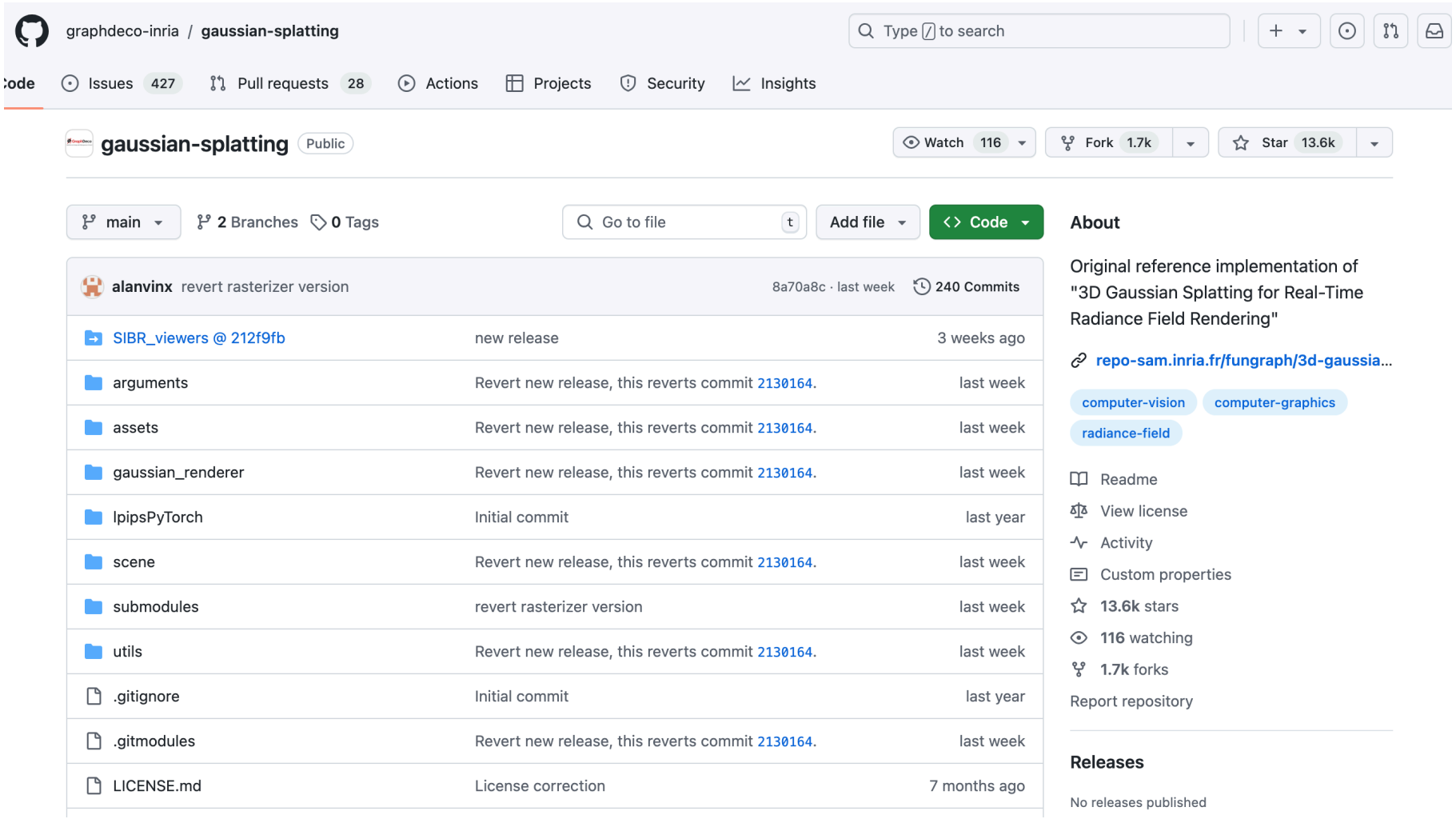


Fig. 4. An illustration of the forward process of 3D GS (see Sec. 3.1). (a) The splatting step projects 3D Gaussians into image space. (b) 3D GS divides the image into multiple non-overlapping patches, *i.e.*, tiles. (c) 3D GS replicates the Gaussians which cover several tiles, assigning each copy an identifier, *i.e.*, a tile ID. (d) By rendering the sorted Gaussians, we can obtain all pixels within the tile. Note that the computational workflows for pixels and tiles are independent and can be done in parallel. Best viewed in color.

Chen, Guikun, and Wenguan Wang. A survey on 3d gaussian splatting. *arXiv preprint arXiv:2401.03890* (2024).

Code



https://github.com/graphdeco-inria/gaussian-splatting

Setup

Setup

Local Setup

Our default, provided install method is based on Conda package and environment management:

```
SET DISTUTILS_USE_SDK=1 # Windows only
conda env create --file environment.yml
conda activate gaussian_splatting
```

Please note that this process assumes that you have CUDA SDK **11** installed, not **12**. For modifications, see below.

Tip: Downloading packages and creating a new environment with Conda can require a significant amount of disk space. By default, Conda will use the main system hard drive. You can avoid this by specifying a different package download location and an environment on a different drive:

```
conda config --add pkgs_dirs <Drive>/<pkg_path>
conda env create --file environment.yml --prefix <Drive>/<env_path>/gaussian_splatting
conda activate <Drive>/<env_path>/gaussian_splatting
```

Setup (install COLMAP)

```
!sudo apt-get install \
    git \
    cmake \
    ninja-build \
    build-essential \
    libboost-program-options-dev \
    libboost-filesystem-dev \
    libboost-graph-dev \
    libboost-system-dev \
    libeigen3-dev \
    libflann-dev \
    libfreeimage-dev \
    libmetis-dev \
    libgoogle-glog-dev \
    libgtest-dev \
    libsqlite3-dev \
    libglew-dev \
    qtbase5-dev \
    libqt5opengl5-dev \
    libcgal-dev \
    libceres-dev

!git clone https://github.com/colmap/colmap.git
cd colmap
!git checkout dev
mkdir build
cd build
!cmake .. -GNinja -DCMAKE_CUDA_ARCHITECTURES=native
!ninja
!sudo ninja install
```

Images + Camera pose

colmap

3D point cloud

https://colmap.github.io/install.html

GaussianModel

Initialization

```
def __init__(self, sh_degree : int):
    self.active_sh_degree = 0
    self.max_sh_degree = sh_degree
    self._xyz = torch.empty(0)
    self._features_dc = torch.empty(0)
    self._features_rest = torch.empty(0)
    self._scaling = torch.empty(0)
    self._rotation = torch.empty(0)
    self._opacity = torch.empty(0)
    self.max_radii2D = torch.empty(0)
    self.xyz_gradient_accum = torch.empty(0)
    self.denom = torch.empty(0)
    self.optimizer = None
    self.percent_dense = 0
    self.spatial_lr_scale = 0
    self.setup_functions()
```

GaussianModel

Create gaussians from point cloud

```
def create_from_pcd(self, pcd : BasicPointCloud, spatial_lr_scale : float):
    self.spatial_lr_scale = spatial_lr_scale
    fused_point_cloud = torch.tensor(np.asarray(pcd.points)).float().cuda()
    fused_color = RGB2SH(torch.tensor(np.asarray(pcd.colors)).float().cuda())
    features = torch.zeros((fused_color.shape[0], 3, (self.max_sh_degree + 1) ** 2)).float().cuda()
    features[:, :3, 0 ] = fused_color
    features[:, 3:, 1:] = 0.0

    print("Number of points at initialisation : ", fused_point_cloud.shape[0])

    dist2 = torch.clamp_min(distCUDA2(torch.from_numpy(np.asarray(pcd.points)).float().cuda()), 0.0000001)
    scales = torch.log(torch.sqrt(dist2))[...,None].repeat(1, 3)
    rots = torch.zeros((fused_point_cloud.shape[0], 4), device="cuda")
    rots[:, 0] = 1

    opacities = inverse_sigmoid(0.1 * torch.ones((fused_point_cloud.shape[0], 1), dtype=torch.float, device="cuda"))

    self._xyz = nn.Parameter(fused_point_cloud.requires_grad_(True))
    self._features_dc = nn.Parameter(features[:,:,0:1].transpose(1, 2).contiguous().requires_grad_(True))
    self._features_rest = nn.Parameter(features[:,:,1:].transpose(1, 2).contiguous().requires_grad_(True))
    self._scaling = nn.Parameter(scales.requires_grad_(True))
    self._rotation = nn.Parameter(rots.requires_grad_(True))
    self._opacity = nn.Parameter(opacities.requires_grad_(True))
    self.max_radii2D = torch.zeros((self.get_xyz.shape[0]), device="cuda")
```

Converts the point cloud colors to spherical harmonics representation

Store the features of the Gaussian model

Computes the squared distances between points in the point cloud

Calculates the scaling factors based on the distances

Initializes rotation parameters

Initializes opacities for the Gaussians

Initializes all the necessary parameters

GaussianModel

Adaptive Gaussian densification -- split

```
def densify_and_split(self, grads, grad_threshold, scene_extent, N=2):
    n_init_points = self.get_xyz.shape[0]
    # Extract points that satisfy the gradient condition
    padded_grad = torch.zeros((n_init_points), device="cuda")
    padded_grad[:grads.shape[0]] = grads.squeeze()
    selected_pts_mask = torch.where(padded_grad >= grad_threshold, True, False)
    selected_pts_mask = torch.logical_and(selected_pts_mask,
                                          torch.max(self.get_scaling, dim=1).values > self.percent_dense*scene_extent)

    stds = self.get_scaling[selected_pts_mask].repeat(N,1)
    means =torch.zeros((stds.size(0), 3),device="cuda")
    samples = torch.normal(mean=means, std=stds)
    rots = build_rotation(self._rotation[selected_pts_mask]).repeat(N,1,1)
    new_xyz = torch.bmm(rots, samples.unsqueeze(-1)).squeeze(-1) + self.get_xyz[selected_pts_mask].repeat(N, 1)
    new_scaling = self.scaling_inverse_activation(self.get_scaling[selected_pts_mask].repeat(N,1) / (0.8*N))
    new_rotation = self._rotation[selected_pts_mask].repeat(N,1)
    new_features_dc = self._features_dc[selected_pts_mask].repeat(N,1,1)
    new_features_rest = self._features_rest[selected_pts_mask].repeat(N,1,1)
    new_opacity = self._opacity[selected_pts_mask].repeat(N,1)

    self.densification_postfix(new_xyz, new_features_dc, new_features_rest, new_opacity, new_scaling, new_rotation)

    prune_filter = torch.cat((selected_pts_mask, torch.zeros(N * selected_pts_mask.sum(), device="cuda", dtype=bool)))
    self.prune_points(prune_filter)
```

Initializes a tensor to hold the gradient values

Creates a mask to identify points that have gradients above the specified threshold

Further refines the mask to include only those points whose maximum scaling value exceeds a certain density threshold relative to the scene extent

Generates new sample points

Updates the properties for the new Gaussian points

Creates a filter combining the selected points mask and a zero tensor for the new points that should not be included in pruning. Calls prune_points to remove points

GaussianModel

Adaptive Gaussian densification – clone

```
def densify_and_clone(self, grads, grad_threshold, scene_extent):
    # Extract points that satisfy the gradient condition
    selected_pts_mask = torch.where(torch.norm(grads, dim=-1) >= grad_threshold, True, False)
    selected_pts_mask = torch.logical_and(selected_pts_mask,
                                          torch.max(self.get_scaling, dim=1).values <= self.percent_dense*scene_extent)
    new_xyz = self._xyz[selected_pts_mask]
    new_features_dc = self._features_dc[selected_pts_mask]
    new_features_rest = self._features_rest[selected_pts_mask]
    new_opacities = self._opacity[selected_pts_mask]
    new_scaling = self._scaling[selected_pts_mask]
    new_rotation = self._rotation[selected_pts_mask]

    self.densification_postfix(new_xyz, new_features_dc, new_features_rest, new_opacities, new_scaling, new_rotation)
```

Further refines the mask to include only those points whose maximum scaling value does not exceed a certain density threshold relative to the scene extent

Clone selected points

GaussianModel

Adaptive Gaussian densification – prune

```
def densify_and_prune(self, max_grad, min_opacity, extent, max_screen_size):
    grads = self.xyz_gradient_accum / self.denom
    grads[grads.isnan()] = 0.0

    self.densify_and_clone(grads, max_grad, extent)
    self.densify_and_split(grads, max_grad, extent)

    prune_mask = (self.get_opacity < min_opacity).squeeze()
    if max_screen_size:
        big_points_vs = self.max_radii2D > max_screen_size
        big_points_ws = self.get_scaling.max(dim=1).values > 0.1 * extent
        prune_mask = torch.logical_or(torch.logical_or(prune_mask, big_points_vs), big_points_ws)
    self.prune_points(prune_mask)

    torch.cuda.empty_cache()
```

Calculates the gradients

Calls the two functions

Initializes a mask for pruning points that have an opacity below the specified minimum opacity threshold

Identifies points that have a radius greater than the maximum screen size

Identifies points whose maximum scaling value exceeds 10% of the scene extent

Training

```
def training(dataset, opt, pipe, testing_iterations, saving_iterations, checkpoint_iterations, checkpoint, debug_from):
    first_iter = 0
    tb_writer = prepare_output_and_logger(dataset)
    gaussians = GaussianModel(dataset.sh_degree)
    scene = Scene(dataset, gaussians)
    gaussians.training_setup(opt)
    if checkpoint:
        (model_params, first_iter) = torch.load(checkpoint)
        gaussians.restore(model_params, opt)

    bg_color = [1, 1, 1] if dataset.white_background else [0, 0, 0]
    background = torch.tensor(bg_color, dtype=torch.float32, device="cuda")

    iter_start = torch.cuda.Event(enable_timing = True)
    iter_end = torch.cuda.Event(enable_timing = True)
```

Initializes counters

Prepares logging

Creates a Gaussian model

Creates a scene using the dataset

Sets up the Gaussian model for training

If a checkpoint is provided, it loads the model parameters and resumes from the specified iteration

Defines the background color based on the dataset

Initializes CUDA events to track time

Training

```
viewpoint_stack = None
ema_loss_for_log = 0.0
progress_bar = tqdm(range(first_iter, opt.iterations), desc="Training progress")
first_iter += 1
for iteration in range(first_iter, opt.iterations + 1):
    if network_gui.conn == None:
        network_gui.try_connect()
    while network_gui.conn != None:
        try:
            net_image_bytes = None
            custom_cam, do_training, pipe.convert_SHs_python, pipe.compute_cov3D_python, keep_alive, scaling_modifer = network_gui.receive()
            if custom_cam != None:
                net_image = render(custom_cam, gaussians, pipe, background, scaling_modifer)["render"]
                net_image_bytes = memoryview((torch.clamp(net_image, min=0, max=1.0) * 255).byte().permute(1, 2, 0).contiguous().cpu().numpy())
            network_gui.send(net_image_bytes, dataset.source_path)
            if do_training and ((iteration < int(opt.iterations)) or not keep_alive):
                break
        except Exception as e:
            network_gui.conn = None

    iter_start.record()

    gaussians.update_learning_rate(iteration)

    # Every 1000 its we increase the levels of SH up to a maximum degree
    if iteration % 1000 == 0:
        gaussians.oneupSHdegree()
    # Pick a random Camera
    if not viewpoint_stack:
        viewpoint_stack = scene.getTrainCameras().copy()
    viewpoint_cam = viewpoint_stack.pop(randint(0, len(viewpoint_stack)-1))
    # Render
    if (iteration - 1) == debug_from:
        pipe.debug = True
    bg = torch.rand((3), device="cuda") if opt.random_background else background
    render_pkg = render(viewpoint_cam, gaussians, pipe, bg)
    image, viewspace_point_tensor, visibility_filter, radii = render_pkg["render"], render_pkg["viewspace_points"], render_pkg["visibility_filter"], render_pkg["radii"]
    # Loss
    gt_image = viewpoint_cam.original_image.cuda()
    Ll1 = l1_loss(image, gt_image)
    loss = (1.0 - opt.lambda_dssim) * Ll1 + opt.lambda_dssim * (1.0 - ssim(image, gt_image))
    loss.backward()
    iter_end.record()
```

Sets up a progress bar for monitoring training

Iterates through the specified number of training iterations

Attempts to connect to a GUI for real-time monitoring

Randomly selects a camera from the available viewpoints for rendering

Render

Computes the loss using L1 loss and SSIM

Training

```
with torch.no_grad():
    # Progress bar
    ema_loss_for_log = 0.4 * loss.item() + 0.6 * ema_loss_for_log
    if iteration % 10 == 0:
        progress_bar.set_postfix({"Loss": f"{ema_loss_for_log:.{7}f}"})
        progress_bar.update(10)
    if iteration == opt.iterations:
        progress_bar.close()
    # Log and save
    training_report(tb_writer, iteration, Ll1, loss, l1_loss, iter_start.elapsed_time(iter_end), testing_iterations, scene, render, (pipe, background))
    if (iteration in saving_iterations):
        print("\n[ITER {}] Saving Gaussians".format(iteration))
        scene.save(iteration)
    # Densification
    if iteration < opt.densify_until_iter:
        # Keep track of max radii in image-space for pruning
        gaussians.max_radii2D[visibility_filter] = torch.max(gaussians.max_radii2D[visibility_filter], radii[visibility_filter])
        gaussians.add_densification_stats(viewspace_point_tensor, visibility_filter)

        if iteration > opt.densify_from_iter and iteration % opt.densification_interval == 0:
            size_threshold = 20 if iteration > opt.opacity_reset_interval else None
            gaussians.densify_and_prune(opt.densify_grad_threshold, 0.005, scene.cameras_extent, size_threshold)

        if iteration % opt.opacity_reset_interval == 0 or (dataset.white_background and iteration == opt.densify_from_iter):
            gaussians.reset_opacity()

    # Optimizer step
    if iteration < opt.iterations:
        gaussians.optimizer.step()
        gaussians.optimizer.zero_grad(set_to_none = True)

    if (iteration in checkpoint_iterations):
        print("\n[ITER {}] Saving Checkpoint".format(iteration))
        torch.save((gaussians.capture(), iteration), scene.model_path + "/chkpnt" + str(iteration) + ".pth")
```

Perform progress bar

Saves the model state at specified iterations

Densification for pruning

Performs an optimization step to update the model weights

Saves a checkpoint of the model at specified iterations

Render

```python
def render(viewpoint_camera, pc : GaussianModel, pipe, bg_color : torch.Tensor, scaling_modifier = 1.0, override_color = None):
    """
    Render the scene.

    Background tensor (bg_color) must be on GPU!
    """
    # Create zero tensor. We will use it to make pytorch return gradients of the 2D (screen-space) means
    screenspace_points = torch.zeros_like(pc.get_xyz, dtype=pc.get_xyz.dtype, requires_grad=True, device="cuda") + 0
    try:
        screenspace_points.retain_grad()
    except:
        pass
    # Set up rasterization configuration
    tanfovx = math.tan(viewpoint_camera.FoVx * 0.5)
    tanfovy = math.tan(viewpoint_camera.FoVy * 0.5)

    raster_settings = GaussianRasterizationSettings(
        image_height=int(viewpoint_camera.image_height),
        image_width=int(viewpoint_camera.image_width),
        tanfovx=tanfovx,
        tanfovy=tanfovy,
        bg=bg_color,
        scale_modifier=scaling_modifier,
        viewmatrix=viewpoint_camera.world_view_transform,
        projmatrix=viewpoint_camera.full_proj_transform,
        sh_degree=pc.active_sh_degree,
        campos=viewpoint_camera.camera_center,
        prefiltered=False,
        debug=pipe.debug
    )

    rasterizer = GaussianRasterizer(raster_settings=raster_settings)

    means3D = pc.get_xyz
    means2D = screenspace_points
    opacity = pc.get_opacity
```

Creates a tensor to store the 2D screen-space points and allows gradients to be tracked for optimization purposes

Calculates the tangent of half the horizontal and vertical field of view angles to set up the rasterization setting

Initializes the rasterization settings including image dimensions, field of view, camera matrices, …

Instantiates a GaussianRasterizer using the defined settings

Extracts the 3D means (positions), 2D means (screen-space points), and opacity information from the Gaussian model

Render

```
    # If precomputed 3d covariance is provided, use it. If not, then it will be computed from
    # scaling / rotation by the rasterizer.
    scales = None
    rotations = None
    cov3D_precomp = None
    if pipe.compute_cov3D_python:
        cov3D_precomp = pc.get_covariance(scaling_modifier)
    else:
        scales = pc.get_scaling
        rotations = pc.get_rotation
    # If precomputed colors are provided, use them. Otherwise, if it is desired to precompute colors
    # from SHs in Python, do it. If not, then SH -> RGB conversion will be done by rasterizer.
    shs = None
    colors_precomp = None
    if override_color is None:
        if pipe.convert_SHs_python:
            shs_view = pc.get_features.transpose(1, 2).view(-1, 3, (pc.max_sh_degree+1)**2)
            dir_pp = (pc.get_xyz - viewpoint_camera.camera_center.repeat(pc.get_features.shape[0], 1))
            dir_pp_normalized = dir_pp/dir_pp.norm(dim=1, keepdim=True)
            sh2rgb = eval_sh(pc.active_sh_degree, shs_view, dir_pp_normalized)
            colors_precomp = torch.clamp_min(sh2rgb + 0.5, 0.0)
        else:
            shs = pc.get_features
    else:
        colors_precomp = override_color
    # Rasterize visible Gaussians to image, obtain their radii (on screen).
    rendered_image, radii = rasterizer(
        means3D = means3D,
        means2D = means2D,
        shs = shs,
        colors_precomp = colors_precomp,
        opacities = opacity,
        scales = scales,
        rotations = rotations,
        cov3D_precomp = cov3D_precomp)
    # Those Gaussians that were frustum culled or had a radius of 0 were not visible.
    # They will be excluded from value updates used in the splitting criteria.
    return {"render": rendered_image,
            "viewspace_points": screenspace_points,
            "visibility_filter" : radii > 0,
            "radii": radii}
```

Checks if covariance data is precomputed or needs to be calculated during rendering

Determines how to handle colors: either using precomputed colors or calculating them from spherical harmonics

Calls the rasterizer to generate the rendered image and retrieve the radii of the visible Gaussians

Returns a dictionary containing the rendered image, the screen-space points, a visibility filter indicating which Gaussians were visible, and their radii

Running & Evaluation

Running

To run the optimizer, simply use

```
python train.py -s <path to COLMAP or NeRF Synthetic dataset>
```

Evaluation

By default, the trained models use all available images in the dataset. To train them while withholding a test set for evaluation, use the `--eval` flag. This way, you can render training/test sets and produce error metrics as follows:

```
python train.py -s <path to COLMAP or NeRF Synthetic dataset> --eval # Train with train/test s
python render.py -m <path to trained model> # Generate renderings
python metrics.py -m <path to trained model> # Compute error metrics on renderings
```