Rendered Image Denoising using Autoencoders

Training your own AI Denoiser

This lab introduces an AI-based image denoiser that dramatically speeds up the removal of noise without a significant loss in image quality. It will be your guide as you explore how the denoiser is implemented and learn to train your own.

This tutorial covers the following topics:

Introduction: What is image noise?

Ray tracing (https://en.wikipedia.org/wiki/Ray_tracing_(graphics) generates an image by tracing many paths of light and simulating the effects of their encounters with virtual objects. Effects such as reflections and shadows are a natural result of ray tracing, making it possible to generate photorealistic imagery.

Ray-traced imagery, however, takes more time to render than other rendering algorithms such as rasterization. Rasterization, for example, uses data coherence to share computations between pixels. Ray tracing, on the other hand, typically re-starts the process, treating each eye ray separately. A pixel's color is actually the result of averaging the colors of many individual point-samples located within the quare region of the image plane that is represented by this pixel. As more samples are accumulated, each pixel's color is progressively replaced by newer values representing more samples, converging towards a final result. Between samples, not only the position inside its pixel is randomized, but many choices during tracing of the ray are made

differently in a probabilistic way, e.g. one sample might only evaluate visibility of one light source, while a second sample might evaluate only a second light source. The algorithms are carefully constructed to converge towards a correct result over time, but as you might expect, intermediate images show artificats, commonly referred to as "noise" because not enough information has been collected to determine the pixel's final color. To the naked eye, these pixels appear to be mis-colored.

Samples per pixel is used to measure the progress of the light simulation in ray tracing. As shown in the following examples, image generation begins with a low sampling rate, where the color of pixels may give little hint of their final coloration. As shown in the following images, progressively higher sample rates result in fewer miss-colored pixels. As you can see in these images, certain effects, such as shadows and reflections, require high rates of sampling in order to converge to a photorealistic final image or beauty shot. Note that the very early iterations look darker, which indicates that for many pixels there has not been a single sample found that successfully connected the pixel to a light source.

(inference_scenes/bistro/rgb/bistro_000001.png)
1 sample per pixel

(inference_scenes/bistro/rgb/bistro_000008.png)
8 samples per pixel

(inference_scenes/bistro/rgb/bistro_000016.png)
16 samples per pixel

(inference_scenes/bistro/rgb/bistro_000032.png)
32 samples per pixel

NVIDIA has developed an AI-accelerated denoiser to speed-up the denoising process without sacrificing much image detail. The denoiser included with OptiX 5.0 detects noise in a path-traced image and fills in the missing, not yet computed, information by using information from neighbouring pixels. By shortening the wait time for final images, interactive experience of users is significantly enhanced.

The following tutorial gives you hands-on experience with the same tools that NVIDIA used to create the denoiser. After completing this tutorial, you should be able to train your own AI accelerated denoiser, one that is good at detecting the noise patterns resulting from your custom sampling algorithm.

Note: No attempt has been made to train the neural network for all possible sampling algorithms. It is possible that AI denoising may not work well in all cases.

Exercise 1: Seeing and measuring noise

Task: Image noise can be seen as oddly colored pixels in an image. It may look similar to grain found in film photographs but can also look like splotches of discoloration. Areas of low light are particularly suceptible to oddly colored pixels. In this task, you identify levels of noise in imagery and learn how to objectively measure this noise.

Exercise 1a: Seeing noise in imagery

Below are three images with high sampling rate. Look for areas in the images where pixels may still be colored incorrectly. If an entire image looks like it is colored correctly, mark it as noise free. If an image displays what appear to be incorrectly colored pixels, mark it as containing noise.

```
Image 1:   Has Noise [ ]            Is Noise Free [ ]
```

Image 1

(inference_scenes/explorerDesk/rgb/explorerDesk_001024.png)

In [ ]:

```
Image 2:   Has Noise [ ]            Is Noise Free [ ]
```

Image 2

(inference_scenes/explorerDesk/rgb/explorerDesk_008192.png)

```
Image 3:   Has Noise [ ]            Is Noise Free [ ]
```

Image 3

(inference_scenes/explorerDesk/rgb/explorerDesk_016384.png)

The correct answer for all three images is "Has Noise". In other words, all three images contain some miscolored pixels. If you have the chance, ask a colleague to evaluate the images for noise and see if you come to the same conclusions.

Exercise 1b: Quantifying noise in imagery

Human sensitivity to noise varies naturally. In order to compensate for individual variation, in this lab a quantitative measure called *structural similarity* SSIM (https://en.wikipedia.org/wiki/Structural_similarity) is used. It measures the similarity between two images. In this lab, the subject image is being compared against the same image rendered to 131,072 samples per pixel. Moreover in this lab the SSIM value is a percentage where 100% means the images are identical.

Run the next cell to see what happens when you compare an image that was rendered to 131,072 samples per pixel against itself.

In [1]:
```
!./ssim inference_scenes/explorerDesk/rgb/explorerDesk_131072.png \
inference_scenes/explorerDesk/rgb/explorerDesk_131072.png
```

```
(R, G & B SSIM index)
100%
100%
100%
```

Equipped with this knowledge, take another look at the preceding three images:

The first image was rendered to 1,024 samples per pixel. For most people it is clear that this image contains noise. What do you think the SSIM value of this image will be when compared to our reference image rendered to 131,072 samples per pixel? Run the next cell to see the SSIM value.

In [2]:
```
!./ssim inference_scenes/explorerDesk/rgb/explorerDesk_131072.png \
inference_scenes/explorerDesk/rgb/explorerDesk_001024.png
```

```
(R, G & B SSIM index)
72.367%
76.6482%
76.8468%
```

The second image was rendered to 8192 samples per pixel. Run the next cell to see the SSIM value of that image compared to the reference image rendered to 131,072 samples per pixel.

In [3]:
```
!./ssim inference_scenes/explorerDesk/rgb/explorerDesk_131072.png \
inference_scenes/explorerDesk/rgb/explorerDesk_008192.png
```

```
(R, G & B SSIM index)
93.412%
95.141%
95.2165%
```

Based on past experience working with the denoiser, images with an SSIM score of approximately 90 still displayed some noise. However, images with a SSIM score over 95 were subjectively indistinguishable from the perfect image. Therefore, the second image is at the tipping point where most people will identify it as being noise free and perfect for creative decision making. However, some people will point to the liquids in the glass bottles, especially the red liquid in the glass on the left, and point out the noise.

The third image was rendered to 16,384 samples per pixel. Most people identify this image as noise free. However, there is still some noise in the liquids in the glass bottles. Run the next cell to see the SSIM value of that image compared to the image rendered to 131,072 samples per pixel.

In [4]:
```
!./ssim inference_scenes/explorerDesk/rgb/explorerDesk_131072.png \
inference_scenes/explorerDesk/rgb/explorerDesk_016384.png
```

```
(R, G & B SSIM index)
96.5955%
97.5334%
97.5238%
```

The following table summarizes the maximum sampling rate and the approximage SSIM value for each of the three images:

IMAGE	Samples per pixel	SSIM
Image 1 (inference_scenes/explorerDesk/rgb/explorerDesk_001024.png)	1,024	75
Image 2 (inference_scenes/explorerDesk/rgb/explorerDesk_008192.png)	8,192	94
Image 3 (inference_scenes/explorerDesk/rgb/explorerDesk_016384.png)	16,384	97
Reference Image (inference_scenes/explorerDesk/rgb/explorerDesk_131072.png)	131,072	100

Exercise 2: Denoising your first image

Recall that working with deep neural networks is a two step process: training and inference (https://devblogs.nvidia.com/parallelforall/inference-next-step-gpu-accelerated-deep-learning/). You will use inference to denoise your first images in this exercise using the trained parameters of the denoiser to classify, recognize, and process unknown images.

We have provided you with 3 test images to use for inference. The bistro, explorerdesk and SLK. These images are stored in the inference_scenes (../tree/inference_scenes) directory.

Bistro **Explorer Desk**


(inference_scenes/bistro/rgb/bistro_131072.png)


(inference_scenes/explorerDesk/rgb/explorerDesk_131072.png

Use the line command `nvdenoise` to specify the model, the noisy image, and the output directory:

```
nvdenoise -t model_path noisy_image_path -o output_image_path
```

To start, run the following example:

In [5]:
```
!./nvdenoise -t inference_models/training_500.bin \
inference_scenes/bistro/rgb/bistro_000032.png \
-o inference_result/bistro_000032_denoise_500.png
```

If you used the default output path, you can see your result in the inference_result (../tree/inference_result) directory. Run the next cell to measure the quality of that denoised image.

In [6]:
```
!./ssim inference_scenes/bistro/rgb/bistro_131072.png \
inference_result/bistro_000032_denoise_500.png
```

```
(R, G & B SSIM index)
76.1653%
77.7787%
78.1805%
```

Try running different scenes. Be sure to try different samples per pixel.

- Noisy images are stored in the inference_scenes (../tree/inference_scenes) directory.
- It is recommended that you store your output images in the inference_result (../tree/inference_result) directory.

The first two tasks have explored the denoiser and how to use an existing model to denoise an image. Next, you start training your own denoiser. But before doing that, let's take a closer look at the training data that you will use.

Overview: Required data for training a denoiser

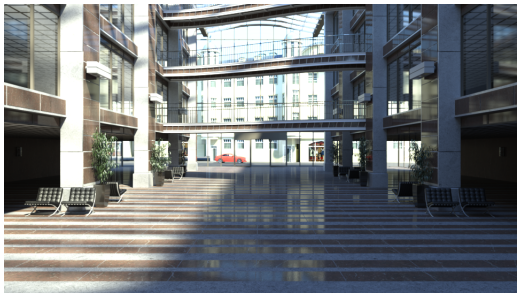
Let's take a look at the types of data that you will use to train your own denoiser network. For tutorial purposes, the data set is small --- about 460 images. To train a real-world denoiser, you will need at least 10,000 images. For example, the denoiser shipped with OptiX used training dataset with over 15,000 images and that training dataset continues to be expanded.

The training data is based on the following 10 scenes. Note that the test scenes are not part of the training set.


(training_data/rgb/JumpStreet_bedroom_131072.png)


(training_data/rgb/Matroschka_131072.png)


(trainin

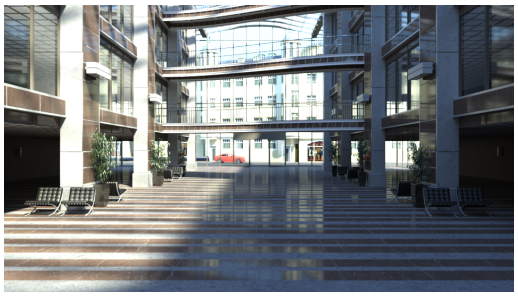

(training_data/rgb/AtriumTower_131072.png)

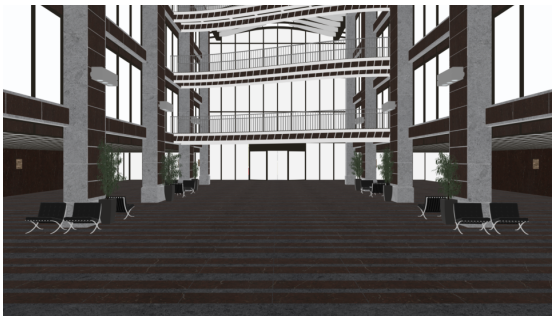

(training_data/rgb/BMW_131072.png)

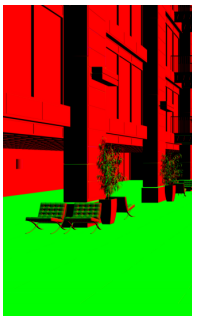

(training_da

For each scene, images were created for RGB, albedo, and normals channels:

RGB **Albedo**


(training_data/rgb/AtriumTower_131072.png)


(training_data/albedo/AtriumTower_131072.png)


(training_data/nc

Note:

- The albedo image represents an approximation of the color of the surface of the object, independent of view direction and lighting conditions. For simple materials with a single color texture, the contents of that texture can be used as-is for creating the albedo image. This information is needed by the denoiser to differentiate, for example, between noise, and a pattern that is actually part of the texture.
- A normal map is stored for the normals parameter. This image stores a direction at each pixel, represting x,y, and z components of the direction in one color channel each. Normals are used to determine edges.

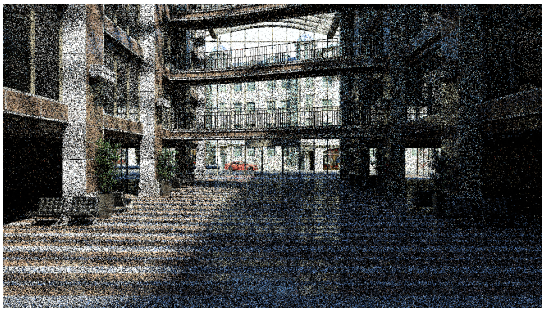
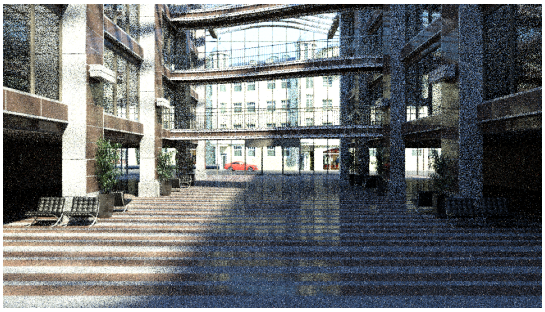
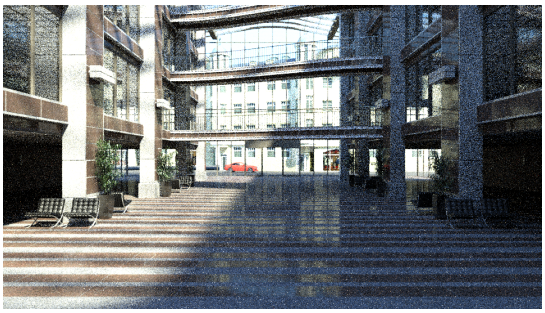
For each channel, sixteen images were rendered at a resolution of 960x540 pixels:

- One noise-free reference image at 131,072 samples per pixel
- Fifteen intermediate images were saved at 1, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768 and 65536 samples per pixel.

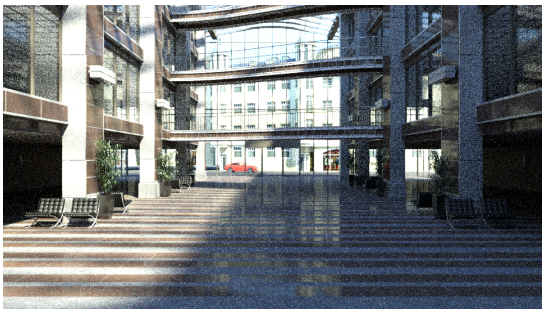
The images are stored in three directories: rgb, albedo, and normal. For training purposes, the image files were converted from .png to the NumPy array format.

Note: You will find the source files for converting .png files to numpy format in the training_convert (../tree/training_convert) directory.

The denoiser training script takes sets of two images and feeds them to the autoencoder (https://en.wikipedia.org/wiki/Autoencoder). Each set has a noisy image X and a noise-free image Y for a total of 15 sets. The 15 RGB sets for the atrium scene are shown in the following table:

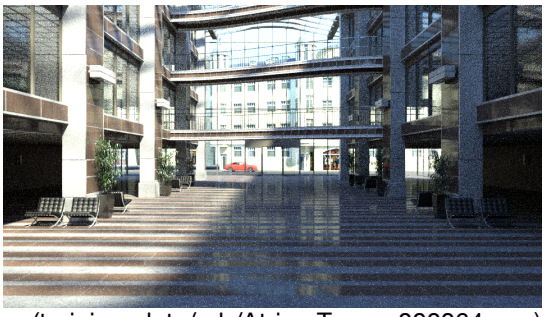
	noisy image X	noise free image Y
Set 1 1 sample per pixel (left) 131072 samples per pixel (right)	 (training_data/rgb/AtriumTower_000001.png)	 (training_data/rgb/AtriumTower_131072.png)
Set 2 8 sample per pixel (left) 131072 samples per pixel (right)	 (training_data/rgb/AtriumTower_000008.png)	 (training_data/rgb/AtriumTower_131072.png)
Set 3 16 sample per pixel (left) 131072 samples per pixel (right)	 (training_data/rgb/AtriumTower_000016.png)	 (training_data/rgb/AtriumTower_131072.png)

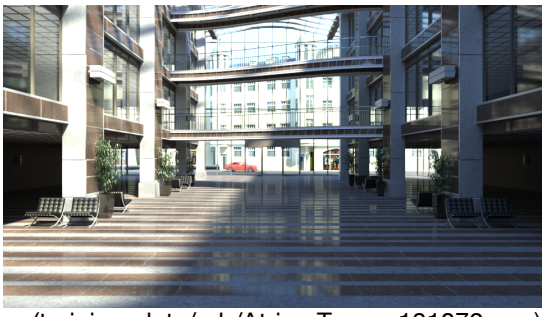
Set 4
32 sample per pixel (left)
131072 samples per pixel (right)


(training_data/rgb/AtriumTower_000032.png)

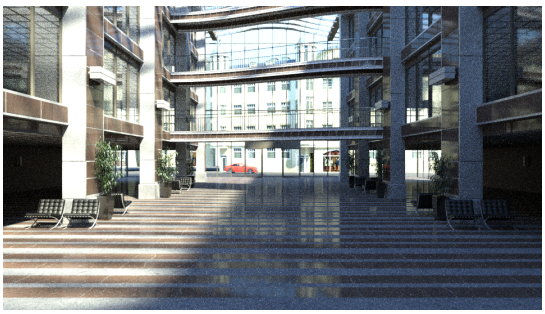

(training_data/rgb/AtriumTower_131072.png)

Set 5
64 sample per pixel (left)
131072 samples per pixel (right)


(training_data/rgb/AtriumTower_000064.png)

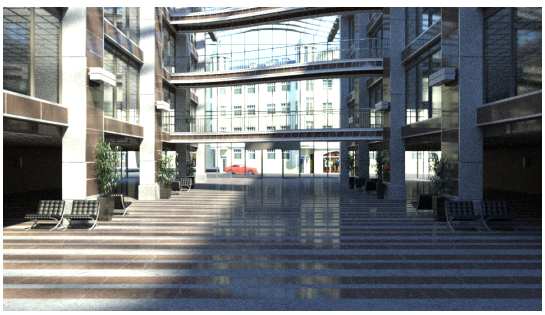

(training_data/rgb/AtriumTower_131072.png)

Set 6
128 sample per pixel (left)
131072 samples per pixel (right)


(training_data/rgb/AtriumTower_000128.png)

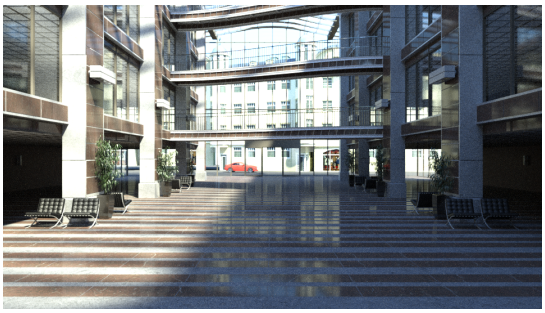

(training_data/rgb/AtriumTower_131072.png)

Set 7
256 sample per pixel (left)
131072 samples per pixel (right)


(training_data/rgb/AtriumTower_000256.png)


(training_data/rgb/AtriumTower_131072.png)

Set 8
512 sample per pixel (left)
131072 samples per pixel (right)


(training_data/rgb/AtriumTower_000512.png)


(training_data/rgb/AtriumTower_131072.png)

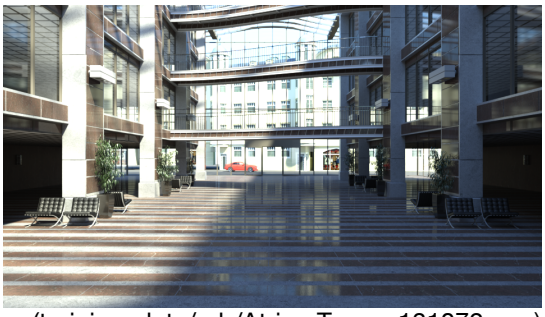
Set 9
1024 sample per pixel (left)
131072 samples per pixel (right)


(training_data/rgb/AtriumTower_001024.png)


(training_data/rgb/AtriumTower_131072.png)

Set 10
2048 sample per pixel (left)
131072 samples per pixel (right)


(training_data/rgb/AtriumTower_002048.png)


(training_data/rgb/AtriumTower_131072.png)

Set 11
4096 sample per pixel (left)
131072 samples per pixel (right)


(training_data/rgb/AtriumTower_004096.png)


(training_data/rgb/AtriumTower_131072.png)

Set 12
8192 sample per pixel (left)
131072 samples per pixel (right)


(training_data/rgb/AtriumTower_008192.png)


(training_data/rgb/AtriumTower_131072.png)

Set 13
16384 sample per pixel (left)
131072 samples per pixel (right)


(training_data/rgb/AtriumTower_016384.png)


(training_data/rgb/AtriumTower_131072.png)

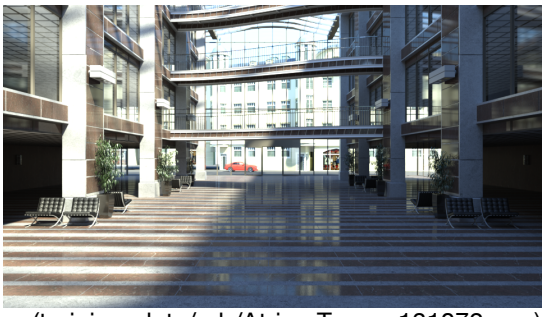
Set 14
32768 sample per pixel (left)
131072 samples per pixel (right)


(training_data/rgb/AtriumTower_032768.png)


(training_data/rgb/AtriumTower_131072.png)

Set 15
65536 sample per pixel (left)
131072 samples per pixel (right)


(training_data/rgb/AtriumTower_065536.png)


(training_data/rgb/AtriumTower_131072.png)

Now, let's really starting training your denoiser using this dataset.

Exercise 3: Training and packaging your denoiser

Task: In this exercise, you train the denoiser using the RGB training images and package the output as a binary file. You will use the binary file to denoise images.

Procedure:

Step 1 - Train your denoiser by running the training script (../edit/training_run.py) in the following cell. By default, the script trains the denoiser using the RGB images only.

In [7]:
```
!python training_run.py
```

```
Complete
Epoch   1 - Learn rate: 0.001000 - train loss: 0.16153 - time 12640.9
ms (remaining 4.0 minutes)
Progress: |xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx| 100.0%
Complete
Epoch   2 - Learn rate: 0.001000 - train loss: 0.08109 - time 9790.7 m
s (remaining 2.9 minutes)
Progress: |xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx| 100.0%
Complete
Epoch   3 - Learn rate: 0.001000 - train loss: 0.06855 - time 9768.7 m
s (remaining 2.8 minutes)
Progress: |xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx| 100.0%
Complete
Epoch   4 - Learn rate: 0.001000 - train loss: 0.05799 - time 9788.9 m
s (remaining 2.6 minutes)
Progress: |xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx| 100.0%
Complete
Epoch   5 - Learn rate: 0.001000 - train loss: 0.05998 - time 9817.5 m
s (remaining 2.5 minutes)
Progress: |xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx| 100.0%
```

Click here (/tensorboard/) to start TensorBoard.

You can look at the training results of each of the layers in SCALARS as it's training. Under GRAPHS tab, the Autoencoder network and connections can be seen.

```
What is the layer name of the Autoencoder's output?

Answer: Fill-in

Can you tell which layers make up the encoder vs decoder to create the
full Autoencoder? If yes, list them below:

Encoder Layers: Fill-in

Decoder Layers: Fill-in
```

Step 2 - Package the training output as a binary file by running the bash script (../edit/genlayer.sh) in the following cell:

In [8]:
```
!./genlayer.sh training_result 20 rgb
```

```
Converted 34 variables to const ops.
154 ops in the final graph.
---> set: training_result/model_20.pb name: rgb
0 conv1 (896,) [32  3  3  3] 32
1 conv1b (9248,) [32 32  3  3] 32
2 conv2 (12716,) [44 32  3  3] 44
3 conv3 (22232,) [56 44  3  3] 56
4 conv4 (38380,) [76 56  3  3] 76
5 conv5 (68500,) [100  76   3   3] 100
6 conv6 (240920,) [152 176   3   3] 152
7 conv6b (208088,) [152 152   3   3] 152
8 conv7 (209776,) [112 208   3   3] 112
9 conv7b (113008,) [112 112   3   3] 112
10 conv8 (118020,) [ 84 156   3   3] 84
11 conv8b (63588,) [84 84  3  3] 84
12 conv9 (66880,) [ 64 116   3   3] 64
13 conv9b (36928,) [64 64  3  3] 64
14 conv10 (38656,) [64 67  3  3] 64
15 conv10b (18464,) [32 64  3  3] 32
16 conv11 (867,) [ 3 32  3  3] 3
wrote training_result/training_20.bin
```

Exercise 4: Denoising an image with your new model

Task: In this exercise, you denoise an image with the new model you just trained and measure the SSIM image quality.

Procedure:

1 - Denoise the image `bistro_000032.png` by running the script in the following cell.

In [9]:
```
!./nvdenoise -t training_result/training_20.bin \
inference_scenes/bistro/rgb/bistro_000032.png \
-o inference_result/bistro_000032_denoise_20.png
```

The denoised image is stored in inference_result (../tree/inference_result) as `bistro_000032_denoise_20.png`.

2 - Test the SSIM value of your image by running the script in the following cell:

In [10]:
```
!./ssim inference_scenes/bistro/rgb/bistro_131072.png \
inference_result/bistro_000032_denoise_20.png
```

```
(R, G & B SSIM index)
64.2009%
67.2897%
68.7326%
```

How does your model compare to the model that you used in Exercise 2?

Exercise 5: Improving the quality of your denoiser

In this exercise, you learn how to improve the quality of your training data. The Overview section describes the arguments for **training_run.py (../edit/training_run.py)** and **genlayer.sh (../edit/genlayer.sh)**. Following the Overview section are four exercises, where you perform the following tasks:

- 5a. Train the network to 50 epochs using RGB and albedo inputs
- 5b. Package the results as a binary file
- 5c. Denoise images using RGB and albedo data
- 5d. Determine the SSIM value of denoised images

Overview

training_run.py

For **training_run.py (../edit/training_run.py)**, you can specify the following arguments:

```
--dimension {3,6,9}
        3 is rgb only (default)
        6 is rgb + albedo
        9 is rgb + albedo + normal


--save e
        Write intermediate checkpoint files every e epochs.


--train dir
        The path to the training input directory where rgb [albedo,
```

```
normal] data are stored.


--result dir
        The location of the output files which contain the trained
tensorflow model as well as metadata of the model.
        By default the files are stored in the training_result folder.
Check that any alternate directory path you
        specify actually exists. The script will not create it for you.


--epochs e
        The total number of epochs where one epoch is equivalent to one
training pass


--restore checkpointfile
        Use checkpointfile to continue training after it has been
paused.
```

genlayer.sh

For the **genlayer bash script (../edit/genlayer.sh)**, you can specify the following arguments:

- The **results_folder**
- The **tensorflow_checkpoint_number**, which in most cases is the number of epochs
- The **model_dimension** which can be one of the following: {rgb, rgb-albedo or rgb-albedo-normal}

Example: **./genlayer.sh training_result 30 rgb-albedo**

This example script does the following:

- Reads the data in the training_result directory
- Takes the network that was saved after 30 epochs and generates a .bin file called **training_30.bin**, which takes rgb and albedo as arguments.

The generated .bin file contains all the weights learned from the training you did on your network. You can then provide the new .bin file as an argument to the nvdenoise process exactly as you did in Exercise 2.

ssim

For the **`ssim`** executable, you specify two arguments:

- The **`reference image`**
- The **`image you want to compare`**

Example: **`./ssim bistro_131072.png bistro_000032_denoise_20.png`**

Exercise 5a

Task: In this task, you train a network to 50 epochs on both RGB and albedo inputs.

Procedure: In the following code block, replace the **`???`** snippets with your own arguments.

Do not specify more than 50 epochs during a live lab because it will take about 10 minutes to run the training on 50 epochs and if you specify a higher number of epochs you may not have enough time to complete the lab.

To train the denoiser included with OptiX we used the NVIDIA DGX-1 (https://www.nvidia.com/en-us/data-center/dgx-1/) to accelerate our training times.

In [11]:
```
!python training_run.py ???
```

```
Start training: dim=3, train dir=training_data, save dir=training_resu
lt, epochs=20

Number of training files: 150
Progress: |xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx| 100.0%
Complete
Epoch   1 - Learn rate: 0.001000 - train loss: 0.16156 - time 12550.5
ms (remaining 4.0 minutes)
Progress: |xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx| 100.0%
Complete
Epoch   2 - Learn rate: 0.001000 - train loss: 0.07770 - time 10012.1
ms (remaining 3.0 minutes)
Progress: |xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx| 100.0%
Complete
Epoch   3 - Learn rate: 0.001000 - train loss: 0.06482 - time 10021.0
ms (remaining 2.8 minutes)
Progress: |xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx| 100.0%
Complete
Epoch   4 - Learn rate: 0.001000 - train loss: 0.05706 - time 10008.7
ms (remaining 2.7 minutes)
```

```
Progress: |xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx| 100.0%
Complete
Epoch   5 - Learn rate: 0.001000 - train loss: 0.05821 - time 10019.4
ms (remaining 2.5 minutes)
Progress: |xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx| 100.0%
Complete
Epoch   6 - Learn rate: 0.001000 - train loss: 0.06901 - time 10030.9
ms (remaining 2.3 minutes)
Progress: |xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx| 100.0%
Complete
Epoch   7 - Learn rate: 0.001000 - train loss: 0.06380 - time 10031.3
ms (remaining 2.2 minutes)
Progress: |xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx| 100.0%
Complete
Epoch   8 - Learn rate: 0.001000 - train loss: 0.05817 - time 10015.6
ms (remaining 2.0 minutes)
Progress: |xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx| 100.0%
Complete
Epoch   9 - Learn rate: 0.001000 - train loss: 0.04875 - time 10029.6
ms (remaining 1.8 minutes)
Progress: |xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx| 100.0%
Complete
Epoch  10 - Learn rate: 0.001000 - train loss: 0.05006 - time 10029.6
ms (remaining 1.7 minutes)
Progress: |xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx| 100.0%
Complete
Epoch  11 - Learn rate: 0.001000 - train loss: 0.04159 - time 10031.7
ms (remaining 1.5 minutes)
Progress: |xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx| 100.0%
Complete
Epoch  12 - Learn rate: 0.001000 - train loss: 0.05764 - time 10061.2
ms (remaining 1.3 minutes)
Progress: |xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx| 100.0%
Complete
Epoch  13 - Learn rate: 0.001000 - train loss: 0.04964 - time 10022.9
ms (remaining 1.2 minutes)
Progress: |xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx| 100.0%
Complete
Epoch  14 - Learn rate: 0.000707 - train loss: 0.04767 - time 10019.9
ms (remaining 1.0 minutes)
Progress: |xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx| 100.0%
Complete
Epoch  15 - Learn rate: 0.000707 - train loss: 0.04197 - time 9997.8 m
s (remaining 50.0 seconds)
Progress: |xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx| 100.0%
Complete
Epoch  16 - Learn rate: 0.000707 - train loss: 0.04293 - time 10002.6
ms (remaining 40.0 seconds)
Progress: |xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx| 100.0%
Complete
```

```
Epoch  17 - Learn rate: 0.000707 - train loss: 0.03808 - time 10026.2
ms (remaining 30.1 seconds)
Progress: |xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx| 100.0%
Complete
Epoch  18 - Learn rate: 0.000707 - train loss: 0.03917 - time 9994.2 m
s (remaining 20.0 seconds)
Progress: |xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx| 100.0%
Complete
Epoch  19 - Learn rate: 0.000707 - train loss: 0.03864 - time 10027.2
ms (remaining 10.0 seconds)
Progress: |xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx| 100.0%
Complete
Epoch  20 - Learn rate: 0.000707 - train loss: 0.03925 - time 9989.9 m
s (remaining 0.0 seconds)
```

Exercise 5b

Task: In this task, you modify the arguments of the genlayer bash script (../edit/genlayer.sh) and generate a .bin file for the training that you did in the previous step.

Procedure: In the following code block, replace the **???** snippets with your own arguments.

In [12]:
```
!./genlayer.sh ???
```

```
use: genlayer.sh result-dir tensorflow_checkpoint_number {rgb, rgb-alb
edo or rgb-albedo-normal}
example: genlayer.sh ../result 100 rgb-albedo will generate a ../resul
t/training_100.bin training file for rgb-albedo input
```

Exercise 5c

Task: In this task, you denoise both a noise-free reference image and a noisy image using RGB, albedo, and normals data.

The **nvdenoise** has the following arguments:

nvdenoise -t [model_path] [noisy_rgb_image_path] **-o** [output_denoised_image_path] **-a** [albedo_image_input] **-n** [normals_image_input]

Procedure: In the following code block, replace the **???** snippets with your own arguments.

In [13]:
```
!./nvdenoise ???
```

```
training file not specified
```

For a list of sample scenes to test with, look in the inference scenes (../tree/inference_scenes) directory.

Suggestions:

- Denoise a noise-free image such as the reference image. What happens?
- Upload your own noisy image using the upload feature of this Jupyter notebook and denoise it.

Exercise 5d

Task: In this task, you test the SSIM value of your denoised images.

Procedure: In the following code block, replace the **`???`** snippets with your own arguments.

In [14]:
```
!./ssim ???
```

How does this new model compare to the model that you trained in Exercise 3?

Exercise 6: Take home assignments

You now have all the tools to train your own network and denoise images. Training is an iterative process where you continuousely add data and fine-tune the number of epochs and the types of inputs you are using to train your network. Following are exercises that are intended to give you additional insights into how to build a robust denoiser for your own renderer:

1. Train an RGB-only network to 50, 100, and 500 epochs.
2. Train a network with RGB and albedo data.
3. Train a network with RGB, albedo, and normals data.
4. Denoise the sample set of images with your different networks.
5. Denoise images with low sample counts.
6. Denoise images with higher sample counts.
7. Denoise images with a 500 epoch network trained with RGB, albedo, and normals data
8. Compare the preceding test with a 500 epoch network that was trained with RGB and albedo data only.
9. Measure the SSIM for each image you denoise.
10. What did you learn?
11. How can you improve your network?
12. Setup your own training environment and train your denoiser with your own training data.
13. Download the OptiX 5.0 SDK and use your custom network instead of the one built into OptiX. Does your custom network do a better job on your data?

When you are training your own network with your own data you will have to use a much higher number for epochs and many more images for the training data to get good results.

The training script is well documented so that you can download it and use it for training in your own denoiser in your own environment. Be sure to look at the system and software requirements to run the script.

When using the OptiX 5.0 SDK, you can also provide the .bin file as an argument to the post processing API so that the OptiX 5.0 denoiser uses your network to do the denoising instead of using the default network that ships with the SDK. For additional information on the OptiX API to specify your own network, refer to the Post Processing API documentation.

Epilogue: Characteristics common to deep-learning applications

Successful deep-learning applications, such as the denoiser shipped with Optix 5.0, are commonly identified by the following characteristics:

- **Dataset size:** Successful deep learning applications typically use large datasets. We used about 1000 different scenes and created a series of 16 progressive images for each scene. To train the denoiser, images were rendered from the scene data at 1 sample per pixel, then 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536, and 131072 samples per pixel. The same process was followed for beauty (final) images and for albedo images. In total, 32 images per scene were rendered. This means that about 32000 images were used to train the denoiser.
- **Reuse:** Successful deep learning applications have models that can be generalized. For denoising, NVIDIA used an auto encoder which leverages existing CNN architecture (https://en.wikipedia.org/wiki/Convolutional_neural_network) for feature extraction. It was assumed that the model could be generalized to understand rendering noise from many different scenes: specifically scenes that the model has never seen. If the model had to be retrained for every new scene, it would be impossible to reuse the model.
- **Feasibility:** Question: Is it possible to describe the problem as an X --> Y mapping? In the case of denoising, the following mapping was used: Given a noisy image X, train a model to uncover a mapping that can produce a noise-free image Y.
- **Payoff:** The tremendous time savings realized by reducing the computation time required to create a high-quality noise-free image.
- **Fault Tolerance:** Finally, deep learning creates statistical models and every model has some level of failure. Although failure in the denoiser may result in one or more miss-colored pixels, the overall feel or perception of noise for the scene is not critically affected.