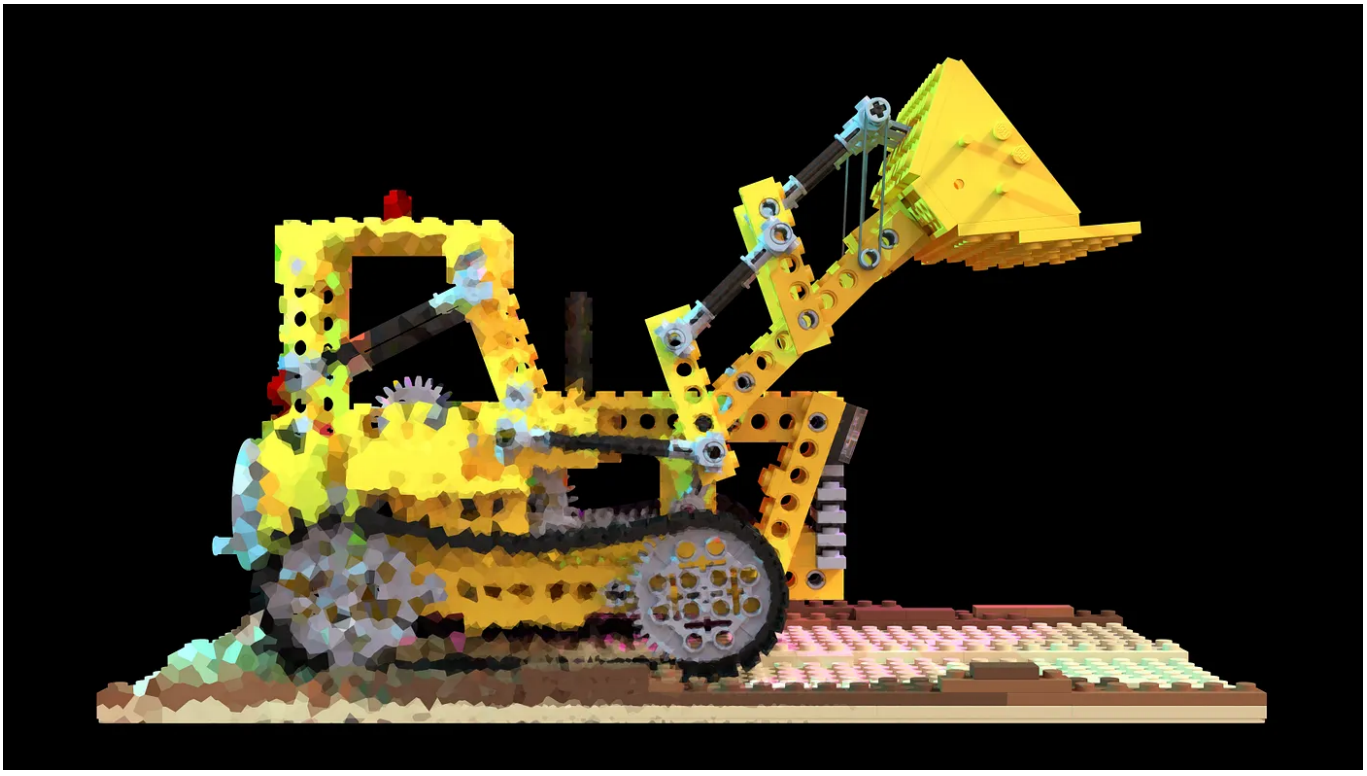
✨ Member-only story

It's NeRF From Nothing: Build A Complete NeRF with PyTorch

A tutorial for how to build your own NeRF model in PyTorch, with step-by-step explanations of each component

Mason McGough · Follow

Published in Towards Data Science · 10 min read · Apr 28, 2022



3D model from Matthew Tancik. Photo by author.

[Note: this article includes a Google Colab notebook. Feel free to use it if you wish to follow along.]

Introduction

NeRF Explosion

The Neural Radiance Field, or NeRF, is a fairly new paradigm in the world of deep learning and computer vision. Introduced in the ECCV 2020 paper "NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis" (which received an honorable mention for Best Paper), the technique has since exploded in popularity and has received nearly 800 citations thus far[1]. The method marks a dramatic change from the conventional ways by which machine learning handles 3D data. The visuals certainly have that "wow" factor as well, which you can see on the project page and in their original video (below).

NeRF introduction video created by Matthew Tancik, one of the authors of the NeRF paper.

In this tutorial, we will walk through the essential components of a NeRF and how to put them all together to train our own NeRF model. Before we get started, however, let's first take a look at what a NeRF really is and what makes it such a breakthrough.

What is a NeRF?

In short, a NeRF is a generative model of sorts, conditioned on a collection of images and accurate poses (e.g. position and rotation), that allows you to generate new views of a 3D scene shared by the images, a process often referred to as "novel view synthesis." Not only that, it explicitly defines the 3D shape and appearance of the scene as a continuous function, with which you can do things like generate a 3D mesh via marching cubes. One thing you may find surprising about NeRFs: although they learn directly from image data, they use neither convolutional nor transformer layers (at least not the original). An understated benefit of NeRFs is compression; at 5–10MB, the weights of a NeRF model may be smaller than the collection of images used to train them.

The proper way to represent 3D data for machine learning applications has been a subject of debate for many years. Many techniques have emerged, ranging from 3D voxels to point clouds to signed distance functions (for more details on some common 3D representations, see my MeshCNN article). Their big common drawback is an initial assumption: most representations require a 3D model, requiring you to either produce 3D data using tools like photogrammetry (which is very time- and data-hungry) and LiDAR (which is often expensive and tricky to use) or to pay an artist to make a 3D model for you. In addition, many types of objects, such as highly reflective objects, "mesh-like" objects such as bushes and chain-link fences, or transparent objects are impractical to scan at scale. 3D reconstruction

methods often have reconstruction errors as well which can cause stair-stepping effects or drift that impact the accuracy of the model.

NeRFs by contrast rely on an old yet elegant concept called light fields, or radiance fields. A light field is a function that describes how light transport occurs throughout a 3D volume. It describes the direction of light rays moving through every $\boldsymbol{x}=(x, y, z)$ coordinate in space and in every direction $\boldsymbol{d}$, described either as θ and ϕ angles or a unit vector. Collectively they form a 5D feature space that describes light transport in a 3D scene. The NeRF, inspired by this representation, attempts to approximate a function that maps from this space into a 4D space consisting of color $\mathbf{c}$=(R,G,B) and a density σ, which you can think of as the likelihood that the light ray at this 5D coordinate space is terminated (e.g. by occlusion). The standard NeRF is thus a function of the form $F : (\boldsymbol{x},\boldsymbol{d}) \rightarrow (\boldsymbol{c},\sigma)$.

The original NeRF paper parameterizes this function with a multilayer perceptron trained directly on a set of images with known poses (which can be obtained via any structure-from-motion application such as COLMAP, Agisoft Metashape, Reality Capture, or Meshroom). This is one method in a class of techniques called *generalized scene reconstruction*, which aim to describe a 3D scene directly from an ensemble of images. This approach gives us some very nice properties:

- Learns from data directly
- Continuous representation of the scene allows for very thin and complex structures such as leaves of a tree or meshes
- Implicitly accounts for physical properties like specularity and roughness
- Implicitly represents lighting in a scene

A litany of papers have since sought to expand the functionality of the original with features such as few-shot and one-shot learning[2, 3], support for dynamic scenes[4, 5], generalizing the light field into feature fields[6], learning from uncalibrated image collections from the web[7], combining with LiDAR data[8], large-scale scene representation[9], learning without a neural network[10], and many more. For some great overviews of NeRF research, see this great overview from 2020 and another overview from 2021 both by Frank Dellaert.

NeRF Architecture

With this function alone, it is still not readily apparent how you can generate novel images. Overall, given a trained NeRF model and a camera with known pose and image dimensions, we construct a scene by the following process:

1. For each pixel, march camera rays through scene to gather a set of samples at (**x, d**) locations.
2. Use (**x, d**) points and viewing directions at each sample as input to produce output $(\boldsymbol{c}, \boldsymbol{\sigma})$ values (essentially rgbσ).
3. Construct an image using classical volume rendering techniques.

The radiance field function is only one of several components that, once combined, allow you to create the visuals seen in the video you saw earlier. There are several more components, each of which I will visit in the tutorial. Overall we will cover the following components:

- Positional encoding
- The radiance field function approximator (in this case, an MLP)
- Differentiable volume renderer

- Stratified sampling
- Hierarchical volume sampling

My intent for this article is maximum clarity, so I have extracted the key elements of each component into the most concise code possible. I have used the original implementation by GitHub user bmild and PyTorch implementations from GitHub users yenchenlin and krrish94 as references.

Tutorial

Positional Encoder

Much like the insanely popular transformer model introduced in 2017[11], the NeRF also benefits from a positional encoder as its input, albeit for a different reason. In short, it maps its continuous input to a higher-dimensional space using high-frequency functions to aid the model in learning high frequency variations in the data, which leads to sharper models. This approach circumvents the bias that neural networks have towards lower frequency functions, allowing NeRF to represent sharper details. The authors refer to a paper at ICML 2019 for further reading on this phenomenon[12].

If you are familiar with positional encoders, the NeRF implementation is fairly standard. It has the same alternating sine and cosine expressions that are a hallmark of the original.

```
class PositionalEncoder(nn.Module):
  r"""
  Sine-cosine positional encoder for input points.
  """
  def __init__(
    self,
    d_input: int
```

```
    d_input: int,
    n_freqs: int,
    log_space: bool = False
  ):
    super().__init__()
    self.d_input = d_input
    self.n_freqs = n_freqs
    self.log_space = log_space
    self.d_output = d_input * (1 + 2 * self.n_freqs)
    self.embed_fns = [lambda x: x]

    # Define frequencies in either linear or log scale
    if self.log_space:
      freq_bands = 2.**torch.linspace(0., self.n_freqs - 1, self.n_freqs)
    else:
      freq_bands = torch.linspace(2.**0., 2.**(self.n_freqs - 1), self.n_freqs)

    # Alternate sin and cos
    for freq in freq_bands:
      self.embed_fns.append(lambda x, freq=freq: torch.sin(x * freq))
```



```
    x
  ) -> torch.Tensor:
    r"""
    Apply positional encoding to input.
    """
    return torch.concat([fn(x) for fn in self.embed_fns], dim=-1)
```

nerf_positional_encoder.py hosted with ❤️ by GitHub view raw

Standard positional encoder implementation.

Radiance Field Function

In the original paper, the radiance field function was represented by the NeRF model, a fairly typical multilayer perceptron that takes encoded 3D points and view directions as inputs and returns RGBA values as outputs. While this paper uses a neural network, any function approximator can be used here. For instance, a follow-up paper by Yu et al. called Plenoxels uses a

basis of spherical harmonics instead for orders of magnitude faster training while achieving competitive results[10].

The NeRF model is 8 layers deep with feature dimension of 256 for most layers. A residual connection is placed at layer 4. After these layers, the RGB and σ values are produced. The RGB values are further processed with a linear layer, then concatenated with the view directions, then passed through yet another linear layer before finally being recombined with σ at the output.

```
class NeRF(nn.Module):
  r"""
  Neural radiance fields module.
  """
  def __init__(
    self,
    d_input: int = 3,
    n_layers: int = 8,
    d_filter: int = 256,
    skip: Tuple[int] = (4,),
    d_viewdirs: Optional[int] = None
  ):
    super().__init__()
    self.d_input = d_input
    self.skip = skip
    self.act = nn.functional.relu
    self.d_viewdirs = d_viewdirs

    # Create model layers
    self.layers = nn.ModuleList(
      [nn.Linear(self.d_input, d_filter)] +
      [nn.Linear(d_filter + self.d_input, d_filter) if i in skip \
       else nn.Linear(d_filter, d_filter) for i in range(n_layers - 1)]
    )

    # Bottleneck layers
    if self.d_viewdirs is not None:
      # If using viewdirs, split alpha and RGB
      self.alpha_out = nn.Linear(d_filter, 1)
      self.rgb_filters = nn.Linear(d_filter, d_filter)
```

```
30        self.rgb_filters = nn.Linear(d_filter, d_filter)
31        self.branch = nn.Linear(d_filter + self.d_viewdirs, d_filter // 2)
32        self.output = nn.Linear(d_filter // 2, 3)
33      else:
34        # If no viewdirs, use simpler output
35        self.output = nn.Linear(d_filter, 4)
36
37    def forward(
38      self,
39      x: torch.Tensor,
40      viewdirs: Optional[torch.Tensor] = None
41    ) -> torch.Tensor:
42      r"""
43      Forward pass with optional view direction.
44      """
45
46      # Cannot use viewdirs if instantiated with d_viewdirs = None
47      if self.d_viewdirs is None and viewdirs is not None:
48        raise ValueError('Cannot input x_direction if d_viewdirs was not given.')
49
50      # Apply forward pass up to bottleneck
51      x_input = x
52      for i, layer in enumerate(self.layers):
53        x = self.act(layer(x))
54        if i in self.skip:
55          x = torch.cat([x, x_input], dim=-1)
56
57      # Apply bottleneck
58      if self.d_viewdirs is not None:
59        # Split alpha from network output
60        alpha = self.alpha_out(x)
61
62        # Pass through bottleneck to get RGB
63        x = self.rgb_filters(x)
64        x = torch.concat([x, viewdirs], dim=-1)
65        x = self.act(self.branch(x))
66        x = self.output(x)
67
68        # Concatenate alphas to output
69        x = torch.concat([x, alpha], dim=-1)
70      else:
71        # Simple output
72        x = self.output(x)
73      return x
```

nerf_module.py hosted with ❤ by GitHub view raw

The NeRF model, implemented as a PyTorch module.

Differentiable Volume Renderer

The RGBA output points are in 3D space, so to composite them into an image we need to apply the volume integration described in Equations 1–3 in Section 4 of the paper. Essentially, we take the weighted sum of all samples along the ray of each pixel to get the estimated color value at that pixel. Each RGB sample is weighted by its alpha value. Higher alpha values indicate higher likelihood that the sampled area is opaque, therefore points further along the ray are likelier to be occluded. The cumulative product ensures that those further points are dampened.

```python
def raw2outputs(
  raw: torch.Tensor,
  z_vals: torch.Tensor,
  rays_d: torch.Tensor,
  raw_noise_std: float = 0.0,
  white_bkgd: bool = False
) -> Tuple[torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor]:
  r"""
  Convert the raw NeRF output into RGB and other maps.
  """

  # Difference between consecutive elements of `z_vals`. [n_rays, n_samples]
  dists = z_vals[..., 1:] - z_vals[..., :-1]
  dists = torch.cat([dists, 1e10 * torch.ones_like(dists[..., :1])], dim=-1)

  # Multiply each distance by the norm of its corresponding direction ray
  # to convert to real world distance (accounts for non-unit directions).
  dists = dists * torch.norm(rays_d[..., None, :], dim=-1)

  # Add noise to model's predictions for density. Can be used to
  # regularize network during training (prevents floater artifacts).
  noise = 0.
  if raw_noise_std > 0.:
    noise = torch.randn(raw[..., 3].shape) * raw_noise_std

  # Predict density of each sample along each ray. Higher values imply
```

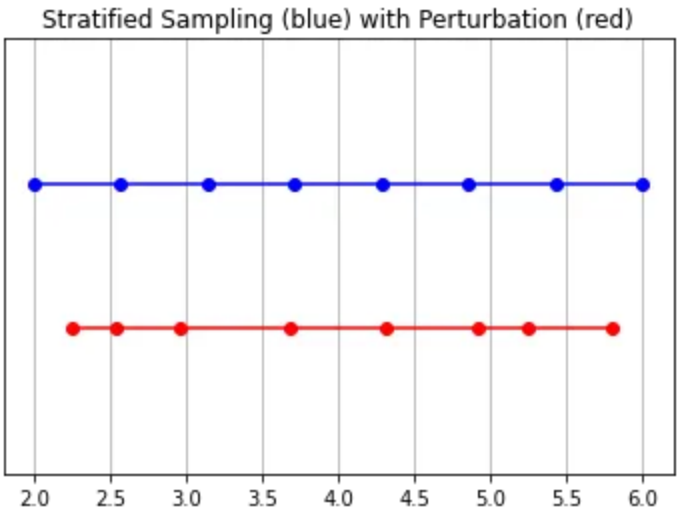
```
    # higher likelihood of being absorbed at this point. [n_rays, n_samples]
    alpha = 1.0 - torch.exp(-nn.functional.relu(raw[..., 3] + noise) * dists)

    # Compute weight for RGB of each sample along each ray. [n_rays, n_samples]
    # The higher the alpha, the lower subsequent weights are driven.
    weights = alpha * cumprod_exclusive(1. - alpha + 1e-10)

    # Compute weighted RGB map.
    rgb = torch.sigmoid(raw[..., :3])  # [n_rays, n_samples, 3]
    rgb_map = torch.sum(weights[..., None] * rgb, dim=-2)  # [n_rays, 3]

    # Estimated depth map is predicted distance.
    depth_map = torch.sum(weights * z_vals, dim=-1)

    # Disparity map is inverse depth.
    disp_map = 1. / torch.max(1e-10 * torch.ones_like(depth_map),
                              depth_map / torch.sum(weights, -1))

    # Sum of weights along each ray. In [0, 1] up to numerical error.
    acc_map = torch.sum(weights, dim=-1)

    # To composite onto a white background, use the accumulated alpha map.
    if white_bkgd:
      rgb_map = rgb_map + (1. - acc_map[..., None])

    return rgb_map, depth_map, acc_map, weights


def cumprod_exclusive(
    tensor: torch.Tensor
) -> torch.Tensor:
    r"""
    (Courtesy of https://github.com/krrish94/nerf-pytorch)

    Mimick functionality of tf.math.cumprod(..., exclusive=True), as it isn't available in

    Args:
    tensor (torch.Tensor): Tensor whose cumprod (cumulative product, see `torch.cumprod`)
      is to be computed.
    Returns:
    cumprod (torch.Tensor): cumprod of Tensor along dim=-1, mimiciking the functionality o
      tf.math.cumprod(..., exclusive=True) (see `tf.math.cumprod` for details).
    """

    # Compute regular cumprod first (this is equivalent to `tf.math.cumprod(..., exclusive
```

```
    cumprod = torch.cumprod(tensor, -1)
    # "Roll" the elements along dimension 'dim' by 1 element.
    cumprod = torch.roll(cumprod, 1, -1)
    # Replace the first element by "1" as this is what tf.cumprod(..., exclusive=True) doe
    cumprod[..., 0] = 1.

    return cumprod
```

nerf_volume_render.py hosted with ❤️ by GitHub view raw

Volume rendering of the raw NeRF model output.

Stratified Sampling



Example of stratified sampling. In this case, 8 samples were collected from the range [2, 6]. (Figure created by author)

In this model, the RGB value that the camera ultimately picks up is the accumulation of light samples along the ray passing through that pixel. The classical volume rendering approach is to accumulate and then integrate points along this ray, estimating at each point the probability that the ray travels without hitting any particle. Each pixel therefore requires a sampling of points along the ray passing through it. To best approximate the integral, their stratified sampling approach is to divide the space evenly into N bins and draw a sample uniformly from each. Rather than simply drawing samples at regular spacing, the stratified sampling approach allows the

model to sample a continuous space, therefore conditioning the network to learn over a continuous space.

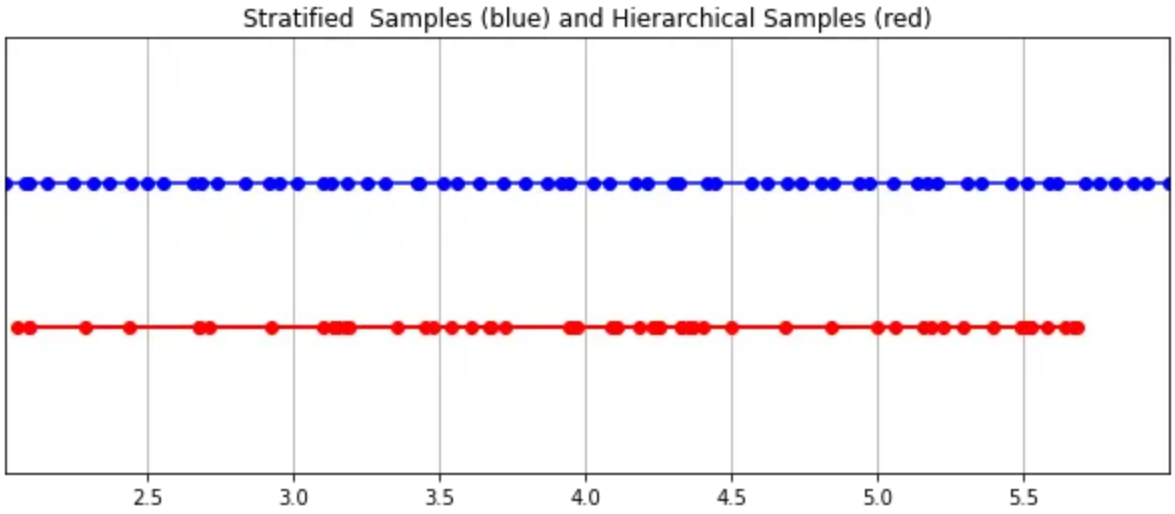
```python
def sample_stratified(
  rays_o: torch.Tensor,
  rays_d: torch.Tensor,
  near: float,
  far: float,
  n_samples: int,
  perturb: Optional[bool] = True,
  inverse_depth: bool = False
) -> Tuple[torch.Tensor, torch.Tensor]:
  r"""
  Sample along ray from regularly-spaced bins.
  """

  # Grab samples for space integration along ray
  t_vals = torch.linspace(0., 1., n_samples, device=rays_o.device)
  if not inverse_depth:
    # Sample linearly between `near` and `far`
    z_vals = near * (1.-t_vals) + far * (t_vals)
  else:
    # Sample linearly in inverse depth (disparity)
    z_vals = 1./(1./near * (1.-t_vals) + 1./far * (t_vals))

  # Draw uniform samples from bins along ray
  if perturb:
    mids = .5 * (z_vals[1:] + z_vals[:-1])
    upper = torch.concat([mids, z_vals[-1:]], dim=-1)
    lower = torch.concat([z_vals[:1], mids], dim=-1)
    t_rand = torch.rand([n_samples], device=z_vals.device)
    z_vals = lower + (upper - lower) * t_rand
  z_vals = z_vals.expand(list(rays_o.shape[:-1]) + [n_samples])

  # Apply scale from `rays_d` and offset from `rays_o` to samples
  # pts: (width, height, n_samples, 3)
  pts = rays_o[..., None, :] + rays_d[..., None, :] * z_vals[..., :, None]
  return pts, z_vals
```

nerf_sample_stratified.py hosted with ❤️ by GitHub view raw

Stratified sampling in PyTorch.

Hierarchical Volume Sampling

Earlier when I said that the radiance field is represented by a multilayer perceptron, I may have lied a little. In fact, it's represented by two multilayer perceptrons! One operates at a coarse level, encoding broad structural properties of the scene. The other refines the details at a fine level, thus enabling thin and complex structures like meshes and branches to be realized. In addition, the samples they receive are different, the coarse model processing broad, mostly regularly spaced samples throughout the ray, and the fine model honing in on areas with strong priors for salient information.



An example of stratified sampling (blue) and hierarchical sampling (red). While stratified sampling captures the whole ray length with even spacing (subject to noise), hierarchical volume sampling samples regions proportionally based on their expected contribution to the render. (Figure created by author.)

This "honing in" process is accomplished by their hierarchical volume sampling procedure. The 3D space is in fact very sparse with occlusions and so most points don't contribute much to the rendered image. It is therefore more beneficial to oversample regions with a high likelihood of contributing to the integral. They apply learned, normalized weights to the first set of samples to create a PDF across the ray. They then apply inverse transform

sampling to this PDF to gather a second set of samples. This set is combined with the first set and fed to the fine network to produce the final output.

```python
def sample_hierarchical(
  rays_o: torch.Tensor,
  rays_d: torch.Tensor,
  z_vals: torch.Tensor,
  weights: torch.Tensor,
  n_samples: int,
  perturb: bool = False
) -> Tuple[torch.Tensor, torch.Tensor, torch.Tensor]:
  r"""
  Apply hierarchical sampling to the rays.
  """

  # Draw samples from PDF using z_vals as bins and weights as probabilities.
  z_vals_mid = .5 * (z_vals[..., 1:] + z_vals[..., :-1])
  new_z_samples = sample_pdf(z_vals_mid, weights[..., 1:-1], n_samples,
                             perturb=perturb)
  new_z_samples = new_z_samples.detach()

  # Resample points from ray based on PDF.
  z_vals_combined, _ = torch.sort(torch.cat([z_vals, new_z_samples], dim=-1), dim=-1)
  pts = rays_o[..., None, :] + rays_d[..., None, :] * z_vals_combined[..., :, None]  # [
  return pts, z_vals_combined, new_z_samples


def sample_pdf(
  bins: torch.Tensor,
  weights: torch.Tensor,
  n_samples: int,
  perturb: bool = False
) -> torch.Tensor:
  r"""
  Apply inverse transform sampling to a weighted set of points.
  """

  # Normalize weights to get PDF.
  pdf = (weights + 1e-5) / torch.sum(weights + 1e-5, -1, keepdims=True) # [n_rays, weigh

  # Convert PDF to CDF.
  cdf = torch.cumsum(pdf, dim=-1) # [n_rays, weights.shape[-1]]
  cdf = torch.concat([torch.zeros_like(cdf[     :1]), cdf], dim=-1) # [n_rays, weights.s
```

```
    cdf = torch.concat([torch.zeros_like(cdf[..., :1]), cdf], dim=-1) # [n_rays, weights.s
41
42    # Take sample positions to grab from CDF. Linear when perturb == 0.
43    if not perturb:
44      u = torch.linspace(0., 1., n_samples, device=cdf.device)
45      u = u.expand(list(cdf.shape[:-1]) + [n_samples]) # [n_rays, n_samples]
46    else:
47      u = torch.rand(list(cdf.shape[:-1]) + [n_samples], device=cdf.device) # [n_rays, n_s
48
49    # Find indices along CDF where values in u would be placed.
50    u = u.contiguous() # Returns contiguous tensor with same values.
51    inds = torch.searchsorted(cdf, u, right=True) # [n_rays, n_samples]
52
53    # Clamp indices that are out of bounds.
54    below = torch.clamp(inds - 1, min=0)
55    above = torch.clamp(inds, max=cdf.shape[-1] - 1)
56    inds_g = torch.stack([below, above], dim=-1) # [n_rays, n_samples, 2]
57
58    # Sample from cdf and the corresponding bin centers.
59    matched_shape = list(inds_g.shape[:-1]) + [cdf.shape[-1]]
60    cdf_g = torch.gather(cdf.unsqueeze(-2).expand(matched_shape), dim=-1,
61                         index=inds_g)
62    bins_g = torch.gather(bins.unsqueeze(-2).expand(matched_shape), dim=-1,
63                          index=inds_g)
64
65    # Convert samples to ray length.
66    denom = (cdf_g[..., 1] - cdf_g[..., 0])
67    denom = torch.where(denom < 1e-5, torch.ones_like(denom), denom)
68    t = (u - cdf_g[..., 0]) / denom
69    samples = bins_g[..., 0] + t * (bins_g[..., 1] - bins_g[..., 0])
70
71    return samples # [n_rays, n_samples]
```

nerf_sample_hierarchical.py hosted with ❤️ by GitHub view raw

Hierarchical sampling in PyTorch.

Training

The standard approach to training NeRF from the paper is mostly what you would expect, with a few key differences. The recommended architecture of 8 layers per network and 256 dimensions per layer can consume a lot of memory during training. Their approach to alleviate this is to chunk the forward pass into smaller pieces and then accumulate the gradients across

these chunks. Note the distinction from minibatching; the gradient is accumulated across a single minibatch of sampled rays, which may have been gathered in chunks. If you do not have an NVIDIA V100 GPU as was used in the paper or something with comparable performance, you will probably have to adjust your chunk sizes accordingly to avoid OOM errors. The Colab notebook uses a much smaller architecture and more modest chunking sizes.

I personally found NeRFs somewhat tricky to train due to local minima, even with many of the defaults chosen. Some techniques that helped include center cropping during early training iterations and early restarts. Feel free to try different hyperparameters and techniques to further improve training convergence.

Conclusion

Radiance fields mark a dramatic change in the way machine learning practitioners approach 3D data. The NeRF model, and more broadly differentiable rendering, are quickly bridging the gap between creation of images and creation of volumetric scenes. While the components we walked through may seem very intricately assembled, countless other methods inspired by this "vanilla" NeRF prove that the basic concept (continuous function approximator + differentiable renderer) is a strong foundation upon which to build a variety of solutions tailored to nearly limitless situations. I encourage you to give it a try for yourself. Happy rendering!

References

[1] Ben Mildenhall, Pratul P. Srinivasan, Matthew Tancik, Jonathan T. Barron, Ravi Ramamoorthi, Ren Ng — NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis (2020), ECCV 2020

[2] Julian Chibane, Aayush Bansal, Verica Lazova, Gerard Pons-Moll — Stereo Radiance Fields (SRF): Learning View Synthesis for Sparse Views of Novel Scenes (2021), CVPR 2021

[3] Alex Yu, Vickie Ye, Matthew Tancik, Angjoo Kanazawa — pixelNeRF: Neural Radiance Fields from One or Few Images (2021), CVPR 2021

[4] Zhengqi Li, Simon Niklaus, Noah Snavely, Oliver Wang — Neural Scene Flow Fields for Space-Time View Synthesis of Dynamic Scenes (2021), CVPR 2021

[5] Albert Pumarola, Enric Corona, Gerard Pons-Moll, Francesc Moreno-Noguer — D-NeRF: Neural Radiance Fields for Dynamic Scenes (2021), CVPR 2021

[6] Michael Niemeyer, Andreas Geiger — GIRAFFE: Representing Scenes as Compositional Generative Neural Feature Fields (2021), CVPR 2021

[7] Zhengfei Kuang, Kyle Olszewski, Menglei Chai, Zeng Huang, Panos Achlioptas, Sergey Tulyakov — NeROIC: Neural Object Capture and Rendering from Online Image Collections, Computing Research Repository 2022

[8] Konstantinos Rematas, Andrew Liu, Pratul P. Srinivasan, Jonathan T. Barron ,Andrea Tagliasacchi, Tom Funkhouser, Vittorio Ferrari — Urban Radiance Fields (2022), CVPR 2022

[9] Matthew Tancik, Vincent Casser, Xinchen Yan, Sabeek Pradhan, Ben Mildenhall, Pratul P. Srinivasan, Jonathan T. Barron, Henrik Kretzschmar — Block-NeRF: Scalable Large Scene Neural View Synthesis (2022), arXiv 2022

[10] Alex Yu, Sara Fridovich-Keil, Matthew Tancik, Qinhong Chen, Benjamin Recht, Angjoo Kanazawa — Plenoxels: Radiance Fields without Neural Networks (2022), CVPR 2022 (Oral)

[11] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, Illia Polosukhin — Attention Is All You Need (2017), NeurIPS 2017

[12] Nasim Rahaman, Aristide Baratin, Devansh Arpit, Felix Draxler, Min Lin, Fred A. Hamprecht, Yoshua Bengio, Aaron Courville — On the Spectral Bias of Neural Networks (2019), Proceedings of the 36th International Conference on Machine Learning (PMLR) 2019

Machine Learning | Computer Vision | Pytorch | 3d Deep Learning | 3d Reconstruction

Written by Mason McGough

312 Followers · Writer for Towards Data Science

Machine learning engineer with a passion for photography and art

Follow

More from Mason McGough and Towards Data Science

Mason McGough in Towards Data Science

Stable Diffusion as an API

Remove people from photos with a Stable Diffusion microservice

12 min read · Feb 4

96

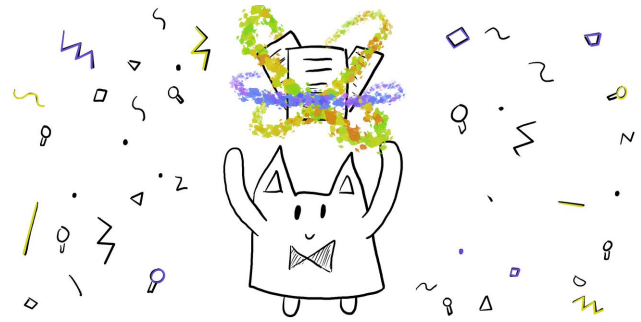
Marco Peixeiro in Towards Data Science

TimeGPT: The First Foundation Model for Time Series Forecasting

Explore the first generative pre-trained forecasting model and apply it in a project...

12 min read · Oct 24

2.1K 18

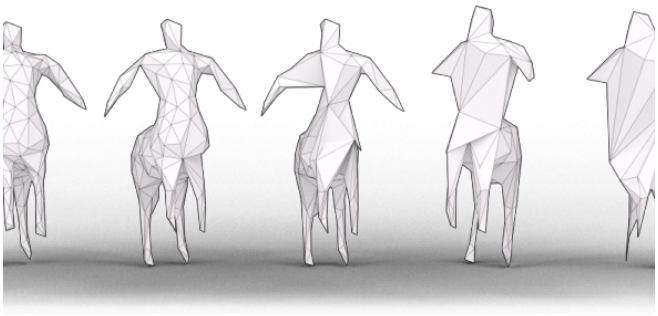


Adrian H. Raudaschl in Towards Data Science

Forget RAG, the Future is RAG-Fusion

The Next Frontier of Search: Retrieval Augmented Generation meets Reciprocal...

10 min read · Oct 6



Mason McGough in Towards Data Science

3D Object Classification and Segmentation with MeshCNN and...

MeshCNN introduces the mesh pooling operation, which enables us to apply CNNs t...

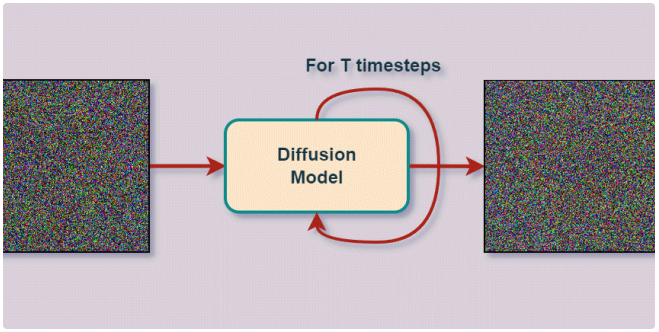
9 min read · Jun 2, 2021

2.4K 24 100 1

See all from Mason McGough

See all from Towards Data Science

Recommended from Medium



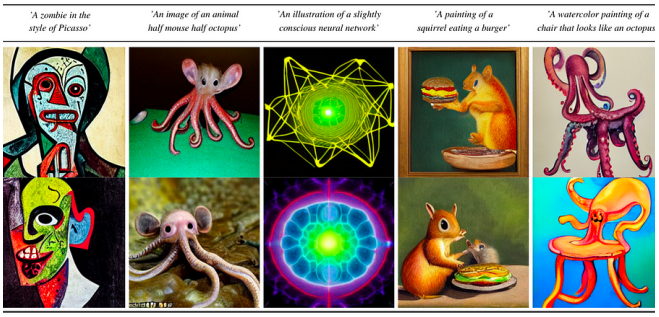
Gabriel Mongaras in Better Programming

Diffusion Models—DDPMs, DDIMs, and Classifier Free Guidance

A guide to the evolution of diffusion models from DDPMs to Classifier Free guidance

28 min read · Mar 13

466 6



Onkar Mishra

Stable Diffusion Explained

How does Stable diffusion work? Explaining the tech behind text to image generation.

6 min read · Jun 8

393 3

Lists

David Cochard in axinc-ai

3D Gaussian Splatting : Real-Time Rendering of Photorealistic Scenes

Overview

6 min read · Oct 2

33

Shaig Hamzaliyev

Voxelization with Python (numpy & scipy)

This article shows how to voxelize point cloud data using only numpy and scipy to have...

7 min read · Aug 5

15

Steins

Abhijat Sarari in Python in Plain English

Stable Diffusion — ControlNet Clearly Explained!

Generating images from line art, scribble, or pose key points using Stable Diffusion and...

· 6 min read · Jun 6

433 3

3D Reconstruction from 2D Images Using Python

3D reconstruction from 2D images is the process of creating a 3D model of an object ...

· 11 min read · Sep 17

65 3

See more recommendations