NeRF Core Concepts

Paper → https://arxiv.org/pdf/2003.08934.pdf

What is NeRF (neural radiance field)?

A neural radiance field (NeRF) is **a fully-connected neural network that can generate novel views of complex 3D scenes, based on a partial set of 2D images**. It is trained to use a rendering loss to reproduce input views of a scene.

It is used for view synthesis. View synthesis is opposite of volume rendering.

```
View synthesis → 2D to 3D
Volume rendering → 3D to 2D
```

One neural network is fit to one scene.

If we want to render a new scene, we take a new neural network.

We overfit one neural network to this particular scene.

Input is the **images** and output is a **scene**.

We train the network with a bunch of images and the goal is that the scene is going to be in the **weights** of the neural network.

> 💡 The weights of the neural network will represent the scene.

This has various advantages. This representation is very compact compared to if stored in voxels or something.

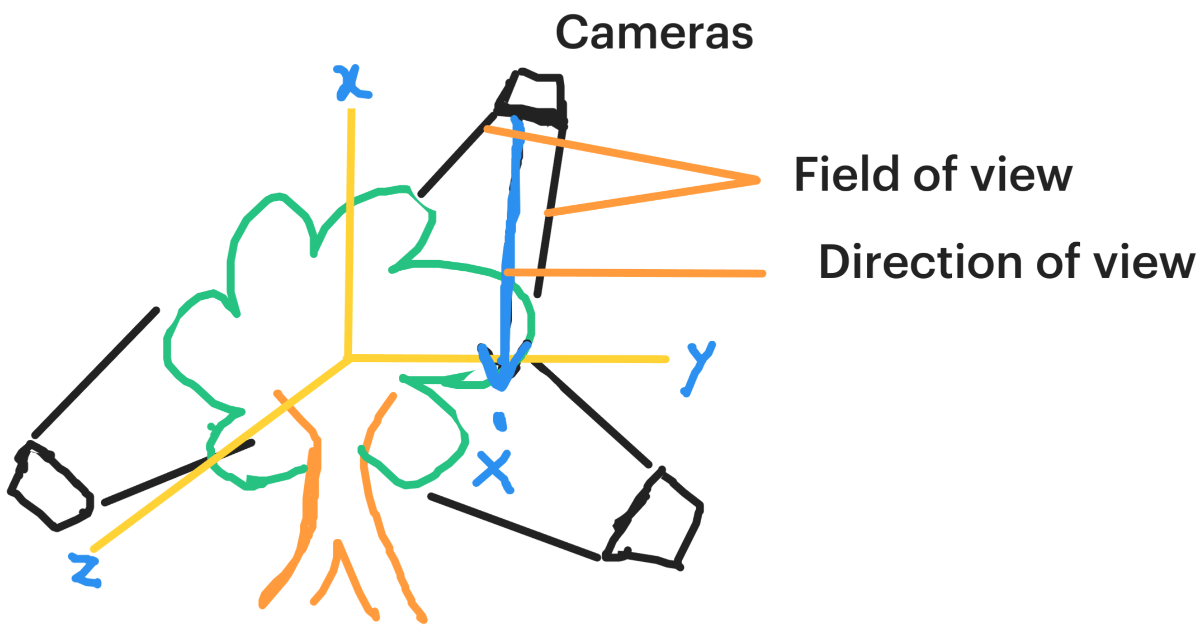
Now the question is what's the input and outputs of the network?

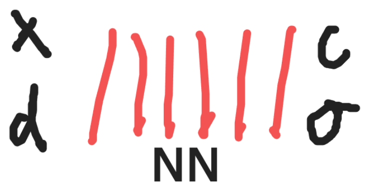
The neural network gets two things as input.

It gets as input a `position` in that coordinate system which we call **'x'** and also it gets a `viewwing direction` **'d'.**

The output of the neural network is going to be a `color` **'c'** i.e. what color is at that particular location, and a `density` **'σ'** i.e. is there even something at that particular location.

So the density tells you if there is something or not, and if there is something, the color tells you what is the color of that thing.



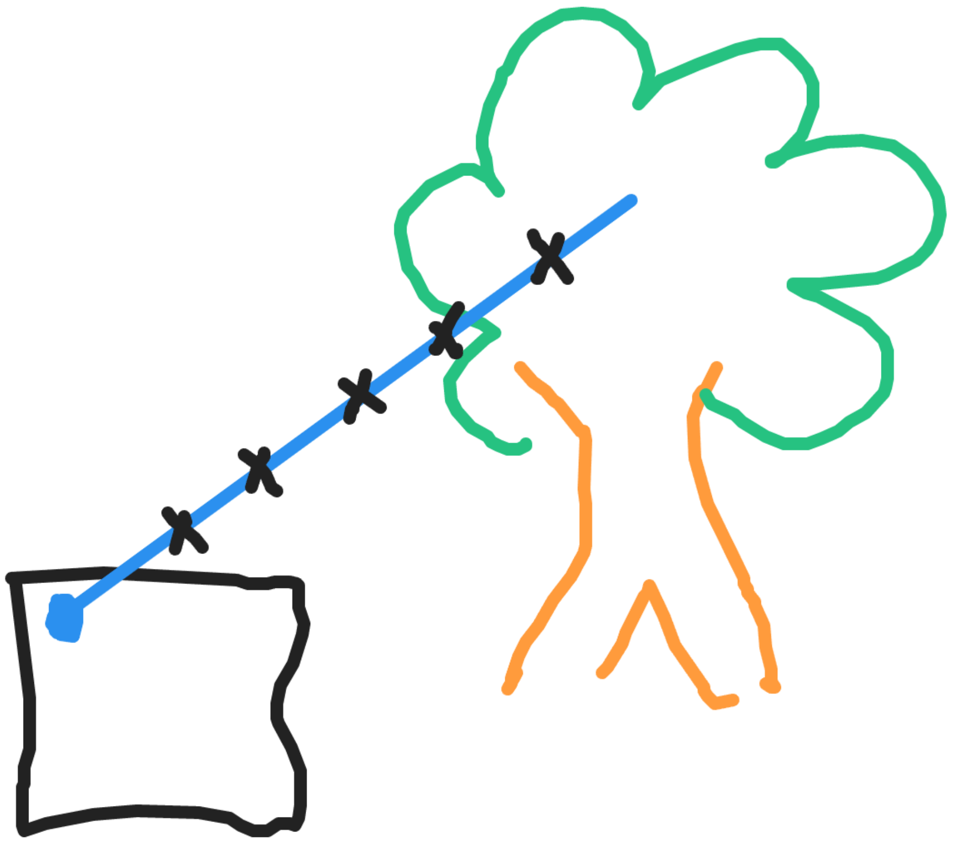


You ask the neural network “Hey neural network, you in your entirety represent this scene. I want to know at a particular location in this scene, viewed from a particular angle, what am I going to see?”

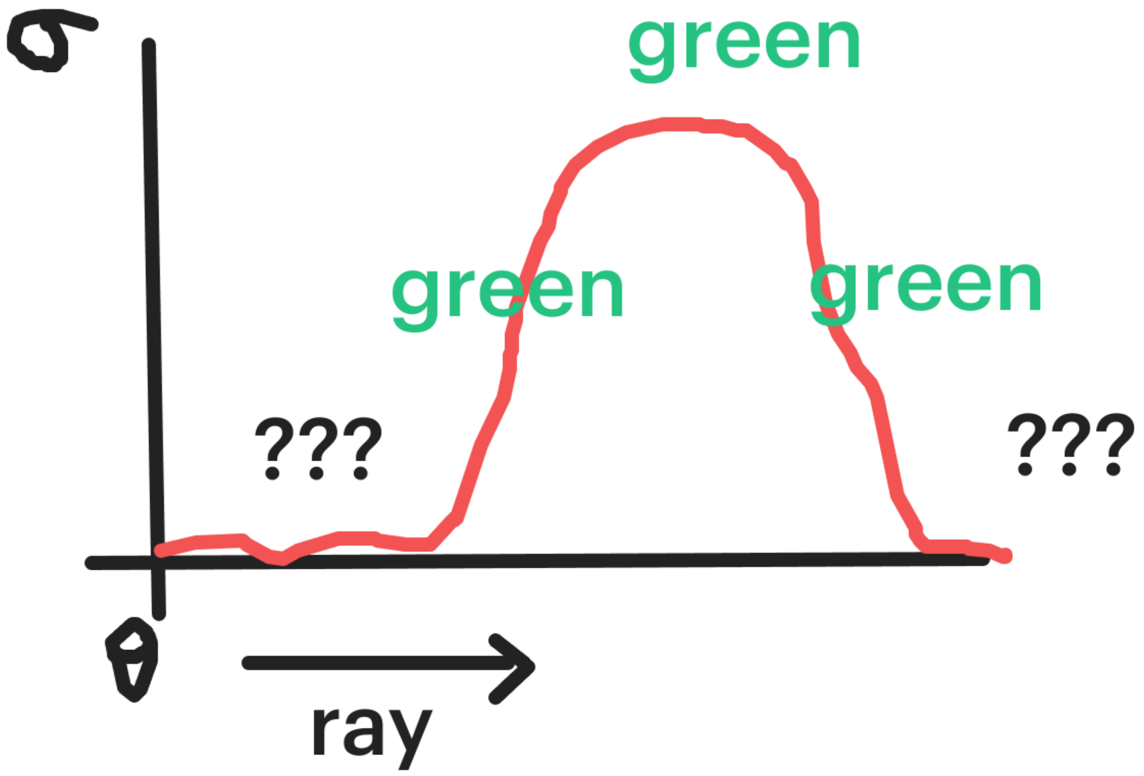
If we send a **ray** to a particular location, what are we going to see?

If there is something, you see a color.

If we gave a frame of the picture, for each pixel we need to send a ray through the scene, and we query the model at each location on the ray. At each location, we ask the neural network **if something is there or not,** and if there is something, what are we going to see (color).

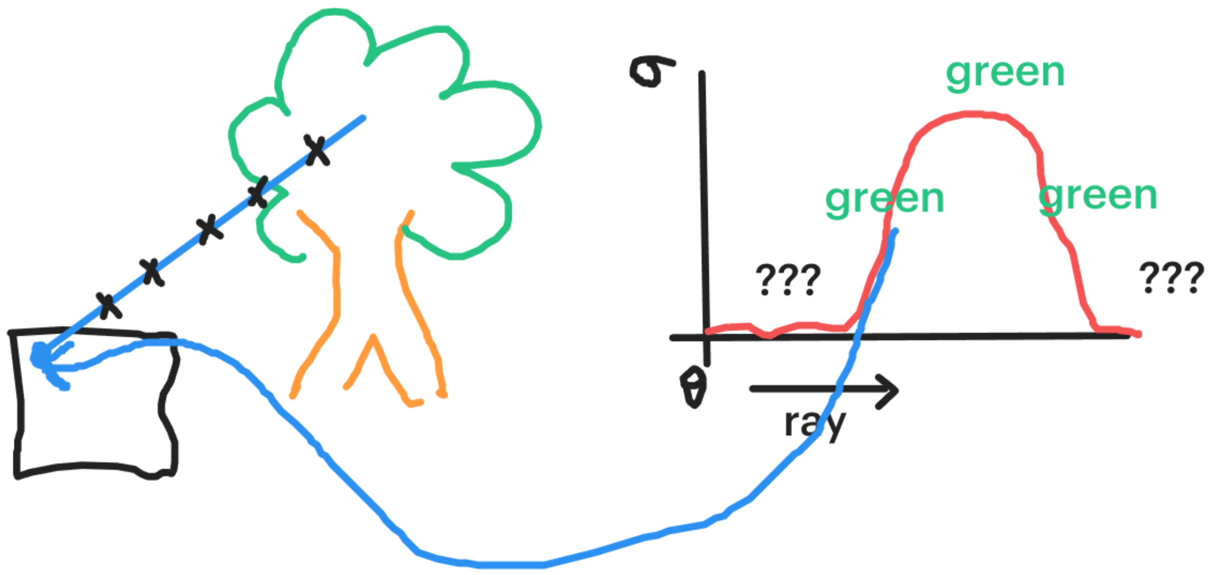


What we get is a bit of a curve.



We send a ray out to the scene. At the beginning maybe we are not going to see anything and the `density` is zero. Then at some point we see something (maybe a tree). We are inside the tree for a while and then outside again. At the same time at every point it also gives us the `color`.

What we can do now is look where we hit the object for the first time and what color is there. Now we know what we need to render at that particular pixel.



Now you can simply do this for all pixels and you got yourself an image.

The neural network is powerful enough that for the same location it can give you different results depending on the different viewing directions. It can capture these **lighting effects** and **reflections**. Also, it can capture **transparency.**

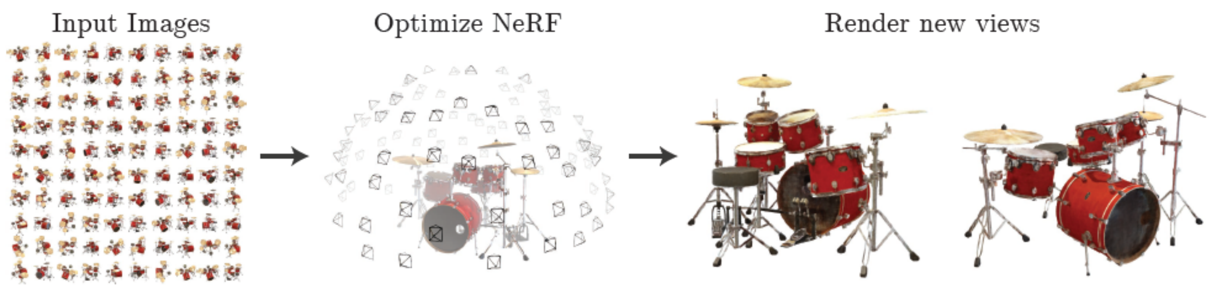
This is a really powerful model. Again, there is no need for a dataset other than the scene that is right in front of you.

So the goal is going to be that if in the future we want to make AR applications or games and so on, you are not actually going to store a mesh or a voxel grid of some scene. What you are going to store is a neural nwtwork, that can be queried from anywhere you want to look at the scene and the neural network will tell you what you are going to see.

So here is the process again.

- You get a set of input images

- You want to find out where they are taken from
    - So for each input image you need to determine where was the camera and which direction did it look
        - This is a known problem with a lot of existing research.
- Then you render the new views.



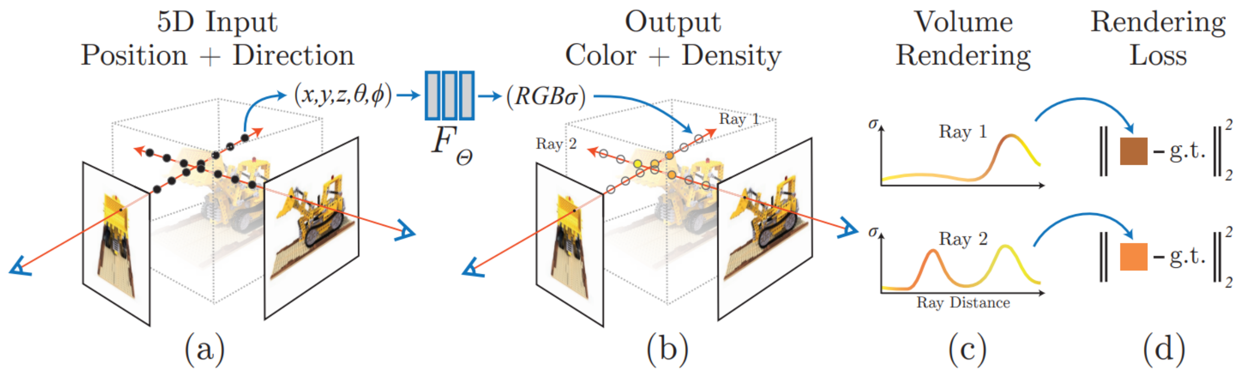
Input is a 3D location `x = (x, y, z)` and 2D viewing direction `(θ, φ)`. The output is a color `c = (r, g, b)` and volume density `σ`.

The authors approximate the continuous 5D scene representation with an **MLP** network.

Now the only question is that we have these images. We don't actually have as a training set kind of the densities at those places. So everything needs to be grounded on the images that we have.

If you want to render an image, you takes an image, pick a pixel and shoot an ray. You sample along the ray and you ask the network what is there? The network will tell you if there is something there, if so then what color. You are going to see the density over time and then you can render an image. You can now calculate a **loss**. What I see and what the network tells me I should see. If the network us not trained yet, that is going to be a pretty big loss. If you make the loss something differentiable, then this whole process is differentiable.

That means we can sue those 30-50 images and construct a pretty big loss. Every pixel in every picture we have defines a ray. So ever y ray essentially is a data point that we can fit to. So at the end we get a pretty sizable dataset for the network which is going to be `number of pixels * number of pictures`.



To render this **NeRF,** there are 3 steps:

1. march camera rays through the scene to sample 3D points

2. use points from step1 and their corresponding 2D viewing directions(θ, φ) as input to the MLP to produce an output set of colors (c = (r, g, b)) and densities σ,

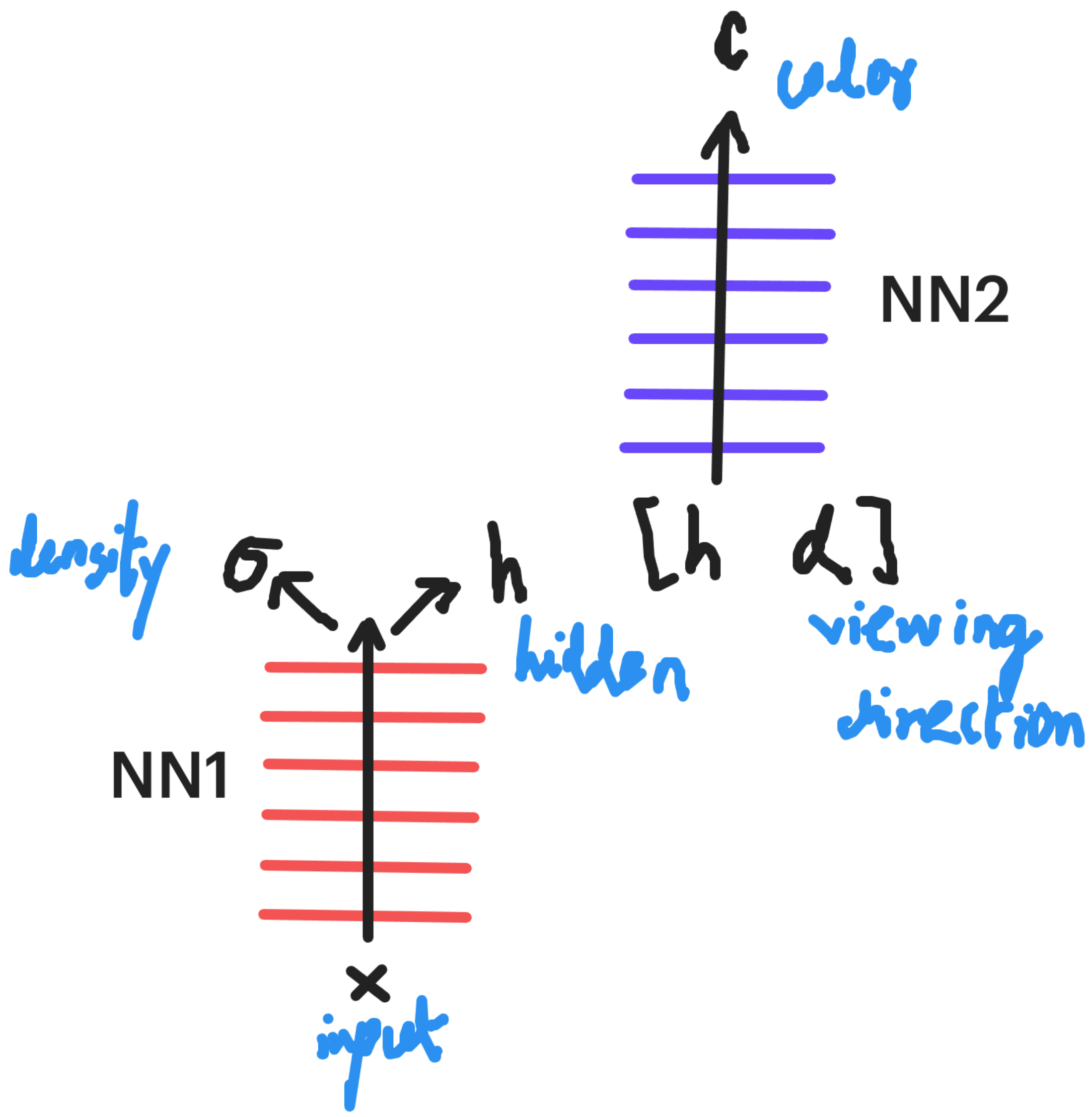
3. use volume rendering techniques to accumulate those colors and densities into a 2D image [Note: Volume rendering refers to creating a 2D projection from sampled 3D points]

The network's weights are going to represent the scene at the end.

There is also some engineering involved.

> We encourage the representation to be multiview consistent by restricting the network to predict the volume density σ as a function of only the location x, while allowing the RGB color c to be

> predicted as a function of both location and viewing direction. To accomplish this, the MLP FΘ first processes the input 3D coordinate x with 8 fully-connected layers (using ReLU activations and 256 channels per layer), and outputs σ and a 256-dimensional feature vector. This feature vector is then concatenated with the camera ray’s viewing direction and passed to one additional fully-connected layer (using a ReLU activation and 128 channels) that output the view-dependent RGB color.



This is the formula for rendering:

$$C(\mathbf{r}) = \int_{t_n}^{t_f} T(t)\sigma(\mathbf{r}(t))\mathbf{c}(\mathbf{r}(t), \mathbf{d})dt\,, \text{ where } T(t) = \exp\left(-\int_{t_n}^{t} \sigma(\mathbf{r}(s))ds\right). \quad (1)$$

This a technique called volume rendering with radiance fields. You send a ray through the scene and integrate along that ray. You have a far and a near bound, and you want to integrate from the near bound to the far bound.

```
T(t):
Probability that the ray does nt hit anything, Probablity of empty space,
Probability that the ray goes on, where the ray continues up until the point
't' or not.

σ:
How that dense that particular point is.

c:
Color at that particular place.
```

These use bunch of tricks and the most important one is **positional encoding**.

It is not that same as in **transformers**. The positional encoding here it simply means that you send the input data point `xyzθφ` to a higher dimensional space.

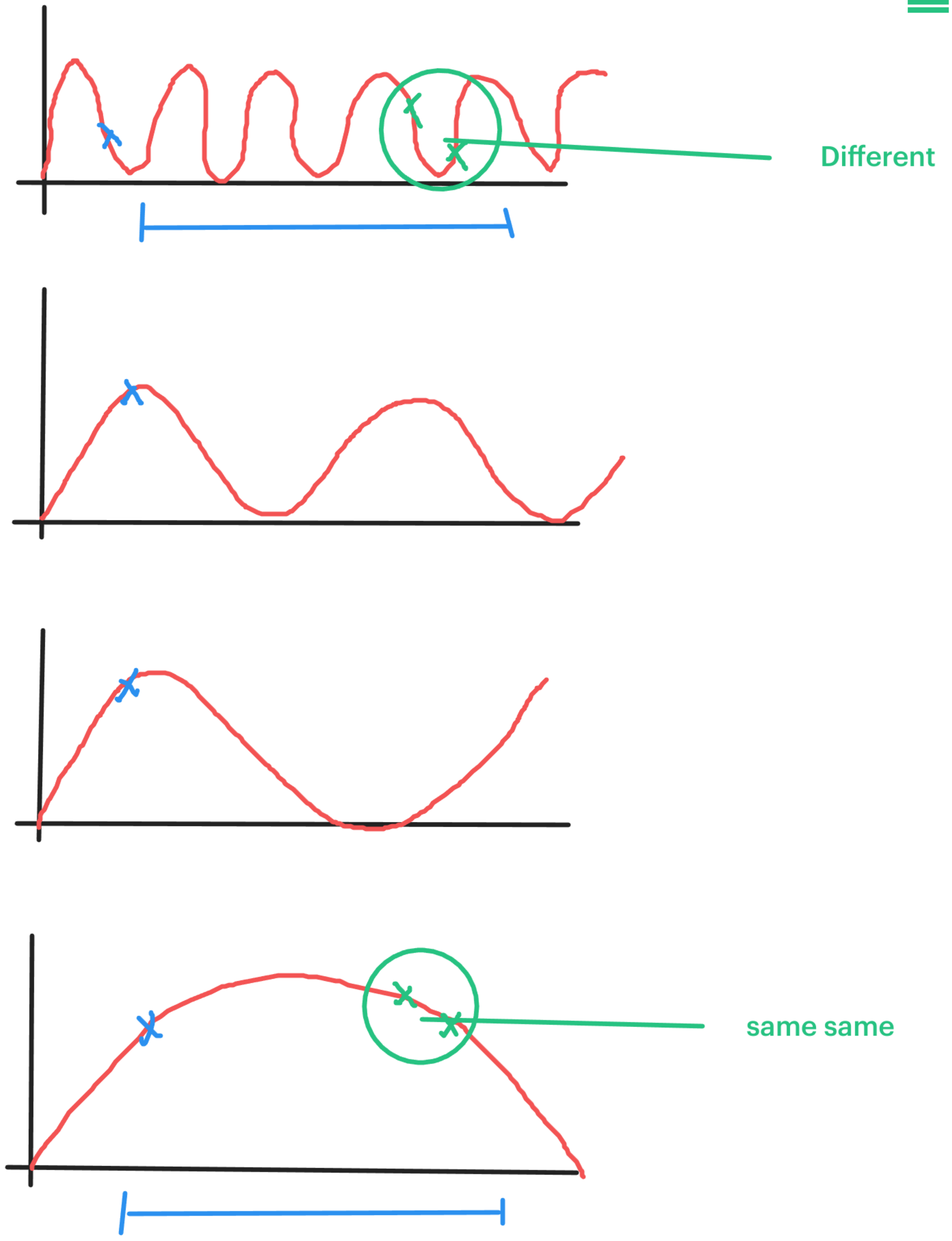
> A similar mapping is used in the popular Transformer architecture [47], where it is referred to as a positional encoding. However, Transformers use it for a different goal of providing the discrete positions of tokens in a sequence as input to an architecture that does not contain any notion of order. In contrast, we use these functions to map continuous input coordinates into a higher dimensional space to enable our MLP to more easily approximate a higher frequency function.

> Despite the fact that neural networks are universal function approximators [14], we found that having the network FΘ directly operate on xyzθφ input coordinates results in renderings that

> perform poorly at representing high-frequency variation in color and geometry

> 💡 Mapping the inputs to a higher dimensional space using high frequency functions before passing them to the network enables better fitting of data that contains high frequency variation.

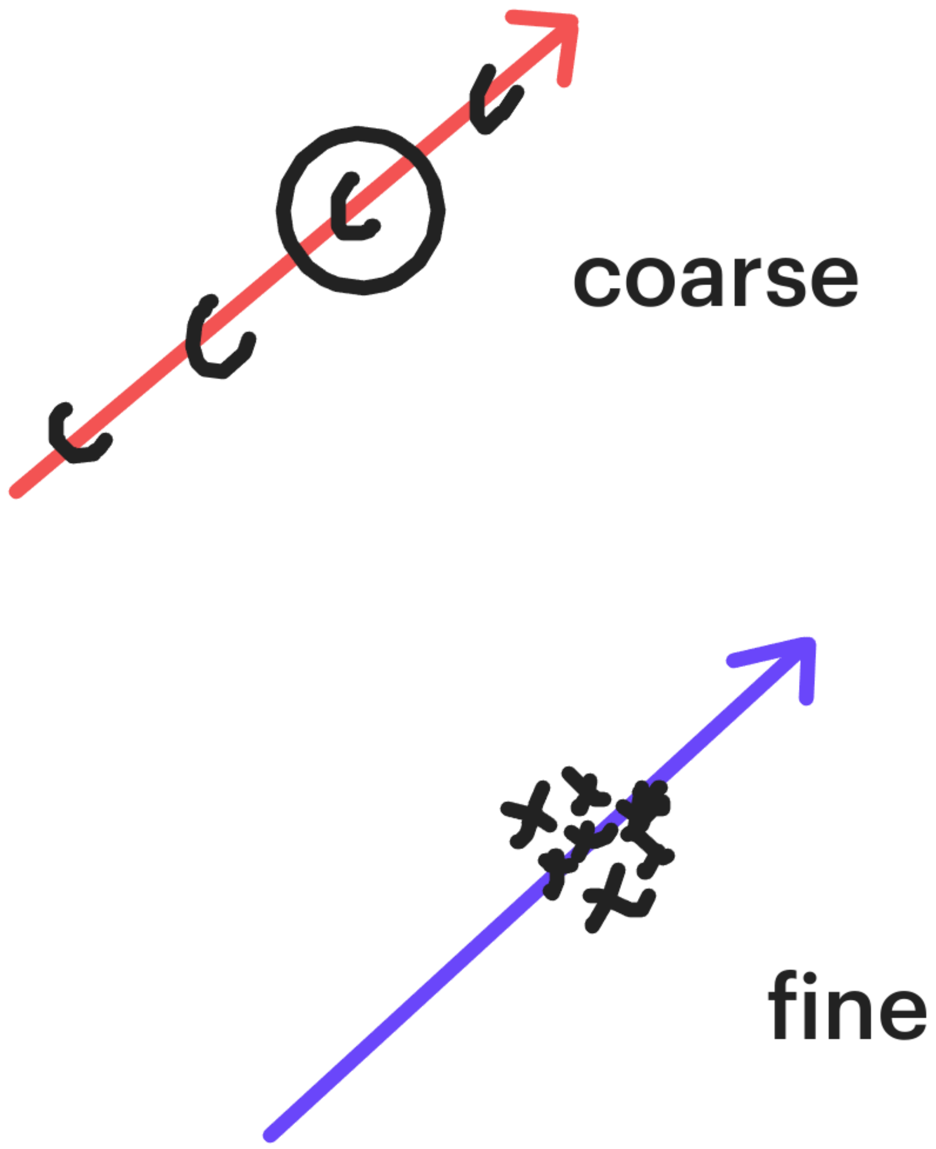
You construct a hierarchy of sine/cosine waves. It gives the network a better chance to focus on the details. In higher wave, tiny changes in x is going to result in long change in the features. In lower coarse wave, the change is really slow. It gives the network choice at which scale it wants to look at for particular locations.



In paper experiments, they consider 10 or 4 layers of these sine/cosine waves.

The second thing they do is **hierarchical volume sampling**.

If you send a ray through the scene and sample along, this would either take a lot of time or it would not be accurate enough. Authors actually use two layers of neural networks `coarse` and `fine`. The first sample with coarse one at coarse locations, and then they use that to evaluate where they should sample more. They optimize both networks at the same time and it actually works out well.



Results show that it can handle easy to fine-grained structures fairly well. The size of network is also really small (5 MB).

The training/fitting time however is long.