View Synthesis

Implementing NeRF in PyTorch

Task Description

"View synthesis" is a task which generating images of a 3D scene from a specific point of view.

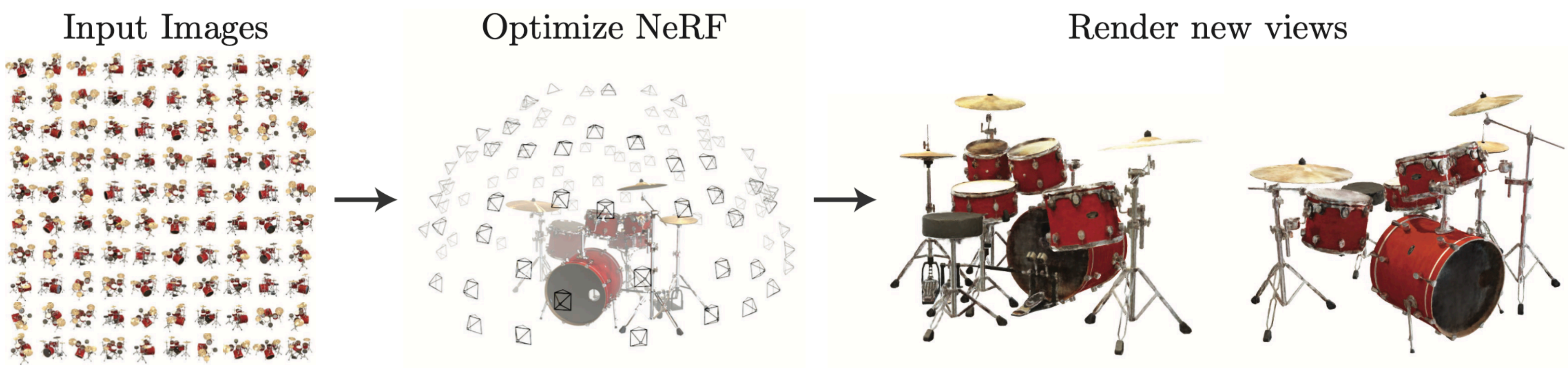
Solution Description (Cont.)

NeRF (Neural Radiance Field)

It can solve "View synthesis" by representing 3D scene using a neural network.

We can render the image $\hat{Y}$ from the view point X by using the function F:

$\hat{Y} = F(X)$



Solution Description (Cont.)

Volume Rendering

We render the 2D image $\hat{Y}$ with height H and width W from the view point X using the 3D scene representation f:

1. The origin o and the direction $d(r=1)$ of the view point X are given.
2. From the origin, we emit $H \times W$ rays through the image plane.
3. For each ray, sample S positions $p(r,i)$ along the direction $d(r)$.
4. Blend the color of the positions to approximate the color of the pixel $C(r)$.
5. Gather all the colors $C(r)$ to form the image $\hat{Y}$.

$$p(r,i) = o + t_i d(r),$$

$$c_i, \sigma_i = f(p(r,i), d(r)),$$

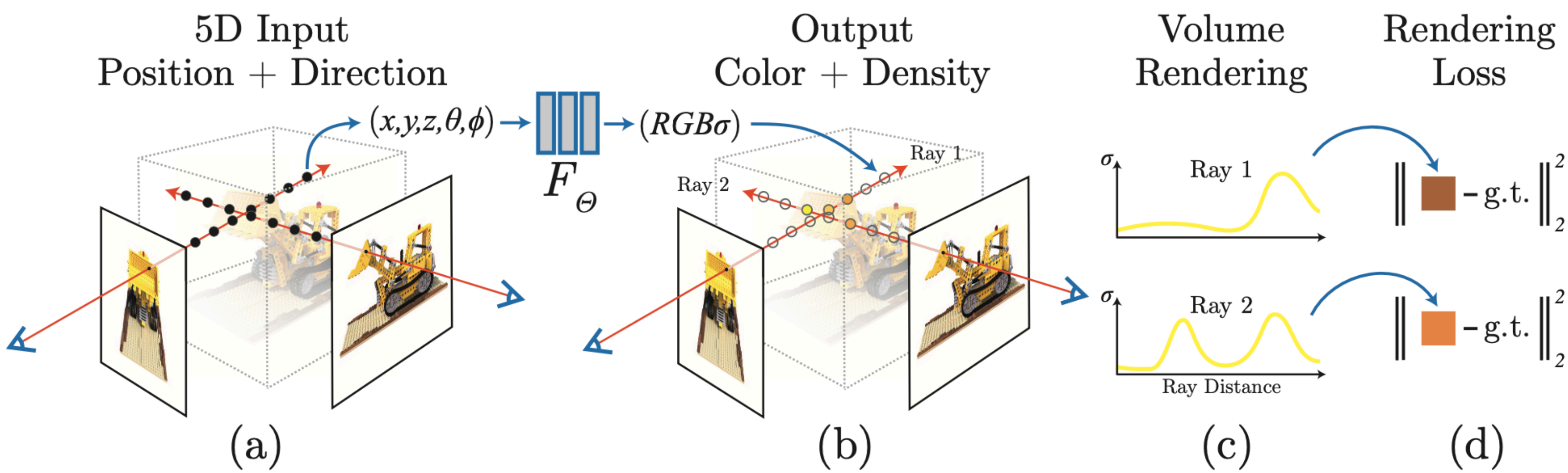
$$l(i) = \exp(-\sigma_i(t_{i+1} - t_i)),$$

$$\hat{C}(r) = \sum_{i=1}^{S} c_i(1 - l(i))(\prod_{j=1}^{i-1} l(j)),$$

$$\hat{Y} = \{\hat{C}(r) | r = 1, 2, \ldots, H \times W\}$$

Solution Description (Cont.)

Volume Rendering (Illustration)



Solution Description (Cont.)

Volumetric Representation Estimation

To estimate the 3D scene representation f, we can train a fully-connected neural network.

Since the volume rendering process is differentiable, we can optimize the neural network using Gradient Descent techniques.

The loss function is defined as the Mean Squared Error between the rendered image $\hat{Y}$ and the ground truth image Y:

$$\mathcal{L} = \frac{\sum_{r=1}^{H \times W} (\hat{Y}_r - Y_r)^2}{H \times W}$$

Solution Conclusion

NeRF is composed of three parts:

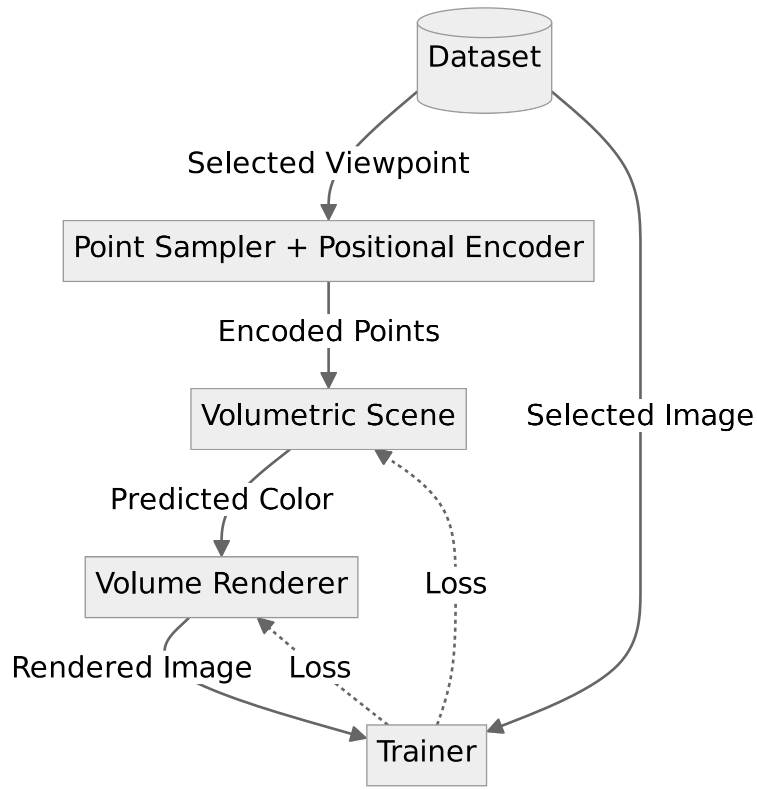
1. f: Volumetric representation estimation using a fully-connected neural network.
2. C: Volume rendering using the implicit scene representation.
3. $\mathcal{L}$: Loss function for training the volume rendering function C.

Modules

We have implemented the following modules using PyTorch `v2.3`:

1. **Point Sampler**: To sample points from batches of rays
2. **Positional Encoder**: To apply Fourier feature encoding for the input points
3. **Volumetric Scene**: To predict the color and desity of the input points
4. **Volume Renderer**: To render the image from the sampled points and colors
5. **Trainer**: To optimize the renderer with the given images and viewpoints

Pipeline



Dataset
Selected Viewpoint
Point Sampler + Positional Encoder
Encoded Points
Volumetric Scene
Selected Image
Predicted Color
Volume Renderer
Loss
Rendered Image
Loss
Trainer

Module Details

Positional Encoder

To explore the high-frequency features in the input points, we apply Fourier feature encoding.

Each coordinate value is encoded as follows:

$$\begin{gathered} Encode_L(p) \\ = \{\sin(2^0\pi p), \cos(2^0\pi p), \ldots, \sin(2^{L-1}\pi p), \cos(2^{L-1}\pi p)\} \\ = \{\sin(2^0\pi p), \sin(\frac{\pi}{2} + 2^0\pi p), \ldots, \sin(2^{L-1}\pi p), \sin(\frac{\pi}{2} + 2^{L-1}\pi p)\} \\ \text{where } p \in \mathbb{R},\ L \in \mathbb{N},\ Encode_L(p) \in \mathbb{R}^{2L} \end{gathered}$$

The raw and encoded coordinate values will be concatenated to form the network input.

The encoded dimensions are calculated as follows:

$$\text{Encoded Dimension} = \text{Input Dimension} \times (2L + 1)$$

Module Details

Volumetric Scene

The fully-connected neural network structure is as follows, and the skip connection is applied to 5th hidden layer:

Constants	Description	Value
I	Input Dimension	$3\ (Position) + 3\ (Direction) = 6$
E	Encoded Dimension	$I \times (2 \cdot 8 + 1) = 102$
H	Hidden Dimension	256
O	Output Dimension	$3\ (Color) + 1\ (Density) = 4$

Layer	Input Dim.	Output Dim.	Activation
0	E	H	ReLU
1	H	H	ReLU
2	H	H	ReLU
3	H	H	ReLU
4	H	H	ReLU
5	$H + E$	H	ReLU
6	H	H	ReLU
7	H	H	ReLU
8	H	O	Sigmoid (Color), ReLU (Density)

Experiment Details

Hyper-parameters

Constants	Description	Value
N	Number of Epochs	≥ 1000
S	Sample points per ray	80
B_r	Rays per batch	250
L	Encoding Factor	10
$\|\|Test\|\|$	Number of Testing Images	21
$\|\|Train\|\|$	Number of Training Images	85

References

1. View synthesis. (n.d.). In Wikipedia. Retrieved from https://en.wikipedia.org/wiki/View_synthesis
2. Neural radiance field. (n.d.). In Wikipedia. Retrieved from https://en.wikipedia.org/wiki/Neural_radiance_field
3. Mildenhall, B., Srinivasan, P. P., Tancik, M., Barron, J. T., Ramamoorthi, R., & Ng, R. (2020). NeRF: Neural radiance fields for image synthesis. arXiv preprint arXiv:2003.08934. Retrieved from https://arxiv.org/pdf/2003.08934
4. Tancik, M., Srinivasan, P. P., Mildenhall, B., Fridovich-Keil, S., Raghavan, N., Singhal, U., Ramamoorthi, R., Barron, J. T., & Ng, R. (2020). Fourier features let networks learn high frequency functions in low dimensional domains. NeurIPS. Retrieved from https://arxiv.org/pdf/2006.10739

Module - Point Sampler

In [3]:
```python
from torch import Tensor
from torch.nn import Module
from torch.types import Device


class PointSampler(Module):
    def __init__(
        self,
        focal: float,
        height: int,
        width: int,
        points_per_ray: int,
        device: Device,
    ):
        """
        `[4, 4] => [height, width, points_per_ray, 3 + 3 + 1]`
        """

        super(PointSampler, self).__init__()

        import torch

        focal = float(focal)
        height = int(height)
        width = int(width)
        points_per_ray = max(int(points_per_ray), 1)

        self.directions = torch.stack(
            torch.meshgrid(
                (torch.arange(float(width), device=device) - width / 2.0) / focal,
                (-torch.arange(float(height), device=device) + height / 2.0) / focal,
                torch.tensor(-1.0, device=device),
                indexing="xy",
            ),
            dim=-1,
        )
        self.points_per_ray = points_per_ray

    def forward(
        self,
        viewpoint: Tensor,
        distance_range: tuple[float, float],
        is_random: bool,
    ) -> Tensor:
        import torch

        viewpoint = torch.as_tensor(viewpoint)[:3]
        device = viewpoint.device
        distance_range = tuple(map(float, distance_range))
        is_random = bool(is_random)
        distance_max = max(distance_range)
        distance_min = min(distance_range)
        interval = (distance_max - distance_min) / self.points_per_ray

        directions = (self.directions * viewpoint[:, :3]).sum(dim=-1).unsqueeze(-2)
        origins = viewpoint[:, -1].expand_as(directions)
        distances = (
            torch.linspace(
```

```
                distance_min,
                distance_max,
                self.points_per_ray,
                device=device,
            )
            .repeat(
                (*origins.shape[:-2], 1),
            )
            .unsqueeze(-1)
        )
        if is_random:
            distances += (
                torch.rand(*origins.shape[:-2], self.points_per_ray, 1, device=device)
                * interval
            )
        positions = origins + directions * distances
        directions = directions.expand_as(positions)
        return torch.cat([positions, directions, distances], dim=-1)
```

Module - Positional Encoder

In [4]:

```
from torch import Tensor
from torch.nn import Module
from torch.types import Device


class PositionalEncoder(Module):
    def __init__(self, encoding_factor: int, device: Device):
        """
        `[..., input_dimension] => [..., input_dimension * (2 * encoding_factor + 1)]`
        """

        import torch

        super(PositionalEncoder, self).__init__()

        encoding_factor = max(int(encoding_factor), 0)

        freq_lvls = torch.arange(encoding_factor, device=device)
        self.freq = ((2**freq_lvls) * torch.pi).repeat_interleave(2).unsqueeze_(-1)
        sine_offsets = torch.tensor([0.0, torch.pi / 2], device=device)
        self.offsets = sine_offsets.repeat(encoding_factor).unsqueeze_(-1)

    def forward(self, inputs: Tensor) -> Tensor:
        import torch

        inputs = torch.as_tensor(inputs).unsqueeze(-2)

        features = (self.freq * inputs + self.offsets).sin()
        features = torch.cat([inputs, features], dim=-2)
        features = features.reshape(*inputs.shape[:-2], -1)
        return features

    def get_last_dimension(self, input_dimension: int) -> int:
        return int(input_dimension) * (self.freq.shape[0] + 1)
```

Module - Volumetric Scene

In [5]:
```python
from torch.nn import Module


class VolumetricScene(Module):
    def __init__(self, encoding_factor: int, device: Device):
        """
        `[..., 3 + 3] => [..., 4]`
        """

        super(VolumetricScene, self).__init__()

        from torch.nn import Linear, ModuleList, ReLU, Sigmoid

        I = 3 + 3
        O = 4
        H = 256

        encoding_factor = max(int(encoding_factor), 0)
        self.encode = PositionalEncoder(encoding_factor, device=device)
        I = self.encode.get_last_dimension(I)

        self.layers = ModuleList(
            [
                Linear(I, H, device=device),
                ReLU(),
                Linear(H, H, device=device),
                ReLU(),
                Linear(H, H, device=device),
                ReLU(),
                Linear(H, H, device=device),
                ReLU(),
                Linear(H, H, device=device),
                ReLU(),
                Linear(H + I, H, device=device),
                ReLU(),
                Linear(H, H, device=device),
                ReLU(),
                Linear(H, H, device=device),
                ReLU(),
                Linear(H, O, device=device),
            ]
        )
        self.skip_indexs = {
            10,
        }
        self.output_rgb_activation = Sigmoid()
        self.output_alpha_activation = ReLU()

    def forward(self, inputs: Tensor) -> Tensor:
        import torch

        inputs = self.encode(inputs)
        outputs = inputs

        for index, layer in enumerate(self.layers):
            if index in self.skip_indexs:
                outputs = torch.cat([outputs, inputs], dim=-1)
            outputs = layer(outputs)
```

```
        outputs = torch.cat(
            [
                self.output_rgb_activation(outputs[..., :3]),
                self.output_alpha_activation(outputs[..., 3:]),
            ],
            dim=-1,
        )
        return outputs
```

Module - Volume Renderer

In [6]:

```
from torch import Tensor
from torch.nn import Module
from torch.types import Device


class VolumeRenderer(Module):
    def __init__(
        self,
        focal: float,
        height: int,
        width: int,
        points_per_ray: int,
        encoding_factor: int,
        device: Device,
    ):
        """
        `[4, 4] => [height, width, 3]`
        """

        super(VolumeRenderer, self).__init__()

        self.sample = PointSampler(
            focal=focal,
            height=height,
            width=width,
            points_per_ray=points_per_ray,
            device=device,
        )
        self.predict = VolumetricScene(
            encoding_factor=encoding_factor,
            device=device,
        )

    def forward(
        self,
        viewpoint: Tensor,
        rays_per_batch: int,
        distance_range: tuple[float, float],
        is_random: bool,
    ) -> Tensor:
        import torch

        rays_per_batch = max(int(rays_per_batch), 1)

        points_per_batch = rays_per_batch * self.sample.points_per_ray
        points: Tensor = self.sample(viewpoint, distance_range, is_random)
        points, distances = points[..., :-1], points[..., -1]
        colors = torch.cat(
            [
                self.predict(batch)
                for batch in TensorDataLoader(
                    data=points.reshape(-1, points.shape[-1]),
                    batch_size=points_per_batch,
                )
            ],
            dim=0,
        )
        colors = colors.reshape(*points.shape[:-1], -1)
```

```
        rgb = colors[..., :3]
        alpha = colors[..., 3]

        intervals = torch.cat(
            [
                distances[..., 1:] - distances[..., :-1],
                torch.tensor([1e9], device=distances.device).expand_as(
                    distances[..., -1:]
                ),
            ],
            dim=-1,
        )
        translucency = (-alpha * intervals).exp().unsqueeze(-1)
        transmittance = (1.0 - translucency) * torch.cumprod(
            translucency + 1e-9, dim=-2
        )
        rgb_planar = (rgb * transmittance).sum(dim=-2)

        return rgb_planar


class TensorDataLoader:
    def __init__(self, data: Tensor, batch_size: int):
        from torch import as_tensor

        self.data = as_tensor(data)
        self.batch_size = max(int(batch_size), 1)

    def __iter__(self):
        return (
            self.data[i : i + self.batch_size]
            for i in range(0, self.data.shape[0], self.batch_size)
        )

    def __len__(self):
        return -(-len(self.data) // self.batch_size)
```

Module - Trainer

In [7]:

```
from dataclasses import dataclass
from torch import Tensor
from torch.types import Device


@dataclass
class Trainer:
    test_dataset: "ViewSynthesisDataset"
    train_dataset: "ViewSynthesisDataset"

    @staticmethod
    def from_dataset(
        dataset: "ViewSynthesisDataset",
        train_ratio: float,
        device: Device,
    ):
        train_data_count = max(int(round(dataset.count * train_ratio)), 1)
        test_data_count = dataset.count - train_data_count
        data_split_index = -test_data_count
        focal = dataset.focal
        height = dataset.height
        images = dataset.images
        viewpoints = dataset.viewpoints
        width = dataset.width

        return Trainer(
            test_dataset=ViewSynthesisDataset(
                count=test_data_count,
                focal=focal,
                height=height,
                images=images[data_split_index:],
                viewpoints=viewpoints[data_split_index:],
                width=width,
            ).set_device(device),
            train_dataset=ViewSynthesisDataset(
                count=train_data_count,
                focal=focal,
                height=height,
                images=images[:data_split_index],
                viewpoints=viewpoints[:data_split_index],
                width=width,
            ).set_device(device),
        )

    def train(
        self,
        render: VolumeRenderer,
        epochs: int,
        rays_per_batch: int,
        distance_range: tuple[float, float],
        show_progress: bool,
    ) -> None:
        import torch
        from torch.nn import MSELoss
        from torch.optim import Adam
        from tqdm import tqdm

        EPOCHS_PER_DEMO = 250
```

```python
        criterion = MSELoss()
        optimizer = Adam(render.parameters(), lr=5e-4)
        progress = tqdm(
            disable=not show_progress,
            desc=f"Fitting the renderer to {self.train_dataset.count}x images and viewpoints",
            colour="green",
            dynamic_ncols=True,
            total=epochs,
        )

        with progress:
            for epoch in range(epochs):
                true_image, viewpoint = self.train_dataset.get_image_and_viewpoint()
                optimizer.zero_grad()
                rendered_image: Tensor = render(
                    viewpoint=viewpoint,
                    rays_per_batch=rays_per_batch,
                    distance_range=distance_range,
                    is_random=True,
                )
                loss = criterion(rendered_image, true_image)
                loss.backward()
                optimizer.step()

                if show_progress and epoch % EPOCHS_PER_DEMO == 0:
                    with torch.no_grad():
                        true_image_demo, viewpoint_demo = (
                            self.train_dataset.get_image_and_viewpoint(0)
                        )
                        rendered_image_demo = render(
                            viewpoint=viewpoint_demo,
                            rays_per_batch=rays_per_batch,
                            distance_range=distance_range,
                            is_random=False,
                        )
                        Trainer.display(
                            torch.cat([true_image_demo, rendered_image_demo], dim=1)
                        )

                progress.update()

    def test(
        self,
        render: VolumeRenderer,
        rays_per_batch: int,
        distance_range: tuple[float, float],
    ) -> None:
        from math import log10
        import torch
        from torch.nn.functional import mse_loss

        with torch.no_grad():
            for index in range(self.test_dataset.count):
                true_image, viewpoint = self.test_dataset.get_image_and_viewpoint(index)
                rendered_image = render(
                    viewpoint=viewpoint,
                    rays_per_batch=rays_per_batch,
                    distance_range=distance_range,
                    is_random=False,
                )
                quality_mse = mse_loss(rendered_image, true_image).item()
                quality_psnr = -10 * log10(quality_mse)
                display(
                    dict(
                        test_index=index,
```

```python
                        quality=dict(mse=quality_mse, psnr=quality_psnr),
                    )
                )

                Trainer.display(torch.cat([true_image, rendered_image], dim=1))

    @staticmethod
    def display(image: Tensor):
        from IPython.display import display
        from PIL import Image
        from torch import uint8

        return display(
            Image.fromarray((image * 255).round().type(uint8).numpy(force=True))
        )


@dataclass
class ViewSynthesisDataset:
    count: int
    focal: float
    height: int
    images: Tensor
    viewpoints: Tensor
    width: int

    def __post_init__(self) -> None:
        if self.images.shape[0] != self.viewpoints.shape[0]:
            raise ValueError("The number of images and viewpoints must be the same")

    @staticmethod
    def from_numpy(url: str) -> "ViewSynthesisDataset":
        from httpx import get
        from io import BytesIO
        from numpy import load

        import torch

        try:
            file = BytesIO(
                get(url, follow_redirects=True, timeout=60).raise_for_status().content
            )
        except:
            file = open(url, "rb")

        with file as file_entered:
            arrays = load(file_entered)
            focal = float(arrays["focal"])
            images = torch.as_tensor(arrays["images"])
            viewpoints = torch.as_tensor(arrays["poses"])

        return ViewSynthesisDataset(
            count=images.shape[0],
            focal=focal,
            height=images.shape[1],
            images=images,
            viewpoints=viewpoints,
            width=images.shape[2],
        )

    def get_image_and_viewpoint(self, index: int | None = None) -> tuple[Tensor, Tensor]:
        from random import randint

        if index is not None:
            index = int(index)
        else:
```

```python
            index = randint(0, self.count - 1)

        return self.images[index], self.viewpoints[index]

    def set_device(self, device: Device) -> "ViewSynthesisDataset":
        self.images = self.images.to(device)
        self.viewpoints = self.viewpoints.to(device)
        return self

    def __repr__(self) -> str:
        repr = f"{self.__class__.__name__}("
        for name, value in self.__dict__.items():
            if isinstance(value, Tensor):
                value = f"Tensor(shape={tuple(value.shape)}, dtype={value.dtype})"
            elif type(value) is float:
                value = f"{value:.7f}"
            repr += f"\n  {name}={value},"
        repr += "\n)"
        return repr
```

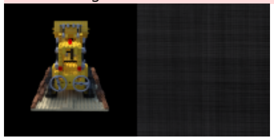
Experiment

In [8]:
```
def main() -> None:
    import torch

    EPOCHS = 10000
    RAYS_PER_BATCH = 250
    POINTS_PER_RAY = 80
    ENCODING_FACTOR = 10
    DISTANCE_RANGE = (2.0, 6.0)

    # The original dataset is retrieved from
    # http://cseweb.ucsd.edu/~viscomp/projects/LF/papers/ECCV20/nerf/tiny_nerf_data.npz
    dataset = ViewSynthesisDataset.from_numpy(
        "https://raw.githubusercontent.com/AsherJingkongChen/nerf/main/tiny_nerf_data.npz"
    )
    device = (
        "cuda:1"
        if torch.cuda.is_available()
        else ("mps" if torch.backends.mps.is_available() else "cpu")
    )
    render = VolumeRenderer(
        focal=dataset.focal,
        height=dataset.height,
        width=dataset.width,
        points_per_ray=POINTS_PER_RAY,
        encoding_factor=ENCODING_FACTOR,
        device=device,
    )
    trainer = Trainer.from_dataset(dataset=dataset, train_ratio=0.8, device=device)

    display(dict(trainer=trainer, render=render))

    trainer.train(
        render=render,
        epochs=EPOCHS,
        rays_per_batch=RAYS_PER_BATCH,
        distance_range=DISTANCE_RANGE,
        show_progress=True,
    )
    trainer.test(
        render=render,
        rays_per_batch=RAYS_PER_BATCH,
        distance_range=DISTANCE_RANGE,
    )
    torch.save(render.state_dict(), "VolumeRenderer.pth")
```

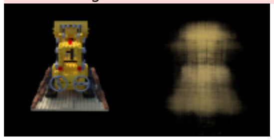
In [9]:
```
if __name__ == "__main__":
    main()
```

```
{'trainer': Trainer(test_dataset=ViewSynthesisDataset(
    count=21,
    focal=138.8888789,
    height=100,
    images=Tensor(shape=(21, 100, 100, 3), dtype=torch.float32),
    viewpoints=Tensor(shape=(21, 4, 4), dtype=torch.float32),
    width=100,
  ), train_dataset=ViewSynthesisDataset(
    count=85,
    focal=138.8888789,
    height=100,
    images=Tensor(shape=(85, 100, 100, 3), dtype=torch.float32),
    viewpoints=Tensor(shape=(85, 4, 4), dtype=torch.float32),
    width=100,
  )),
  'render': VolumeRenderer(
    (sample): PointSampler()
    (predict): VolumetricScene(
      (encode): PositionalEncoder()
      (layers): ModuleList(
        (0): Linear(in_features=126, out_features=256, bias=True)
        (1): ReLU()
        (2): Linear(in_features=256, out_features=256, bias=True)
        (3): ReLU()
        (4): Linear(in_features=256, out_features=256, bias=True)
        (5): ReLU()
        (6): Linear(in_features=256, out_features=256, bias=True)
        (7): ReLU()
        (8): Linear(in_features=256, out_features=256, bias=True)
        (9): ReLU()
        (10): Linear(in_features=382, out_features=256, bias=True)
        (11): ReLU()
        (12): Linear(in_features=256, out_features=256, bias=True)
        (13): ReLU()
        (14): Linear(in_features=256, out_features=256, bias=True)
        (15): ReLU()
        (16): Linear(in_features=256, out_features=4, bias=True)
      )
      (output_rgb_activation): Sigmoid()
      (output_alpha_activation): ReLU()
    )
  )}
```

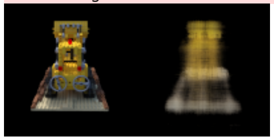
```
Fitting the renderer to 85x images and viewpoints:   0%|          | 0/10000 [00:00<?, ?it/s]
```



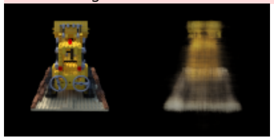
```
Fitting the renderer to 85x images and viewpoints:   2%|▏         | 250/10000 [02:18<1:34:22,  1.72it/s]
```



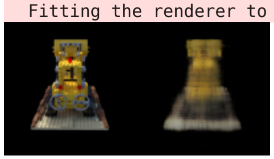
```
Fitting the renderer to 85x images and viewpoints:   5%|▌         | 500/10000 [04:43<1:32:26,  1.71it/s]
```



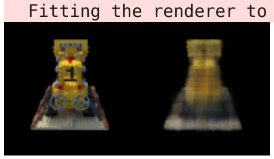
```
Fitting the renderer to 85x images and viewpoints:   8%|          | 750/10000 [07:09<1:29:36,  1.72it/s]
```



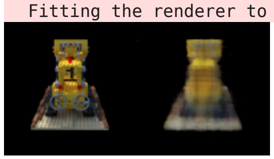
```
Fitting the renderer to 85x images and viewpoints:  10%|          | 1000/10000 [09:34<1:26:56,  1.73it/s]
```



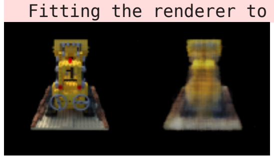
```
Fitting the renderer to 85x images and viewpoints:  12%|          | 1250/10000 [11:59<1:24:12,  1.73it/s]
```



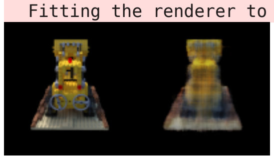
```
Fitting the renderer to 85x images and viewpoints:  15%|          | 1500/10000 [14:24<1:22:21,  1.72it/s]
```



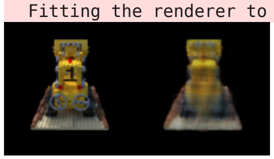
```
Fitting the renderer to 85x images and viewpoints:  18%|          | 1750/10000 [16:49<1:19:55,  1.72it/s]
```



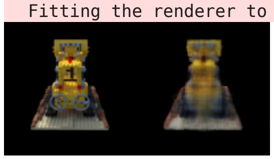
```
Fitting the renderer to 85x images and viewpoints:  20%|          | 2000/10000 [19:15<1:17:31,  1.72it/s]
```



```
Fitting the renderer to 85x images and viewpoints:  22%|          | 2250/10000 [21:40<1:15:21,  1.71it/s]
```

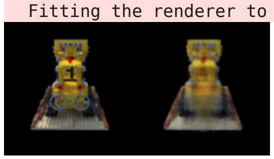


```
Fitting the renderer to 85x images and viewpoints:  25%|          | 2500/10000 [24:05<1:12:23,  1.73it/s]
```

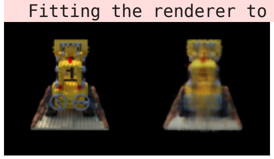


```
Fitting the renderer to 85x images and viewpoints:  28%|          | 2750/10000 [26:31<1:10:01,  1.73it/s]
```

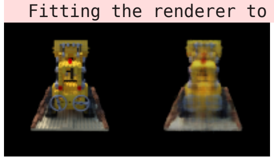
Fitting the renderer to 85x images and viewpoints: 30%| | 3000/10000 [28:56<1:07:43, 1.72it/s]



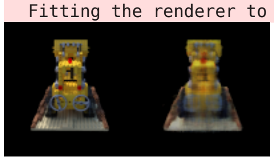
Fitting the renderer to 85x images and viewpoints: 32%| | 3250/10000 [31:21<1:05:16, 1.72it/s]



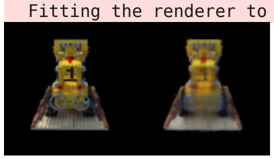
Fitting the renderer to 85x images and viewpoints: 35%| | 3500/10000 [33:46<1:02:36, 1.73it/s]



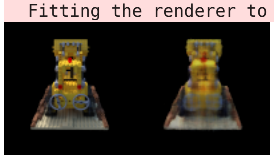
Fitting the renderer to 85x images and viewpoints: 38%| | 3750/10000 [36:12<1:00:41, 1.72it/s]



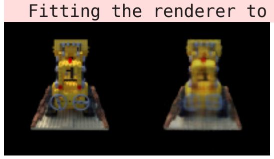
Fitting the renderer to 85x images and viewpoints: 40%| | 4000/10000 [38:37<58:03, 1.72it/s]



Fitting the renderer to 85x images and viewpoints: 42%| | 4250/10000 [41:02<55:40, 1.72it/s]

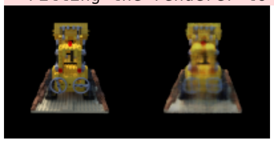


Fitting the renderer to 85x images and viewpoints: 45%| | 4500/10000 [43:28<53:05, 1.73it/s]

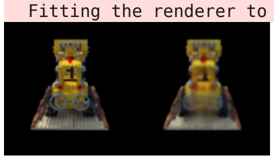


Fitting the renderer to 85x images and viewpoints: 48%| | 4750/10000 [45:53<50:39, 1.73it/s]

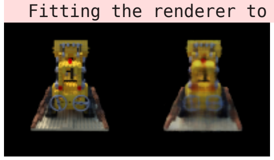
```
Fitting the renderer to 85x images and viewpoints:  50%|██████████          | 5000/10000 [48:18<48:33,  1.72it/s]
```



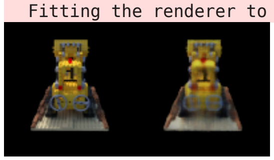
```
Fitting the renderer to 85x images and viewpoints:  52%|██████████          | 5250/10000 [50:44<46:12,  1.71it/s]
```



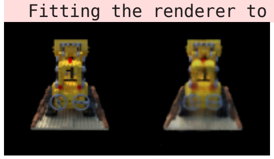
```
Fitting the renderer to 85x images and viewpoints:  55%|███████████         | 5500/10000 [53:09<43:40,  1.72it/s]
```



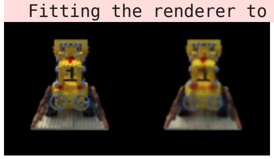
```
Fitting the renderer to 85x images and viewpoints:  57%|███████████         | 5750/10000 [55:34<41:04,  1.72it/s]
```



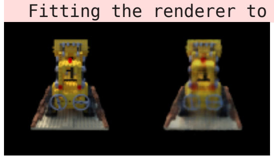
```
Fitting the renderer to 85x images and viewpoints:  60%|████████████        | 6000/10000 [58:00<38:56,  1.71it/s]
```



```
Fitting the renderer to 85x images and viewpoints:  62%|████████████        | 6250/10000 [1:00:25<36:12,  1.73it/s]
```

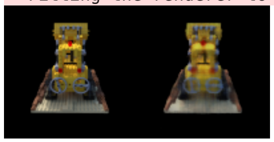


```
Fitting the renderer to 85x images and viewpoints:  65%|█████████████       | 6500/10000 [1:02:51<33:53,  1.72it/s]
```

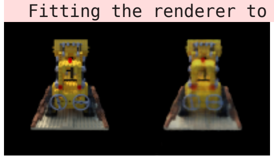


```
Fitting the renderer to 85x images and viewpoints:  68%|█████████████       | 6750/10000 [1:05:16<31:27,  1.72it/s]
```

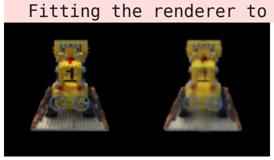
```
Fitting the renderer to 85x images and viewpoints:  70%|██████████  | 7000/10000 [1:07:41<28:54,  1.73it/s]
```



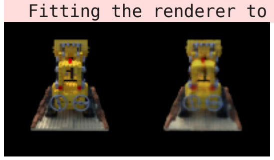
```
Fitting the renderer to 85x images and viewpoints:  72%|██████████  | 7250/10000 [1:10:07<26:27,  1.73it/s]
```



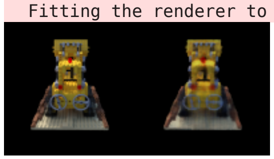
```
Fitting the renderer to 85x images and viewpoints:  75%|██████████  | 7500/10000 [1:12:32<24:10,  1.72it/s]
```



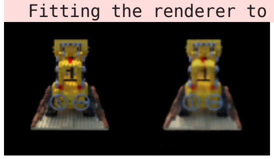
```
Fitting the renderer to 85x images and viewpoints:  78%|██████████  | 7750/10000 [1:14:58<21:50,  1.72it/s]
```



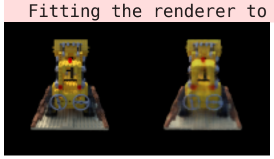
```
Fitting the renderer to 85x images and viewpoints:  80%|██████████  | 8000/10000 [1:17:23<19:20,  1.72it/s]
```



```
Fitting the renderer to 85x images and viewpoints:  82%|██████████  | 8250/10000 [1:19:48<16:56,  1.72it/s]
```

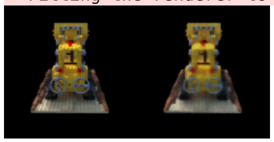


```
Fitting the renderer to 85x images and viewpoints:  85%|██████████  | 8500/10000 [1:22:13<14:30,  1.72it/s]
```

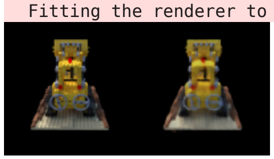


```
Fitting the renderer to 85x images and viewpoints:  88%|██████████  | 8750/10000 [1:24:38<11:58,  1.74it/s]
```

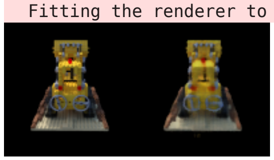
```
Fitting the renderer to 85x images and viewpoints:  90%|█████████████████████████████████████      | 9000/10000 [1:27:02<09:36,  1.73it/s]
```



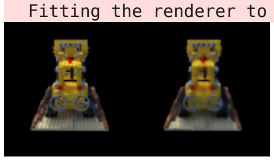
```
Fitting the renderer to 85x images and viewpoints:  92%|██████████████████████████████████████     | 9250/10000 [1:29:27<07:14,  1.73it/s]
```



```
Fitting the renderer to 85x images and viewpoints:  95%|███████████████████████████████████████    | 9500/10000 [1:31:52<04:51,  1.72it/s]
```

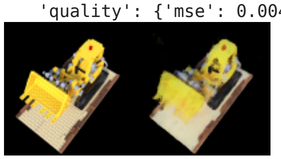


```
Fitting the renderer to 85x images and viewpoints:  98%|████████████████████████████████████████   | 9750/10000 [1:34:18<02:25,  1.72it/s]
```

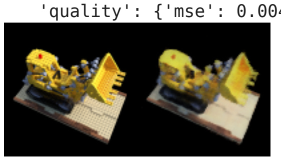


```
Fitting the renderer to 85x images and viewpoints: 100%|█████████████████████████████████████████| 10000/10000 [1:36:44<00:00,  1.72it/s]
{'test_index': 0,
 'quality': {'mse': 0.0018234748858958483, 'psnr': 27.391002137003284}}
```

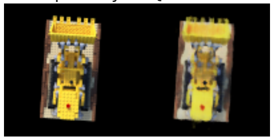
```
{'test_index': 1,
 'quality': {'mse': 0.004227453842759132, 'psnr': 23.73921125526659}}
```



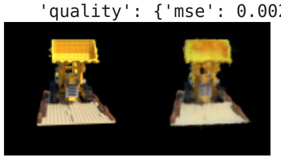
```
{'test_index': 2,
 'quality': {'mse': 0.004046123940497637, 'psnr': 23.929608180926998}}
```



```
{'test_index': 3,
 'quality': {'mse': 0.00615657540038228, 'psnr': 22.10660797298503}}
```

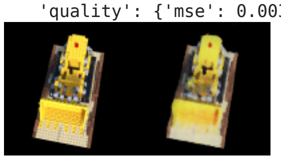


```
{'test_index': 4,
 'quality': {'mse': 0.002972586778923869, 'psnr': 25.268654581364643}}
```

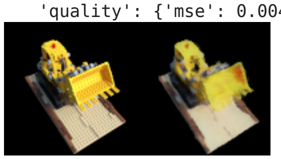


```
{'test_index': 5,
 'quality': {'mse': 0.002282568719238043, 'psnr': 26.415761386845162}}
```

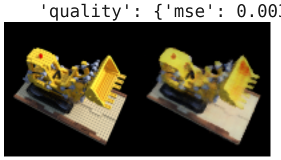
```
{'test_index': 6,
 'quality': {'mse': 0.0034460092429071665, 'psnr': 24.626835620191}}
```



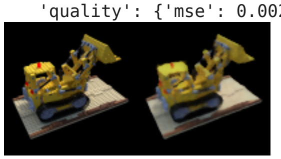
```
{'test_index': 7,
 'quality': {'mse': 0.00469020614400506, 'psnr': 23.28808068748372}}
```



```
{'test_index': 8,
 'quality': {'mse': 0.003257329575717449, 'psnr': 24.8713829741482}}
```



```
{'test_index': 9,
 'quality': {'mse': 0.0020031288731843233, 'psnr': 26.982911090627034}}
```

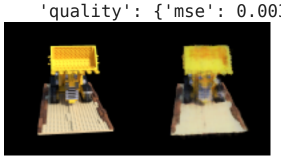


```
{'test_index': 10,
 'quality': {'mse': 0.006571581121534109, 'psnr': 21.82330126673976}}
```

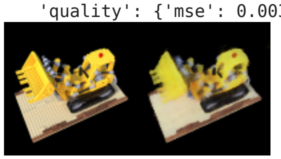
```
{'test_index': 11,
 'quality': {'mse': 0.0018614789005368948, 'psnr': 27.301418820522663}}
```

```
{'test_index': 12,
 'quality': {'mse': 0.0017940590623766184, 'psnr': 27.46163263609637}}
```

```
{'test_index': 13,
 'quality': {'mse': 0.0031179401557892 56, 'psnr': 25.061322247019934}}
```



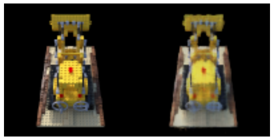
```
{'test_index': 14,
 'quality': {'mse': 0.0039011535700410604, 'psnr': 24.088069532235394}}
```



```
{'test_index': 15,
 'quality': {'mse': 0.003562702564522624, 'psnr': 24.482204336070325}}
```

```
{'test_index': 16,
 'quality': {'mse': 0.004903879947960377, 'psnr': 23.094601703364074}}
```

```
{'test_index': 17,
 'quality': {'mse': 0.0024387307930737734, 'psnr': 26.128361379951386}}
```



```
{'test_index': 18,
 'quality': {'mse': 0.0027348005678504705, 'psnr': 25.63074338595643}}
```

```
{'test_index': 19,
 'quality': {'mse': 0.0015439643757417798, 'psnr': 28.11362724464825}}
```

```
{'test_index': 20,
 'quality': {'mse': 0.0028872385155409575, 'psnr': 25.395173374547365}}
```