

**IMAGE-BASED RELIGHTING USING IMPLICIT NEURAL
REPRESENTATION**

An Undergraduate Research Scholars Thesis

by

SHUYU WANG

Submitted to the LAUNCH: Undergraduate Research office at
Texas A&M University
in partial fulfillment of requirements for the designation as an

UNDERGRADUATE RESEARCH SCHOLAR

Approved by
Faculty Research Advisor:

Dr. Nima Kalantari

May 2022

Major:

Computer Science

Copyright © 2022. Shuyu Wang.

RESEARCH COMPLIANCE CERTIFICATION

Research activities involving the use of human subjects, vertebrate animals, and/or biohazards must be reviewed and approved by the appropriate Texas A&M University regulatory research committee (i.e., IRB, IACUC, IBC) before the activity can commence. This requirement applies to activities conducted at Texas A&M and to activities conducted at non-Texas A&M facilities or institutions. In both cases, students are responsible for working with the relevant Texas A&M research compliance program to ensure and document that all Texas A&M compliance obligations are met before the study begins.

I, Shuyu Wang, certify that all research compliance requirements related to this Undergraduate Research Scholars thesis have been addressed with my Research Faculty Advisor prior to the collection of any data used in this final thesis submission.

This project did not require approval from the Texas A&M University Research Compliance & Biosafety office.

TABLE OF CONTENTS

	Page
ABSTRACT.....	1
ACKNOWLEDGEMENTS.....	3
NOMENCLATURE.....	4
SECTIONS	
1. INTRODUCTION.....	5
1.1 Research Problem.....	5
1.2 Related Work.....	6
1.3 Proposed Approach.....	7
2. METHODS.....	9
2.1 Setting up the SIREN Network.....	12
2.2 Training of the SIREN Network.....	13
2.3 Testing the SIREN Network.....	17
3. IMPLEMENTATION.....	21
3.1 Converting Light Positions and Choosing Input Images.....	21
3.2 SIREN Network.....	24
4. RESULTS.....	33
5. CONCLUSION.....	39
REFERENCES.....	41

ABSTRACT

Image-based Relighting Using Implicit Neural Representation

Shuyu Wang
Department of Computer Science & Engineering
Texas A&M University

Research Faculty Advisor: Dr. Nima Kalantari
Department of Computer Science & Engineering
Texas A&M University

Rendering a scene under novel lighting has been a problem in all fields that require computer graphics knowledge, and Image-based relighting is one of the best ways to reconstruct the scene correctly.

Current research on Image-based relighting uses discrete convolutional neural networks, which tend to be less fit-able to different spatial resolutions and take up massive memory spaces. However, the implicit neural representation solves the problem by mapping the coordinates of the image directly to the value of the coordinate with a continuous function modeled through the neural network. In this way, despite the changing of the image resolution, the parameters taken in by the neural network stay the same, so the complexity stays the same.

Also, the rectified linear activation unit (ReLU) based network used in current research lacks the representation of information of second and higher derivatives. On the other hand, the sinusoidal representation networks (SIREN) provide a new way to solve this problem by using periodic activation functions like the sin curve. Hence, my research intends to leverage implicit neural representation with periodic activation functions in image-based relighting.

To tackle the research question, we proposed to base our image-relighting network on the SIREN network in the research by Sitzmann. Our method is to modify the SIREN network so that it takes in not only coordinates but also light positions. Then we train it with a set of input images depicting the same set of sparse objects in different lighting conditions and their corresponding light positions, as in previous image-based relighting research. We test our network by giving the network new lighting positions, and the result we aim for is to acquire a good representation of optimal sparse samples under novel lighting with high-frequency details.

Eventually, we run the training and test with several different input sets and acquire their results. We also compare and evaluate the results, in order to find the advantage or limitation of the method.

ACKNOWLEDGEMENTS

Contributors

I would like to thank my faculty advisor, Dr. Nima Kalantari, and my graduate student mentors, Avinash Paliwal and Libing Zeng, for their guidance and support throughout the course of this research.

Thanks also go to my friends and colleagues and the department faculty and staff for making my time at Texas A&M University a great experience.

Finally, thanks to my friends for their encouragement and to my parents for their patience and love.

The models used for generating input image and light coordinates data in this research were provided in Zexiang Xu's research. The code used for training in this research is based on Vincent Sitzmann's code for fitting an image to SIREN. The code used for saving and loading data is from Vortana Say.

All other work conducted for the thesis was completed by the student independently.

Funding Sources

This undergraduate research received no funding.

NOMENCLATURE

NeRF	Neural Radiance Field
NeLF	Neural Light-transport Field
SIREN	Sinusoidal Representation Networks
ReLU	Rectified Linear Unit
MLP	Multilayer Perceptron

1. INTRODUCTION

1.1 Research Problem

In many computer graphics related fields, image-based relighting is an important technique that could be pervasively used in solving the problem of rendering a scene under novel lightings. Rendering means building models of a scene and generating images from it. The term “relighting” then refers to re-rendering a scene under new lighting conditions. The scene often contains multiple objects, so typically, to achieve the goal of relighting, the precise reconstruction of all objects’ materials, textures, and their physical properties in the model are required. However, in many situations, reconstructing all objects in a complicated scene from scratch would be a massive workload for the engineer. As a result, image-based relighting is developed as an easier and more effective approach to relighting the scene.

The basic process of image relighting is to set up and train a neural network with a series of input images depicting the same objects under different lighting conditions. Then, the network is used to model the objects under new lighting conditions. However, the previous image-based relighting researches often use discrete neural representation, which has distinct limits comparing to implicit neural representation.

Implicit neural representation is a new way of parameterizing signals as a continuous function that maps the domain of the signals to the feature at each point on the domain, according to Sitzmann [1]. The benefit of implicit neural representation is that, using a differentiable, continuous function, the network model would be independent from the grid resolutions. It could also model fine detail that is only limited by the capacity of the underlying network architecture, which is more memory efficient than the traditional discrete grid-based

representations [2]. As a result, the research problem we are focusing on is to leverage implicit neural representation in image-based relighting.

1.2 Related Work

One of the examples of image-based relighting is Xu et al.'s work [3], where five images of an optimal sparse sample were passed into a deep convolutional neural network that's modeling the scene's light transport function. Many improvements on the technique of image-based relighting were also achieved in recent years. For instance, in Bi et al.'s work on Neural Reflectance Fields [4], the image-based relighting method was improved by solving the problem of failing to remodel accurate hard shadows. The neural reflectance fields used reflectance model and normal to calculate the hard shadows instead of passing in view or lighting information. Also, in Mildenhall et al.'s work with image-based 5D neural radiance field representation [5], a continuous volume with a fully connected neural network was used. This approach avoided the discrete sampling of the convolutional neural network, so that the time and space complexity were improved and more promising results was generated. What's more, Sun et al.'s research introduced the concept of a Neural Light-transport Field [6], which could generate volume density and light transport coefficients at any point. This improved the image-based relighting method by making it possible to do portrait relighting and view synthesis simultaneously.

In these previous approaches for image relighting, some of them already used implicit neural representation. For example, the Neural Reflectance Fields Bi et al. developed [4], is an implicit neural representation to model scene geometry and reflectance. Mildenhall et al. [5] also used an implicit continuous 5D vector-valued function for their Neural Radiance Field (NeRF). What's more, the Neural Light-transport Field (NeLF) developed by Sun et al. [6] is inspired by convolutional neural network-based radiance fields like NeRF and its extended researches, where

the NeLF outputs the volume density and light transport coefficients instead of the view-dependent radiance in NeRF.

1.3 Proposed Approach

Although we are also using implicit neural representation, our aim differs from these researches. Our research focuses on abstractly modeling the result pixel color instead of reflectance or radiance fields using the implicit neural network. In our research, the continuous function will be mapping the pixel coordinates directly to the color channels on each pixel. The continuous function of implicit representation can not be written out by hand, so it would have to be approximated via neural network.

Out of the implicit neural networks, we chose sinusoidal representation networks (SIREN) specifically over rectified linear activation unit (ReLU) networks, because ReLU has limitations in representing fine details. For example, natural signals often contain a lot of information in higher-order derivatives, but ReLU-based networks have zero information everywhere on their second derivative. SIRENs, on the other hand, address this existing problem by approaching implicit neural representation with periodic activation functions which have valid second and higher-order derivatives. Based on Sitzmann et al.'s research [2], SIRENs are proven to generate better signal representation results with even high-frequency details in comparison to other networks having non-periodic activation function.

As a result, my research presents a method that renders a scene of optimal sparse samples in novel lighting using implicit neural representation. The algorithm I used for this project is based on SIREN presented in Sitzmann et al's work. I modified SIREN so that it would take in both the coordinates and the corresponding light coordinates of the input image. SIREN was trained with an input images dataset depicting the same set of optimal sparse samples under

different lighting positions. Finally, the testing of the network used an input of the coordinates and a new light position. The expected result should be an image of the same optimal sparse objects under the novel light, generated through the network.

2. METHODS

As in the previously mentioned deep image-based relighting researches, our inputs are multiple images depicting the same set of sparse objects under different lighting conditions and their lighting coordinates. In the process of generating these images, the only variant is the position of the light, which means the camera position, camera angle, objects position, etc. are all kept the same. Our goal is to train the neural network with these images and their light positions, then use the network to predict the color of each pixel in the result image according to the new light position.

Before setting up and training the network, we first need to figure out how we are representing the input information. For the domain of the continuous function, we have both pixel coordinates and light positions to take in. The pixels of the input images are represented using 2D Cartesian Coordinates. Each pixel of the image is nominated as (x,y) , corresponding to the rows and columns, respectively.

Afterwards, the problem of how to represent the light position arises. The light source used in our project is a point light source in 3D spaces, which is normally represented in the form of (x,y,z) in a 3D Cartesian Coordinate system, as shown in Figure 2.1.

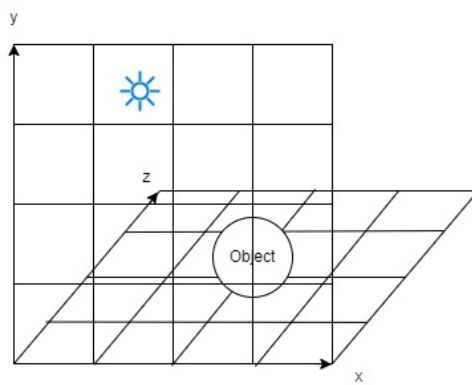


Figure 2.1: point light source in cartesian coordinates

However, in our algorithm, to describe the light position, we see the position of the light source as a certain point on the surface of a hemisphere surrounding the object with a fixed radius r . In this respect, we are expressing the light position in a spherical coordinate system. A benefit is that, since the r is fixed, we would only need to parameterize the coordinate in terms of the two angles, θ and φ , instead of three parameters (x,y,z) , as shown in Figure 2.2.

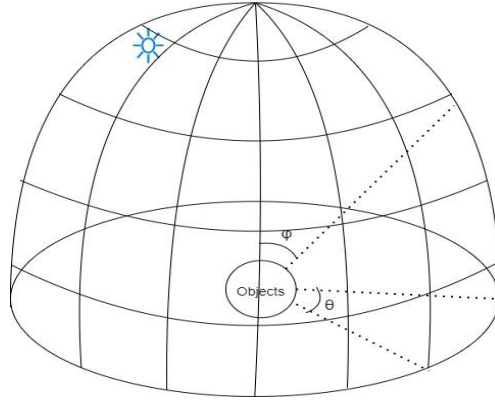


Figure 2.2: point light source in spherical coordinates

Another benefit is that, after the network is trained with a smaller sample of single input lights, we can simulate the scene under an environment map light. An environment map light is composed by many point light sources arranged as a mesh grid on the surface of a hemisphere surrounding the object. The goal of using this light dome is to create the effect that the light source is infinitely far away from the objects, and shines on the objects from all directions. The environment map light is as shown in Figure 2.3. Although we are only doing our testing simulation with point lights, we can ensure that the algorithm we developed is able to be tested with an environment map light.

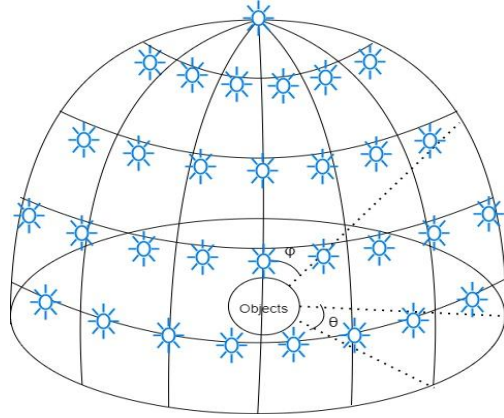


Figure 2.3: Environment map light

As a result, the light coordinate taken into the network would be represented as a pair φ , θ . To get higher quality results, we picked our input images based on the method described in Xu et al.'s research [7]. For example, here we sampled five images, where one of the images has the central light source right above the objects and the other four images have light sources distributed around the central light source equally, as shown in Figure 2.4.

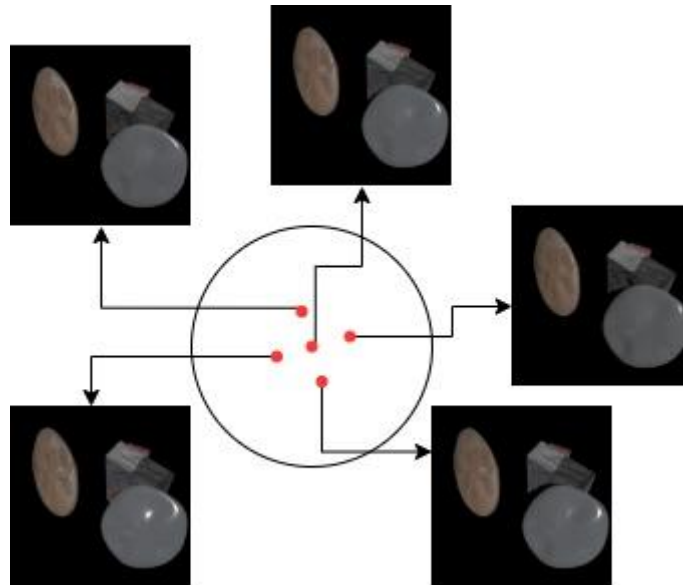


Figure 2.4: input images and their corresponding light positions

2.1 Setting up the SIREN Network

The network we used in our research is the SIREN network presented in Sitzmann et al's research [2]. SIREN is developed based on Multilayer Perceptron (MLP) neural networks. Traditional MLPs are composed by an input layer, an output layer, and many hidden layers in between. The nodes in each layer are fully connected. At each hidden layer, the output is generated by taking the dot product of the input and the weights existing between this layer and the previous layer. After that, the outputs are pushed through an activation function, and then pushed forward to the next layer as an input. Eventually, we perform the backpropagation at the output layer [8].

The difference between SIRENs and traditional MLPs is that SIRENs replace the piecewise linear ReLU activation function with a periodic sinusoidal function, for example, a sin curve. When derived, linear functions have zero everywhere on second and higher derivatives, but sinusoidal functions have valid values on its derivatives. In this way, SIREN is able to model the signal better with higher-ordered derivatives information than original MLPs [2].

In our research, SIREN needs to take in both pixel and light coordinates, and then output color channels. In this process, each pixel and light position are mapped to the corresponding RGB color at the pixel. As a result, the SIREN network is set up to have 4 input features in the input layer, in the format of (x, y, φ, θ) , where x, y represents the pixel coordinate, and φ, θ represents light coordinate. There are 3 output features in the output layer, in the format of (r, g, b) , representing the 3 color channels corresponding to each pixel. Between the input and output layers are 3 hidden layers, each have 256 nodes fully connected to each other. Thus, we finished setting up the SIREN network, and the fully-connected diagram for the network is shown in Figure 2.5.

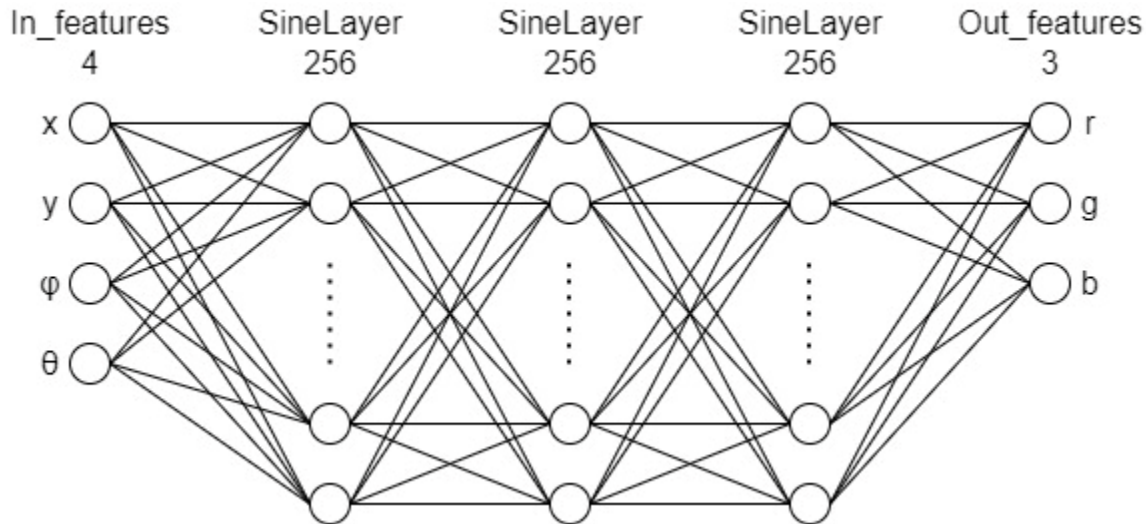


Figure 2.5: fully-connected SIREN network

2.2 Training of the SIREN Network

Our goal for training the SIREN network is to make it reconstruct all input images as well as possible and then determine when to stop training. The training algorithm we used is based on the training loop provided in the section of fitting an image in Sitzmann’s research.

We inherited the optimization method from Sitzmann et al’s code. In our training loop, Adam optimizer is used with a learning rate of $1 * 10^{-4}$. We calculate the gradients and do backpropagation for each individual input images, so our minibatch size is just one input image.

Also, as in Sitzmann et al’s research [2], with pixel coordinates denoted as $x_i = (x_i, y_i)$, corresponding RGB colors denoted as $f(x_i)$, continuous function denoted as Φ , and output at each epoch denoted as $\Phi(x)$, the loss function we used in the SIRENs is as Equation 2.1.

$$\mathcal{L} = \sum_i \|\Phi(x_i) - f(x_i)\|^2 \tag{2.1}$$

First of all, we started the training by fitting all input images into the network with 512*512 pixels resolution and comparing the reconstructed images side by side to the ground truth images. I trained the network for a total of 30000 epochs here and printed out the result of

step 0, 10000, and 20000 as examples, with ground truth on the left and result images on the right, as shown in Figure 2.6.

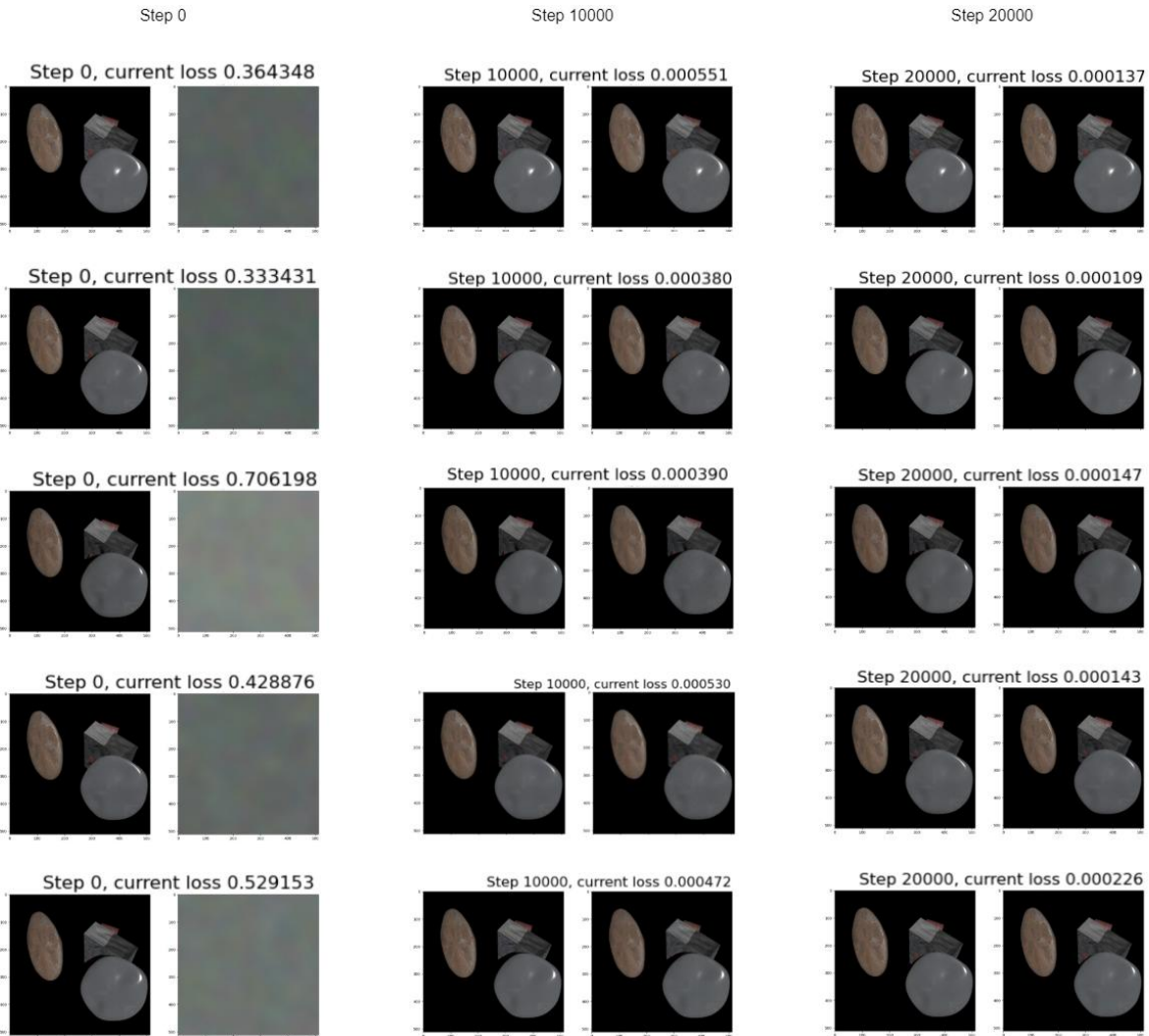


Figure 2.6: Step 0, 10000, and 20000.

Comparing the images reconstructed by the network, between step 0 and step 20000, we can clearly see that the output images are restoring the ground truth better and better. However, as the step number gets bigger, like from 10000 to 20000, it would be very hard to tell the difference between the ground truth images and the result images to the naked eye. What's

noticeable is that the print-out of the loss value in the figure does show a valid decrease despite the similarity of the graphs. As a result, the best way of seeing how many steps it would take to optimize the training result is to graph the loss and find where it starts to minimize. In our research, each point on the loss graph would be the logarithm of the average loss for all input images over every 50 steps. We took the logarithm of the average loss, because the loss data tend to dispartate a lot between the beginning and the end of the training, and logarithm would help us in seeing the slight differences between the losses when it gets to the end, thus determine when to stop. We used the average loss instead of the loss every 50 steps, because in this case the curve can be smoothed out. Figure 2.7 shows the loss graph we get when training the network for 30000 steps.

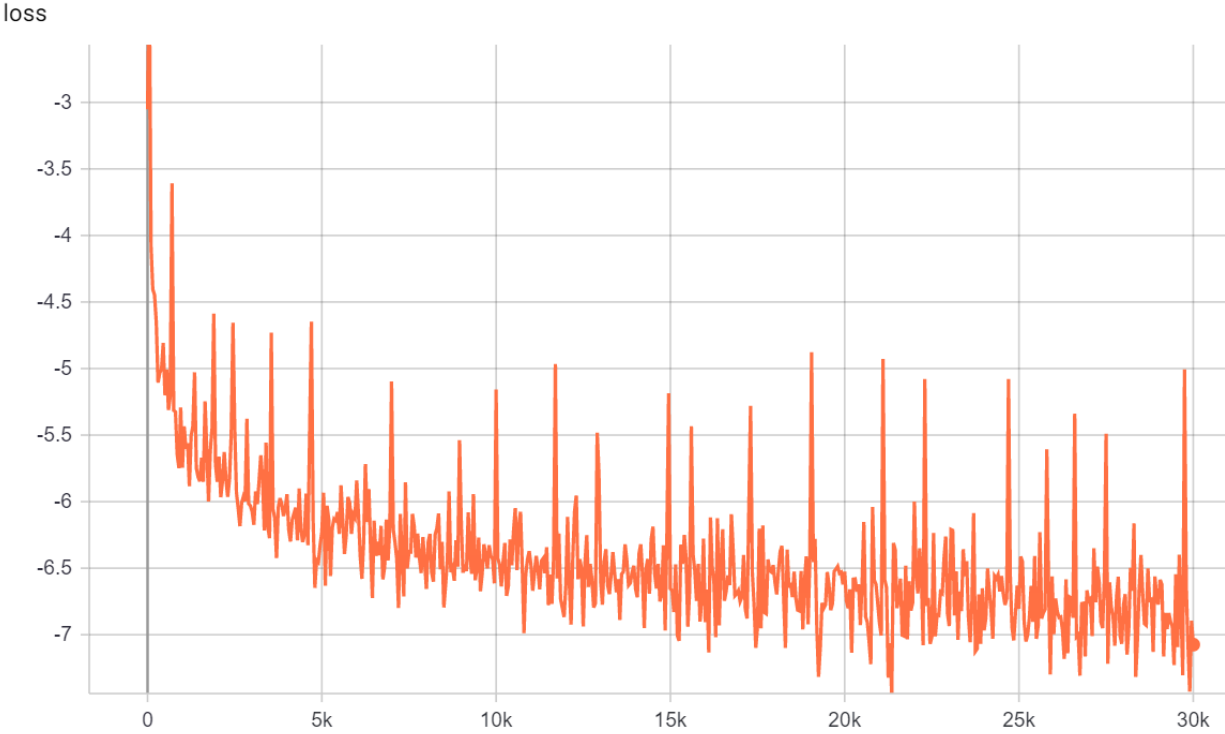


Figure 2.7: loss graph for 30000 training steps.

From the loss graph we can see that, although the decrease rate is getting lower and lower, the loss is still decreasing. As a result, we were still not sure if the network is trained well

or already overly trained at this point. In order to decide when to stop training, we have to see an actual result when testing the model with a new lighting coordinate. Hence, we need to save the current best trained model, so that we can load the model in the testing process.

To achieve the goal of saving and loading models, I used the save and load method code from an online resource by Say [9]. The saved model contains four kinds of information: the epoch number it starts on, its current minimum loss, model architecture information, and its optimizer. Since our total epochs goes to above 30000, we save the model at every 1000 epochs. If the average loss showed improvement, i.e. decreased comparing to the previous average loss, we save this model as the best model into the local device. Otherwise, it would be saved simply as the most recent model.

Eventually, the best model was saved successfully after training the SIREN network with our input images for a total of 30000 epochs. The saved minimal loss is -7.074147143990179. Figure 2.8 is the training result we get, comparing the output of each images side by side with its ground truth.

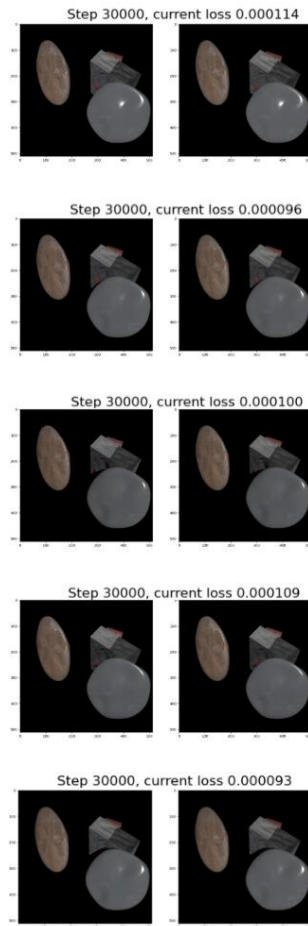


Figure 2.8: Training output after 30000 steps.

2.3 Testing the SIREN Network

The eventual goal for testing the SIREN network is to give the trained SIREN model a new light position and generate a new image from it. However, before going through with new lighting positions, we first needed to make sure that our algorithm works with no error.

As a result, after loading all four features of the saved best model into the program, we passed an input light coordinate along with the pixel coordinates first into the model, trying to see if the algorithm could reproduce the input image. We used the $[0.0,0.0]$ light position for this test, which gave us a successful result, as in Figure 2.9.

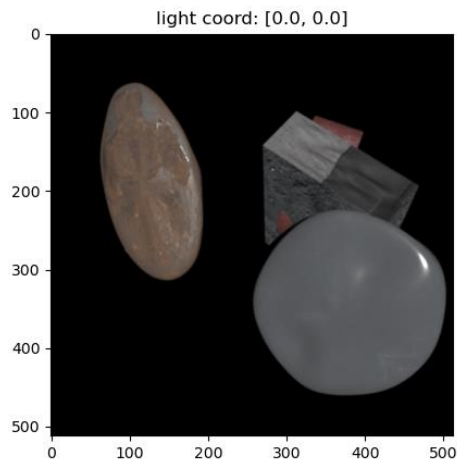


Figure 2.9: Testing the algorithm with existing light position [0.0, 0.0]

Then, to test how good the model works, we need to pass in a new light coordinate. At first, we chose a new light coordinate that existed in the original dataset with 1053 images, because in this way, we would have a ground truth image to compare with the image generated by the testing process. However, the new light position seemed to be giving a faulty result, as in Figure 2.10.

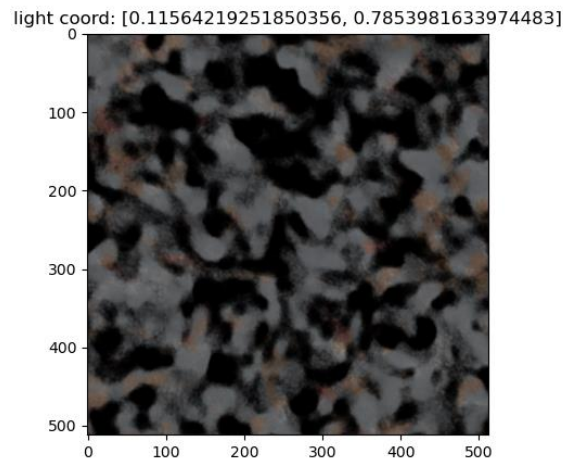


Figure 2.10: Testing the algorithm with new light position.

This is because the network is overfitting the input data. Overfitting means that the network is trained so well for reconstructing the set of input images, that would not recognize new light source that's far away from the input light positions and generate the output image accordingly. To further understand the logic in this situation, we generated a series of images with each of their light positions differ slightly from the input light position (0.0, 0.0). We increased both φ and θ by 0.01 each time, from (0.01, 0.01) to (0.05, 0.05). Figure 2.11 shows the result images and their light positions accordingly.

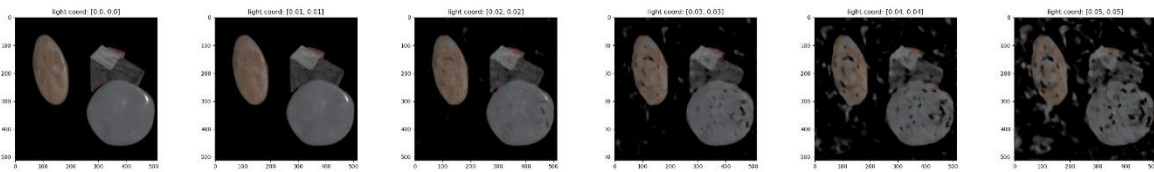


Figure 2.11: Testing with slightly different light position.

From the series of results, we can see that the network does generate good quality results at first, but the quality of the image drops generally quickly when slowly changing the light position. In order to test if the level of training affects the result, we also tried to train the network for 5000 steps and 1000 steps. These results shows that the more training steps, the better the training results quality, but the quicker the quality drops when switching to another lighting position, as in Figure 2.12.

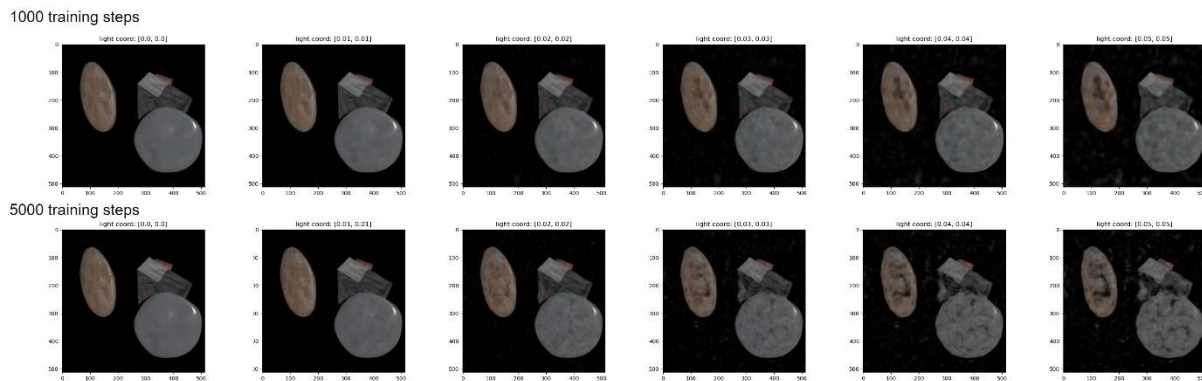


Figure 2.12: Testing with slightly different light position after 5000 and 1000 training epochs

However, even though the results show that less training might make the network less overfitting to the input images, it doesn't provide a solid solution to our situation, since even only training for 1000 steps, the results still share a similar quality dropping tendency, and gives out faulty images at a new light position. As a result, we decided to solve this problem without sacrificing image quality by compressing the light coordinate. We scaled down the light positions φ and θ by 10 and 100 respectively for both training and testing process, which will bring the input light coordinates much closer to each other in the space. In this way, it's easier for the network to interpolate a new lighting position in between smaller gaps and generate the result image. After scaling down φ and θ , we tested the network with the same new light position as before. The network gave a successful result this time, as in Figure 2.13.

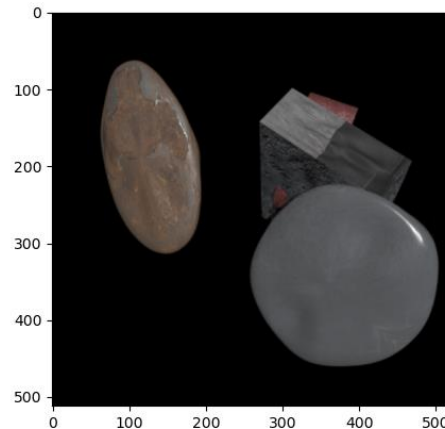


Figure 2.13: Testing the algorithm with new light position after compressing light coordinates.

Thus, we managed to get promising testing results from the network. The stability and quality of the testing results will be analyzed and discussed in the results section.

3. IMPLEMENTATION

In the implementation section, we will further discuss the process of training and testing with coding details.

3.1 Converting Light Positions and Choosing Input Images

The dataset we used for the research contains 1053 images in total, and the lighting condition is the only variable during the generation of these images. Each image is generated under a single light source and the light positions of these images were recorded as 3D Cartesian coordinates in a text file. These light coordinates composed a full light dome when combined. In our research, only five input images were needed for training, and their corresponding light positions should be passed in as 3D Spherical coordinates. As a result, we needed to convert their light positions so that they fit into our algorithm and to sample these images in a reasonable way.

First of all, we needed to convert the light positions into spherical system. The recorded light coordinates were in (x,y,z) format in the text file. However, in spherical coordinate, we denote the position with θ , φ , and r instead. As explained in the method section, θ is the angle on xy-plane, and φ is the angle according to z-axis. r is the radius between the light positions and the objects, which is deliberately kept as 1 in the original dataset. In this case, we can convert the coordinates using the mathematical Equations 3.1 and 3.2 below:

$$\theta = \tan^{-1}\left(\frac{y}{x}\right) \quad (3.1)$$

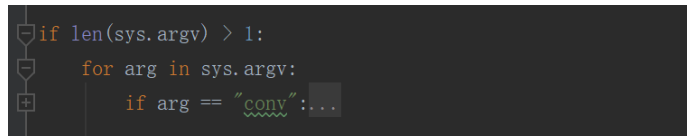
$$\varphi = \cos^{-1}(z) \quad (3.2)$$

What's worth noticing is that, when calculating θ , we have to take the quadrants into account. For example, when y is -1 and x is -1, the θ angle they made lands in the third quadrant.

When y is 1 and x is 1, the θ angle they made lands in the first quadrant. However, when we are calculating them using equation 2.1, we will get the same answer regardless of quadrants since y/x are both 1. As a result, when implementing the conversion, we used the `atan2(y,x)` function in the math packet to calculate θ instead of `atan(y/x)`. This is because `atan2()` takes in the negativity information of both y and x into count when calculating θ .

There exists an edge case in this conversion, where the light position is right above the objects. This light coordinate is on the z axis, which means φ would be 0, but θ would be undefined since x and y are both 0. In this case, we just assigned θ as 0 for our implementation.

When implementing this part, it'd be the best if we separate the conversion from the training and testing of the network, since we only need to convert the light position file once. To achieve this, we used `sys.argv` command to make sure it could take in a keyword when running the program. For this part, we used “conv” as our keyword, as shown in Figure 3.1. When running the program, if “conv” is added after the filename, the program goes into this if-statement.



```
if len(sys.argv) > 1:
    for arg in sys.argv:
        if arg == "conv":...
```

Figure 3.1: taking in “conv” as keyword

Inside the if statement, we built the conversion section algorithm by first opening the input file in reading mode and output file in writing mode. Then, we read in the cartesian coordinates text file line by line into a list and looped through them. Inside the loop, we first stripped the spaces around the line, split the line by spaces into a list of strings, then converted each string into Decimal type variable. The reason we chose Decimal instead of float is that it preserves more decimal places than float type does. Afterwards, we did the conversion for θ and

φ using the functions above and wrote the spherical coordinates data into the output text file. The reading and writing are done using `readlines()` and `writelines()` functions. The whole process is shown in Figure 3.2.

```

print("converting!")
#-----conversion from cartesian to spherical-----
cartesian = open("C:/Users/shuyuwang99/Desktop/muti-img-siren/resources/images/images/3d_pos.txt","r")
spherical = open("C:/Users/shuyuwang99/Desktop/muti-img-siren/resources/images/images/sph_pos.txt","w")
all_lines = cartesian.readlines()
spherical.writelines(["theta", " ", "phi", "\n"])
#print(all_lines)
for line in all_lines:
    line = line.strip()
    line = line.split(' ')
    #print(line)
    x = Decimal(line[0])
    y = Decimal(line[1])
    z = Decimal(line[2])
    #print(x, y, z)
    #z = math.sqrt(1-x*x-y*y)
    #print(z)
    #r = Decimal(1.0)
    theta = math.acos(z)
    if x != 0:
        phi = math.atan2(y,x)
    else:
        phi = math.atan(0)
    print(theta)
    print(phi)
    #the decimal number is shortened after calculation, is this ok???
    spherical.writelines([str(theta), " ", str(phi), "\n"])
cartesian.close()
spherical.close()
quit()

```

Figure 3.2: conversion from cartesian to spherical

After converting the coordinates of the light from the cartesian to the spherical system, we needed to pick out the input images we needed for training. In order to get higher quality results from the network, the method we used in choosing images was based on the method used in Xu et al.'s research [7]. The light coordinate of the image in the middle has a light right above the depicted objects is $[0.0, 0.0]$. For the other four, we kept φ the same so that they are all equally far away from the middle light source. Then, we calculated the angles between each θ

and made sure that they are about 90 degrees, which keeps the four light sources equally distributed.

3.2 SIREN Network

3.2.1 Setting up

The SIREN network used in Sitzmann et al’s research [2] for demonstrating fitting a grayscale image in the network was set to have 2 input channels in the input layer, 1 output channel in the output layer, and 256 nodes in each of the 3 hidden layers. The input is a tensor of (x,y) coordinates, representing each pixel of the input image. The output, on the other hand, is the grayscale color value corresponding to each pixel, ranged between 0 and 1. See Figure 3.3.

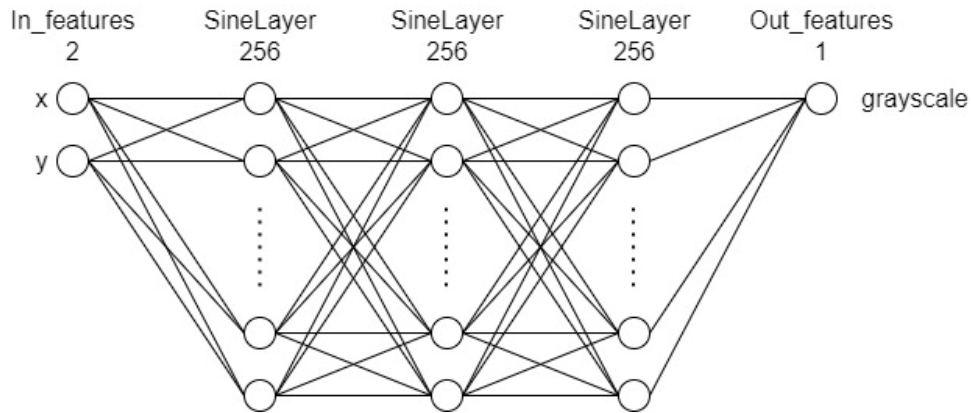


Figure 3.3: fully connected graph for the original SIREN

However, in our research, multiple colored input images are used instead of one grayscale image. Due to the fact, we first changed the out features variable from 1 to 3 when setting the SIREN network, so that the network has 3 output channels representing (r,g,b) values. As shown in the simplified Figure 3.4.

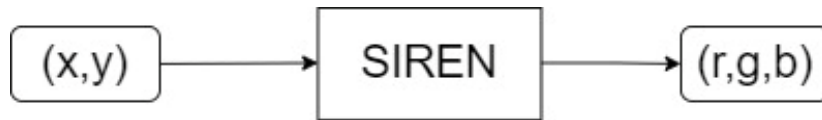


Figure 3.4: simplified graph for fitting colored image in SIREN.

To achieve the goal of image relighting, we also need to make sure that the network can take in and be trained with multiple images. As a result, instead of using the cameraman sample image from the skimage data pack, we acquired the path of the folder containing all the input images. Then, we modified the ImageFitting(Dataset) class, so that it takes in the path, sorts all images, resizes them, takes them in as normalized tensors, and finally returns them along with coordinates and shape of the tensor.

Also, we need to modify the SIREN network so that it takes both pixel coordinates and the light coordinate of each image in the set of input images. In this case, light coordinates of the images are hard coded in the ImageFitting(Dataset) class as a 2D tensor. This tensor has (number of images) rows and 2 columns, since each image has a light coordinate (φ, θ) . After that, the tensor is returned along with images, coordinates, and shape. Finally, when setting the SIREN network, we change the in features as 4 in the format of (x, y, φ, θ) .

3.2.2 Training and Plotting Loss Graph

First of all, we used the ImageFitting() class and DataLoader() function to load in multiple images. Then, we looped through all five images at each step of the training with a nested for loop. Also, since most of the training loop from the original code by Sitzmann et al. can be kept for our training, we just moved this part into our inner loop. We also removed and changed some of the print out and showing of training results for efficiency.

The major issue during training is the plotting of loss graph, since it's the most intuitive way to see the training progress. At first, we used matplotlib.pyplot package to graph the loss. We first created an empty list named "losses" before the loop. Then, we created a variable named "total_loss", in order to sum up all the loss of all images between every 50 steps. Afterwards, at every 50th steps, we calculated and pushed the logarithm of the average loss into the list, and

restored `total_loss` to zero for the next 50 steps. Eventually, after gaining all loss values from the training, we plotted out the loss graph.

However, this graphing method only shows the graph after the training loop ended. Not being able to see the data during the training is very inconvenient, especially for long training processes like this, because we have to wait for a long time and consume a lot of GPU resources before we are able to see if the results are ideal or there are actually errors within our algorithm and the graphing was wrong from the beginning.

Looking for a solution to graph the loss while training, we eventually switched our graphing tool to TensorBoard. TensorBoard is a visualization toolkit under TensorFlow package, and it can be used to display data in many forms such as graphs, images, audios, etc.

TensorBoard package can be used to train the model and graph the loss all together, but in our case, we only need to graph loss as custom scalars, since we are already calculating the loss data in our training.

To graph the data, we first need to create a local directory for the data log. Once the training starts, all the data would be written in a file in this directory. In order to distinguish different training processes, I added the local time into the directory. In this case, each training process data log would be stored in a different folder, named in the format of “directory + localtime”. I printed out this directory for future references.

Then, we needed a file writer variable to write the data in the directory we set. Here we use the `SummaryWriter()` function in `tensorboardX` package, where the directory is passed in as a string parameter, to create a new file writer. Then, inside our training loop, at each 50 steps, once we get the logarithm of the average loss, instead of putting the data into a list and graph it at the end, we use the file writer we created to write the loss into the file. In this case, we used

`add_scalar()` function from `tensorboardX` package, and passed in the graph name as “loss”, data as current loss, and step as the current epoch.

Now, after setting up the file writer, we started running this program. Once we started running, the data log directory will be printed out. However, to see the actual graph while running, we need to activate the TensorBoard web interface. This required us to open up a new command prompt, and copy the printed-out directory from the original command prompt into the new one. Then, after the directory, we typed in “`--host localhost --port 8088`” to direct the interface to a specific port, and ran this command in the new command prompt. This way, the running program will give out a website link “`http://localhost:8088`”, and that is where the TensorBoard web interface will show up. We can see the interface by copying this link into the browser, like shown in Figure 3.5.

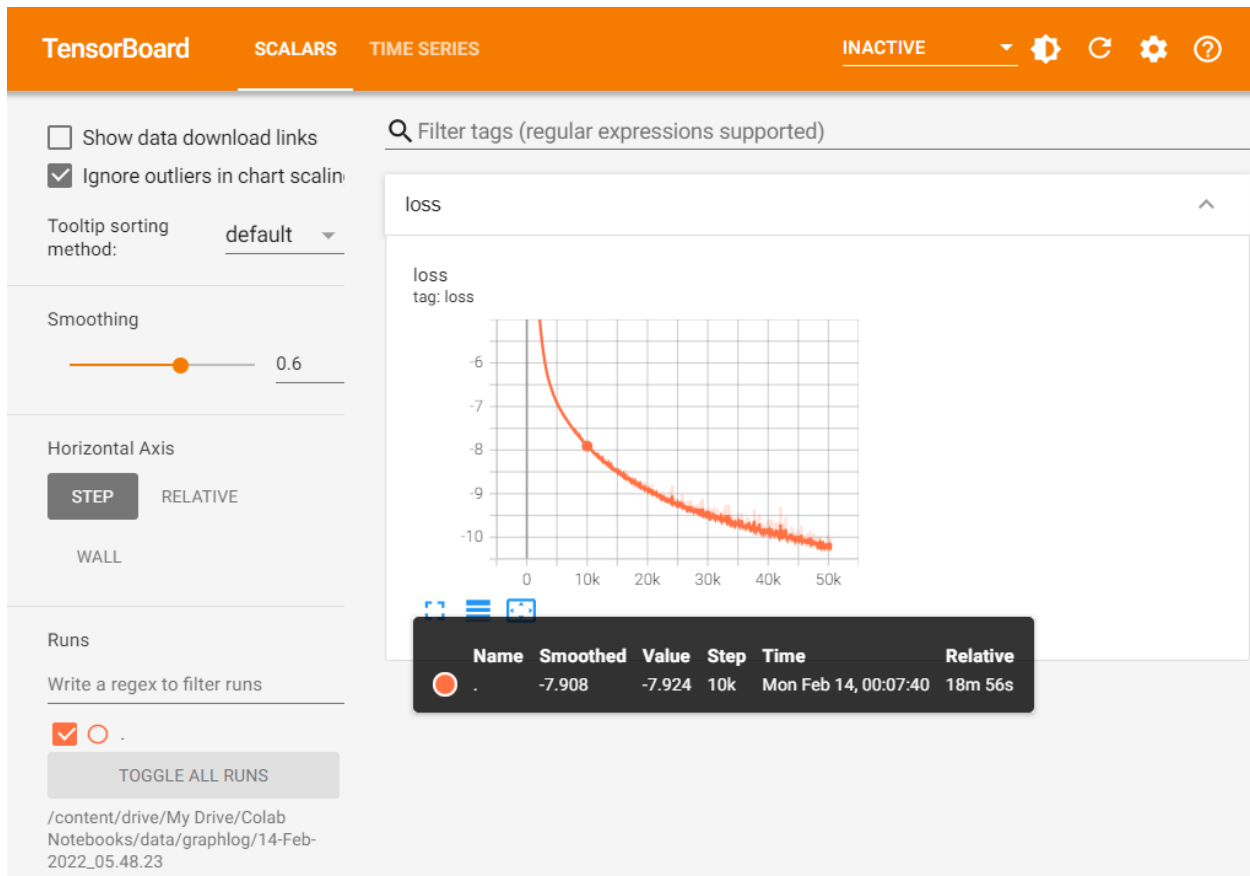


Figure 3.5: TensorBoard interface for loss graph.

On this web interface, we can see that our graph appears under the tab “Scalar”, because we are graphing the scalar loss data point after point. The Horizontal Axis should be set to “step”, which is the epoch number we passed into the file writer. We can adjust how smooth our graph looks by dragging or input a number at the “Smoothing” section. Also, to see the new data being written in while running, we need to use the refresh button on the upper-right corner, which looks like a circling arrow. Every time we click the refresh button, it will show new data on the graph, but in order to see the whole graph, we would need to manually adjust the domain by clicking the “Fit domain to data” button under the graph, which looks like a box with four arrows inside at each direction.

What's more, if we hover the mouse on the graph, we can see the value of the original data, the smoothed data, the current step, the time that the data was written in, and the relative written-in time to the whole graph. Also, to see the graph more clearly, we can expand the graph to fit out browser window by clicking the "full screen" button, which is the left first button under the graph.

3.2.3 *Saving, Loading, and Testing Model*

Implementing the saving and loading would give us the benefit of separating the training and testing process, as a result, we separated the training and testing procedures using the same `sys.argv` command to take in keywords when running as in the conversion section. For training, we take in the keyword "train", and for testing, we take in the keyword "test". The code for saving model goes in training section and loading model goes in testing section.

First of all, I used the `save_ckp()` and `load_ckp()` functions provided by Vortana Say [9], as in Figure 3.6. In this part, we used a variable called `checkpoint`. Each checkpoint contains four parameters: "epoch", "valid_loss_min" - minimum loss, "state_dict" - model architecture information, and "optimizer". Basically, a checkpoint contains all the information we needed for successfully saving a model.


```

#-----saving & loading model-----
#source: https://towardsdatascience.com/how-to-save-and-load-a-model-in-pytorch-with-a-complete-example-c2920e617dee

def save_ckp(state, is_best, checkpoint_path, best_model_path):
    """
    state: checkpoint we want to save
    is_best: is this the best checkpoint; min validation loss
    checkpoint_path: path to save checkpoint
    best_model_path: path to save best model
    """
    f_path = checkpoint_path
    # save checkpoint data to the path given, checkpoint_path
    torch.save(state, f_path)
    # if it is a best model, min validation loss
    if is_best:
        best_fpath = best_model_path
        # copy that checkpoint file to best path given, best_model_path
        shutil.copyfile(f_path, best_fpath)

def load_ckp(checkpoint_fpath, model, optimizer):
    """
    checkpoint_path: path to save checkpoint
    model: model that we want to load checkpoint parameters into
    optimizer: optimizer we defined in previous training
    """
    # load check point
    checkpoint = torch.load(checkpoint_fpath)
    # initialize state_dict from checkpoint to model
    model.load_state_dict(checkpoint['state_dict'])
    # initialize optimizer from checkpoint to optimizer
    optimizer.load_state_dict(checkpoint['optimizer'])
    # initialize valid_loss_min from checkpoint to valid_loss_min
    valid_loss_min = checkpoint['valid_loss_min']
    # return model, optimizer, epoch value, min validation loss
    return model, optimizer, checkpoint['epoch'], valid_loss_min.item()

```

Figure 3.6: save_ckp and load_ckp functions.

The `save_ckp` function is used to save checkpoints. Each time this function is called, the old checkpoint file in “`checkpoint_path`” is overwritten by the current checkpoint “`state`” using `torch.save()` function. Then, if “`is_best`” variable is true, that means the current checkpoint is the best checkpoint with lowest loss value. In this case, we copy the current checkpoint file to the “`best_model_path`” using `shutil.copyfile()` function, which will also overwrite the original content in the path.

The `load_ckp` function is for loading checkpoints. When this function is called, we first load in the checkpoint with `torch.load()` function from the “`checkpoint_fpath`”. Then, we use `load_state_dict()` function to load the “`state_dict`” and “`optimizer`” parameters of current

checkpoint into the passed in “model” and “optimizer” variables accordingly. Next, we create a variable called “valid_loss_min” to store the minimum loss from the checkpoint. Finally, we return all four variables as results.

After setting up these functions, we first set the directories for “checkpoint_path” and “best_model_path” in the local folder for storing checkpoints. Then, we use the save_ckp() function in our training loops to save both the current and the best model.

```
#-----saving model-----
if epoch !=0 and epoch % 1000 == 0:
    # create checkpoint variable and add important data
    checkpoint = {
        'epoch': epoch + 1,
        'valid_loss_min': log_loss,
        'state_dict': img_siren.state_dict(),
        'optimizer': optim.state_dict(),
    }
    # save checkpoint
    save_ckp(checkpoint, False, checkpoint_path, best_model_path)

    ## TODO: save the model if validation loss has decreased
    if log_loss <= loss_min:
        print('Validation loss decreased ({:.6f} --> {:.6f}). Saving model ...'.format(loss_min,log_loss))
        # save checkpoint as best model
        save_ckp(checkpoint, True, checkpoint_path, best_model_path)
        loss_min = log_loss
```

Figure 3.7: Saving model.

As in Figure 3.7, we also used the code by Vortnana Say for this implementation [9]. First, we created a “loss_min” variable outside the training loop, with the original value as positive infinity, to keep track of the minimum loss. Then, we decided that we would update the checkpoint every 1000 steps. Hence, at every 1000th step, we create and parameterize a checkpoint variable and save it into the checkpoint_path. Here we set the second parameter of save_ckp() function, “is_best”, as “False”, meaning that current checkpoint is not the best model. Afterwards, we check if the loss is decreased by comparing current loss to the “loss_min”. If there is a decrease, we save this model as the best model with the second parameter “is_best” as “True”.

After saving the model, the loading and testing began. To distinguish the testing process from the training process, I created a new set of variables. First, since the `load_ckpt()` function takes in model and optimizer parameters, I created a “test_optim” variable and a new siren network model called “test_siren” as I did for the training process, and activated cuda using `.cuda()` function for the “test_siren” model. Then, I loaded model using `load_ckpt()`, as in Figure 3.8. At this point, test_siren has become the best saved network model.

```
#load model
test_siren, test_optim, loaded_start_epoch, loaded_loss_min = load_ckpt(checkpoint_path, test_siren, test_optim)
```

Figure 3.8: Loading the model.

Then, to test the model, we need pixel coordinates and a new light coordinate. Also, we need an input shape to show the result image. For the pixel coordinates and the input shape, we created a new set of variables named “test_model_input” and “test_shape”, and got the information from dataloader directly. For the new light coordinate, on the other hand, we hard coded the light position as a new tensor named “test_lightcoord”. Finally, we activated cuda, tested the model with the pixel and light coordinate, and displayed the results, as in Figure 3.9.

```
test_model_input, test_lightcoord = test_model_input.cuda(), test_lightcoord.cuda()

#test with loaded model
test_model_output, test_coords = test_siren(test_model_input, test_lightcoord)

#for displaying groundtruth
denormalize = Normalize(torch.Tensor([-1]), torch.Tensor([2]))
plt.imshow(denormalize(test_model_output).cpu().view(test_shape[1], test_shape[2], 3).detach().numpy())
plt.show()
```

Figure 3.9: Testing the model and showing results.

4. RESULTS

When training, we used three set of images, named “close to z,” “in the middle,” and “far from z,” according to how big their φ angle is. Each set of images shares a same φ angle in order to keep them on a circle surrounding the object. The five lighting coordinates covers an area on the hemisphere above the object. The smaller the φ angle is, the smaller the area is covered by the input lighting coordinates on the hemisphere, and vice versa, as shown in Figure 4.1.

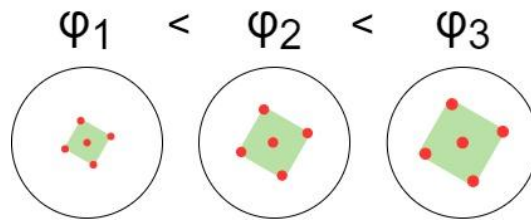
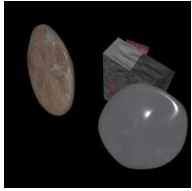
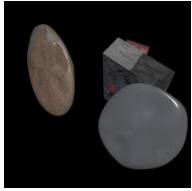
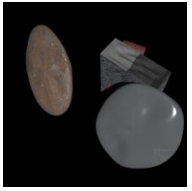

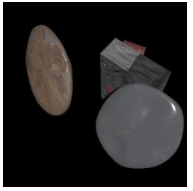
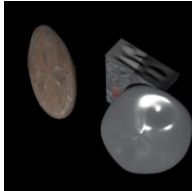
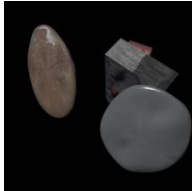
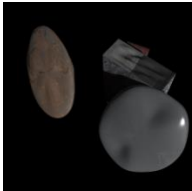
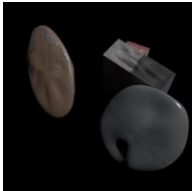
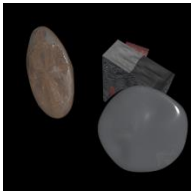
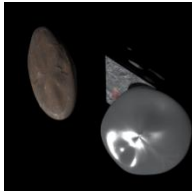
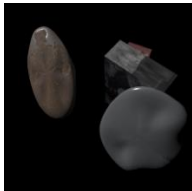


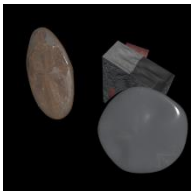


Figure 4.1: The area covered by input lighting coordinates

The sets of images we used and their corresponding uncompressed light coordinates are shown in Table 4.1.

Table 4.1: Training input image sets.

Close to z					
Image					
φ	0.1935131925145 134	0.193513192514 5134	0.193513192514 5134	0.193513192514 5134	0.0
θ	-2.99969559898 5629	1.712693381399 0606	-1.42889927219 07332	0.141897054604 1634	0.0
In the middle					
Image					
φ	0.7661626497120 905	0.766162649712 0905	0.766162649712 0905	0.766162649712 0905	0.0
θ	-2.41169299543 80617	2.300695984946 628	-0.84089666864 31651	0.729899658151 7314	0.0
Far from z					
Image					
φ	1.2925495041266 992	1.292549504126 6992	1.292549504126 6992	1.292549504126 6992	0.0
θ	-2.356194490192 345	2.356194490192 345	-0.78539816339 74483	0.785398163397 4483	0.0

Note: Image with light position (0,0) exists in all three sets because it has the lighting position in the middle.

The first aspect we want to test with our network is if the image quality is still good while moving the light position away from the input light position. Using the “close to z” image batch, we decided to test with 10 lighting positions, equally distanced from each other, transferring from input light position $[0,0]$ to another input light position $[0.1935131925145134, 1.7126933813990606]$, as in Figure 4.2.

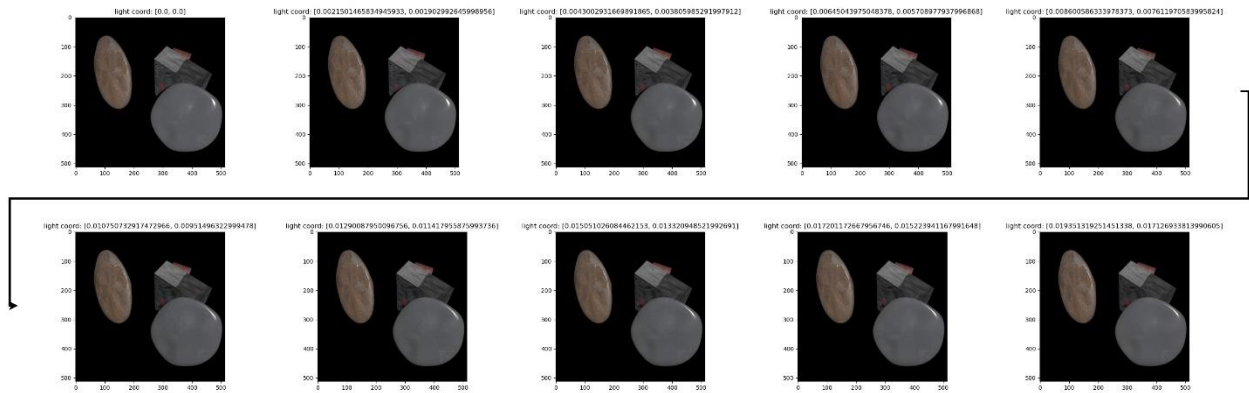


Figure 4.2: Transition between two input images.

The transition test results are very promising, since we can see that the image quality did not drop while moving the testing light position away from the input light positions. As a result, now we can try testing with new lighting positions in the area covered by the input lighting positions. In this part, we chose light coordinates that already exist in the original dataset of 1053 images, so that the result can be compared to a ground truth image. Figure 4.3 shows the five testing images comparing to their ground truth images, with their compressed light position and their loss corresponding to the ground truth.

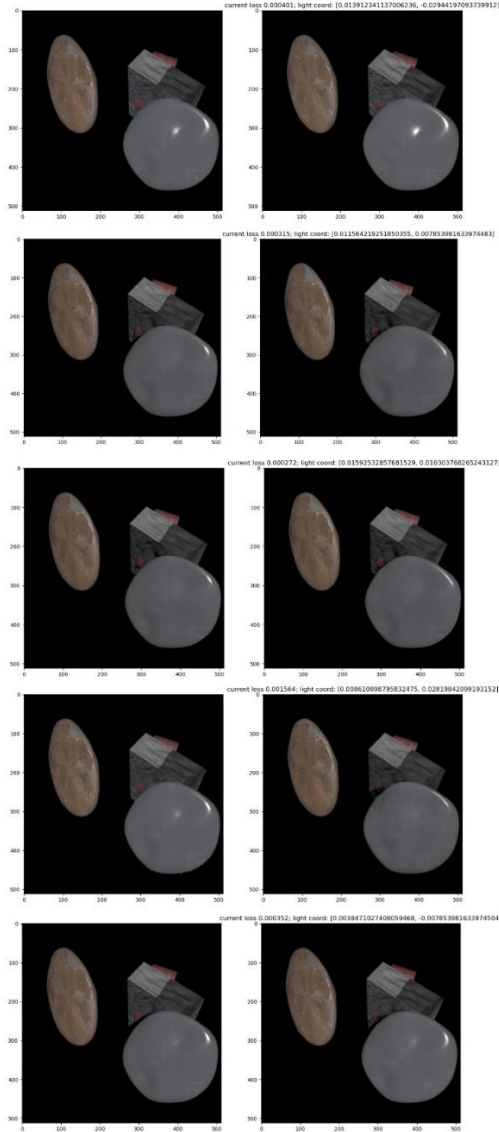


Figure 4.3: Testing results for “close to z” images.

From the results, we can see that despite the good quality of result images, some of the light conditions are not precisely restored. For example, at the fourth image in Figure 4.3, with light coordinate (0.008610898795832475, 0.028198420991931523), the difference can be seen with the naked eye and the loss is higher than the other testing images. However, this could just be an outlier in the testing, since the other four images showed reasonable output images and loss values. Therefore, our testing results can still be considered promising.

When evaluating our results, we not only want to see how good the result is under new light positions, but we also want to know if the change of φ would affect the quality of outputs. Since the input sets with larger φ has a larger area covered by the input light positions, we spread out the testing light positions into the covered area correspondingly. Figure 4.4 shows the testing results for “in the middle” and “far from z” input sets, comparing to their ground truth images, with their compressed light position and their loss corresponding to the ground truth.

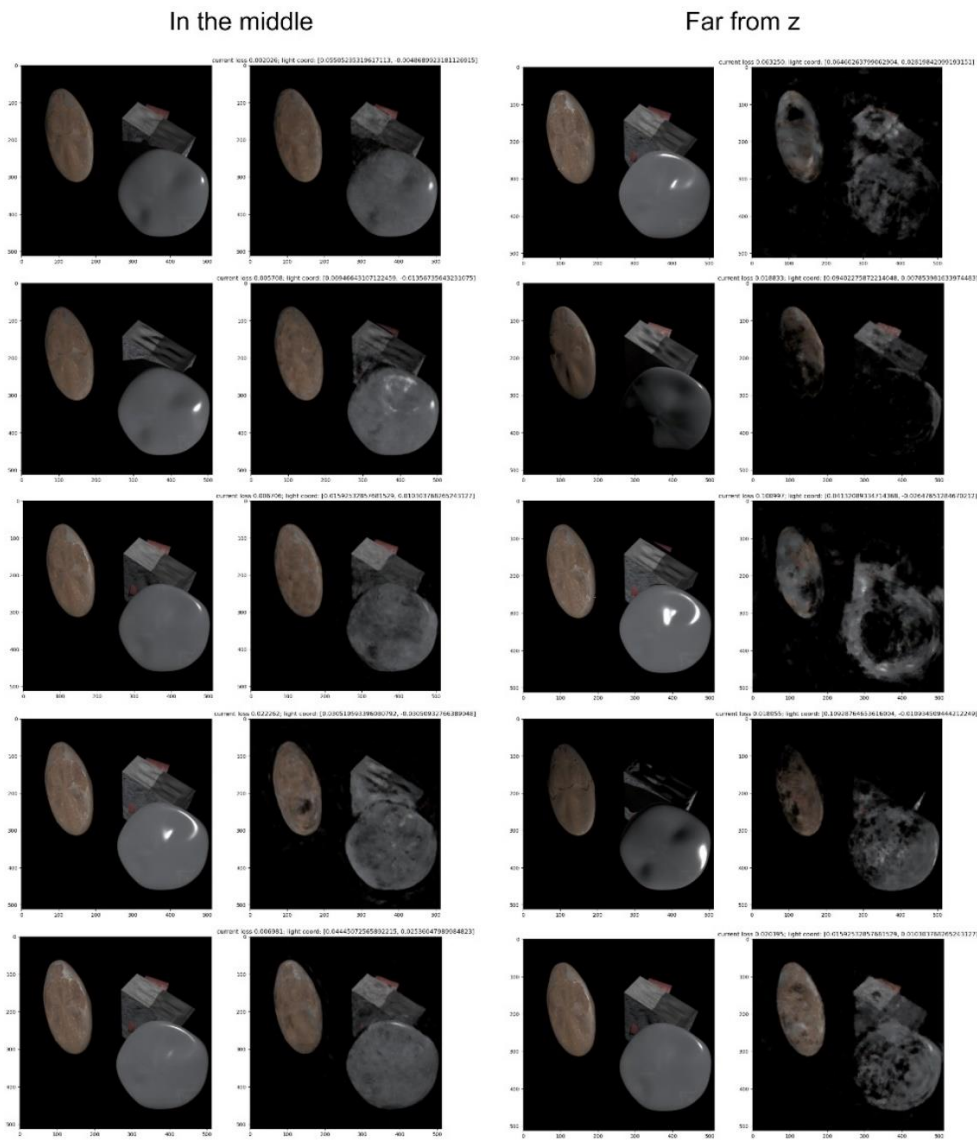


Figure 4.4: Testing results for “in the middle” and “far from z” images.

We can easily see from the comparison that the image quality drops as φ gets larger. The loss between the generated output and their ground truth also shows a similar trend. To look into the differences more clearly, we compared the results of the testing light position (0.01592532857681529, 0.010303768265243127), which was tested in all three sets, as shown in Figure 4.5.



Figure 4.5: Comparing results of the three sets with the same ground truth.

What we noticed here is that the quality decreasing shows a similar tendency as we were moving the new light position slightly away from the input light position before we compressed the light coordinates. As a result, we deduced that the reason behind this situation is the same, where the network is overfitted to the input images. As φ becomes larger, the covered area is larger and the test cases are further away from the input images, which is harder for the network to interpolate. A possible solution for this problem could be dividing φ and θ by a larger number, which would bring the light coordinates even closer in space, as we have done in our method. Another possible solution is to increase the number of input images. In this way, the network will have more light coordinate references in the area covered by training and the gaps between each input images will be shortened.

5. CONCLUSION

Generally, our research provides a method that leverages implicit neural representation in image-based relighting by abstractly modeling the image coordinates and light coordinates to the color channels of the images. To achieve this, we first set up SIREN so that it would take in both image pixel coordinates and light coordinates, and map the coordinates to RGB channels at each pixel as the output. Then, we trained the network with a set of input images and saved the best training model, which has the minimal loss out of all the training epochs. Finally, we loaded the saved best model and tested the algorithm with a new light position. At first, the result looked faulty because the network is overfitted to the input images, so we tried to optimize the testing results by dividing the light coordinates by a constant, in order to bring these positions closer so that it's easier for the network to simulate. After compressing the light coordinates, the general results are promising.

We did a series of testing in three aspects: how smooth the transition is between different light coordinates, how well the output quality is comparing to the ground truth, and how would the distance between the input images affect the results. From the first two aspects, our testing results seem to be generally positive, since the image quality stays good in the transition and the comparison to the ground truth is reasonable. However, the third aspect shows some limit. The network tends to overfit to the input images when the distance between the input light coordinates becomes larger in space, so the output quality drops.

Overall, our method gives a promising result. In future related researches, we could address the limits by either dividing the light coordinates by a larger number, or expanding the

input dataset and include more images. Either way, each method will bring the light coordinates closer to each other in space and refine the results.

REFERENCES

- [1] V. Sitzmann, “Awesome Implicit Representations - A curated list of resources on implicit neural representations,” *GitHub*, 28-Dec-2020. [Online]. Available: <https://github.com/vsitzmann/awesome-implicit-representations>. [Accessed: 27-Feb-2022].
- [2] V. Sitzmann, J. N. P. Martel, A. W. Bergman, D. B. Lindell, en G. Wetzstein, “Implicit Neural Representations with Periodic Activation Functions”, *CoRR*, vol abs/2006.09661, 2020.
- [3] Z. Xu, K. Sunkavalli, S. Hadap, en R. Ramamoorthi, “Deep image-based relighting from optimal sparse samples”, *ACM Transactions on Graphics*, vol 37, bll 1–13, 07 2018.
- [4] S. Bi *et al.*, “Neural Reflectance Fields for Appearance Acquisition”, *CoRR*, vol abs/2008.03824, 2020.
- [5] B. Mildenhall, P. P. Srinivasan, M. Tancik, J. T. Barron, R. Ramamoorthi, en R. Ng, “NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis”, *CoRR*, vol abs/2003.08934, 2020.
- [6] T. Sun, K.-E. Lin, S. Bi, Z. Xu, en R. Ramamoorthi, “NeLF: Neural Light-transport Field for Portrait View Synthesis and Relighting”, *CoRR*, vol abs/2107.12351, 2021.
- [7] Z. Xu, S. Bi, K. Sunkavalli, S. Hadap, H. Su, en R. Ramamoorthi, “Deep View Synthesis from Sparse Photometric Images”, *ACM Trans. Graph.*, vol 38, no 4, Jul 2019.
- [8] DeepAI, “Multilayer Perceptron,” *DeepAI*, 17-May-2019. [Online]. Available: <https://deepai.org/machine-learning-glossary-and-terms/multilayer-perceptron>. [Accessed: 22-Feb-2022].
- [9] V. Say, “How to save and load a model in pytorch with a complete example,” *Medium*, 28-May-2021. [Online]. Available: <https://towardsdatascience.com/how-to-save-and-load-a-model-in-pytorch-with-a-complete-example-c2920e617dee>. [Accessed: 27-Feb-2022].