CIS 580, Machine Perception, Spring 2023
Homework 5
Due: Thursday April 27th 2023, 11:59pm ET

1 Part 1: Fitting a 2D image (25 points)

Let's consider a color image to be a mapping $I : \mathbb{R}^2 \rightarrow \mathbb{R}^3$, where the input is the pixel coordinates (x, y) and the output is the RGB channels. We define a Multilayer Perceptron (MLP) network F_Ω that will learn this mapping I. We say we fit the network F to the image I.

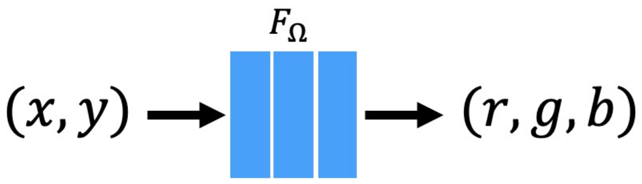

Figure 1: A network F with learnable parameters Ω learns a mapping from 2D pixel coordinates to RGB color.

1.1 Positional Encoding (5 points)

Positional encoding is used to map continuous input coordinates into a higher dimensional space to enable a neural network to approximate a higher frequency function. Despite the fact that neural networks are universal function approximators, it has been shown that having a network directly operate on input coordinates (x, y) results in renderings that perform poorly at representing high-frequency variation in color and texture. Mapping the inputs to a higher dimensional space using high-frequency functions before passing them to the network enables better approximation of high-frequency variation.

In this homework, we will use the sinusoidal periodic function for positional encoding. Note that other periodic functions can be used to map your data into higher dimensions or non-periodic functions. Let $\mathbf{x} \in \mathbb{R}^D$ be a D-dimensional vector. Then we define the positional encoding of $\mathbf{x}$ as:

$$\gamma(x) = \begin{bmatrix} sin(2^0 \pi x) \\ cos(2^0 \pi x) \\ \vdots \\ sin(2^{L-1} \pi x) \\ cos(2^{L-1} \pi x) \end{bmatrix}$$

and thus the positional encoding γ is a mapping from $\mathbb{R}^D \rightarrow \mathbb{R}^{2DL}$, where L is fixed and chosen. Note for example, that for a $2D$-dimensional input $\mathbf{x} = (x_1, x_2)$, $sin(\mathbf{x}) = (sin(x_1), sin(x_2))$.

For this part, complete the function `positional_encoding()` that takes a $[N, D]$ vector as input, where N is the batch size and D the feature dimension, and returns the positional encoding mapping of the input.

1.2 MLP Design (5 points)

The architecture of your MLP should be the following:

- It should consist of three linear layers. The first one will map the feature dimensions (after the positional encoding) to the `filter_size` dimension. The second one should keep the feature space dimension the same. The final linear layer should map the feature space dimension to the output dimension of the MLP.
- The first two linear layers should be followed by a ReLu activation while the last linear layer should be followed by a Sigmoid activation function.

For this part, complete the class `MLP()` and specifically the functions `__init()__` and `forward()` that define the neural network that will be used to fit the 2D image.

1.3 Fitting the image (15 points)

For this part, complete the function `train_2d_model()`. We define the learning rate and number of iterations for you. For the optimizer we will use the Adam optimizer and for the loss function, the mean square error. You should transform the image to a vector of 2D points and then apply the positional encoding to them. The pixel coordinates should be normalized between $[0, 1]$. Train the model by fitting the points to the MLP, transforming the output back to an image, and computing the loss between the original and reconstructed image. Finally, calculate the PSNR between the images which is given by:

$$PSNR = 10 \cdot log_{10}\left(\frac{R^2}{MSE}\right)$$

where R is the maximum valid value of a pixel and MSE is the mean squared error between the images. The PSNR computes the peak signal-to-noise ratio, in decibels, between two images and is used as a quality measurement between the original and the reconstructed image.

After completing the function, train the model to fit the given image without applying positional encoding to the input, and by applying positional encoding of two different frequencies to the input; $L = 2$ and $L = 6$. What's the effect of positional encoding and the effect of different numbers of frequencies? In order to pass the autograder for this one, you need a PSNR higher than $15.5, 16.2$, and 26, for the three cases accordingly, after $10,000$ iterations.

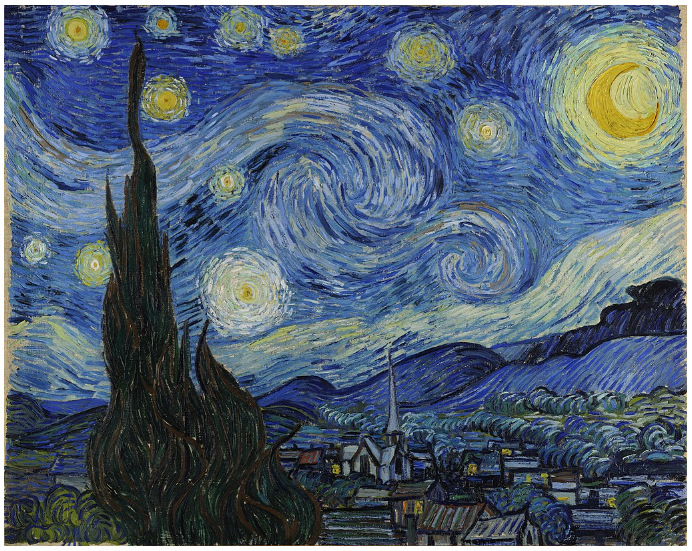

Figure 2: Starry Night by Van Gogh; the image we try to fit the model to.

2 Part 2: Fitting a 3D scene (75 points)

In this part of the homework, our goal is to represent a 3D scene in a convenient and compact way, to be later used for rendering 2D images. A 3D scene can be considered a set of points that contain color and density. Though every fixed point in the scene has a fixed density, that's not the case with the color. Each point can have a different color depending on the viewing direction if we assume the surface to be non-Lambertian.

Thus a 3D scene is representing by the mapping $(x, y, z, \theta, \phi) = (\mathbf{x}, \mathbf{d}) \rightarrow (\mathbf{c}, \sigma)$ where $\mathbf{x} = (x, y, z)$ is the 3D position of a point in the scene, $\mathbf{d} = (\theta, \phi)$ is the viewing direction, $\mathbf{c} = (r, g, b)$ is the color of the point and σ is its density. The problem we will try to solve here, and the one that the NeRF paper approached, is that given a number of 2D views of the same static scene, to be able to render novel views of that scene.

2.1 Computing the images' rays (10 points)

For every 2D image I, we are given the transformation between the camera and the world coordinates, along with the intrinsic parameters of the camera. We need to calculate the origins and the directions of each camera frame with respect to the world coordinate frame.

For this part, you need to complete the function `get_rays()` that returns the origins and the directions of $H \times W$ rays of an image. Note that all the origins should be the same for each ray, while the directions should slightly differ depending on the pixel the ray is passing through.

Finally, you should complete the function `plot_all_poses()` which plots the vectors (origin and direction) from all the frames used to capture the 3D scene. This will offer a good visualization of the whole setup of the data. The function should calculate two vectors that contain all the origins from each image and all the directions that pass through each image's center (u_0, v_0). The plotting part is implemented for you.

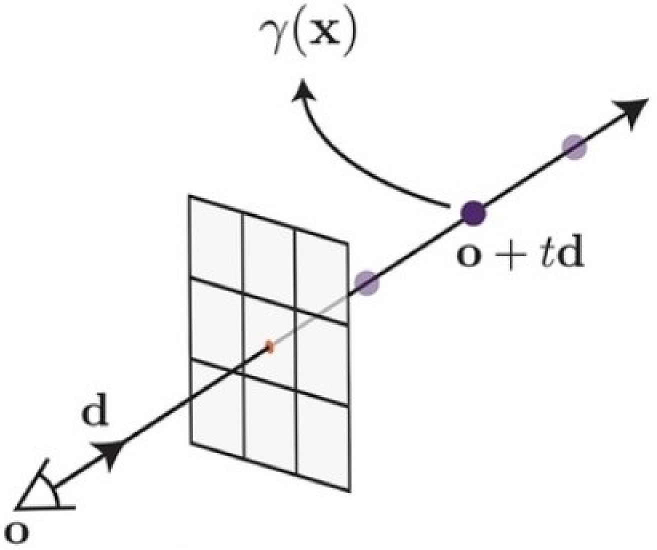

Figure 3: A ray cast through an image plane with origin $\mathbf{o}$ and direction $\mathbf{d}$. Sampled points along the ray follow the equation $\mathbf{r} = \mathbf{o} + t \cdot \mathbf{d}$, for $t \in \mathbb{R}$. The sampled points shall be mapped to a higher dimension with the use of positional encoding.

2.2 Sampling points along a ray (10 points)

Now that we have all rays calculated, we need to be able to sample a number of points along a ray. Recall that for a ray with an origin $\mathbf{o}$ and direction $\mathbf{d}$, all the points that belong on the ray are given by the equation: $\mathbf{r} = \mathbf{o} + t \cdot \mathbf{d}, \forall t \in \mathbb{R}$.

For this part, you need to complete the function `stratified_sampling()` where you need to sample a number of t_i, $i = 1, \ldots, N$ between t_{near} and t_{far}. The points should be chosen evenly along this space, which means that:

$$t_i = t_{near} + \frac{i-1}{N}\left(t_{far} - t_{near}\right), i = 1, \cdots, N$$

The function should return the (x, y, z) position of each point sampled from each ray of the given image, along with the depth points (t_i) of each ray.

2.3 NeRF MLP Design (20 points)

The neural network architecture of the NeRF paper is rather simple for the task it approaches to solve. The input of the neural network is the position (x, y, z) of the sample points along the ray, and the direction of these points (θ, ϕ), after applying positional encoding to both of them. The figure below depicts the architecture in more detail.

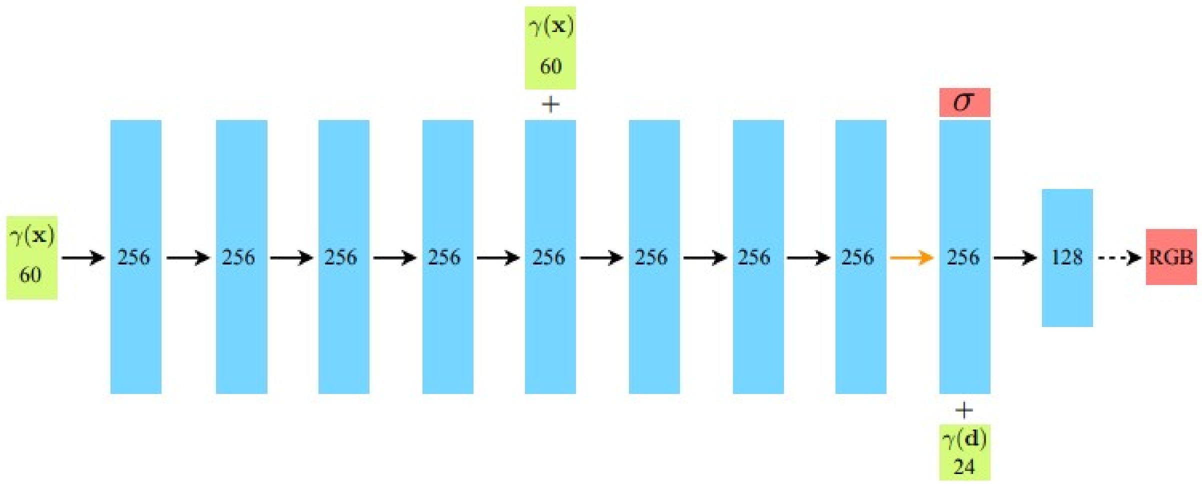

Figure 4: The architecture of the MLP that is used to learn a mapping between the position and direction of a point along a ray with its color and density.

The network is comprised of 10 fully-connected layers, followed by ReLU activation (black arrows), no activation (orange arrow), or sigmoid activation (dashed black arrow). The number inside the fully-connected layer depicts the number of channels. A skip connection is included that concatenates the input to the fifth layer's activation. After the eight layer with no activation function, there is a layer that outputs the volume density σ, and a layer that produces a 256-dimensional feature vector. This feature vector is concatenated with the positional encoding of the input viewing direction ($\gamma(\mathbf{d})$) and is processed by an additional fully-connected ReLU layer with 128 channels. A final layer (with a sigmoid activation) outputs the emitted RGB radiance at position $\mathbf{x}$, as viewed by a ray with direction $\mathbf{d}$. Note that the network predicts the density of a point before it is given the direction of that point, to enforce that the density is view-invariant. On the other hand, the color needs both the position and view direction of the point to model plausible non-Lambertian properties of the 3D scene. Note as well that we will be using 10 and 4 as the number of frequencies for the positional encoding of the position and direction input, instead of 60 and 24.

For this part, complete the class `nerf_model()` and specifically the functions `__init()__` and `forward()` that define the neural network that will be used to fit the 3D scene. Except for this function, you need to complete the function `get_batch()` as well, which prepares the data for the neural network. Specifically, this function takes as input the ray points and directions with $[H, W, n_{samples}, 3]$ and $[H, W, 3]$ dimensions respectively. The function should normalize the directions, populate them along each ray (repeat the direction of the ray to every point), flatten the vector, apply positional encoding to it, and then call the helper function `get_chunks()`. Similarly, the function should flatten the vector of the ray positions and then call `get_chunks()`.

2.4 Volume Rendering: Computing the color of a pixel (15 points)

The quintessence of the NeRF paper is the volumetric rendering formula. According to the volumetric rendering equation, if we are given the density and color of an adequate number of samples along a ray, we can approximate the color of that ray and thus the pixel color perceived by the camera with the direction of that ray. We remind the integral that calculates the ray color:

$$C(r) = \int_{t_{near}}^{t_{far}} T(t)\,\sigma(\mathbf{r}(t))\,\mathbf{c}\left(\mathbf{r}(t), \mathbf{d}\right) dt$$

where

$$T(t) = exp\left(-\int_{t_{near}}^{t} \sigma\left(\mathbf{r}(s)\right) ds\right)$$

denotes the accumulated transmittance along the ray from t_{near} to t, or the probability that the ray does not hit any other particle from t_{near} to t. Given N number of sample points along a ray, we can numerically and differentiable estimate the continuous integral as follows:

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

where

$$T_i = \exp\left(-\sum_{j=1}^{i-1} \sigma_j \delta_j\right)$$

and $\delta_i = t_{i+1} - t_i$ is the distance between adjacent sampled depth values t_i. For $i = N$ you should let $\delta_N = 10^9$. Note that for the computation of the T_i's, if your implementation is not vectorized, it will have a complexity of $\mathcal{O}(HWN^2)$ which is of quartic degree, and it will not be able to pass the autograder (Hint: use the function `cumprod()` from the PyTorch library). You should also pass σ from a ReLU activation function before multiplying it with the δ_i's, to avoid infinity values.

For this part, complete the function `volumetric_rendering()` that takes as input the sampled depth values of the rays of an image, along with the outputs of the MLP (density and RGB color of all points) and calculates the final color of each ray, or equally of each pixel of the image.

2.5 Render an image (10 points)

For this part you should complete the function `one_forward_pass()` that used all the functions implemented earlier. Given one image from the dataset, the function should first calculate all the rays - origins, and directions - of the image (Section 2.1) and then sample a number of points from all these rays (Section 2.2). Next, to avoid memory errors, you should divide all the points into chunks, and forward pass them through the neural network separately (Section 2.3). After concatenating the separate outputs of the MLP, the function should apply the volumetric equation to produce the reconstructed image (Section 2.4) and which shall be the output of the function as well.

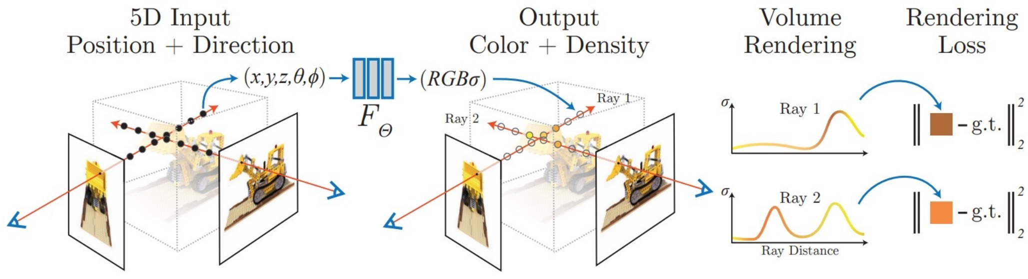

Figure 5: The full pipeline of the NeRF method, from sampling points to rendering the color of the rays.

2.6 Putting everything together (10 points)

For this part, complete the remaining commands in the training iteration loop to be able to train the NeRF model. All the hyperparameters (learning rate, number of samples, near and far ray distance, and so on) are defined for you. For optimizing we will use the Adam optimizer and for the loss function, the mean square error, similarly to Section 1.3. For every iteration of the model, you should choose a random image from the training set and use it for the forward pass of the model. Then, you should calculate the error between the original image and the reconstructed one, back-propagate the loss and step the optimizer to train the model.

You should expect to reach a PSNR of 25 after 3000 iterations. Note that the original paper reaches a PSNR of 40, using further engineering tricks that we do not employ here. The view we try to reconstruct is shown in the Figure below, though you should expect a fuzzier one. Upload the reconstructed image, the original one, and the plot of the PSNR during the training, for the one image that managed to reach a PSNR of 25.

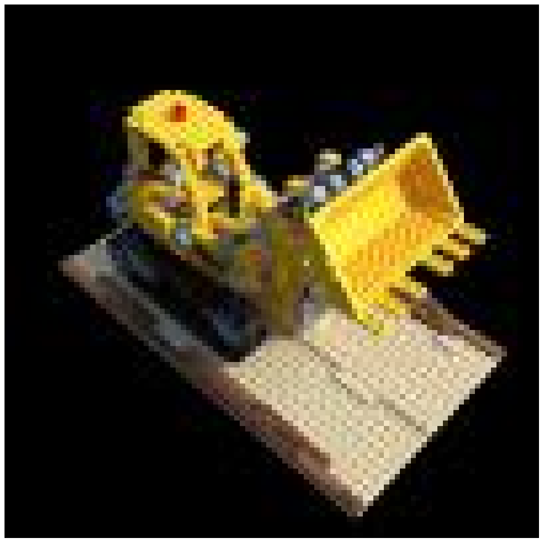

Figure 6: Lego Toy 3D Scene; the novel view we try to render.