UNIVERSITÄT
DES
SAARLANDES

# BACHELOR THESIS

## Denoising Monte-Carlo Noise with Deep Learning; A Rodent-Specific Approach


submitted by

Hendrik Junkawitsch



Saarbrücken, 2021
Faculty of Mathematics and Computer Science,
Department of Computer Science


| | |
|---|---|
| Supervisor: | Prof. Dr. Ing. Philipp Slusallek |
| Second Reviewer: | Dr. Ing. Thomas Leimkühler |

## Erklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

## Statement

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

## Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

## Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

| | |
|---|---|
| Date | Signature |

# Acknowledgements

First and foremost, I want to thank Prof. Dr. Ing. Philipp Slusallek and the computer graphics chair at Saarland University for the great opportunity to work on this topic. Furthermore, I want to thank my advisor Ömercan Yazici, M.Sc., for his patient support and valuable insight into different important key points.

Additionally, I want to acknowledge Joshua Meyer, who concurrently worked at the efficient implementation of the denoising approach in Rodent and was always willing to discuss various ideas.

Lastly, I want to thank my father, Dr. Ing. Jochen Junkawitsch, for proofreading the thesis, providing his honest opinion, and guiding me to a finished thesis.

# Abstract

In recent years, path tracing has seen major advances regarding efficient software implementation and hardware acceleration. However, creating noise-free images with Monte-Carlo renderers incorporating global illumination is still computationally expensive and almost not feasible in a reasonable amount of time. Deep Learning has been used to train denoising filters that can efficiently create visually good-looking images with little computational effort. This thesis investigates neural network architectures and learning methods used for denoising images and evaluates their performance. Training neural network models to remove Monte-Carlo noise is done with data sets created by the high-performance renderer *Rodent* to set up a Rodent-specific denoising approach. We train our models with little data to make learning manageable without large rendering clusters, examine generalizability, and compare different results. This thesis uses the off-the-shelf and open-source denoising library, Open Image Denoise by Intel, as the baseline for a hyperparameter analysis. The ultimate goal is to provide concise information and empirical results regarding Monte-Carlo denoising with deep learning.

# Contents

**Acknowledgements** **iii**

**Abstract** **v**

**1 Introduction** **1**

**2 Path Tracing** **3**
2.1 The Rendering Equation . . . 3
2.2 Solving the Rendering Equation . . . 4
2.2.1 The Finite Element Method . . . 4
2.2.2 Monte Carlo Integration . . . 4
2.3 Ray Tracing . . . 5
2.4 Noise Reduction . . . 6

**3 Denoising** **9**
3.1 Pixel- and Sample-based Denoising . . . 9
3.2 "Hand-Coded" Algorithms . . . 10

**4 Deep Learning** **11**
4.1 Neural Network Architecture . . . 11
4.1.1 Neurons . . . 11
4.1.2 Multilayer Perceptrons . . . 12
4.1.3 Convolutional Neural Networks . . . 13
4.1.4 Convolutional Autoencoders . . . 19
4.1.5 U-Nets . . . 20
4.2 Loss Functions . . . 21
4.3 The Training Process . . . 22
4.3.1 Stochastic Gradient Descent . . . 22
4.3.2 Overfitting and Underfitting . . . 23
4.4 Learning Rate Scheduling . . . 24
4.4.1 Adam . . . 25

**5 Existing Software** **27**
5.1 Rodent . . . 27
5.2 Open Image Denoise by Intel . . . 28
5.2.1 Neural Network Model . . . 28
5.2.2 Problems with OIDN . . . 28

**6 Data Setup** **31**
6.1 Albedo and Normal Images . . . 31

6.2 Diffuse and Specular Images . . . . . 32
6.3 Data Set Generation . . . . . 33
6.4 Data Augmentation . . . . . 35

**7 Empirical Hyperparameter Evaluation** **37**
7.1 Overfitting and Underfitting . . . . . 37
7.2 The Loss Function . . . . . 40
7.3 The Optimization Technique . . . . . 43
7.4 The Auxiliary Feature Image Selection . . . . . 44
7.5 The Network Architecture . . . . . 46
7.5.1 MLP Denoising . . . . . 46
7.5.2 OIDN Architecture Modifications . . . . . 47

**8 Denoising Quality Comparison** **51**
8.1 The Cornellbox . . . . . 52
8.2 The Office Scene . . . . . 53
8.3 The Spaceship Scene . . . . . 54

**9 Conclusion and Future Work** **55**

**Bibliography** **57**

**List of Figures** **61**

**List of Tables** **63**

# 1 Introduction

The generation of realistic images from a non-physical environment was and still is the particular focus of research in universities and companies. This follows from the great demand for efficient and feature-rich renderers. Renderers have to reproduce scenes exactly as the human visual system perceives them to create realistic-looking images. This process incorporates the physically correct modeling of objects and materials, realistic global illumination, and the simulation of various effects caused by the human eye. Unfortunately, the growing complexity of rendering algorithms often results in long rendering times, raising the need for efficient software solutions and hardware acceleration.

This thesis primarily considers noise produced by the *path tracing* algorithm and specifically the high-performance renderer *Rodent* implemented within the AnyDSL[1] compiler framework. Path tracing is an unbiased rendering algorithm with high variance recognizable as noise in the image. Creating noise-free images in feasible rendering times or in real-time for interactive applications is always coming at the expense of physical accuracy. Rendering physically correct images requires enormous computational power and, therefore, often results in unfeasible time consumption. Consequently, noise is frequently considered as the archenemy in realistic image synthesis. Even highly optimized renderers like Rodent generally need a tremendous amount of time to create relatively noise-free images.

There are several approaches to reduce the noise in a given amount of time. These techniques either directly reduce the noise produced by a renderer or denoise the images after rendering, thus indirectly reducing the noise. Direct methods range from carefully choosing sampling strategies and complex algorithm optimizations to modern hardware acceleration. In this thesis, we do not want to decrease the noise directly. We instead add a post-processing step that removes the noise from the final image. Denoising filters are commonly present in every state-of-the-art image editing software. However, the quality of the resulting image and time efficiency of those hand-coded denoising filters are often poor.

*Deep learning* has effectively been used to remove noise from images to create visually good-looking results. One of the most successful denoising solutions is Open Image Denoise published by Intel and based on deep neural networks. Unfortunately, even though Intel's approach is entirely open-source, there is no easy-to-access publicly available information about the exact training methods and parameters.

[1] `https://github.com/AnyDSL`

The main goal of this thesis is to provide a thorough study of different deep learning techniques to denoise Monte-Carlo rendered images. In the first chapters, we present the necessary theoretical and mathematical background of path tracing and Monte-Carlo integration to introduce the general Monte-Carlo denoising task and determine the source of Monte-Carlo noise. We identify various reasons for the necessity of advanced denoising approaches using deep learning methods. Then, based on the theoretical foundations of machine learning techniques used in image processing, we want to analyze neural network topologies, efficient learning strategies, and important parameters that enable successful denoising.

We compare neural networks trained with our training data set and learning strategies with the baseline Open Image Denoise and the non-machine-learning denoising algorithm BM3D. The importance of a large and general training data set is one of the key aspects of the final comparison.

# 2 Path Tracing

Path tracing is one of the simplest and most common rendering algorithms based on ray tracing. Monte-Carlo integration is the underlying mathematical concept and the origin of noise in rendered images. Research dates back to 1968 where Arthur Appel [App68] introduced the first ray-tracing algorithm. Since then, researchers have continuously pushed the limits of realistic image synthesis. To start our thesis, we will have a quick look at the mathematical and theoretical foundations of path tracing. Mainly we want to understand why path-traced images contain such a high amount of noise.

## 2.1 The Rendering Equation

In 1986, James T. Kajiya published a SIGGRAPH paper [Kaj86] and stated that every rendering algorithm wants to model the same physical phenomenon: light perception after complex reflection and refraction in space. He presented an integral equation that describes the simulation of light transport: *the rendering equation*[1].

$$L_o(x, \omega_o) = L_e(x, \omega_o) + \int_{\omega_i \in \Omega_+} L_i(x, \omega_i) f_r(\omega_i, x, \omega_o) \cos\theta_i \mathrm{d}\omega_i \tag{2.1}$$

The rendering equation (equation 2.1) is the mathematical foundation of almost every rendering algorithm and also of path tracing. It describes the outgoing radiance $L_o$ at a point $x$ in a direction $\omega_o$ as the sum of the emitted radiance $L_e$ at $x$ towards $\omega_o$ and the reflected radiance. The reflected radiance is the integral of the incident radiance $L_i$ from all directions $\omega_i$ multiplied by the bidirectional reflectance distribution function (BRDF) $f_r$ and the cosine of the incident angle $\theta_i$. Rendering algorithms like path tracing try to solve the Rendering Equation to achieve a good lighting simulation.

$L_i(x, \omega_i)$, i.e., the incoming radiance at a point $x$ from a direction $\omega_i$, in the integral is just $L_o(y, -\omega_i)$, i.e., the outgoing radiance from another point in the scene $y$ with direction $-\omega_i$. We can directly see this by reformulating the rendering equation using a visibility function $h$ or the ray tracing operator [SKP99].

$$L_o(x, \omega_o) = L_e(x, \omega_o) + \int_{\omega_i \in \Omega_+} L_o(\boldsymbol{h(x, -\omega_i)}, \omega_i) f_r(\omega_i, x, \omega_o) \cos\theta_i \mathrm{d}\omega_i \tag{2.2}$$

[1] The rendering equation presented here is not the original formulation introduced by James T. Kajiya but an equivalent one.

The visibility function $h(x, -\omega_i)$ maps to the hit point of a ray with origin $x$ and direction $-\omega_i$, hence *"ray tracing operator"*. In equation 2.2, the infinite recursive property is clearly visible because the incoming radiance in equation 2.1 is now the outgoing radiance of another surface $L_o(h(x, -\omega_i), \omega_i)$. Every point in a scene can potentially influence the light arriving at all other points. This property leads to a high dimensionality in practice. Mathematically the rendering equation is called a *Fredholm integral equation of the second kind*.

## 2.2 Solving the Rendering Equation

One of the main tasks in computer graphics is to solve the rendering equation. Since a Fredholm integral equation is hard to solve analytically, the rendering industry uses numerical methods. Essentially there are two common ways to solve the rendering equation.

### 2.2.1 The Finite Element Method

The *finite element method* is based on the radiosity formulation of the rendering equation. It aims to convert the integral equation into a discrete formulation of the rendering equation. The scene gets subdivided into mesh elements with constant illumination, and the final solution is approximated by solving a matrix equation. However, some problems make this approach impractical for most rendering tasks. Firstly the surfaces have to be subdivided into appropriate patches to visualize high-frequency illumination changes. This partitioning of the space into smaller patches can be arbitrarily hard. Another disadvantage is that the subdivision of surfaces into patches with constant illumination is an abstraction that leads to a biased solution. In general, the algorithm complexity, memory requirements, and little support for more advanced BRDFs is the reason why another method is used to solve the rendering equation in path tracing.

### 2.2.2 Monte Carlo Integration

*Monte Carlo integration* is the most common way to find solutions to a Fredholm integral equation. The technique is used in path tracing and constitutes the reason for the visible noise in rendered images. In the following section, we want to introduce the fundamental theory of Monte Carlo integration. For further understanding, refer to [Wei00].

The particular goal is to solve definite integrals numerically:

$$I = \int_\Omega f(x) \, \mathrm{d}x \tag{2.3}$$

The idea of Monte Carlo integration is to randomly sample $N$ points $x_1, \ldots, x_N \in \Omega$ according to a probability density function (PDF) $p(x)$, also called the importance function. The PDF ensures that more important samples are chosen with a higher probability[2].

[2] A constant PDF represents a uniform distribution.

If our PDF $p(x) \neq 0$ whenever $f(x) \neq 0$, we can rewrite our integral:

$$I = \int_\Omega f(x)\ \mathrm{d}x = \int_\Omega f(x)\frac{p(x)}{p(x)}\ \mathrm{d}x \tag{2.4}$$

We can clearly identify the definition of the expected value in the equation 2.4. Therefore, the expected value $\mathbb{E}\left[\frac{f(x)}{p(x)}\right]$ is the solution of the initial integral $I$.

We define a *primary estimator* $\langle F\rangle_1$ where $x$ is one random sample chosen according to our importance function $p(x)$:

$$\langle F\rangle_1 = \frac{f(x)}{p(x)} \tag{2.5}$$

Due to the *law of large numbers* [HR47], one can finally average many primary estimators to obtain the expected value of the primary estimator and, therefore, the final solution of the integral $I$. Ergo the result of the sampling process converges to the exact result. Therefore, Monte Carlo integration is an unbiased algorithm that yields the correct solution.

$$\lim_{n\to\infty}\left(\frac{1}{n}\sum_{i=1}^{n}\frac{f(x_i)}{p(x_i)}\right) = \mathbb{E}\left[\langle F\rangle_1\right] = I \tag{2.6}$$

The estimated error of Monte Carlo integration scales proportionally to $\frac{1}{\sqrt{N}}$ [Wei00]. In rendering, the variance of Monte-Carlo integration becomes visible as noise in the images. A significant advantage over a simple method using quadratures is that the error is entirely independent of the integral's dimensionality. This independence is particularly relevant in rendering, considering that the rendering equation has an infinite dimensionality.

The information we have to memorize after this section is that Monte Carlo integration is used in path tracing to solve the rendering equation. Therefore path tracing is an unbiased rendering algorithm that converges relatively slowly ($\sqrt{N}$) to the exact result. The convergence rate is the reason why rendering realistic images with no visible noise is time-demanding. Unfortunately, noise in path-traced images is still an enormous issue after more than 50 years of consistent research.

## 2.3 Ray Tracing

James T. Kajiya is also considered the inventor of the path tracing algorithm. Ray tracing is used to solve the rendering equation with Monte Carlo integration. Samples are collected by tracing rays for each pixel starting from the camera. Rays carry a color contribution that gets modulated every time the ray gets reflected or refracted by surfaces or hits a light source. The contribution finally represents the color of a pixel and ultimately provides a primary estimator (equation 2.5).

Repeating the process and averaging the pixel colors results in the Monte Carlo integration algorithm. Eventually, the image converges to the exact result (equation 2.6). Figure 2.1 visualizes the process of ray tracing in the path tracing algorithm. We trace a ray through the scene until it gets terminated with a certain probability or hits a light source. The procedure that randomly terminates the path is called Russian roulette and depends on the ray's current throughput. This dependency implies that darker paths statistically terminate earlier than brighter paths.

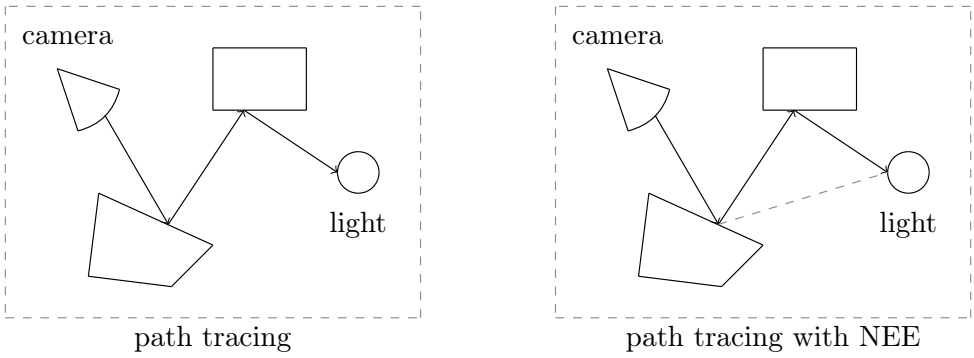



Figure 2.1: Visualization of path tracing with and without Next Event Estimation.

## 2.4 Noise Reduction

Noise in rendered images has been one of the dominant problems in the research history of computer graphics. Many improvements and optimizations have been developed to speed up the rendering process and consequently reduce the visible noise. We want to present the most basic ones before diving into the noise reduction approaches investigated in this thesis.

The most common enhancement of the classical path tracing algorithm is *Next Event Estimation* (NEE). One of the most significant issues of path tracing is that the color contribution of a ray path is non-zero only when the ray hits a light source. Depending on the scene, the probability of hitting a light source can be close to zero. When only using point light sources, the rays will never have a non-zero contribution. This characteristic leads to particularly noisy images. NEE solves this problem by directly connecting the ray path to a randomly chosen light source at every hit point. This technique ensures that the contribution of a path is rarely zero and thereupon drastically improves the convergence rate. Figure 2.1 shows the direct connection between the first ray hitpoint in the scene and the light source when using NEE.

Another way to reduce the variance of Monte Carlo integration in path tracing is *importance sampling* [KvD78]. We can reduce the variance enormously by intelligently choosing a probability density function similar to the desired integrand. A perfect PDF $p(x)$ that is proportional to the integrand $f(x)$ would result in zero variance. Of course, constructing such a PDF would already require the solution of the integral $\int f(x)\ dx$. A good approximation of the integrand can often be created by combining multiple functions via *Multiple Importance Sampling* (MIS). Refer to [VG95] for further information.

Other *sampling strategies* can also help to reduce variance [PJH16]. There exist many different sampling strategies, but we only mention a few important examples.

- *Stratified sampling*: Subdividing the domain into smaller patches and sampling in each patch separately reduces clustering and gaps between sampled points.
- *Poisson-Disk sampling*: Every sampled point is the center of a circular area. If the following samples fall into the area of an already existing sample, the point gets rejected.
- *Quasi Monte Carlo sampling*: Samples get selected according to a low-discrepancy sequence, as described in [KPR12].

Recent advancements in hardware acceleration also improved rendering times. There are many more approaches to improve the performance of Monte Carlo renderers. Nevertheless, Monte-Carlo noise is undeniably still one of the biggest challenges in modern computer graphics research.

# 3 Denoising

We already mentioned numerous noise-reducing methods for path tracing (chapter 2.4). All of them either directly improve the variance of Monte Carlo integration or increase the performance of the rendering process in another way. This thesis aims *not* to further reduce the noise produced by the renderer. We instead want to add a *denoising* post-processing step.

Denoising is not only targeting image restoration tasks. It essentially describes the process of removing any corruption from an input signal. The corruption is called noise and is, in most cases, a signal that overlaps or interferes with the original information [Vas08]. An ideal denoising process aims to perfectly separate the noise from the original signal.

We can classify the noise in this thesis as processing noise [Vas08]. The reason for this naming is the fact that path tracing is a kind of signal processing. The following sections address *Monte Carlo noise* caused by Monte Carlo integration in path tracing.

Before considering different denoising algorithms, we have to define the input signal. The input signal of our denoising algorithm is the noisy/corrupted data. There are essentially two different ways to define that signal in Monte Carlo rendering.

## 3.1 Pixel- and Sample-based Denoising

A path tracer provides a noisy image. The evident approach is to directly take the output image of the renderer as the corrupted signal. This method is called *pixel-space* or *pixel-based* denoising. In pixel-based denoising, the algorithm directly works with the pixel values of the rendered image, hence the name.

Another approach is called *sample-based* denoising. Instead of only using the averaged result of the Monte Carlo samples, sample-based denoising takes every Monte Carlo sample into account. The advantage of sample-based denoising is that a larger amount of information can be used to restore the noise-free image. The additional knowledge might improve the performance of the denoising algorithm. However, this would strictly require an explicit integration of the denoising process into the renderer and add additional complexity. Therefore we decided to work with the images produced by the renderer directly.

The input data $I$ is a tensor with dimension $width \times height \times 3$. An image consists of three $width \times height$ matrices representing the RGB color channels. $I_{gt}$ depicts the ground truth image without noise, i.e., the expected value of the primary estimator in the Monte Carlo integration (equation 2.6).

## 3.2 "Hand-Coded" Algorithms

We can describe denoising as a function $d$ that maps the noisy image $I$ to a denoised output image $I_d$.

$$d(I) = I_d \tag{3.1}$$

Optimally $I_d$ should be close to the ground truth $I_{gt}$. Chapter 4.2 will introduce various metrics that decide how close an image is to the ground truth. There has been much research regarding image denoising, and many different algorithms have been developed [FZFZ19]. All of them are carefully designed by hand. We can divide denoising methods into linear and non-linear spatial filtering, which directly works in the image space, and transform domain filtering, which first applies a domain transform. A standard non-linear spatial domain filtering method is bilateral filtering [TM98], and the most popular transform domain method is BM3D [DFKE07].

Hand-coded filters like the previously mentioned ones often have several problems. Simple linear spatial domain algorithms often smooth the image or blur edges. Thus non-linear filters have been introduced. However, the problem with non-linear methods, e.g., bilateral filtering, is large time consumption. Transform domain filtering also has some disadvantages like time consumption and hyperparameter setting. Manually deciding which denoising filter works best for Monte Carlo noise and choosing the best values for its parameters is hard, and thus another approach is introduced in the following chapter 4. We can use machine learning to avoid hand-coded denoising algorithms and train algorithms to recognize corruption.

# 4 Deep Learning

Deep learning is a successful technique applied to denoise images. Machine learning methods have been used to learn denoising filters instead of manually developing them. We can apply supervised learning to train models which represent the denoising function (equation 3.1) directly. Training is done by iterating over a data set and continuously optimizing the model with respect to a loss function. In the following sections, we will cover the foundation of deep learning methods used in this thesis. The sections explain the core building blocks and finally combine them into working neural network denoising approaches.

## 4.1 Neural Network Architecture

We train artificial neural network models that represent the denoising function when using deep learning as a denoising approach. *Neural networks* try to emulate the functioning of the human brain. They are highly parallelizable constructs that allow fast inference on modern GPUs or application-specific integrated circuits (ASICs). Generally, neural networks work by repeatedly optimizing internal weights and thus progressively learning a non-linear function. Once a model finishes training, we can use it to apply the learned non-linear function to input data. This is called *inference*. The model architecture is one of the most important design choices we have to make. Different topologies affect the ability to fit data successfully, highly influence generalizability, and determine the computational effort needed for inference.

### 4.1.1 Neurons

Every neural network builds upon the first mathematical definition of neurons [MP43]. These mathematical constructs resemble the biological neurons in the human brain. Connections between artificial neurons emulating synapses in the human brain establish networks that transport and modify signals. Figure 4.1 depicts the mathematical definition of an artificial neuron. Let $(x_0, x_1, \ldots, x_n)^T = x \in \mathbb{R}^n$ be the input values for the given neuron $\mathcal{N}$ and $y \in \mathbb{R}$ the output. $x_0$ is called a *bias* with a fixed input, usually 1. $(w_0, w_1, \ldots, w_n)^T = w \in \mathbb{R}^n$ are the learnable weights optimized during the neural network training process. Finally, $\sigma$ is the activation function (section 4.1.3).

The output $y$ of the neuron is then given by:

$$y = \sigma\left(\sum_{i=0}^{n} w_i x_i\right) \tag{4.1}$$

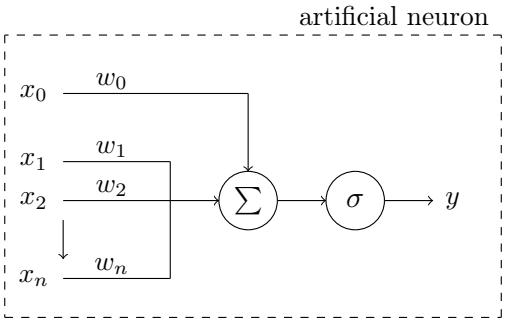



Figure 4.1: Depiction of an artificial neuron.

Any arbitrary network topology, meaning every possible conjunction of neurons, is called an artificial neural network. However, neurons are commonly organized in layers. Each layer is a set of neurons, which usually only connect to the neurons of the next layer. Therefore, the number of layers and neurons within a layer and the exact layout of the connections are hyperparameters and a critical design choice.

### 4.1.2 Multilayer Perceptrons

The *Multilayer Perceptron* (MLP) is one of the most superficial neural networks, which has its roots in work from F. Rosenblatt [VDM86]. An MLP comprises an input layer, an output layer, and several hidden layers. Each layer consists of multiple neurons connected to the neurons in the next layer. How exactly the neurons are connected is a crucial layout option. *Fully connected* layers often result in *overfitting*, while a lack of connections may hinder the network from learning relevant data [LL05]. Section 4.3.2 will describe the training process and overfitting more in-depth.

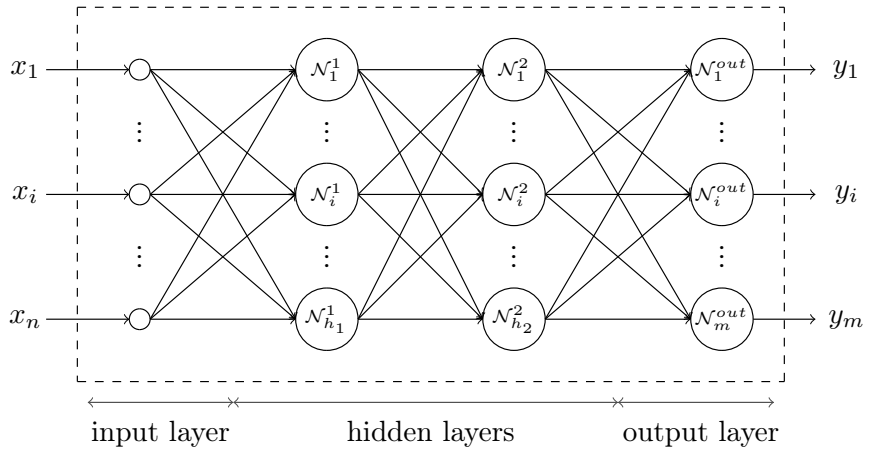



Figure 4.2: Depiction of a fully connected MLP with *two* hidden layers.

Figure 4.2 illustrates the architecture of a fully connected Multilayer Perceptron with two hidden layers, where

- $x = (x_1, \ldots, x_n)^T \in \mathbb{R}^n$ denotes the input vector with dimension $n$,

- $y = (y_1, \ldots, y_m)^T \in \mathbb{R}^m$ denotes the output vector with dimension $m$,
- $h_1$ and $h_2$ denote the number of neurons in the respective hidden layer.

We calculate the output of each neuron in the first hidden layer of the network (equation 4.1) and propagate the result to the neurons in the following layer. Repeating this process for every layer ultimately results in the output vector $y$ and is called the *forward pass* or *forward propagation*.

Optimizing the weights works with backpropagation in the *backward pass*. The goal is to minimize an error estimate, i.e., the loss between the desired and the actual output of the network. In section 4.3, we cover the basics of the weight learning process, but we also refer to other papers for different optimization strategies: [HN92], [PM92].

If we want to define an MLP, we must set specific constant parameters for the network model.

- $N :=$ number of hidden layers.
- $n_0 := |x| =$ number of input values.
- $n_{N+1} := |y| =$ number of neurons in the output layer.
- $h_i =$ number of neurons in the $i$-th hidden layer where $0 < i \leq N$.
- $\sigma :=$ activation function.
- A concrete connection pattern between the layers.

### 4.1.3 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are a subset of neural networks and take their origin from the human visual system. Therefore, they are especially well suited for 2-dimensional data such as images. Kunihiko Fukushima introduced the first kind of CNN, the *neocognitron* [FM82]. Small visual areas, called *receptive fields*, decide each neuron's activation. Consequently, neighboring pixels influence the output of a neuron in image processing. This characteristic is realized by integrating convolutional layers into the neural networks. The main difference to MLPs is that we only connect the neurons that lie in the next neuron's receptive field instead of connecting, e.g., every neuron (fully connected). Yann LeCun [LBD$^+$89] is regarded as the founder of modern CNN architecture and is recognized as a pioneer in machine learning and computer vision. Most of the commonly used deep CNN architectures also incorporate *Pooling*, *Upscaling*, and *fully connected* layers [ABAAU18]. We introduce the theory of CNNs used in image processing tasks in the following sections.

### Convolutional Layer

Figure 4.3 illustrates a 3-dimensional convolution that is typical for an image processing CNN. The input of a convolutional layer is a tensor with dimension $width \times height \times \#channels$, also called the input volume. In our case, that might be the input tensor $I$ defined in section 3.1. A convolutional layer convolves the input tensor into several neuron activation maps or *feature maps*, creating an output tensor with dimension $width_{fm} \times height_{fm} \times \#featuremaps$.

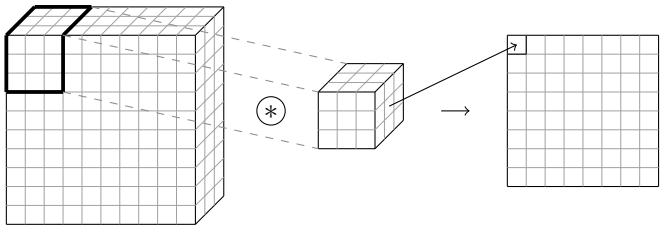



Figure 4.3: Depiction of a 3-dimensional convolution.

In the forward pass, a distinct 3-dimensional learnable filter moves over the input tensor for each feature map we want to create. The activation values are then calculated by an element-wise multiplication of the filter and the underlying receptive volume and then summing up the values of the resulting 3-dimensional tensor. The product between the kernel and the receptive volume is called the Frobenius inner product. In section 4.1.3, we will see that an activation function is commonly used to introduce non-linearity to the linear convolution operation. It is essential to mention that the convolution we are using is a special kind of 3-dimensional convolution. This is because the filter and the input volume must have the same number of channels in the third dimension. Therefore the dimension of the filter is $width_{kernel} \times height_{kernel} \times \#channels$. This results in the filter only moving in two dimensions[1].

The feature map size, $width_{fm}$, and $height_{fm}$, is controlled by the kernel size, the size of the input volume, and the following hyperparameters. For a good visualization and a better understanding of different hyperparameters, we refer to a paper from Vincent Dumoulin and Francesco Visin [DV18].

- *Stride* $(s_x, s_y) = s \in \mathbb{N}^2$ controls the step size of the filter movement. If we use small values for the stride, we get heavily overlapping receptive fields. On the other hand, using a large stride may ultimately lead to "unseen" space between receptive fields.

- *Padding* $(p_x, p_y) = p \in \mathbb{N}_0^2$ extends the input volume on the borders and fills the new cells with specific values. The most common padding value is 0. We can use the padding parameter to ensure that $width = width_{fm}$ and $height = height_{fm}$.

[1] This is why the convolutional layer is called *"Conv2d"* in the open-source machine learning framework PyTorch.

- *Dilation* $(d_x, d_y) = d \in \mathbb{N}^2$ is a relatively rare hyperparameter. It can be used to increase the size of the receptive field without increasing the number of learnable parameters. This is done by inflating the kernel and adding a spatial gap between single kernel values.

Figure 4.4 depicts the movement of a $5 \times 5$ kernel when $p_x = 1$ and $s_x = 2$. The 2-dimensional illustration is just a 3-dimensional convolution, as explained above, but with $\#channels = 1$.

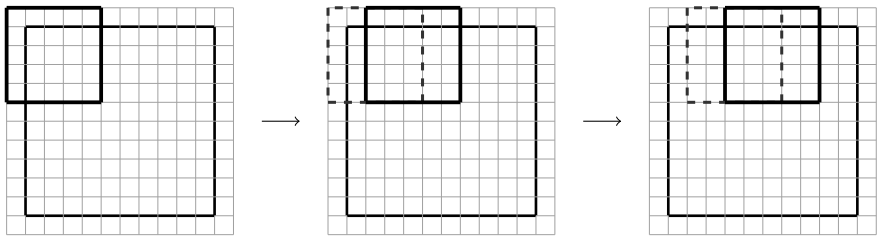


Figure 4.4: Depiction of the kernel movement with $p_x = 1$ and $s_x = 2$.

We can finally formulate a relationship between the hyperparameters and the size of the feature map.[2]

$$height_{fm} = \left\lfloor \frac{height + 2 \cdot p_y - d_y \cdot (height_{kernel} - 1) - 1}{s_y} + 1 \right\rfloor \tag{4.2}$$

$$width_{fm} = \left\lfloor \frac{width + 2 \cdot p_x - d_x \cdot (width_{kernel} - 1) - 1}{s_x} + 1 \right\rfloor \tag{4.3}$$

### Pooling Layer

Computation time and memory space are limited resources dramatically needed in deep learning. Therefore, fast training and inference are one of the main requirements for our denoising solution. Pooling layers are an efficient and computationally cheap method to reduce the size of the feature maps for the next convolutional layer. This spatial size reduction decreases the memory footprint and increases the inference speed. Similar to how the convolutional layer works, a kernel moves over the input feature map and creates an output matrix. The hyperparameter stride, padding, and dilation control the movement of the kernel. The pooling layer operates independently on every feature map, depicting a 2-dimensional operation. There are different types of pooling kernels which represent various down-sampling methods.

1. Max pooling:
   The max pooling operation takes the largest value of the underlying area.
2. Average pooling:
   The average pooling operation averages the values of the underlying area.

[2] This exactly matches the relationship in the PyTorch documentation.
(`https://pytorch.org/docs/stable/generated/torch.nn.Conv2d.html`)

3. Global pooling:
   Global pooling is a specific kind of pooling where we map the input matrix to precisely one value. There are different ways to do this, e.g., global max pooling or global average pooling.

Figure 4.5 depicts the max pooling and the average pooling operation. In this particular example, the memory consumption is halved.

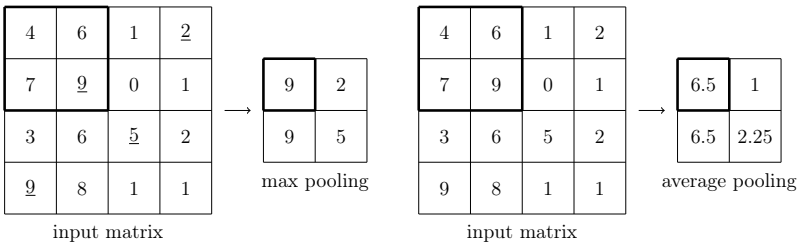



Figure 4.5: Depiction of the max pooling and the average pooling operation. We used a $2 \times 2$ kernel and $p = (0, 0)$, $s = (2, 2)$, and $d = (1, 1)$.

### Upsampling Layer

Upsampling layers try to invert pooling layers (unpooling). They are used to recreate feature maps with larger spatial sizes. We obviously can not completely reverse pooling layers because information may get lost depending on the pooling method. There are a lot of different algorithms used in upsampling layers. A straightforward way of upsampling is using the nearest neighbor algorithm depicted in figure 4.6.

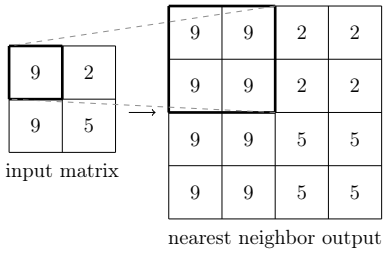



Figure 4.6: Depiction of the nearest neighbor upsampling algorithm.

Another way of increasing the spatial dimension of the input is using *transposed convolutions*, also known as *fractionally strided convolutions* [DV18]. Figure 4.7 depicts a transposed convolution. An easy way of understanding the mathematical operation between the $2 \times 2$ input matrix $I$ and the $3 \times 3$ kernel $K$ in figure 4.7 is by swapping the operation's forward pass and backward pass and considering a direct convolution. The matrix $I$ then is the result when directly convolving the output $O$ with the kernel $K$. The transposed convolution aims to recover a matrix with the same shape as $O$ when only given the result of the direct convolution $I$. Note that the transposed convolution is *not* an inverse convolution, i.e., it is not necessarily possible to exactly recover the matrix $O$.

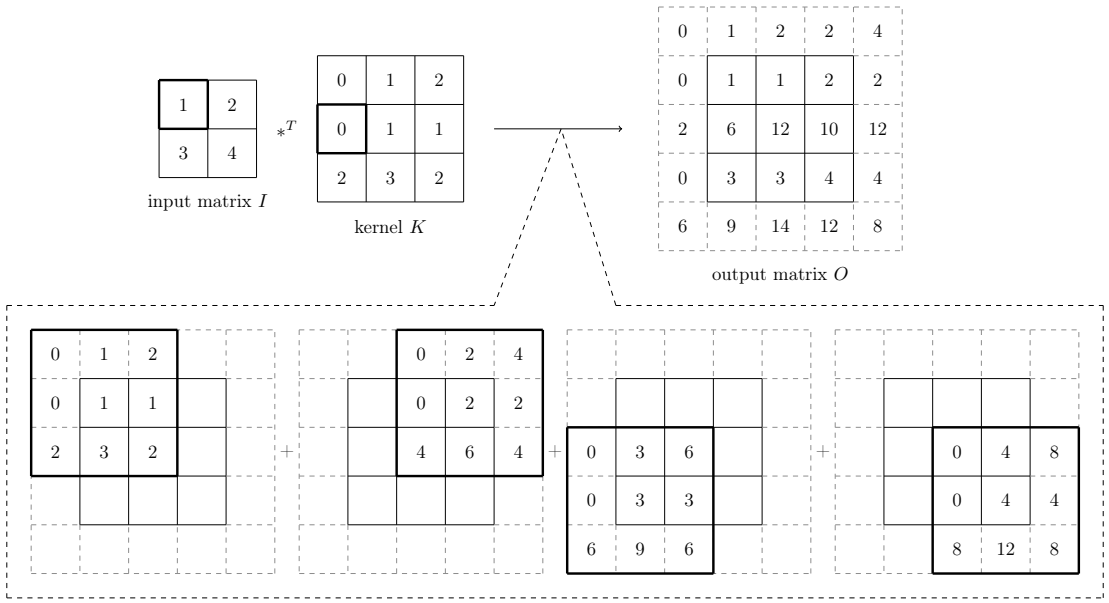



Figure 4.7: Depiction of a transposed convolution between a $2 \times 2$ input matrix and a $3 \times 3$ kernel using stride $s = (2, 2)$ and padding $p = (1, 1)$.

In figure 4.7, we use a stride $s = (2, 2)$ and a padding $p = (1, 1)$. A difference to direct convolution is that we apply stride, padding, and dilation in the output matrix of the transposed convolution. The padding $p = (1, 1)$, therefore, removes the columns and rows at the borders.

Transposed convolutions are an advanced upsampling method because it is possible to learn the kernel weights. However, this also comes at the expense of computational efficiency. The additional computational effort is why we use more straightforward methods like nearest neighbor in modern CNN architectures. Combining trivial upsampling algorithms like nearest neighbor with a direct convolution is a common approach to avoid transposed convolutions. For an in-depth view of transposed convolutions, we refer to [DV18].

### Activation Function

Activation functions are a crucial component of an artificial neuron in deep learning (section 4.1.1). They map the output of a neuron to a meaningful activation value. Activation functions commonly reduce the image space and are therefore called *squashing functions*. Various activation functions have been developed over the years [SS17]. Most of the activation functions are non-linear to integrate a non-linearity into the neural network. The non-linearity is the most significant difference to other statistical learning methods such as linear regression. Without a non-linear activation function, an MLP would simply represent a linear combination of the input values. We want to present the four most commonly used non-linear activation functions. Figure 4.8 shows various activation functions explained hereafter.

- Binary Step Function:

$$\sigma_{bsf}(v) = \begin{cases} 1, & \text{if } v \geq 0 \\ 0, & \text{otherwise} \end{cases} \tag{4.4}$$

The binary step function represents a threshold-based activation function. It defines a hard limit that decides whether the neuron is active or not. For example, in the definition above, the threshold is 0. We typically use a binary step function in neural networks with boolean decisions such as binary classification networks.

- Sigmoid Function:

$$\sigma_{sig}(v) = \frac{1}{1 + \exp(-av)} \tag{4.5}$$

The sigmoid function maps the input value $v$ to a probability in the range $[0, 1]$. It is often used in neural networks where the output needs to be a probability. The sigmoid function introduces the *vanishing gradient problem*. A large change in the input space may only result in a small change in the output. The derivative may become so small during the backpropagation process that weights are not updated properly. The vanishing gradient problem is particularly visible if small derivatives of multiple sigmoids get multiplied due to the chain rule in the gradient descent algorithm.

- ReLU (Rectified Linear Unit):

$$\sigma_{relu}(v) = \max(0, v) \tag{4.6}$$

The ReLU function is the most successful activation function in deep learning [RZL17]. For values $v \geq 0$, it just represents the identity, but all values $v < 0$ are mapped to 0. This introduces the so-called *dying ReLU problem*. Neurons may reach a point where the derivative is zero and the neuron is "dead". That means that the gradient will always be zero during backpropagation, resulting in the weights not being changed. Ultimately this can lead to substantial network parts being inactive because the weights will never update again. Large negative bias or learning rate can cause the dying ReLU problem. Both of the reasons can lead to the inputs being negative and ReLU returning zero.

- Leaky ReLU:

$$\sigma_{lrelu}(v) = \begin{cases} 0.01v, & \text{if } v < 0 \\ v, & \text{otherwise} \end{cases} \tag{4.7}$$

The leaky ReLU function is an approach to solve the dying ReLU problem. For negative $v$, the leaky ReLU function ensures that the weights keep updating with a slight positive gradient. However, we will not encounter the dying ReLU problem in our empirical evaluation (section 7). Nevertheless, if one encounters an unexpected plateau during the training process, switching to the leaky ReLU function might be a possible solution.

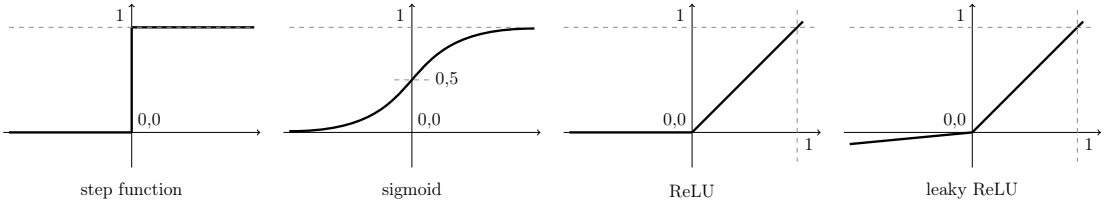



Figure 4.8: Depiction of various activation functions.

### 4.1.4 Convolutional Autoencoders

We now have the main building blocks to put our first convolutional neural network together. The most important CNN in image denoising is the *denoising autoencoder* (DAE). Figure 4.9 shows the basic schema of a denoising autoencoder, consisting of two main parts: the *encoder* and the *decoder*. The final goal of the network is to denoise the corrupted input volume $I$ to obtain a clean output $O$. In our case, the input and output volume may be an RGB image.

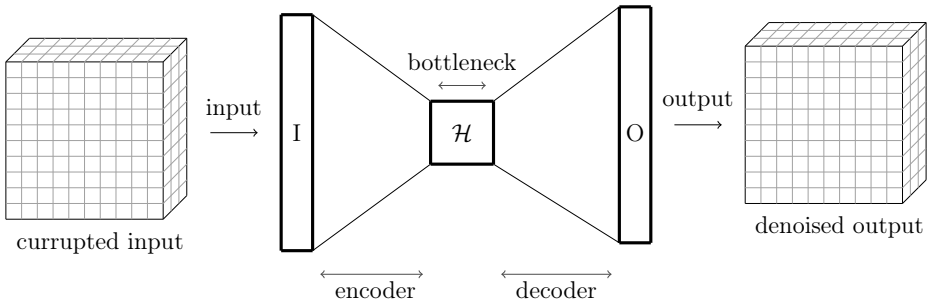



Figure 4.9: Depiction of a simple denoising autoencoder which denoises the corrupted input data given as a 3-dimensional tensor.

We can define the encoder and decoder as two mappings $\epsilon$, $\delta$:

$$\epsilon : I \to \mathcal{H} \tag{4.8}$$
$$\delta : \mathcal{H} \to O \tag{4.9}$$

We can furthermore specify our denoising function $d$ (equation 3.1):

$$d(I) = (\delta \circ \epsilon)(I) = O = I_d \tag{4.10}$$

We finally want an encoder and a decoder such that the error or $loss$[3] between $I_d$ and $I_{gt}$ is minimal.

$$\epsilon, \delta = \arg\min_{\epsilon,\delta} [loss(d(I), I_{gt})] \tag{4.11}$$

[3]We will define various loss functions in section 4.2.

The tensor $\mathcal{H}$ is a latent representation of the input data. Meaning a hidden representation that the DAE learns. This tensor should contain all feature maps, and its purpose is to precisely comprise every pattern in the input. In machine learning, latent space representation often compresses the original data to only learn the important features and drop the input's noise. The compression is the reason why $\mathcal{H}$ is often called the *bottleneck* of an autoencoder.

Having discussed the basic idea of DAEs, we can now take a look inside the encoding and decoding pass. This thesis only considers convolutional autoencoders, i.e., autoencoders that work with convolutions in the encoder and decoder. Constructing a convolutional autoencoder works by connecting various layers, such as convolutional, pooling, and upsampling layers. The most important thing to bear in mind is that the input dimension has to match the output dimension of the previous layer. Figure 4.10 depicts a particular example of a convolutional autoencoder. Note that the input width $w$ and height $h$ do not have to be constant. After every convolution, we use ReLU or leaky ReLU functions as the activation function.

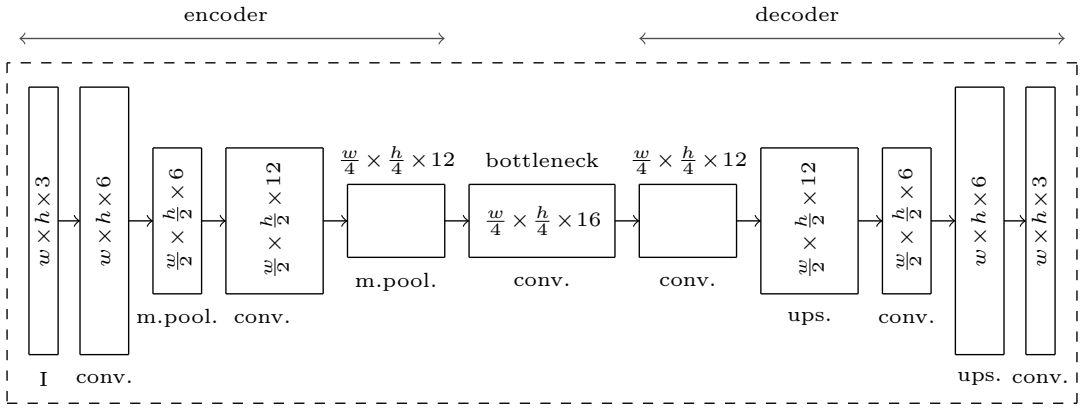



Figure 4.10: Example of a convolutional autoencoder using 6 convolutional layers (conv.), 2 max pooling layers (m.pool.) and 2 upsampling layers (ups.). $w$ and $h$ are the input and output width and height.

### 4.1.5 U-Nets

U-nets are a variation of classical autoencoders. O. Ronneberger, P. Fischer, and T. Brox introduced this concept for biomedical image segmentation [RFB15]. The critical modification is the integration of skip connections between layers of the encoding path and the corresponding layers of the decoding path. Those skip connections copy the feature maps from the encoder to the decoder. Finally, we concatenate the copied feature maps from the encoding pass and the corresponding feature maps of the decoding pass.

Figure 4.11 depicts the convolutional autoencoder from figure 4.10 with skip connections in the first and second layers. By directly transferring the feature maps from the encoding pass to the decoding pass, we lose the notion of a bottleneck. We will compare the denoising performance of this autoencoder variation in our empirical evaluation (section 7). The name U-net is derived from the U shape seen in the figure.

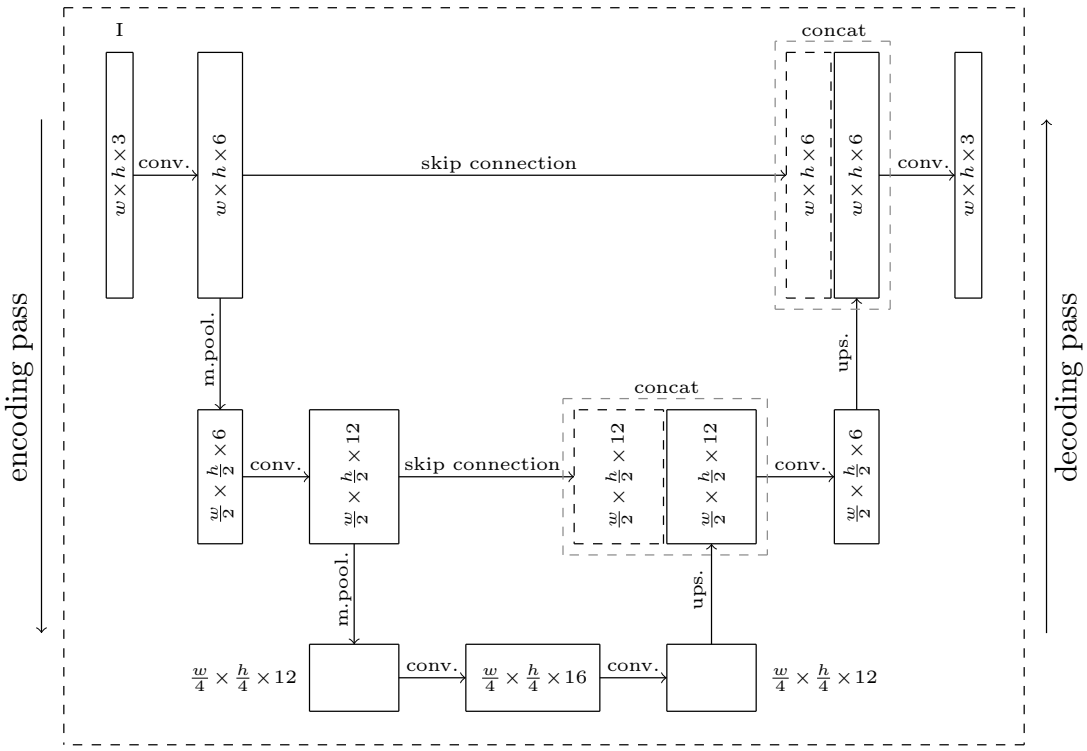



Figure 4.11: The autoencoder from figure 4.10 with two skip connections. This type of neural network is called a U-net.

## 4.2 Loss Functions

Loss functions define the error between the output $I_d$ of a neural network and the desired result, also called ground truth $I_{gt}$. Training a neural network is an optimization problem. We want to update the weights such that the loss is minimal (equation 4.11). How we define the loss is a crucial parameter of the training process and highly influences the success of our denoising approach. Different functions result in visually different images.

In the following loss definitions, $n$ is the number of values, and $y_i^{gt}$ is the target value for $y_i$. In our case, $y_i$ would be a pixel value and $y_i^{gt}$ the ground truth pixel value.

- MAE (Mean Absolute Error):

$$\mathcal{L}_{mae} = \frac{1}{n} \sum_{i=1}^{n} |y_i^{gt} - y_i| \tag{4.12}$$

- MSE (Mean Squared Error):

$$\mathcal{L}_{mse} = \frac{1}{n} \sum_{i=1}^{n} (y_i^{gt} - y_i)^2 \tag{4.13}$$

Mean Absolute Error and Mean Squared Error are both pixel-wise loss functions. They directly compare the pixel values of the ground truth and the result of a network.

However, obtaining a visually good-looking image is the main goal in image processing tasks. Because a human observer is the primary quality evaluator, we need perceptually motivated loss functions, also called structure-based metrics. These loss functions also capture structural similarities between images.

- SSIM (Structural Similarity Index):
  The SSIM loss function evaluates three different components: *luminance* $l$, *contrast* $c$, and *structure* $s$:

  $$\mathcal{L}_{SSIM} = [l(I_d, I_{gt})]^{\alpha} \cdot [c(I_d, I_{gt})]^{\beta} \cdot [s(I_d, I_{gt})]^{\gamma} \tag{4.14}$$

  A Gaussian filter with standard deviation $\sigma_G$ is used to emulate the human visual system and to finally calculate the luminance, contrast, and structural difference.

- MS-SSIM (Multi-Scale Structural Similarity Index):
  MS-SSIM is an enhancement of the SSIM index using multiple scales and subsampling processes.

We refer to [ZGFK15] for the exact definitions of SSIM and MS-SSIM and additional information.

## 4.3 The Training Process

Training is the process where a neural network progressively learns the optimal weights. When setting up a new model, the initial weights are randomly chosen. Then, we provide the network with example data points to obtain early predictions. Finally, the optimizer calculates the loss between those predictions and a ground truth value in supervised learning and changes the weights accordingly. Various learning algorithms for neural networks update the weights based on the example data points. Repeating this process for multiple dataset sweeps will hopefully increase the model's accuracy and produce better predictions. One entire training data sweep is called an epoch. Section 6 will precisely define the data sets used in our approaches.

### 4.3.1 Stochastic Gradient Descent

Finding the optimal weights is a local optimization problem, where we want to find a local optimum of an objective function $\mathcal{O}(w)$. There exists a wide variety of optimization algorithms that try to optimize the weights of a network. *Stochastic gradient descent* (SGD) is the most common weight adjustment technique in machine learning. The algorithm is an iterative process that modifies the weights according to the true gradient approximated by the gradient of an example data point $x$, the prediction $y$, and the ground truth value $y^{gt}$ (equation 4.15). $\mu$ is called the learning rate and represents the step size in each weight update (section 4.4).

$$w = w - \mu \cdot \nabla \mathcal{L}(y, y^{gt}) \tag{4.15}$$

Instead of approximating the gradient with only one training sample, we can use batches of multiple data points (equation 4.16). As a result, the true gradient is not approximated with only one sample but with $n$ sample points instead. Furthermore, this approach enables the use of modern vectorization libraries that greatly enhance performance on modern GPUs.

$$w = w - \mu \cdot \frac{1}{n} \sum_{i=1}^{n} \nabla \mathcal{L}_i(y_i, y_i^{gt}) \tag{4.16}$$

Researchers developed multiple optimizations for the classic stochastic gradient descent algorithm. Adaptively choosing the learning rate of the algorithm is one way to improve weight optimization (section 4.4). For a profound explanation of different optimization algorithms, we refer to [Mur22].

### 4.3.2 Overfitting and Underfitting

Training data is an essential part of machine learning. The quantity and quality of data points are crucial factors that decide any learned model's success. In supervised learning, we require labeled data, i.e., we need a target value for each training sample. In our case, that is a noise-free image.

The composition of the training data set and the model complexity, also referred to as *capacity* [GBC16], are crucial parameters determining the success of a learning process. The capacity of a neural network model generally describes the ability to fit the training data. If the capacity is too high or the training data set is too small, it is possible to achieve zero loss on the training data. This is because the model memorizes the exact target value of every training sample. When feeding new (unseen) data to the network, the model prediction is bad. This phenomenon is called *overfitting*. On the other hand, when the network architecture does not provide enough capacity to learn features from the training data, it is impossible to reduce the model's loss. This is called *underfitting*. The ability to transfer a trained model to unseen data is called *generalization* [GBC16]. Therefore, underfitting and overfitting are properties that determine the generalizability of a neural network model. Choosing a suitable model capacity is a crucial part of network building.

Deciding if a network overfits or underfits can be hard. Generally, we use a test set to validate the performance of the model with inputs not present in the training data. Consequently, we can evaluate the generalizability of the trained model. We finally choose a network architecture that achieves the lowest loss when testing it with the data points in the test set.

Figure 4.12 visualizes three common learning scenarios. Commonly, the training set's loss is constantly decreasing during the training process, while the test set's loss may behave entirely differently. Ideally, we want to choose a neural network with the optimal capacity, such that training for too long is not affecting the generalizability of the model. In general, reaching the optimal point with minimal test data loss is the desired objective, even though the training set's loss might not be optimal.

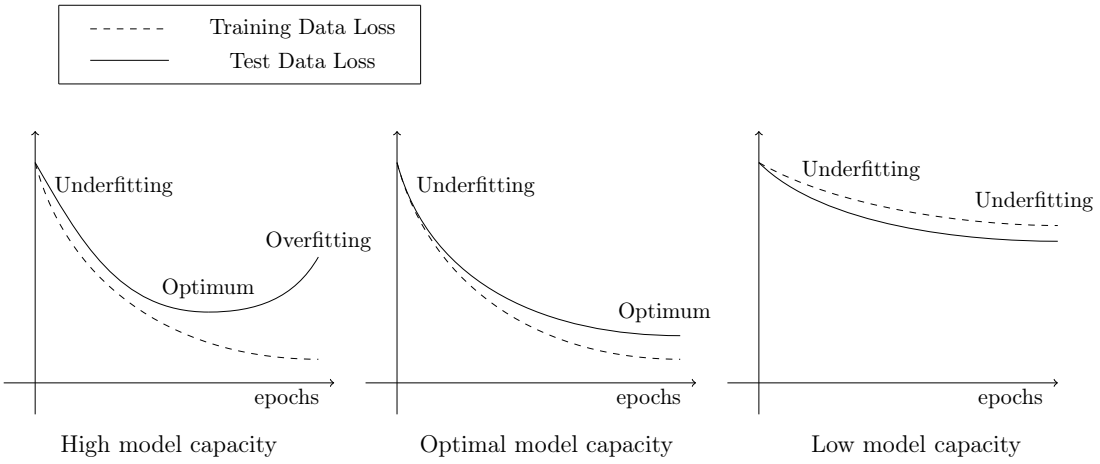



Figure 4.12: Depicts three common training scenarios: high model capacity, optimal model capacity, and low model capacity. High model capacity is prone to overfitting, while low model capacity can not fit the training data.

## 4.4 Learning Rate Scheduling

The learning rate $\mu$ is another essential hyperparameter of the training process. The value determines the step size with which the weights in a neural network are updated (section 4.3.1). Thus, a large learning rate indicates that we take a large step in the descent direction, that is, the direction towards the closest local minimum.

Learning rate scheduling depicts the management of the learning rate. Instead of just using a constant learning rate $\mu$ for our training process, we can make the learning rate epoch-dependent. That means we define a learning rate function that maps the epoch to a specific learning rate (equation 4.17). This modification allows us to perform larger updates at the beginning of the learning process and later smaller updates to avoid overshooting the minimum.

$$\mu : \mathbb{N} \to \mathbb{R}^+ \tag{4.17}$$

We can even adaptively modify the learning rate depending on the current state of our learning process.

We will now take a look at some basic learning rate schedules that are visualized in figure 4.13[4].

- Constant Learning Rate:

$$\mu(e) = c \tag{4.18}$$

  A constant learning rate function always maps to the same learning rate $c$.

[4]These learning rate schedules are also provided by the PyTorch training library.

- Lambda Learning Rate:

$$\mu(e) = \mu_{init} \cdot \lambda(e) \tag{4.19}$$

  A custom lambda function $\lambda(e)$ dictates a lambda learning rate schedule. Every epoch, the initial learning rate $\mu_{init}$ is scaled with $\lambda(e)$.

- Exponential Learning Rate:

$$\mu(1) = \mu_{init} \tag{4.20}$$
$$\mu(e+1) = \gamma \cdot \mu(e) \tag{4.21}$$

  This schedule exponentially decays the learning rate of each epoch by $\gamma$. We can also define a step size or particular epoch milestone to specify when the learning rate will be modified. Note that the exponential learning rate schedule is a special case of the lambda learning rate schedule.

- Cosine Annealing:

$$\mu(e) = \mu_{min} + \frac{1}{2}(\mu_{max} - \mu_{min}) \left(1 + \cos\left(\frac{T_{cur}}{T_{max}}\pi\right)\right) \tag{4.22}$$

  Cosine annealing is a cyclic learning rate schedule based on the cosine function. I. Loshchilov and F. Hutter [LH16] introduced cosine annealing with warm restarts: Stochastic Gradient Descent with Warm Restarts (SGDR).

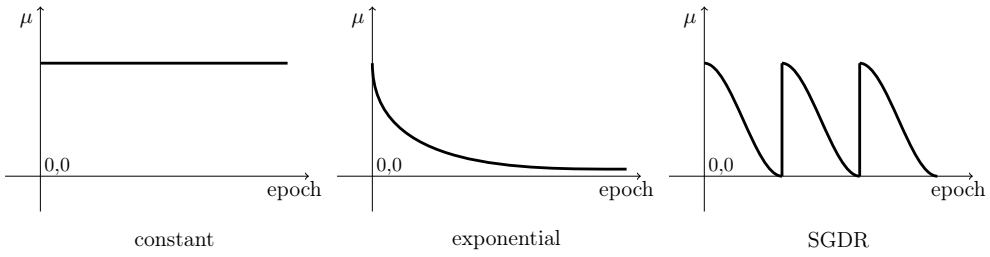



Figure 4.13: Depiction of different learning rate schedules.

### 4.4.1 Adam

Adaptive learning rate scheduling is natively integrated into different gradient descent algorithms. *Adam* (Adaptive Moment Estimation) is a state-of-the-art optimization algorithm that calculates individual learning rates for parameters using estimates of the first and second moments of the gradients [KB17]. The most considerable improvements of the Adam algorithm are that we do not use only one learning rate $\mu \in \mathbb{R}$ for the neural network but instead a learning rate $\mu_{Adam} \in \mathbb{R}^d$ for each weight in the model, where $d$ is the total number of weights in the neural network. In general, Adam is based on the Adaptive Gradient Algorithm (AdaGrad) and Root Mean Square Propagation (RMSProp) [KB17] and tries to combine the benefits of both. Adam is commonly used in computer vision and works well for noise reduction.

# 5 Existing Software

## 5.1 Rodent

In this thesis, we want to focus on *one* specific path tracer implementation. The main goal is to create a CNN-based post-processing approach that reduces the rendering time needed to create visually good-looking images with no visible noise. Eventually, the renderer-specific results can be used to create similar approaches for any other Monte Carlo renderer. Ultimately it might also be possible to build a more general solution combining the solutions from multiple renderers or stay completely renderer-independent. It is important to note that although we restrict ourselves to one renderer, this thesis also serves as a universal source of information.

The Monte Carlo renderer we have chosen is *Rodent*[1]. Introduced by [PGML+19] at Saarland University, it builds upon the partial evaluation framework AnyDSL [LBH+18]. Rodent is based on the classical path tracing algorithm with Next Event Estimation and includes a real-time rendering option. The renderer is highly optimized and works on modern CPUs and GPUs. For a deeper understanding of Rodent refer to the original paper.

There is also a follow-up thesis [Mey21] providing a framework inside AnyDSL, which allows us to directly integrate our denoising solutions into the codebase of Rodent. For example, using matrix multiplications natively implemented in AnyDSL could speed up the network inference time, leading to optimal rendering times. In addition, working with Rodent would enable sample-based denoising covered in section 3.1. However, this would go beyond the scope of this thesis and is left for future work.

[1] `https://github.com/AnyDSL/rodent`

## 5.2 Open Image Denoise by Intel

Monte Carlo denoising utilizing deep learning methods is a commonly used technique. There are a bunch of existing denoising frameworks developed by companies and researchers. The most successful ones are Nvidia OptiX and Open Image Denoise by Intel. Open Image Denoise (OIDN) was initially released in 2018 at SIGGRAPH. It is an open-source denoising library by Intel[2]. OIDN contains a neural network training framework based on the python PyTorch library. One can use this framework to train denoising filters with own training data. The library also contains an API written in C/C++ to integrate the denoising filter into existing renderers. The network inference is based on the high-performance Intel oneAPI Deep Neural Network Library (DNN). Open Image Denoise has also been integrated into state-of-the-art rendering software such as Blender[3].

### 5.2.1 Neural Network Model

The heart of OIDN is a pre-trained denoising neural network model $\mathcal{M}_{OIDN}$ that can denoise Monte Carlo noise. The neural network architecture is a U-net, depicted in figure 5.1. Internally the model uses convolutional layers conv(*in*,*out*) with $3 \times 3$ kernels and a padding $p = (1, 1)$ where *in* and *out* are the incoming and outgoing channels. Note that the framework allows easy modification of the input channels to support more than three input channels (section 6). The ReLU activation function is used after every convolutional layer. In addition, max pooling and nearest neighbor upsampling layers scale the feature maps in the encoding and decoding pass. Last but not least, the network includes four skip connections that directly copy the feature maps from the encoder to the decoder.

### 5.2.2 Problems with OIDN

The neural network architecture of OIDN served as the fundamental baseline of this thesis. Therefore, hyperparameter testing in the following sections is primarily based on the OIDN approach. Even though OIDN is an open-source library, there is little to no concise information publicly available.

The following list presents issues regarding the network architecture and training process that are not addressed in a compact manner.

- There is no Monte Carlo noise training set available.
- Investigations of different model architectures are split up into multiple research papers that might not even directly handle Monte Carlo denoising.
- Information about various hyperparameters is ambiguous and hard to find.
- There is no easy training guideline that provides explanations for good learning strategies.

[2] `https://www.openimagedenoise.org/`
[3] `https://www.blender.org/`

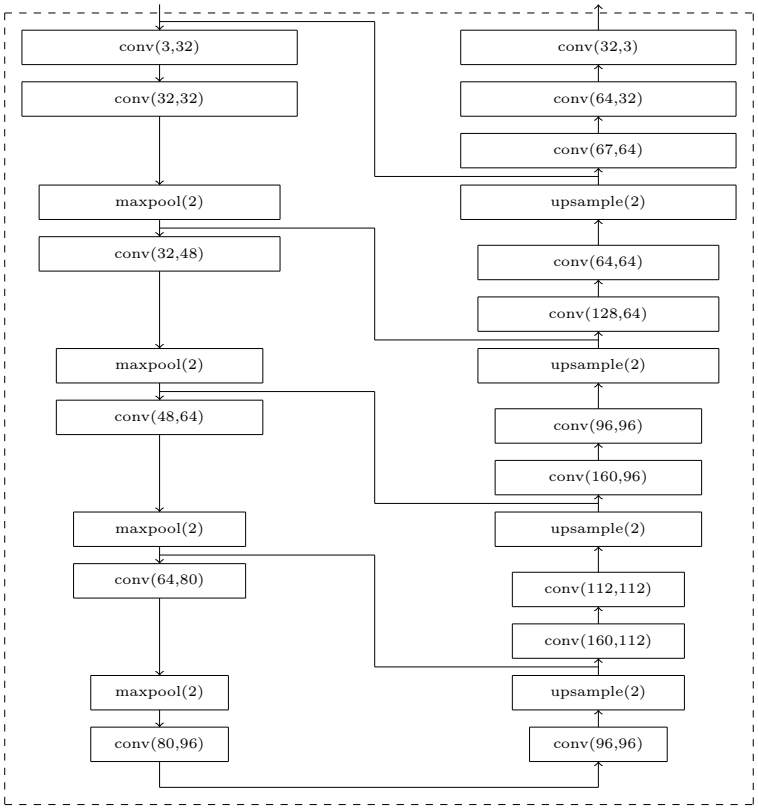



Figure 5.1: The U-net architecture used in Open Image Denoise by Intel.

This thesis aims to provide the missing information and work as a good starting point in Monte Carlo denoising. We used OIDN as a starting point of our work and integrated it into Rodent using the C/C++ API. This allows us to use the Intel denoiser as a reference solution for comparison. It might not be possible to recreate the quality of the denoising neural networks trained by Intel, but we certainly can provide an insight into Monte Carlo denoising.

# 6 Data Setup

One of the most important elements of a machine learning task is the training data set. It is the foundation of training a machine to complete complex tasks like separating Monte Carlo noise from a rendered image. There are several requirements to the samples in the data set deciding the quality and performance of a trained model. First, the data set has to be large enough to prevent the neural network from overfitting. Simultaneously the data needs to be general, meaning that it has to represent every scene that could potentially get denoised with the finished model. Both requirements try to ensure a good translation of the model to unseen data. Depending on the purpose and the later usage, one might consider using different training data sets for different tasks. Since there is no sufficient Monte Carlo noise training data set publicly available, we generated our own training data set. Furthermore, considering the aim of creating a Rodent-specific denoising approach, we rendered all of our training and test samples with the path tracer Rodent (section 5.1). In this chapter, we will cover the training data set generation and test data set generation.

## 6.1 Albedo and Normal Images

In section 3.1, we defined the input to our neural network models. We want to create pixel-based denoising approaches that directly use the noisy output image of our renderer as the input tensor $I$. Recall that we use a denoiser $d(I)$ to map a noisy input image to a noise-free output $I_d$ in classical denoising (equation 3.1). The great advantage of denoising Monte Carlo noise caused by path tracing and having direct access to the path tracer implementation is that we can slightly modify our renderer to generate additional feature images. These *auxiliary feature images* provide more information to the neural network model and can improve the denoising quality. Ultimately, we feed our network with noisy images and provide other images containing additional information. The modification to our renderer such that we also render the auxiliary images obviously should only cause a small amount of extra rendering time.

Various papers experiment with different auxiliary feature images [KBS15], [CKS+17], e.g., depth information, visibility maps, and ambient occlusion. However, the most common features are the albedo and normal image, also supported and used by OIDN.

- Albedo Image:
  The albedo image represents the material's base color without considering the viewing angle, complex lighting, reflection, or refraction. The issue with albedo images is that there is no universally accepted definition. Various renderers might implement the generation of albedo images slightly differently. We use a combination of the diffuse $k_d$ and specular $k_s$ values in the material file and only consider the first hit

of a ray path. When hitting a specular material, the ray bounces, and we take the first non-specular hit.

- Normal Image:
  The normal image maps the normal $n = (n_x, n_y, n_z)^T$ of every surface to a color $c$. We can create the normal feature map using the following formula and the first hit of a ray path.

$$c = n \cdot \frac{1}{2} + (0.5, 0.5, 0.5)^T \tag{6.1}$$

  When hitting a specular material, the ray bounces, and we take the first non-specular hit.

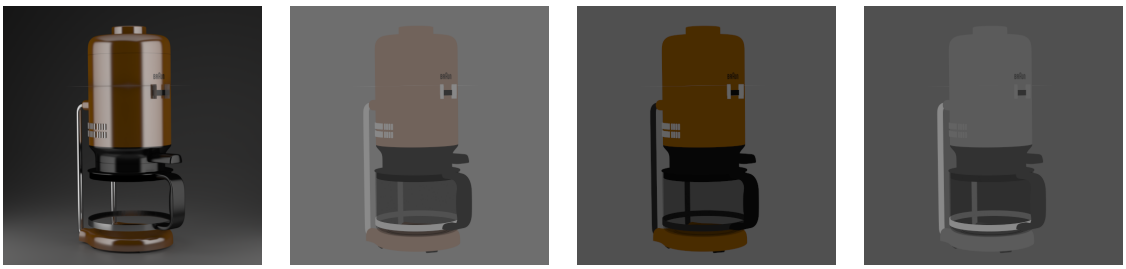
The albedo images are particularly beneficial when reconstructing high-frequency details not present in the corrupted image. In addition, the normal images provide the neural network model with spatial information of the scene. We modified Rodent to render the albedo and normal images alongside the actual rendering. Therefore, the auxiliary feature images in our training data set are Rodent-specific. Our neural networks now take three images as input instead of only one. Therefore, the dimension of the tensor $I$ is $width \times height \times 9$. Figure 6.1 shows an example of the three input images of a sample in the training data set.

(a) noisy (32 spp) (b) albedo (c) normal

Figure 6.1: Depiction of the albedo and normal auxiliary input images.

## 6.2 Diffuse and Specular Images

Since the generation of the albedo image is not universally defined, the combination of the diffuse values $k_d$ and specular values $k_s$ is rather arbitrary. Figure 6.2b depicts an example of an albedo image not really representing the ground truth (6.2a). The material values of the orange metal are $k_d = (0.595, 0.192, 0.0)$ and $k_s = (0.5, 0.5, 0.5)$. The combination of those values results in a color not representing the ground truth color of the coffee maker. Instead of finding the perfect combination of the $k_d$ and $k_s$ values, we split the albedo image into the diffuse and specular parts. This separation allows the neural network to learn the optimal combination of the $k_d$ and $k_s$ values.



(a) ground truth (b) albedo (c) diffuse (d) specular

Figure 6.2: Comparison between the albedo image and the diffuse and specular images.

A simple modification to Rodent allowed us to output the diffuse and specular images. We are considering the first hit for non-specular materials and using the $k_d$ and $k_s$ values, respectively. For specular materials (glass, mirror), we use the first non-specular hit. Figure 6.2c and 6.2d show the diffuse and specular images for the coffee maker example. We will investigate the difference between the usage of the auxiliary input features in our empirical evaluation in section 7.

## 6.3 Data Set Generation

The training data set $T$ and the test set $V$ are sets of 6-tuples $t$ and $v$. The tuple $t$ denotes a *training sample* and $v$ denotes a *test sample*. Both contain noisy, albedo, diffuse, specular, and normal images. We can use a selection of those images as the input to our neural network, meaning that even though every auxiliary input image is present in a training sample, we might not use it during a specific training run. For example, OIDN uses the common albedo-normal combination (section 6.1). Of course, $t$ and $v$ also include the ground truth image.

$$(noisy, albedo, diffuse, specular, normal, ground\ truth) = t \in T \tag{6.2}$$

$$(noisy, albedo, diffuse, specular, normal, ground\ truth) = v \in V \tag{6.3}$$

All of the scenes we used to render our data sets are from Benedikt Bitterli [Bit16]. In total, we used eight scenes for the training data set $T$ and five scenes for the test set $V$. Every scene was rendered multiple times, changing the camera position, viewing angle, and field of view, creating various perspectives of a scene. Multiple perspectives allowed us to generate more training samples without needing more scenes. Altogether we created 71 perspectives for the training data set and 12 for the test set.

Figure 6.3 and figure 6.4 show example perspectives of the data sets. Our training data set mainly consists of inner architecture scenes. Thus, every neural network model trained with this data is specialized to inner architecture. On the other hand, the test data set contains a wider variety of scenes to measure the generalizability of the trained neural networks. This restriction to our training data set reduces the training data set size needed to learn the key features at the expense of generalizability.

(a) bedroom (b) hall (c) bathroom (d) kitchen

Figure 6.3: Depiction of four example perspectives from the training data set.

(a) spaceship (b) coffeemaker (c) office (d) cornellbox

Figure 6.4: Depiction of four example perspectives from the test data set.

We rendered the following images for each perspective:

- 32× noisy images with a $512 \times 512$ resolution and different samples per pixel (spp). The spp values reach from 4 to 32000. The wide range provides a data set containing many different noise levels facilitating generalizability concerning the amount of noise.
- 1× albedo image with a $512 \times 512$ resolution.
- 1× diffuse image with a $512 \times 512$ resolution.
- 1× specular image with a $512 \times 512$ resolution.
- 1× normal image with a $512 \times 512$ resolution.
- The generation of the 1× ground truth image is a more challenging task compared to the other images. Even after 33000 spp, a $512 \times 512$ image is not entirely noise-free and often contains fireflies and noise in parts that are difficult to render. We finally rendered the scenes with a $2048 \times 2048$ resolution and 33000 spp. We denoised the noisy images with the "despeckle" algorithm provided by ImageMagick[1]. Downsizing the despeckled images to $512 \times 512$ ultimately results in noise-free ground truth images.

[1] `https://imagemagick.org/`

We finally arrived at $32 \cdot 71 = 2272$ training samples and $32 \cdot 12 = 384$ test samples. Overall we attained $6 \cdot 2272 = 13632$ images for the training data set and $6 \cdot 384 = 2304$ images for the test set. Every image was rendered with Rodent on a single Nvidia RTX 2070 Super GPU. The whole data set rendering took around 100 hours.

## 6.4 Data Augmentation

*Data Augmentation* is a technique used in machine learning to increase the number of training samples without really increasing the size of the training data set. It is a procedure that decreases the chance of overfitting by enhancing the data [SK19]. Researchers developed many different data augmentation methods, but we are only using a simple transformation.

Instead of directly training our neural network with the $512 \times 512$ images, we use a random crop transformation that provides a random $256 \times 256$ crop of the input images. Therefore, we only train our neural networks with $256 \times 256$ images. This data augmentation technique not only prevents overfitting it also speeds up the training process.

# 7 Empirical Hyperparameter Evaluation

In the following sections, we want to use the theory introduced in previous chapters to set up different denoising approaches, compare learning strategies, and find optimal values for different hyperparameters. Each of the following sections focuses on a different hyperparameter or learning strategy and provides information about denoising Monte Carlo noise with deep learning. The knowledge is most likely transferrable to other renderers, even though our training data is based on Rodent, and we only test our neural networks with Rodent. Note that a complete study of all hyperparameters would require an exponentially growing amount of tests. However, this is not feasible for our evaluation. Thus, we have to rely on certain independences between single hyperparameters.

Each neural network in the following sections is trained on a single Nvidia RTX 2070 Super GPU for 1270 epochs (section 4.3.2). We chose the batch size such that the training process fully uses the 8 GB GPU memory. Multithreaded training data loading during the learning averts a data loading bottleneck.

We recommend viewing all of the evaluation images in the digital version of the thesis. This is because a printer might not be able to reproduce the differences between some images. This is especially important for the spaceship scene.

## 7.1 Overfitting and Underfitting

Section 4.3.2 covered the importance of choosing a suitable model capacity and number of epochs. When training our neural networks, we have to determine the correct number of epochs such that the model is not underfitting and overfitting. We use the OIDN neural network architecture $\mathcal{M}_{OIDN}$ as a starting point and evaluate the generalizability of the network at different training stages. For the training, we used the combined loss $\mathcal{L}_{MIX}$ that we will introduce in section 7.2, the Adam optimization algorithm with a constant learning rate $\mu = 0.0001$, and diffuse, specular and normal images as additional feature images. It is possible to investigate if the model is overfitted or underfitted by measuring the difference between the training data loss and test data loss, also known as the *generalization gap* [GBC16]. The training data loss and test data loss are obtained by iterating over the training set and test set and averaging the loss when denoising with a specific denoising network. The question of how long one should train a neural network model is hard to answer. Generally, we aim at a reasonable training time of around 12 hours, which allows us to train every model on a single GPU. For most of our neural networks, 12 hours correspond to 1270 epochs.

Figure 7.1 illustrates the development of the training set loss and test set loss during the learning process of the network $\mathcal{M}_{OIDN}$. We used the SSIM loss function to capture the error between the denoised images and the ground truth. The training set error rapidly decreases until we reach epoch 400. From there, the error seems to converge to a fixed value. The convergence indicates that training for more epochs would most likely not significantly improve the denoising quality of the neural network. However, the test set loss also decreases but at a much slower rate and hits a global minimum at epoch 230. After 230 epochs, the test set loss rises, and we can conclude that the model is overfitted when using a higher number of epochs. Therefore, training a neural network for final usage always requires evaluating the test set loss to determine the best number of epochs and thus the best denoising model. Because 230 epochs might not be optimal for every neural network in the following sections, we train all models for 1270 epochs to maintain comparability.

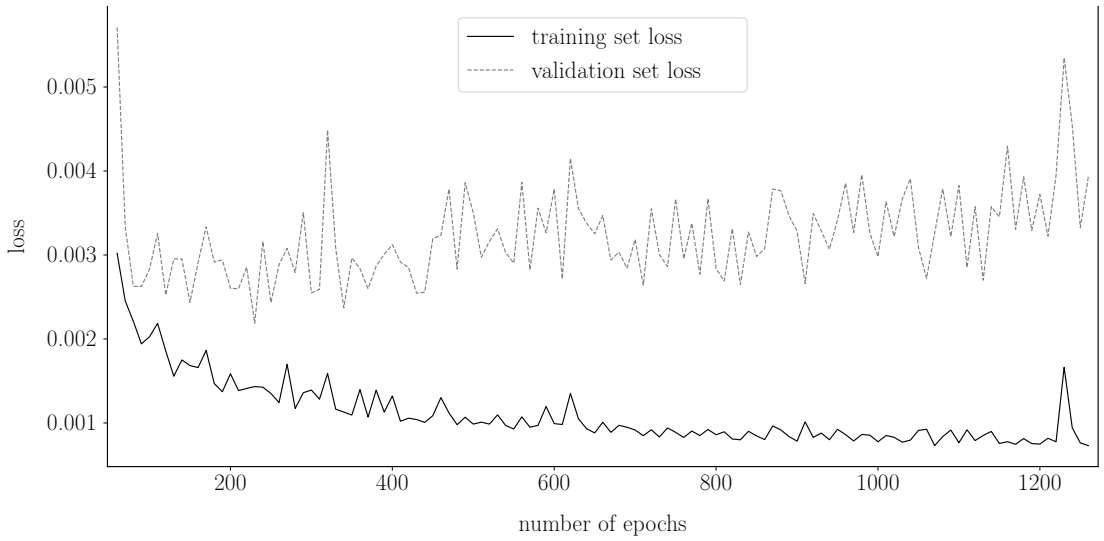



Figure 7.1: Development of the training set loss and test set loss during the learning process.

Figure 7.2 visualizes the underfitting and overfitting of our neural network. Denoising the corrupted image (7.2a) using the neural networks trained for 10, 230, and 1270 epochs result in very different quality of denoising. By looking at the outcome, we clearly see that denoising at 230 epochs provides significantly better results. It seems like the network is underfitted at 10 epochs and overfitted at 1270 epochs. Our observation is confirmed when we consider the actual SSIM loss between the denoised image and the ground truth. For the denoised images at epoch 10 and 1270, we obtain an SSIM error of 0.1121 and 0.1061, respectively, whereas for the denoised image at epoch 230 we obtain a substantially better SSIM error of 0.013. Figure 7.3 shows the pixel-wise difference between the different denoising results.

(a) noisy (b) ground truth (c) 10 epochs (d) 230 epochs (e) 1270 epochs

Figure 7.2: Denoising the spaceship scene using neural networks trained with different numbers of epochs.

(a) 10 epochs (b) 230 epochs (c) 1270 epochs

Figure 7.3: Visualization of the pixel-wise difference between the images and the ground truth. A difference greater than 10% is marked in black.

The interesting question is, why overfitting happens so early. To answer this, recall that overfitting and underfitting are also highly dependent on the training and test data sets. When a neural network overfits, it learns features and correlations present in the training data that might not be true for the test data.

For example, our training data primarily consists of interior design scenes, whereas our test set includes completely different scenes like the spaceship from figure 7.2. (Recall figure 6.3 and figure 6.4.) When only considering different interior design scenes in our test set, overfitting would most likely happen later. Our training data does not include a dark scene which might explain why the example in figure 7.2 provides bad results. Figure 7.4 demonstrates that test data that is similar to the training data is not affected by overfitting. The denoising result at 1270 epochs looks best when judged by a human observer.

(a) noisy (b) ground truth (c) 10 epochs (d) 230 epochs (e) 1270 epochs

Figure 7.4: Denoising the office scene using neural networks trained with different numbers of epochs.

We can finally conclude that our training scenes and test scenes are not perfect. A more general training data set is a strict requirement for a universal and general Monte Carlo denoising neural network that works for any rendered input image. Ideally, the training and the test data should reflect all possible scenes denoised with the neural network in later usage. Whenever this requirement is met, it is possible to maximize the generalization.

## 7.2 The Loss Function

The loss function is another deciding hyperparameter. We want to explore the impact of various loss functions that we defined beforehand (section 4.2) on the denoising result. To do so, we trained the OIDN network $\mathcal{M}_{OIDN}$ using different loss functions ($\mathcal{L}_{MAE}$, $\mathcal{L}_{MSE}$, $\mathcal{L}_{SSIM}$, and $\mathcal{L}_{MS-SSIM}$). Every training process utilized the Adam optimization algorithm with a constant learning rate $\mu = 0.0001$. We also used diffuse, specular, and normal images as auxiliary input features.

Table 7.1 depicts the training times for the network models with different loss functions. As expected, we notice an increasing time consumption with increasing loss function complexity. However, the slowest training process using $\mathcal{L}_{MS-SSIM}$ is only 5.5% slower than the fastest training process using $\mathcal{L}_{MAE}$.

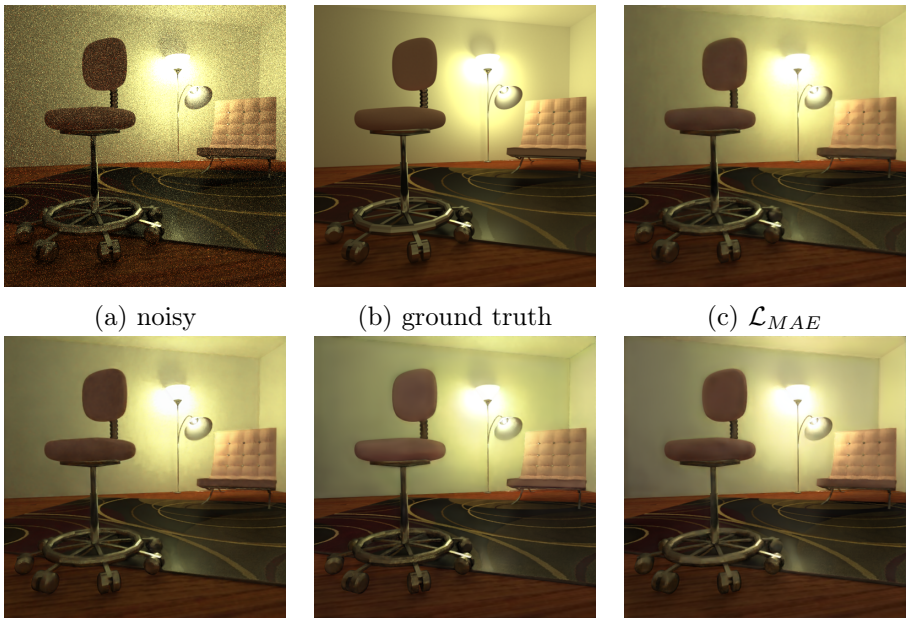
| Loss Function | Training Time [*min*] | Time per Epoch [*min*] |
|---|---|---|
| $\mathcal{L}_{MAE}$ | 671 | 0.528 |
| $\mathcal{L}_{MSE}$ | 682 | 0.537 |
| $\mathcal{L}_{SSIM}$ | 698 | 0.54 |
| $\mathcal{L}_{MS-SSIM}$ | 708 | 0.557 |
| $\mathcal{L}_{MIX}$ | 706 | 0.56 |

Table 7.1: Training time (in minutes) needed to learn for 1270 epochs.

The loss function is a quality metric that significantly changes the appearance of the denoised images. The quality of a loss function primarily depends on the visual quality of the predictions. Since humans mainly observe the resulting denoised images, we should choose loss functions that produce visually pleasing images as seen by the human visual system. Consequently, the loss function that results in the best denoising result when judged by a human observer is considered best.

Figure 7.5 shows an arbitrary image from the test set denoised with various neural networks trained with different loss functions. One can notice that $\mathcal{L}_{MSE}$, the most common loss function for image processing, performs worse than $\mathcal{L}_{MAE}$. This is because $\mathcal{L}_{MSE}$ creates splotchy artifacts in low-frequency regions of the denoised image. This observation matches with the result of Zhao et al. [ZGFK15]. However, both pixel-wise loss functions, $\mathcal{L}_{MAE}$ and $\mathcal{L}_{MSE}$, introduce some problems when a human observer assesses the resulting images. They seek to minimize the averaged difference between the exact pixel values. This characteristic might result in blurry edges and the loss of high-frequency details. This is especially problematic because sharp edges are a main quality indication for the human visual system. Therefore, perceptually motivated loss functions have been developed. In figure 7.5, we can see that the neural networks trained with $\mathcal{L}_{SSIM}$ and

$\mathcal{L}_{MS-SSIM}$ produce very good denoised images. Nonetheless, it is important to note that $\mathcal{L}_{SSIM}$ or $\mathcal{L}_{MS-SSIM}$ might introduce a slight shift in color or luminance. For example, when looking at the chair or the background wall, we recognize a minor color change compared to the ground truth image. This property is caused by the mathematical definition of $\mathcal{L}_{SSIM}$ and $\mathcal{L}_{MS-SSIM}$, which is not sensitive to a uniform bias ([ZGFK15]).



(a) noisy (b) ground truth (c) $\mathcal{L}_{MAE}$

(d) $\mathcal{L}_{MSE}$ (e) $\mathcal{L}_{SSIM}$ (f) $\mathcal{L}_{MS-SSIM}$

Figure 7.5: Denoising results of neural networks trained with different loss functions.

To finally combine the advantages of $\mathcal{L}_{MAE}$ and $\mathcal{L}_{SSIM}$ / $\mathcal{L}_{MS-SSIM}$, we use the approach from Zhao et al. [ZGFK15]. By connecting $\mathcal{L}_{MAE}$ and $\mathcal{L}_{MS-SSIM}$, we get an edge-preserving and perceptually motivated loss function $\mathcal{L}_{MIX}$ that alleviates the bias. The exact values of the coefficients are directly taken from the paper and are empirically chosen.

$$\mathcal{L}_{MIX} = 0.16 \cdot \mathcal{L}_{MAE} + 0.84 \cdot \mathcal{L}_{MS-SSIM} \tag{7.1}$$

Figure 7.6 shows the result of denoising a noisy image from the test set with the neural network $\mathcal{M}_{OIDN}$ and trained with $\mathcal{L}_{MIX}$. We can report a good denoising outcome because this scene is not in the training data, and the input image contains significant noise.

To make the difference of the loss functions visible, we used an image comparison tool[1]. This allowed us to highlight the exact differences between the denoised images. Figure 7.7 shows the pixel-wise difference visualization. Because we highlight the pixel value difference, $\mathcal{L}_{MAE}$ is looking really good. Additionally, we can see the splotchy artifacts caused by $\mathcal{L}_{MSE}$. In the visualization for $\mathcal{L}_{SSIM}$ and $\mathcal{L}_{MS-SSIM}$, the color shift is why the

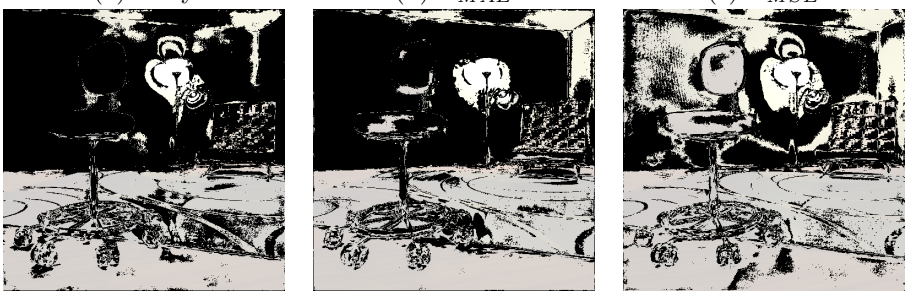
[1] `https://online-image-comparison.com/`

(a) noisy (b) ground truth (c) $\mathcal{L}_{MIX}$

Figure 7.6: Depiction of the denoising result when using a network trained with $\mathcal{L}_{MIX}$.

background wall and the chair are predominantly highlighted black. Finally, we conclude that the combined loss $\mathcal{L}_{MIX}$ partly removes the uniform bias and provides a solid loss function for image denoising using convolutional neural networks.

(a) noisy (b) $\mathcal{L}_{MAE}$ (c) $\mathcal{L}_{MSE}$



(d) $\mathcal{L}_{SSIM}$ (e) $\mathcal{L}_{MS-SSIM}$ (f) $\mathcal{L}_{MIX}$

Figure 7.7: Visualization of the pixel-wise difference between the denoised images and the ground truth. A difference greater than 3% is marked in black.

## 7.3 The Optimization Technique

The optimization algorithm and a potential learning rate schedule are other influential choices. Better learning strategies improve the efficiency and speed of the training process. For example, a better update rule for modifying the weights in each backward pass may reduce the steps needed to reach a global minimum. Training the OIDN network $\mathcal{M}_{OIDN}$ using various optimization strategies allows us to compare the approaches. Every training process in this section uses the combination loss $\mathcal{L}_{MIX}$ (equation 7.1) and the diffuse, specular, and normal images as auxiliary input features.

Stochastic gradient descent (SGD) (section 4.3.1) is the most common and basic optimization algorithm. In section 4.4, we introduced various learning rate scheduling techniques that we can apply to SGD. Figure 7.8 shows the training set combined loss convergence of a constant learning rate schedule (SGD / const), a cosine annealing learning strategy (SGD / CSA), and a cosine annealing strategy with warm restarts (SGD / CAR). The graphs visualize that different learning rate schedules do only have a small impact on the learning speed, though the constant learning rate slightly outperforms the other techniques. However, remember that every training process might behave somewhat differently because arbitrarily choosing the initial weight values and the subsequent training batch introduces some randomness to the learning. Consequently, based on the slight difference between the learning rate schedules, we can not conclude that one significantly outperforms the others.

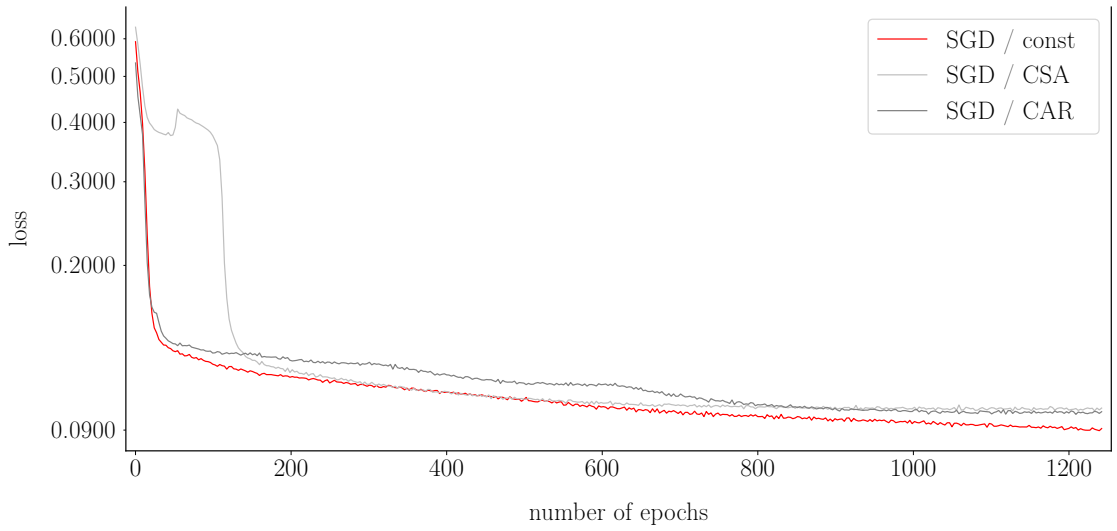



Figure 7.8: Comparison between three learning rate schedules applied to stochastic gradient descent.

The problem with simple learning rate schedules like cosine annealing is that they only depend on the epoch. To further enhance the learning of neural networks, researchers developed learning rate schedules that also use other information about the current state of the training. In section 4.4.1, we mentioned the optimizer Adam. Figure 7.9 visualizes the significant increase in performance of the Adam optimizer compared to classical stochastic gradient descent. Furthermore, we also applied the epoch-dependent learning

rate schedules on top of the Adam optimizer and concluded that they do not significantly change the loss convergence. Note that we can clearly see the learning rate resets of the cosine annealing schedule with warm restarts.

Finally, we can conclude that the correct choice of the optimization strategy heavily impacts the training of a neural network. Specialized algorithms like Adam outperform stochastic gradient descent and are well suited for image denoising with deep learning. Of course, there are many more optimization algorithms to choose from, and there might even be better-performing ones, but Adam is a good choice. Finding even better optimization algorithms is an ongoing goal of current research and is left for future work. For the rest of this thesis, we use the Adam optimizer with a constant learning rate schedule.

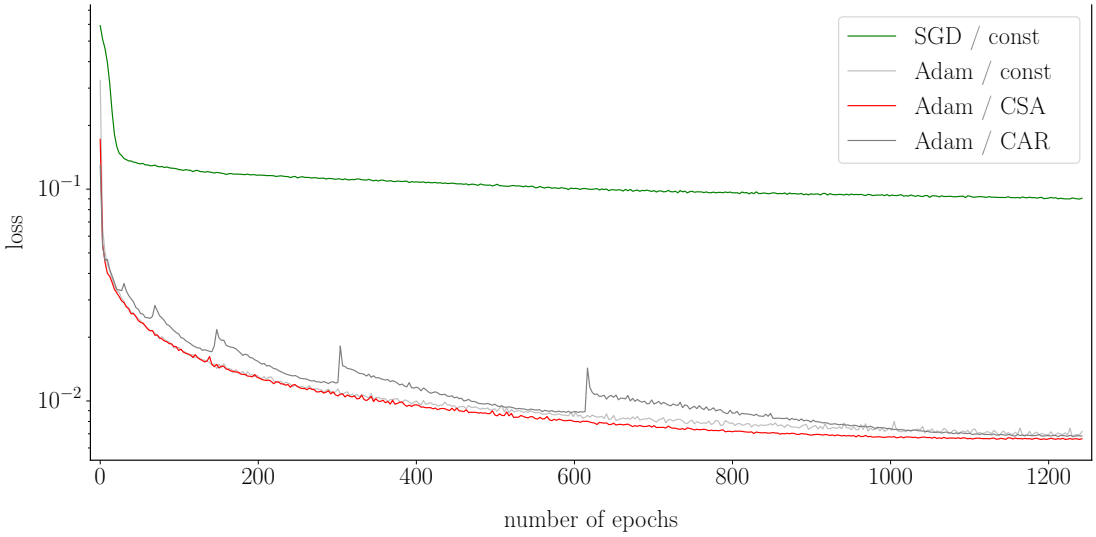



Figure 7.9: Comparison between three learning rate schedules applied to the Adam optimizer. We also included the best performing stochastic gradient descent technique for reference.

## 7.4 The Auxiliary Feature Image Selection

In section 6, we introduced the concept of auxiliary feature images. We now want to evaluate the impact on the quality of our denoising neural networks. We trained three networks based on the OIDN architecture $\mathcal{M}_{OIDN}$. The first network $\mathcal{M}_n$ takes only the noisy image as input, the second one $\mathcal{M}_{ann}$ uses the albedo and normal images as additional features, and lastly, we trained a network $\mathcal{M}_{dsnn}$ utilizing the diffuse, specular, and normal images. Every training process utilized the Adam optimization algorithm with a constant learning rate $\mu = 0.0001$ and the combined loss $\mathcal{L}_{MIX}$ (equation 7.1).

Figure 7.10 visualizes the training set combined loss convergence when using additional input information like albedo/normal or diffuse/specular/normal images. The network only trained with the noisy images converges to a significantly higher error than the others. This observation might imply that the denoising quality is worse than the models

trained with additional images. However, the difference between the albedo/normal and diffuse/specular/normal approach is minor when looking at the graph.

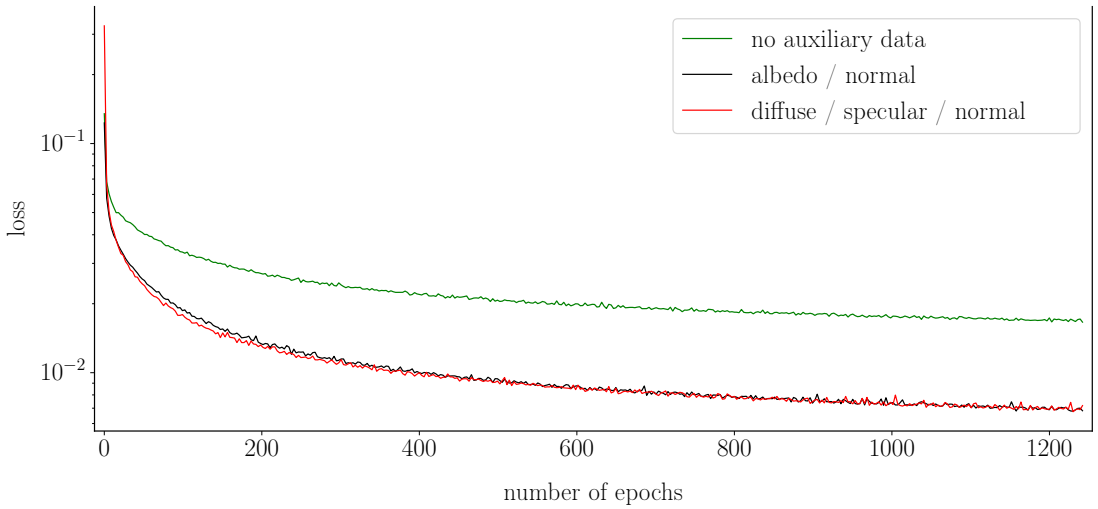



Figure 7.10: Comparison of the loss convergence during training when using different auxiliary feature image selections.

Especially with only a few samples per pixel, the noisy input image does not provide enough information to reconstruct a good denoised image. The auxiliary feature images do *not* assure perfect denoising but substantially improve the quality of the result. Figure 7.11 depicts the results when denoising the office scene from the test data set with the different neural network models. Only using the noisy images as inputs results in bad reconstruction. $\mathcal{M}_{ann}$ and $\mathcal{M}_{dsnn}$ yield exceptionally better results. The denoised images still contain high-frequency details even though the noisy input image alone does not contain enough information. The human observer would unambiguously rate the quality of the denoised images with auxiliary feature images higher. Although the difference between $\mathcal{M}_{ann}$ and $\mathcal{M}_{dsnn}$ is less visible, we can still identify some inequalities.

For example, when considering the clock at the background wall, we can observe an improvement of the network, which uses the separated diffuse and specular images compared to the network with the albedo image.

(a) noisy (b) ground truth (c) $\mathcal{M}_n$ (d) $\mathcal{M}_{ann}$ (e) $\mathcal{M}_{dsnn}$

Figure 7.11: Denoising the office scene using neural networks trained with different selections of auxiliary feature images.

It is important to note that the additional input features only change the neural network's input dimension. The inner architecture stays completely identical. This property is why the different approaches do not significantly change the training times of the neural networks.

The exceptional performance of the diffuse, specular, and normal images is why we used this combination of input features in all of our hyperparameter tests. Auxiliary input features are the key to pixel-based image denoising with machine learning. Finding even better additional input images could be an exciting goal for future research.

## 7.5 The Network Architecture

The network architecture is one of the most critical design choices when using machine learning to solve any task. However, till now, we only used the OIDN network architecture $\mathcal{N}_{OIDN}$ for our evaluation. In the following sections, we want to discuss and compare different designs. Every training process in the following sections uses the combined loss $\mathcal{L}_{MIX}$ (equation 7.1), the Adam optimization algorithm, and diffuse, specular, and normal auxiliary feature images.

### 7.5.1 MLP Denoising

The approach using a plain Multilayer Perceptron is based on the work from H. C. Burger, C. J. Schuler, and S. Harmeling published by the Max Planck Institute for Intelligent Systems in Tübingen, Germany [BSH12].

Using MLPs for image processing tasks is not trivial. We already explained that the number of input values and output values is constant (section 4.1.2). Therefore, we can only use images with a fixed size as the input to our network. Thus, H. C. Burger et al. [BSH12] introduced patch-based denoising. First, the noisy images get subdivided into patches with a fixed size. Then, we can denoise each patch separately and finally merge every denoised patch to obtain the clean image. Note that every patch is denoised with the same Multilayer Perceptron. Figure 7.12 illustrates the process.

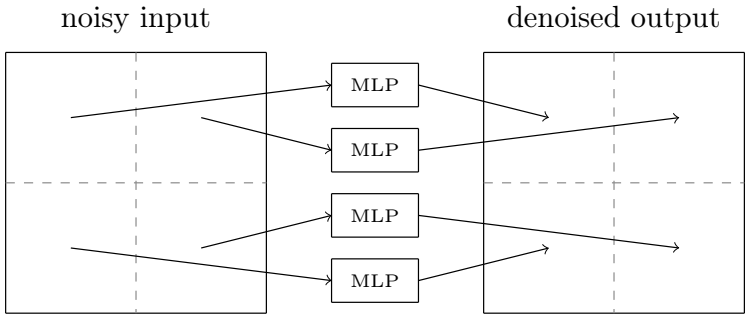



Figure 7.12: Depiction of the patch-based MLP denoising procedure.

The authors of the original paper further enhanced the decomposition of the input image. They use overlapping patches and weight the denoised patches with a Gaussian window.

The patches are basically obtained by moving a kernel over the input image (kernel size $17 \times 17$, stride $s = (3, 3)$, padding $p = (0, 0)$). In the final merging process, the overlapping parts get averaged to avoid artifacts.

This approach has multiple disadvantages compared to CNN denoising.

1. Complexity:
   The input image decomposition and the merging procedure introduce significant complexity to the denoising algorithm.

2. Time Consumption:
   The MLP forward pass for each patch, and the decomposition and merging take much time. For example, H. C. Burger et al. write that they need approximately one minute to denoise a $512 \times 512$ image.

3. Hyperparameter Optimization:
   This patch-based method requires hyperparameter optimization. For example, the size of the patches, the exact layout of the neural networks, and optimizations like weighting the denoised patches with a Gaussian window are parameters that are empirically chosen and might not be optimal. Admittedly, every deep learning approach requires a study of hyperparameters, but the number of seemingly arbitrarily chosen parameters is greater than when using CNNs.

The severe disadvantages are the reason why we chose *not* to implement the approach. However, we find it essential to mention the patch-based MLP denoising procedure. It reveals the substantial benefit of using Convolutional Neural Networks for image processing tasks.

### 7.5.2 OIDN Architecture Modifications

#### Classical Autoencoder

In section 4.1.4, we introduced convolutional autoencoders as a typical denoising architecture. Since Open Image Denoise uses the U-net variation (section 4.1.5) of convolutional autoencoders, we directly assessed the performance of the U-Net architecture. Using the OIDN architecture as a baseline allowed us to evaluate other hyperparameters first. Indeed, we never looked at classical denoising autoencoders. Do the skip connections in the U-Net architecture improve the denoising performance compared to a convolutional autoencoder? To investigate the difference between autoencoder and U-nets, we analyze the OIDN $\mathcal{M}_{OIDN}$ network and the OIDN network without skip connections $\mathcal{M}_{DAE}$. We trained both neural networks using the combined loss function $\mathcal{L}_{MIX}$ (equation 7.1) and the Adam optimization algorithm with a constant learning rate $\mu = 0.0001$. We also utilized the diffuse, specular, and normal images as additional input features.

Figure 7.13 depicts the difference between the loss during the training process when using the original OIDN network and the OIDN that does not contain skip connections. The OIDN architecture without skip connections basically represents a classical autoencoder. We observe a large gap between the two curves. Hence the U-Net architecture seems to work much better.

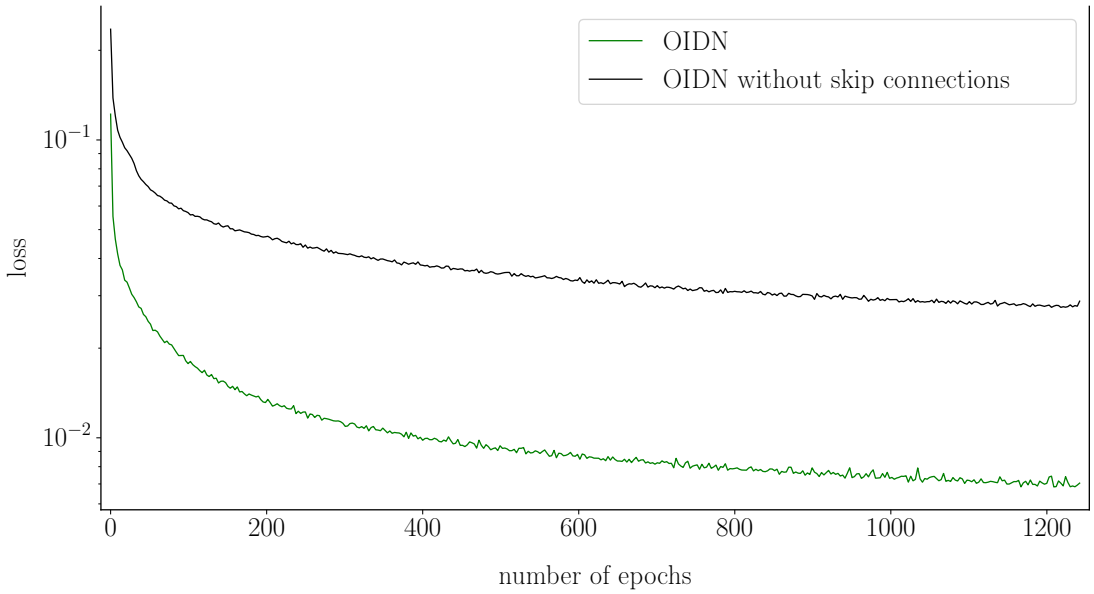



Figure 7.13: Comparison of the loss convergence during training when using the OIDN architecture and the OIDN architecture without skip connections.

We also compared the difference of the test set loss between the two network architectures at various stages during the training, depicted in table 7.2. As shown in section 7.1, the OIDN network reaches its global loss minimum of 0.0076 at around 230 epochs. From there, the network is overfitting. On the other hand, the test set loss of the classical autoencoder is relatively constant for the different epoch numbers, meaning that the network already reached the convergence value at 100 epochs. Furthermore, the original OIDN architecture heavily outperforms the autoencoder architecture. As a result, we observe a lower SSIM loss of the U-net architecture at 100, 230, 460, and 1270 epochs. We can directly see the difference in denoising quality by looking at figure 7.14.

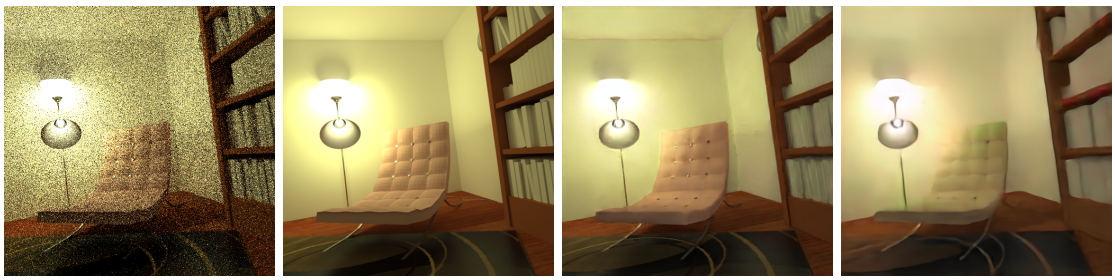
| Neural Network Model | 100 epochs | 230 epochs | 460 epochs | 1270 epochs |
|---|---|---|---|---|
| OIDN | 0.0103 | 0.0076 | 0.0168 | 0.0252 |
| OIDN without skip connections | 0.0536 | 0.0557 | 0.0537 | 0.0527 |

Table 7.2: SSIM loss of the test data when denoising with the OIDN architecture and the OIDN architecture without skip connections.

### Smaller Capacity

A crucial parameter to avoid overfitting is the model capacity (section 4.3.2). Since overfitting is already happening at epoch 230 (section 7.1), we try to decrease the model capacity and compare the denoising performance of the resulting models. Figure 5.1 shows the exact U-net architecture of Open Image Denoise.

Every convolutional layer is defined, among other things, by the number of input and output channels. These values determine the number of feature images in each layer.



(a) noisy (b) ground truth (c) $\mathcal{M}_{OIDN}$ (d) $\mathcal{M}_{DAE}$

Figure 7.14: Comparison of the U-net architecture and the classical autoencoder.

Therefore, we can reduce the model capacity by decreasing the number of input and output channels accordingly.

We compare two smaller neural network models $\mathcal{M}_1$ and $\mathcal{M}_2$ with the original OIDN U-net $\mathcal{M}_{OIDN}$. We define two functions that transform the original number of channels $c_{oidn}$ into smaller values $c_1$, $c_2$ for $\mathcal{N}_1$ and $\mathcal{N}_2$ respectively.

$$c_1 = \lfloor c_{oidn} \cdot 66\% \rfloor \tag{7.2}$$
$$c_2 = \lfloor c_1 \cdot 50\% \rfloor \tag{7.3}$$

We trained the neural networks using the combined loss $\mathcal{L}_{MIX}$ (equation 7.1) and the Adam optimization algorithm with a constant learning rate $\mu = 0.0001$. Each network also utilizes diffuse, specular, and normal images as auxiliary input data.

Table 7.3 depicts the training times needed to learn for 1270 epochs. As expected, the smaller network capacities significantly speed up the training process. This speedup most likely implies a similar speedup of the network inference.

| Neural Network Model | Training Time [*min*] | Time per Epoch [*min*] |
|---|---|---|
| $\mathcal{M}_{OIDN}$ | 710 | 0.560 |
| $\mathcal{M}_1$ | 525 | 0.413 |
| $\mathcal{M}_2$ | 408 | 0.321 |

Table 7.3: Training time (in minutes) needed to learn for 1270 epochs.

Figure 7.15 visualizes the test loss of the different neural networks. We can observe that the original model $\mathcal{M}_{OIDN}$ has the lowest loss and thus provides the best denoising results, which means that decreasing the model capacity lowers the denoising quality. Furthermore, contrary to the expectation, we see that the middle-sized network $\mathcal{M}_1$ yields the worst loss values. We can not give a reason for that result. It might just reflect the notion of unpredictability when working with neural networks. In some cases, machine learning is more like art than science. Figure 7.16 shows the visual difference between the networks with different capacity.

To summarize this section, we can certainly say that a smaller network does not improve the denoising quality. It instead has a negative influence on the quality. Additionally, we decided not to evaluate networks with larger capacity. Larger and deeper networks would further increase the training time and most likely not improve the denoising performance. This is because the original OIDN network $\mathcal{M}_{OIDN}$ is already overfitting very early. Finally, we must admit that the model capacity is a complex hyperparameter and needs to be evaluated for each individual application.

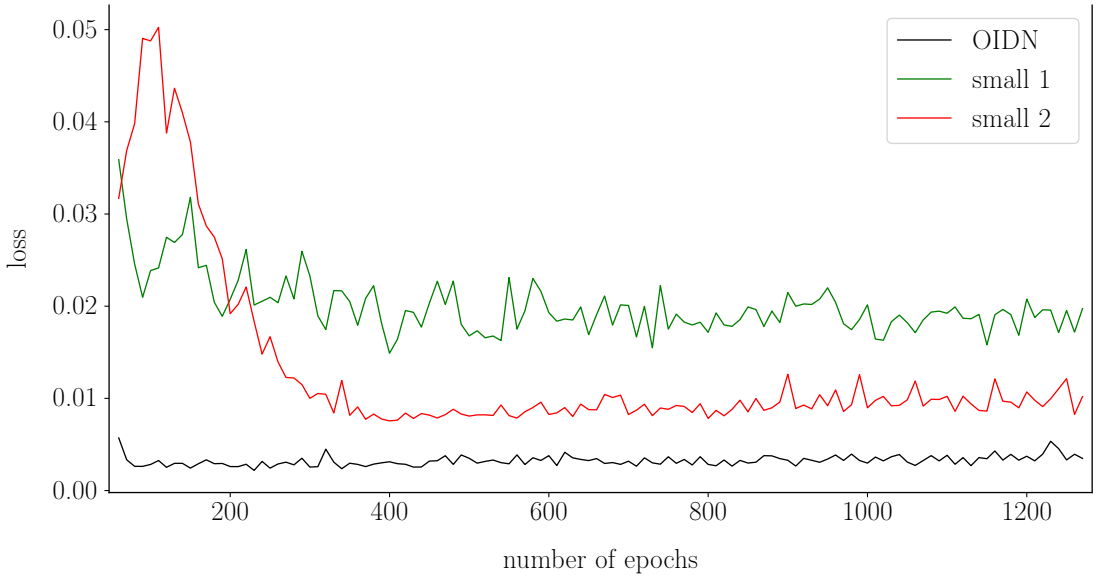



Figure 7.15: Test set loss comparison of $\mathcal{M}_{OIDN}$, $\mathcal{M}_1$ (small 1), and $\mathcal{M}_2$ (small 2).

(a) noisy (b) ground truth (c) $\mathcal{M}_{OIDN}$ (d) $\mathcal{M}_1$ (e) $\mathcal{M}_2$

Figure 7.16: Denoising results of the neural networks with different capacities.

# 8 Denoising Quality Comparison

This is the final evaluation chapter in this thesis. After investigating the influence of various hyperparameters, the OIDN network architecture $\mathcal{M}_{OIDN}$ is performing best. However, we changed the number of input channels such that the neural network takes diffuse, specular, and normal images instead of only albedo and normal images as additional feature values. This modification has proven to work better in our tests. We trained the network with the best working loss function $\mathcal{L}_{MIX}$ that merges the Multi-Scale Structural Similarity Index and Mean Absolute Error and ultimately yields the best denoising quality. The Adam optimization algorithm allows fast training of our model. Section 4.3.2 covered overfitting and underfitting and concluded that the OIDN network trained for 230 epochs provides the best denoising results and, therefore, will be the final comparison network.

The denoising algorithms we want to use as comparison are the official Open Image Denoise library and Block-matching and 3D filtering (BM3D) [DFKE07]. BM3D is a state-of-the-art denoising algorithm that is not based on deep learning. We do not expect to get near Intel's denoising quality of the OIDN model, but we want to present our results. The following pages provide a side-by-side comparison of the denoising results and the associated loss values. Every noisy image is from our test set and, therefore, has not been used to train our model.

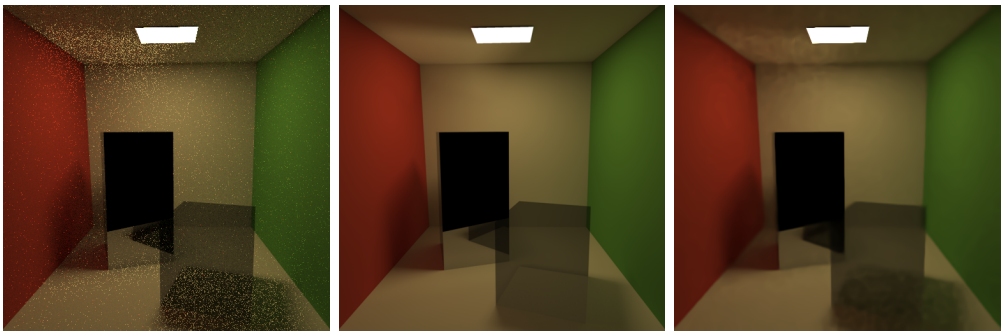
Remember that a loss value is only a mathematical metric that somehow calculates the similarity of two images. In Monte Carlo denoising, we want visually good-looking images. Consequently, the quality metric provided by a human observer is the best we can use.

In the following examples, we notice that BM3D takes significantly longer than the machine learning approaches. When using neural networks for denoising, the main computational effort is shifted to the training process. The advantage is that training a neural network model is a one-time effort that is amortized over the time in use.

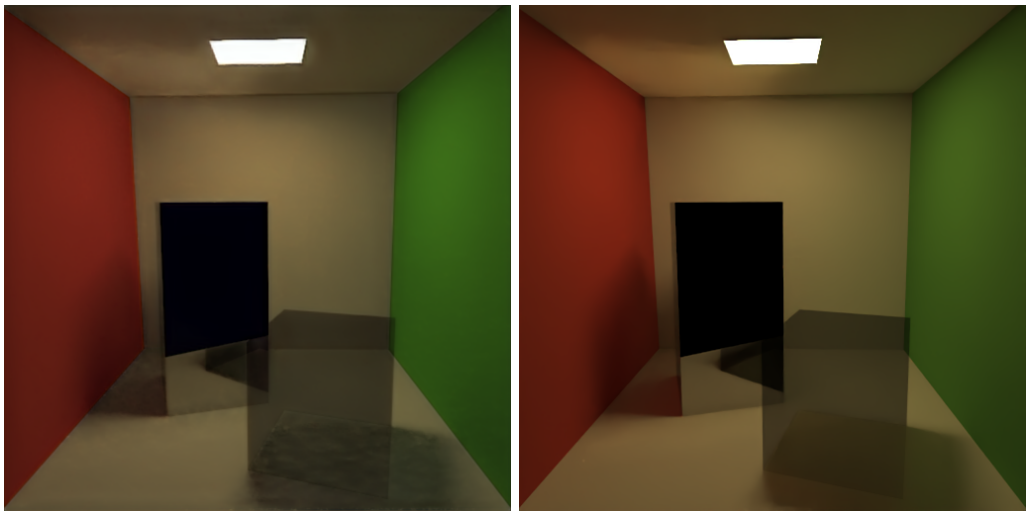
The official OIDN neural network model works remarkably well and outperforms the other approaches. BM3D results in good loss values but often lacks high-frequency details and contains artifacts when observed by a human. Furthermore, our learned model introduces a slight bias to the denoised image. That is most likely due to an insufficient training data set. We believe that with a large and general training data set, it is possible to reach the denoising quality of the Intel denoiser. However, this requires an enormous amount of computational power. Nonetheless, our denoising approach provides okay results even though we test it with images utterly different from the training data.

## 8.1 The Cornellbox

The cornellbox scene is a typical test scene in computer graphics. BM3D clearly blurs the room's edges and introduces smear artifacts in regions with much noise. Our network preserves sharp edges but slightly brightens up the image. The original OIDN network significantly outperforms the other approaches in shadow areas.



(a) noisy (b) ground truth (c) BM3D



(d) $\mathcal{M}_{OIDN}$ (e) official OIDN

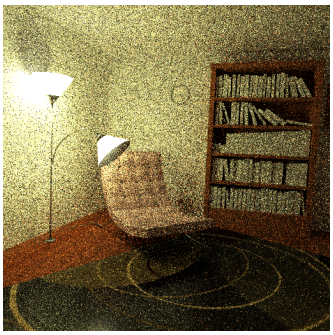
Figure 8.1: The denoising results of the cornellbox when using different denoisers.

| Neural Network Model | BM3D | $\mathcal{M}_{OIDN}$ | official OIDN |
|---|---|---|---|
| denoising time | 8.9205 sec | 0.3146 sec | 0.11 sec |
| MAE | 705.705 | 1279.56 | 442.001 |
| MSE | 32.9903 | 41.0952 | 17.7012 |
| PSNR | 32.9809 | 32.0268 | 35.6847 |
| SSIM | 0.982815 | 0.9561 | 0.9923 |

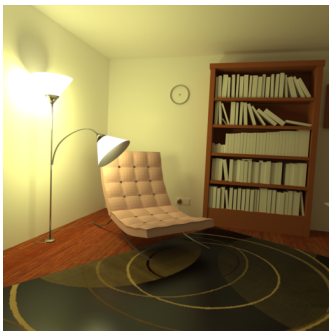
Table 8.1: Denoising time and loss comparison of the cornellbox.

## 8.2 The Office Scene

The office scene example provides a highly corrupted noisy image. This displays the disadvantage of BM3D, which does not use auxiliary feature images and therefore can not reconstruct a noise-free image. Both deep learning approaches provide better results than the hand-coded BM3D algorithm.



(a) noisy



(b) ground truth

(c) BM3D

(d) $\mathcal{M}_{OIDN}$

(e) official OIDN

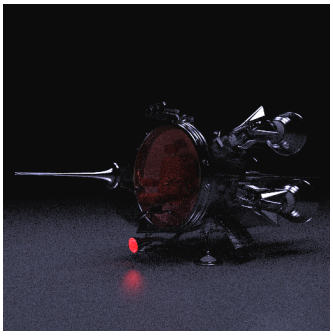
Figure 8.2: The denoising results of the office scene when using different denoisers.

| Neural Network Model | BM3D | $\mathcal{M}_{OIDN}$ | official OIDN |
|---|---|---|---|
| denoising time | 9.1319 sec | 0.4645 sec | 0.121 sec |
| MAE | 4594.12 | 2178.2 | 2060.6 |
| MSE | 445.27 | 130.9 | 102.784 |
| PSNR | 21.6785 | 26.9953 | 28.0455 |
| SSIM | 0.7990 | 0.9125 | 0.932653 |

Table 8.2: Denoising time and loss comparison of the office scene.

## 8.3 The Spaceship Scene

The spaceship scene provides an example that is entirely different from the training data samples. Firstly, the official OIDN network significantly outperforms the other approaches. BM3D partly blurs the spaceship, whereas our network introduces a significant color shift.



(a) noisy

(b) ground truth

(c) BM3D

(d) $\mathcal{M}_{OIDN}$

(e) official OIDN

Figure 8.3: The denoising results of the spaceship scene when using different denoisers.

| Neural Network Model | BM3D | $\mathcal{M}_{OIDN}$ | official OIDN |
|---|---|---|---|
| denoising time | 8.92504 sec | 0.3468 sec | 0.108 sec |
| MAE | 637.106 | 1195.04 | 567.369 |
| MSE | 36.5436 | 64.0954 | 43.3993 |
| PSNR | 32.5366 | 30.0965 | 31.7899 |
| SSIM | 0.935532 | 0.77789 | 0.9583 |

Table 8.3: Denoising time and loss comparison of the spaceship scene.

# 9 Conclusion and Future Work

This thesis introduced the theoretical background of path tracing and the origin of one of the most dominant issues in rendering: Monte Carlo noise. We briefly presented different approaches that directly reduce the variance of Monte Carlo integration in path tracing. We show that using denoising algorithms as a post-processing step is another method to decrease the noise per rendering time effectively.

Machine learning approaches that use neural networks to learn a denoising function directly constitute a significant advancement in Monte-Carlo denoising. We concisely introduced the deep learning foundations needed to create denoising neural networks. When designing denoising solutions, loss functions, learning strategies, and the overall network topology are important key points that have to be considered. The combined loss function, merging MS-SSIM and MAE, the Adam optimization algorithm, and a modification of a classical denoising autoencoder, called U-net, are important hyperparameters that positively influence the denoising quality.

We provided an in-depth evaluation of the hyperparameters mentioned beforehand. Additionally, this thesis verified the importance of additional feature images required to reconstruct heavily corrupted images. Our solution using diffuse, specular, and normal images instead of the standard albedo and normal images also used by Open Image Denoise worked best in testing. Overfitting and underfitting are critical points that have to be considered carefully. Analyzing the generalization gap between a test data set and the training data set allows conclusions about the network's performance when used with unseen input data.

We figured that a large and general training data set that completely covers all edge cases and includes a wide variety of different images is indispensable for success. However, the computational effort of creating large training and test sets is significant and requires state-of-the-art rendering clusters. Therefore, our training and test data are imperfect and leave much room for improvement. Nevertheless, we certainly showed that learning denoising filters is promising even though we do not reach the performance of Open Image Denoise by Intel.

Lastly, we compared our training results with a non-machine-learning denoising algorithm BM3D and the original Open Image Denoise model. The comparison visualizes the disadvantages of hand-coded denoising algorithms, such as BM3D. They particularly lack the ability to reconstruct images with low sample counts because they can not benefit from additional feature images.

We can finally use deep convolutional neural networks to denoise images rendered by Rodent. The direct integration of the denoising solution into the Rodent codebase is presented in the follow-up thesis [Mey21]. The denoising post-processing step reduces the rendering time needed to obtain noise-free images.

Of course, this thesis does not cover all possible hyperparameters that could influence the quality and performance of a machine learning denoising solution. Therefore, finding even better optimization algorithms, network topologies, and important hyperparameters that affect denoising is essential for future research.

Another way to improve the denoising quality of path tracers using Monte Carlo integration might be to use sample-based denoising directly. Instead of only using the exact pixel values of the noisy output image, a denoiser could directly use every primary estimator of the Monte Carlo integration. There already exist approaches that use convolutional neural networks that directly use sample information to denoise the noisy image [GLA+19].

Machine learning is currently a hot topic in computer science and constantly evolving. For example, deep learning approaches might also be useful for other noise-reducing techniques, such as path guiding.

# Bibliography


[ABAAU18] Saad Albawi, Oguz Bayat, Saad Al-Azawi, and Osman N. Ucan. Social touch gesture recognition using convolutional neural network. *Computational intelligence and neuroscience*, 2018:6973103–6973103, Oct 2018. 30402085[pmid].

[App68] Arthur Appel. Some techniques for shading machine renderings of solids. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, AFIPS '68 (Spring), page 37–45, New York, NY, USA, 1968. Association for Computing Machinery.

[Bit16] Benedikt Bitterli. Rendering resources, 2016. https://benedikt-bitterli.me/resources/.

[BSH12] Harold C. Burger, Christian J. Schuler, and Stefan Harmeling. Image denoising: Can plain neural networks compete with bm3d? In *2012 IEEE Conference on Computer Vision and Pattern Recognition*, pages 2392–2399, 2012.

[CKS+17] Chakravarty R. Alla Chaitanya, Anton S. Kaplanyan, Christoph Schied, Marco Salvi, Aaron Lefohn, Derek Nowrouzezahrai, and Timo Aila. Interactive reconstruction of monte carlo image sequences using a recurrent denoising autoencoder. *ACM Trans. Graph.*, 36(4), July 2017.

[DFKE07] Kostadin Dabov, Alessandro Foi, Vladimir Katkovnik, and Karen Egiazarian. Image denoising by sparse 3-d transform-domain collaborative filtering. *IEEE Transactions on Image Processing*, 16(8):2080–2095, 2007.

[DV18] Vincent Dumoulin and Francesco Visin. A guide to convolution arithmetic for deep learning, 2018.

[FM82] Kunihiko Fukushima and Sei Miyake. Neocognitron: A self-organizing neural network model for a mechanism of visual pattern recognition. In *Competition and cooperation in neural nets*, pages 267–285. Springer, 1982.

[FZFZ19] Linwei Fan, Fan Zhang, Hui Fan, and Caiming Zhang. Brief review of image denoising techniques. *Visual Computing for Industry, Biomedicine, and Art*, 2(1):7, Jul 2019.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[GLA+19] Michaël Gharbi, Tzu-Mao Li, Miika Aittala, Jaakko Lehtinen, and Frédo Durand. Sample-based monte carlo denoising using a kernel-splatting network. *ACM Trans. Graph.*, 38(4), jul 2019.

[HN92] ROBERT HECHT-NIELSEN. Iii.3 - theory of the backpropagation neural network**based on "nonindent" by robert hecht-nielsen, which appeared in proceedings of the international joint conference on neural networks 1, 593–611, june 1989. © 1989 ieee. In Harry Wechsler, editor, *Neural Networks for Perception*, pages 65–93. Academic Press, 1992.

[HR47] Pao-Lu Hsu and Herbert Robbins. Complete convergence and the law of large numbers. *Proceedings of the National Academy of Sciences of the United States of America*, 33(2):25, 1947.

[Kaj86] James T. Kajiya. The rendering equation. In *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '86, page 143–150, New York, NY, USA, 1986. Association for Computing Machinery.

[KB17] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[KBS15] Nima Khademi Kalantari, Steve Bako, and Pradeep Sen. A machine learning approach for filtering monte carlo noise. *ACM Trans. Graph.*, 34(4), July 2015.

[KPR12] Alexander Keller, Simon Premoze, and Matthias Raab. Advanced (quasi) monte carlo methods for image synthesis. In *ACM SIGGRAPH 2012 Courses*, SIGGRAPH '12, New York, NY, USA, 2012. Association for Computing Machinery.

[KvD78] T. Kloek and H. K. van Dijk. Bayesian estimates of equation system parameters: An application of integration by monte carlo. *Econometrica*, 46(1):1–19, 1978.

[LBD+89] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1(4):541–551, 1989.

[LBH+18] Roland Leißa, Klaas Boesche, Sebastian Hack, Arsène Pérard-Gayot, Richard Membarth, Philipp Slusallek, André Müller, and Bertil Schmidt. Anydsl: A partial evaluation framework for programming high-performance libraries. *Proc. ACM Program. Lang.*, 2(OOPSLA), October 2018.

[LH16] Ilya Loshchilov and Frank Hutter. SGDR: stochastic gradient descent with restarts. *CoRR*, abs/1608.03983, 2016.

[LL05] A.P.S. Lins and T.B. Ludermir. Hybrid optimization algorithm for the definition of mlp neural network architectures and weights. In *Fifth International Conference on Hybrid Intelligent Systems (HIS'05)*, pages 6 pp.–, 2005.

[Mey21] Joshua Meyer. Integrating a denoising neural network into a renderer using a high-level functional language and partial evaluation. 2021.

[MP43] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, Dec 1943.

[Mur22] Kevin P. Murphy. *Probabilistic Machine Learning: An introduction.* MIT Press, 2022.

[PGML+19] Arsène Pérard-Gayot, Richard Membarth, Roland Leißa, Sebastian Hack, and Philipp Slusallek. Rodent: Generating renderers without writing a generator. *ACM Transactions on Graphics (TOG) (Proceedings of SIGGRAPH 2019)*, 38(4):40:1–40:12, July 2019.

[PJH16] Matt Pharr, Wenzel Jakob, and Greg Humphreys. *Physically Based Rendering: From Theory to Implementation (3rd ed.).* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, October 2016.

[PM92] Sankar K Pal and Sushmita Mitra. Multilayer perceptron, fuzzy sets, classifiaction. 1992.

[RFB15] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*, pages 234–241. Springer, 2015.

[RZL17] Prajit Ramachandran, Barret Zoph, and Quoc V Le. Searching for activation functions. *arXiv preprint arXiv:1710.05941*, 2017.

[SK19] Connor Shorten and Taghi M. Khoshgoftaar. A survey on image data augmentation for deep learning. *Journal of Big Data*, 6(1):60, Jul 2019.

[SKP99] László Szirmay-Kalos and Werner Purgathofer. Analysis of the quasi-monte carlo integration of the rendering equation. In *Proceedings of Winter School of Computer Graphics*, volume 99, pages 281–288. Citeseer, 1999.

[SS17] Sagar Sharma and Simone Sharma. Activation functions in neural networks. *Towards Data Science*, 6(12):310–316, 2017.

[TM98] C. Tomasi and R. Manduchi. Bilateral filtering for gray and color images. In *Sixth International Conference on Computer Vision (IEEE Cat. No.98CH36271)*, pages 839–846, 1998.

[Vas08] Saeed V Vaseghi. *Advanced digital signal processing and noise reduction.* John Wiley & Sons, 2008.

[VDM86] C. Van Der Malsburg. Frank rosenblatt: Principles of neurodynamics: Perceptrons and the theory of brain mechanisms. In Günther Palm and Ad Aertsen, editors, *Brain Theory*, pages 245–248, Berlin, Heidelberg, 1986. Springer Berlin Heidelberg.

[VG95] Eric Veach and Leonidas J. Guibas. Optimally combining sampling techniques for monte carlo rendering, 1995.

[Wei00] Stefan Weinzierl. Introduction to monte carlo methods. In *Proceedings of High Energy Physics - Phenomenology.* Topical lectures given at the Research School Subatomic Physics, 2000.

[ZGFK15] Hang Zhao, Orazio Gallo, Iuri Frosio, and Jan Kautz. Loss functions for neural networks for image processing. *CoRR*, abs/1511.08861, 2015.

# List of Figures


2.1 Visualization of path tracing with and without Next Event Estimation. . . 6

4.1 Depiction of an artificial neuron. . . . . 12
4.2 Depiction of a fully connected MLP with *two* hidden layers. . . . . 12
4.3 Depiction of a 3-dimensional convolution. . . . . 14
4.4 Depiction of the kernel movement with $p_x = 1$ and $s_x = 2$. . . . . 15
4.5 Depiction of the max pooling and the average pooling operation. We used a $2 \times 2$ kernel and $p = (0, 0)$, $s = (2, 2)$, and $d = (1, 1)$. . . . . 16
4.6 Depiction of the nearest neighbor upsampling algorithm. . . . . 16
4.7 Depiction of a transposed convolution between a $2 \times 2$ input matrix and a $3 \times 3$ kernel using stride $s = (2, 2)$ and padding $p = (1, 1)$. . . . . 17
4.8 Depiction of various activation functions. . . . . 19
4.9 Depiction of a simple denoising autoencoder which denoises the corrupted input data given as a 3-dimensional tensor. . . . . 19
4.10 Example of a convolutional autoencoder using 6 convolutional layers (conv.), 2 max pooling layers (m.pool.) and 2 upsampling layers (ups.). $w$ and $h$ are the input and output width and height. . . . . 20
4.11 The autoencoder from figure 4.10 with two skip connections. This type of neural network is called a U-net. . . . . 21
4.12 Depicts three common training scenarios: high model capacity, optimal model capacity, and low model capacity. High model capacity is prone to overfitting, while low model capacity can not fit the training data. . . . . 24
4.13 Depiction of different learning rate schedules. . . . . 25

5.1 The U-net architecture used in Open Image Denoise by Intel. . . . . 29

6.1 Depiction of the albedo and normal auxiliary input images. . . . . 32
6.2 Comparison between the albedo image and the diffuse and specular images. 33
6.3 Depiction of four example perspectives from the training data set. . . . . 34
6.4 Depiction of four example perspectives from the test data set. . . . . 34

7.1 Development of the training set loss and test set loss during the learning process. . . . . 38
7.2 Denoising the spaceship scene using neural networks trained with different numbers of epochs. . . . . 39
7.3 Visualization of the pixel-wise difference between the images and the ground truth. A difference greater than 10% is marked in black. . . . . 39
7.4 Denoising the office scene using neural networks trained with different numbers of epochs. . . . . 39
7.5 Denoising results of neural networks trained with different loss functions. . 41

7.6 Depiction of the denoising result when using a network trained with $\mathcal{L}_{MIX}$. 42
7.7 Visualization of the pixel-wise difference between the denoised images and the ground truth. A difference greater than 3% is marked in black. . . . . 42
7.8 Comparison between three learning rate schedules applied to stochastic gradient descent. . . . . 43
7.9 Comparison between three learning rate schedules applied to the Adam optimizer. We also included the best performing stochastic gradient descent technique for reference. . . . . 44
7.10 Comparison of the loss convergence during training when using different auxiliary feature image selections. . . . . 45
7.11 Denoising the office scene using neural networks trained with different selections of auxiliary feature images. . . . . 45
7.12 Depiction of the patch-based MLP denoising procedure. . . . . 46
7.13 Comparison of the loss convergence during training when using the OIDN architecture and the OIDN architecture without skip connections. . . . . 48
7.14 Comparison of the U-net architecture and the classical autoencoder. . . . 49
7.15 Test set loss comparison of $\mathcal{M}_{OIDN}$, $\mathcal{M}_1$ (small 1), and $\mathcal{M}_2$ (small 2). . . 50
7.16 Denoising results of the neural networks with different capacities. . . . . 50

8.1 The denoising results of the cornellbox when using different denoisers. . . 52
8.2 The denoising results of the office scene when using different denoisers. . . 53
8.3 The denoising results of the spaceship scene when using different denoisers. 54

# List of Tables

7.1 Training time (in minutes) needed to learn for 1270 epochs. . . . . 40
7.2 SSIM loss of the test data when denoising with the OIDN architecture and the OIDN architecture without skip connections. . . . . 48
7.3 Training time (in minutes) needed to learn for 1270 epochs. . . . . 49

8.1 Denoising time and loss comparison of the cornellbox. . . . . 52
8.2 Denoising time and loss comparison of the office scene. . . . . 53
8.3 Denoising time and loss comparison of the spaceship scene. . . . . 54