Acknowledgments

The time I spent working on this thesis has been very enjoyable thanks to the great people at Disney Research Zürich. I would like to especially express my gratitude to my advisors, Brian, Fabrice and Jan, for their persistent commitment to the project and the many enlightening conversations we had. I also appreciate the help of and conversations with my supervisor, Andreas Krause, and other colleagues at Disney that made the project a pleasant experience.

During the project, I have made extensive use of several Open Source software products. I am thankful to all contributors to TensorFlow and to the author of the Tungsten renderer, Benedikt Bitterli.

Finally, a word of thanks to the following blendswap.com artists for creating the scenes in our dataset: thecali (car and spaceship), MrChimp2313 (house), UP3D (lamp), SlykDrako (bedroom), nacimus (bathroom), Delatronic (dragon), Wig42 (horse room, dining room, red room, staircase), Jay-Artist (living room), and NickWoelk (shaving kit). The museum scene was created by Alvaro Luna Bautista and Joel Anderson.
# Table of Contents

1. **Introduction** ................................................. 1

2. **Theoretical Background and Related Work** .......................... 3
   2.1. Rendering .................................................................. 3
   2.2. Denoising ................................................................. 5
       2.2.1. Bilateral filter ....................................................... 6
       2.2.2. Non-Local Means ............................................... 6
       2.2.3. Weighted Linear Regression ................................. 7
       2.2.4. Nonlinearly Weighted First-order Regression .......... 7
       2.2.5. Adaptive Polynomial Rendering ......................... 8
   2.3. Machine Learning .................................................... 8
       2.3.1. Neural networks ................................................ 9
       2.3.2. Stochastic gradient descent ................................. 10
       2.3.3. Convolutional neural networks ............................. 12
       2.3.4. Neural networks for low-level image processing .... 13

3. **Problem statement & challenges** .................................. 15
   3.1. The denoiser as an estimator ..................................... 15
   3.2. Challenges ............................................................. 15
       3.2.1. Cost of generating training data ............................ 16
       3.2.2. High-dynamic-range data ................................. 16
       3.2.3. Choosing the loss function ................................. 17
       3.2.4. Architecture .................................................... 17

4. **Single-scale denoising** ............................................. 19
   4.1. Pre-processing ......................................................... 20
       4.1.1. Non-linear feature transformations ...................... 20
       4.1.2. Standardization ................................................. 25
       4.1.3. ZCA Whitening ................................................ 25
   4.2. Network architecture .............................................. 26
       4.2.1. Patches versus ConvNets .................................... 26
       4.2.2. Repeated architecture ...................................... 28
       4.2.3. Residual architecture ...................................... 29
   4.3. Reconstruction ...................................................... 30
       4.3.1. Direct prediction .............................................. 30
       4.3.2. Kernel-prediction .......................................... 31
Introduction

In photo-realistic image synthesis, Monte Carlo (MC) path tracing is a powerful and general algorithm to simulate light transport. In MC path tracing, the color of a pixel is computed by randomly sampling light paths that connect the camera to light sources through multiple interactions with the scene. The mean intensity of many such samples constitutes a noisy estimate of the total illumination. Unfortunately, in realistic scenes with complex light transport, these samples might have large variance, and the variance of their mean only decreases linearly with respect to the number of samples per pixel. Typically, thousands of samples per pixel are required to achieve a visually converged rendering. This results in prohibitively long render times.

Several solutions have been proposed to improve the performance of probabilistic path tracing algorithms. Apart from clever sampling schemes, the two main approaches are: (1) adaptive sampling and (2) denoising. In adaptive sampling [Overbeck et al. 2009; Rousselle et al. 2011; Zwicker et al. 2015] the number of sampled paths per pixel is inhomogeneous, and determined based on an estimate of the variance of the pixel’s sample distribution. The goal of denoising [Takeda et al. 2007; Moon et al. 2016; Bitterli et al. 2016] is to reduce the pixel variances by leveraging spatial redundancy in the image. The current state-of-the-art denoisers achieve this by replacing pixel colors by a weighted sum of their neighborhoods, based on a local regression. Such models can make use of rendering-specific guiding data, such as the pixel’s depth in the scene, normal directions, texture colors, etc. These features can be extracted with less noise than the final color, and can greatly help preserving sharp edges and image texture.

Traditionally, denoising algorithms have been engineered and hand-tuned by researchers. Recently Kalantari et al. [2015] started to apply machine learning to denoise MC renderings by learning the optimal tuning parameters of an existing filter from data. The aim of this thesis is to take this a step further, and present an end-to-end learned denoiser. We employ several variants of convolutional neural networks [LeCun et al. 1995] that take noisy renderings and rendering features as input, and generate noise free versions of them.

This research is motivated by great successes that have been achieved by other machine learning algorithms in computer vision. After reaching state-of-the-art performance in many high-level computer vision tasks such as object recognition and detection [LeCun et al. 2015; Russakovsky et al. 2015], excellent results have been achieved in more fundamental subfields such as super-resolution [Bruna et al. 2015] and inpainting [Xie et al. 2012] as well. Convolutional neural networks have also shown to be successful in copying complex non-linear Photoshop filters [Xu et al. 2015]. Several authors, such as Burger et al. [2012] and Jain and Seung [2009] have presented results for natural image denoising that compete with the state-of-the-art.

Our work aims to evaluate and extend techniques proposed by machine learning researchers in several low level image processing areas to incorporate additional guiding features such as nor-
mals, depth, direct visibility, texture colors and variance information. Another aspect in which denoising MC renderings is different from regular image denoising is that in contrast to regular images, the colors in renderings are not constrained to a fixed range. Pixels that directly show a light source can be orders of magnitude brighter than dark regions in the image. The high-dynamic-range (HDR) nature of renderings forms a significant challenge for machine learning algorithms in this field. A third challenge lies in the availability of example data for the learning algorithm. MC noise is inhomogeneous and its distribution is not well understood. Therefore, the only feasible way to generate training data is to actually render many images to a high quality: an expensive process.

We will present several solutions to these challenges, our main contributions being: An end-to-end learned approach to denoising MC renderings, a framework for generating and augmenting training data, evaluation of several loss functions and their effect on visual quality of the denoised images, and the development of methodology to deal with HDR image data and additional feature buffers in machine learning. The performance of our best denoisers is competitive with the current state-of-the-art.
2 | Theoretical Background and Related Work

This section briefly discusses the basics of rendering, denoising and machine learning that are required for a good understanding of this thesis. It is by no means a complete reference on these topics. References to more elaborate sources are provided in the respective sections. In Section 2.2 on denoising and 2.3 on machine learning, relevant related work will be discussed as well.

2.1 Rendering

In a multitude of fields including games, the movie industry and medical simulations, images are synthesized from 3D models. The goal of these ‘renderings’ mostly is to imitate physical lighting as accurately as possible. The different fields have different constraints, though. In games, real time performance is crucial, whereas in the movie industry, the computation of a movie-frame may take hours.

To be able to simulate any effect in geometric optics, the movie industry recently started to use path tracing for rendering [Keller et al. 2015]. With path tracing, the light leaving an object in a certain direction is computed by integrating all incoming and generated light at that point. The nature of this computation is recursive, and it is governed by the rendering equation:

\[
L_o(\vec{x}, \vec{\omega}_o) = L_e(\vec{x}, \vec{\omega}_e) + \int \phi(\vec{x}, \vec{\omega}_i, \vec{\omega}_o) L_i(\vec{x}, \vec{\omega}_i) (\vec{\omega}_i \cdot \vec{n}) d\vec{\omega}_i,
\]

where \(L_o\) represents the total radiant power transmitted from an infinitesimal region around a point \(\vec{x}\) into an infinitesimal cone in the direction \(\vec{\omega}_o\). This quantity is called radiance. In the equation \(L_e\) is the emitted radiance (for light sources), \(\vec{n}\) is the normal direction at position \(\vec{x}\), \(\Omega\) is the unit hemisphere centered around \(\vec{n}\) containing all possible values for incoming directions \(\vec{\omega}_i\), and \(L_i\) represents the incoming radiance from \(\vec{\omega}_i\). The function \(\phi\) is called the bidirectional reflectance distribution function (BSDF). It captures the material properties of an object at \(\vec{x}\) [Kajiya 1986]. The basic path tracing algorithm is illustrated in Figure 2.1.

The recursive integrals in the rendering equation are usually evaluated using a MC approximation. To compute the pixel’s color, light paths are randomly sampled throughout the different bounces. We will use the notation

\[
\bar{p}_i = \frac{1}{n} \sum_{k=1}^{n} p_{i,k}, \quad p_{i,k} \sim \mathcal{S}, \quad \forall i \in [1,n],
\]
Figure 2.1: Path tracing. On the left, a simple setup shows a screen with a pixel in the middle. A hypothetical viewing ray that connects an object in the screen (a box) to the viewing point is drawn. The color of the pixel is determined by the incoming light, which depends on the ray that connects this point with the light source (left top). On the right image, the same scene is depicted, but now with indirect lighting. The image shows that to determine a pixel’s color, not only light sources contribute, but actually many objects in the scene might reflect some light.

to denote the MC estimate of the color of a pixel \( i \) as the mean of \( n \) independent samples \( p_{i,k} \) from the pixel’s sample distribution \( S_i \). The MC approximated \( \hat{p}_i \) is an unbiased estimate for the converged pixel color mean \( \bar{p}_i \) that would be achieved with an infinite number of samples:

\[
\hat{p}_i = \lim_{n \to \infty} \frac{1}{n} \sum_{k=1}^{n} p_{i,k}.
\]  

In unbiased path tracing, the mean of \( S_i \) equals \( \bar{p}_i \), and its variance depends on several factors. One cause might be that light rays sometimes just hit an object, and sometimes just miss it, or that they sometimes hit a light source, and sometimes not. This makes scenes with indirect lighting and many reflective objects particularly difficult to render. In these cases, the sample distribution is very skewed, and the samples \( p_{i,k} \) can be orders of magnitude apart.

The variance of the MC estimate \( \bar{p}_i \) based on \( n \) samples, follows from the variance of \( S_i \) as

\[
\text{Var} [\bar{p}_i] = \frac{1}{n} \text{Var} [S_i].
\]

Because the variance decreases linearly with respect to \( n \), the expected error \( \sqrt{\text{Var}[\bar{p}_i]} \) decreases as \( 1/\sqrt{n} \). The convergence of MC renderings is illustrated in Figure 2.2.

For a complete discussion on rendering and Monte Carlo path tracing, please refer to [Pharr and Humphreys 2010].
2.2 Denoising

To deal with the slow convergence of MC renderings, several denoising techniques have been proposed to reduce the variance of rendered pixel colors by leveraging spatial redundancy in images. Zwicker et al. [2015] classify denoising algorithms into *a priori* methods and *a posteriori* methods. The difference between the two categories is that *a priori* methods make use of analytical models based on intricate knowledge of the 3D scene, such as analytical material descriptions and geometry descriptions. Through this analytical approach, optimal local denoising filters can be constructed from theory. It is a powerful approach, but to make it feasible, existing approaches are limited to specific effects such as motion blur, depth of field, or ambient occlusion and often assume diffuse surfaces. *A posteriori* denoisers, on the other hand, operate from per-pixel statistics such as the mean and variance of the sample colors and possibly statistics of guiding features recorded during rendering, such as normal directions, texture information, direct visibility and camera depth. The aim for both kinds of denoising approaches is to estimate the ground truth pixel colors \( \hat{p} \), achieved when the number of samples goes to infinity.

In our work, we focus on *a posteriori* denoising, because of the inherent generality that can be achieved in this way. Bitterli et al. [2016] recognize that all current *a posteriori* denoisers estimate \( \hat{p}_i \) by a weighted sum of the observed pixels \( \bar{p}_k \) in a region of pixels around pixel \( i \):

\[
\hat{p}_i = \sum_{k \in R_i} \bar{p}_k w(i, k),
\]

(2.5)

where \( R_i \) is an (in practice square) region around pixel \( i \) and \( \sum_{k \in R} w(i, k) = 1 \). The weights \( w(i, k) \) follow from different kinds of weighted regressions on \( R_i \). Like Bitterli et al. [2016] we will describe the denoisers as algorithms that solve a local regression on patches.
2.2.1 Bilateral filter

The bilateral filter [Tomasi and Manduchi 1998; Elad 2002] is an edge preserving extension to the Gaussian filter. It solves a zero-order regression that fits a constant color $\hat{\rho}^{|t}$ to $R_i$ by minimizing a quadratic cost function

$$\hat{\rho}_i^{|t} = \arg\min_{\rho} \sum_{k \in R_i} (\rho - \bar{\rho}_k)^2 \omega(i, k),$$

where $\bar{\rho}_k$ is the MC estimated color of a pixel $k$ and the regression weights $\omega(a, b)$ are computed as

$$\omega(a, b) = \exp\left(-\frac{\|a - b\|^2}{2\sigma_s^2}\right) \exp\left(-\frac{\|\bar{\rho}_a - \bar{\rho}_b\|^2}{2\sigma_r^2}\right),$$

where $\|a - b\|$ is the spatial $L_2$ distance between pixels $a$ and $b$ and $\|\bar{\rho}_a - \bar{\rho}_b\|$ is the $L_2$ distance in color. The spatial bandwidth is determined by the parameter $\sigma_s$, and the color bandwidth is determined by the parameter $\sigma_r$. The weights are large when pixels $a$ and $b$ are similar in both color and location. Since this is a zero-order regression, the regression weights $\omega(a, b)$ translate directly into pixel weights $w(a, b)$ after normalization:

$$w(a, b) = \frac{\omega(a, b)}{\sum_{b' \in R_i} \omega(a, b')}.$$  

The bilateral filter gains its edge-preserving properties from the color-distance term in Equation 2.7. It will not smooth out edge if the difference in color is large enough.

2.2.2 Non-Local Means

The bilateral filter suffers inherently from the noise in the Monte Carlo estimates $\bar{\rho}$, since the weight estimates in Equation 2.7 are not robust. The noise makes the resulting weights noisy as well. The non-local means filter [Buades et al. 2005] tackles this instability by generalizing the bilateral filter to compute the color-distance on small image patches $P$ surrounding pixels instead of only on the pixels themselves only. This reduces the variance of the weights. This color distance $d(P(a), P(b))$ of two patches around pixels $a$ and $b$ follows as the average pixel-wise error

$$d(i, j) = \|\bar{\rho}_i - \bar{\rho}_j\|^2 - 2\sigma_r^2$$

over corresponding pixel pairs $i$ and $j$ in $P(a)$ and $P(b)$ respectively, clamped to $[0, \infty)$. The value $\sigma_r$ represents the variance of the input colors, and serves to correct the over-estimation of the distance due to the pixel variances [Buades et al. 2005].
The resulting formulation for the regression weights is
\[
\omega(a, b) = \exp\left(\frac{-||a - b||^2}{2\sigma^2_s}\right) \exp\left(\frac{-d(P(a), P(b))^2}{k^22\sigma^2_r}\right),
\]
where \(k\) can be tuned to set the strength of the filter. The usual patch size for non-local means is \(7 \times 7\) [Bitterli et al. 2016].

Rousselle et al. [2012] extended the NL-means distance measure to work on images with non-uniform variance. They replace the metric in Equation 2.9 by
\[
d(i, j) = ||\hat{p}_i - \bar{p}_i||^2 - (\mathcal{V}_i + \min(\mathcal{V}_i, \mathcal{V}_j)),
\]
where \(\mathcal{V}_i\) is the estimated variance of \(\bar{p}_i\), and \(\varepsilon\) is small constant to regularize the division.

Extra per-pixel guiding features can be easily incorporated in the distance functions by adding factors for distances in each feature. This is demonstrated by the RDFC denoiser [Rousselle et al. 2013].

### 2.2.3 Weighted Linear Regression

Moon et al. [2014] recognize that the zero-order regression model to determine the weights in Equation 2.5 can be made more powerful by extending it to a first-order regression. Instead of fitting a constant, a linear model is used, with spatial ramps and guiding rendering information as regression features. Let \(\bar{p}\) be a column vector of size \(n\) that contains all \(n\) pixels in a patch \(\mathcal{R}_i\) around pixel \(i\). The colors of \(\bar{p}\) are estimated as as \(\hat{p} = X\hat{\beta}\), where
\[
\hat{\beta} = \arg\min_{\beta} \|W(X\hat{\beta} - \bar{p})\|_2^2,
\]
where \(\vec{v}\) and \(\vec{p}\) are column vectors and the columns of \(X\) are the regression features of each pixel. \(\bar{p}\) is a vector of Monte Carlo estimated colors \(\hat{p}_k, k \in \mathcal{R}_i\) in the patch around \(i\) and \(W\) is a diagonal matrix, the diagonals of which contain weights that determines the amount by which each pixel-error should be weighted.

### 2.2.4 Nonlinearly Weighted First-order Regression

Bitterli et al. [2016] analysed the results discussed above, and combined them into a single high-quality denoising solution, which they name **Nonlinearly Weighted First-Order Regression**.

In summary their method combines:

- **feature pre-filtering**: features such as normals, direct visibility, texture and depth are pre-filtered independently using a non-local means filter (Section 2.2.2).
- **a first order regression**: a patch-wise linear regression is fit, using the pre-filtered features as a basis
• **non-local means weights**: the regression weights are shaped by the non-local means weights described in Equation 2.10, to reject pixels that are too different from the center pixel.

• **collaborative filtering**: A patch-wise model is fit at every pixel, and the final denoised pixel values  are reconstructed as a weighted combination of their values in each of the windows.

• **selection-based bandwidth estimation**: the bandwidth-selection parameters in the NL-means weights are tuned by optimizing an estimate of the denoising error.

### 2.2.5 Adaptive Polynomial Rendering

Moon et al. [2016] take a similar approach to Bitterli et al. [2016], evaluate a linear regression on patches surrounding each pixel. Instead of NL-means weights to shape the regression, they use a binary threshold, rejecting any pixels more than 3 standard deviations away from the center pixel’s value. Instead of tuning the NL-means bandwidth, the authors adaptively add spatial features of increasing orders. They either add no spatial features, linear ramps in $x$ and $y$, quadratic combinations $x^2$, $y^2$ and $xy$, or even cubic interaction terms. To determine which polynomial degree is locally optimal, they perform a multi-stage error estimation for each of the degrees.

### 2.3 Machine Learning

In supervised machine learning, the aim is to create models that accurately predict the value of a **response variable** as a function of **explanatory variables**. One could for example try predict a car’s price based on its color, age, initial price and brand. Such a relationship is typically modeled by a function that estimates the response variable $y$ as a function $\hat{y} = f(\vec{x}, \vec{w})$ of the explanatory variables $\vec{x}$ and tunable parameters $\vec{w}$ that are adjusted to make the model describe the relationship accurately.

The parameters $\vec{w}$ are learned from data. They are set to minimize a cost function, or **loss**, $L(D_{\text{train}}, \vec{w})$ over a **training set** $D_{\text{train}}$, which is typically the sum of errors on the entries of the dataset:

$$L(D_{\text{train}}, \vec{w}) = \frac{1}{|D_{\text{train}}|} \sum_{(\vec{x}, \vec{y}) \in D_{\text{train}}} \ell(\vec{y}, f(\vec{x}, \vec{w})), \quad \text{2.13}$$

where $\ell$ is a per-element loss function. The optimal parameters satisfy

$$\vec{w} = \arg\min_{\vec{w}} L(D_{\text{train}}, \vec{w}). \quad \text{2.14}$$

Typical loss functions for continuous variables are the quadratic or $L_2$ loss $\ell_2(y, \hat{y}) = (y - \hat{y})^2$ and the $L_1$ loss $\ell_1(y, \hat{y}) = |y - \hat{y}|$. 

---

8
As an example, linear least squares regression fits this framework with the model

$$f_{\hat{w}}(\tilde{x}, \tilde{w}) = \tilde{x} \cdot \tilde{w},$$  \hspace{1cm}  \text{(2.15)}$$

and the quadratic loss function $\ell_2$. In our notation, the feature vector $\tilde{x}$ is augmented with a constant feature 1, to model the intercept. This model has one degree of freedom per explanatory variable in $\tilde{x}$, and the optimal weights $\tilde{w}$ can be directly computed from the normal equation

$$\tilde{w} = (A^T A)^{-1} A^T \tilde{x},$$  \hspace{1cm}  \text{(2.16)}$$

where $A$ is an $n \times m$ matrix containing the explanatory variables of the training data samples as rows. A model like this can be quite restrictive. Many real-world relationships are not accurately captured by this linear model.

This brings us to a central issue in machine learning. The model $f$ should be chosen such that it does not under-fit, i.e. the model space spanned by $\tilde{w}$ should ideally contain the real-world relationship between $\tilde{x}$ and $y$. At the same time it should not over-fit either, i.e. fit the particular noise in the training set $D_{\text{train}}$. To control over-fitting, the data in a machine learning problem is usually split into three disjoint subsets: the training set $D_{\text{train}}$, a test set $D_{\text{test}}$ and a validation set $D_{\text{val}}$. After a model is optimized to fit $D_{\text{train}}$, its generalization behavior can be evaluated by its loss on $D_{\text{test}}$. After the best model is selected based on its performance on $D_{\text{test}}$, it is ideally re-evaluated on a fresh set of data $D_{\text{val}}$.

### 2.3.1 Neural networks

Neural networks are a very general class of models with potentially large numbers of parameters that have shown to be very successful in capturing patterns in complex data.

The model function $f$ of a neural network is composed of atomic building blocks called neurons or nodes. Each neuron behaves similar to the linear regression function in Equation 2.15. A neuron $n_i$ has inputs $\tilde{x}_i$ and an scalar output value $y_i$, and it computes the output as

$$y_i = n_i(\tilde{x}_i, \tilde{w}_i) = \phi_i(\tilde{x}_i \cdot \tilde{w}_i),$$  \hspace{1cm}  \text{(2.17)}$$

where $\tilde{w}_i$ are the neuron’s parameters and $\tilde{x}_i$ is augmented with a constant feature. $\phi$ is a non-linear activation function that is important to make sure a composition of several neurons can be non-linear. Popular activation functions are hyperbolic tangent $\tanh(x)$, sigmoid function $\phi_{\text{sigmoid}}(x) = (1 + \exp(-x))^{-1}$ and the rectified linear unit (ReLU) $\phi_{\text{ReLU}}(x) = \max(x, 0)$.

The neurons in a neural network are organized into layers, as illustrated in Figure 2.3. We write $\tilde{N}_k$ to denote a vector containing the outputs of all neurons $n_i$ in a layer $k > 0$. The input layer $\tilde{N}_0$ contains the model’s input features $\tilde{x}$. The neurons in the output layer return the model prediction $\tilde{y}$. The outputs of the neurons in each layer $k$ form the input of layer $k + 1$. 
The activity of a layer $N_i$ can be conveniently written in matrix notation:

$$\begin{align*}
\tilde{N}_0 &= \tilde{x}, \\
\tilde{N}_k &= \phi_k(\mathbf{W}_k \tilde{N}_{k-1}) \forall k \in \{1, n\},
\end{align*}$$

where $\mathbf{W}_k$ is a matrix that contains the model parameters $\tilde{w}_j$ for each neuron in the layer as rows. The activation function $\phi_k$ now operates element wise on its vector input.

The layers in a neural network can be interpreted as feature extractors. A layer combines the features from the previous layer into ‘higher level’, more meaningful, features that in the end enable a more powerful prediction in the output layer.

### 2.3.2 Stochastic gradient descent

In the linear regression example, the weights that optimize the loss function over the training set could simply be computed directly. For neural networks, this is no longer possible, because of their nonlinearities. Instead, they are generally optimized with stochastic gradient descent.

Gradient descent is a versatile tool for optimizing functions for which the gradient can be computed analytically. Let $F$ be the convex objective of a minimization problem. Let $\tilde{x}^*$ be the unique minimum of $F$:

$$\tilde{x}^* = \arg\min_{\tilde{x}} F(\tilde{x}).$$
With the gradient descent algorithm, \( \hat{x}^2 \) can be approached iteratively, starting at an arbitrary \( \hat{x}_0 \):

\[
\hat{x}_k = \hat{x}_{k-1} - \gamma \nabla F(\hat{x}_{k-1}) \quad \forall k \in \mathbb{N},
\]

where \( \gamma \) is the step size. For a small enough step size \( \gamma \), \( \hat{x}_k \leq \hat{x}_{k-1} \), and the series converges to \( \hat{x}^* \).

Recall the general minimization objective of machine learning, presented in Equation 2.13. The gradient of the associated cost function follows as

\[
\nabla L(\mathcal{D}_{\text{train}}, \hat{w}) = \frac{1}{|\mathcal{D}_{\text{train}}|} \sum_{(\hat{x},y) \in \mathcal{D}_{\text{train}}} \nabla \ell(y, \hat{f}(\hat{x}, \hat{w}))
\]

which can be expensive to compute when the dataset \( \mathcal{D}_{\text{train}} \) is large. Stochastic gradient descent (SGD) solves this issue. Instead of computing the full gradient in each iteration of algorithm, the full gradient is replaced by many cheap updates, based on only one element \( i \in \mathcal{D}_{\text{train}} \). The parameter update equation would be

\[
\hat{w}_k = \hat{w}_{k-1} - \lambda \nabla \ell(y, \hat{f}(\hat{x}, \hat{w}_{k-1})), \quad i \sim \mathcal{U}(0, |\mathcal{D}_{\text{train}}|),
\]

where \( \mathcal{U}(a, b) \) represents a discrete uniform distribution with inclusive lower bound \( a \) and exclusive upper bound \( b \). The parameter \( \lambda \) is called the learning rate. It controls the size of the steps taken. It is important to choose an appropriate learning rate or learning rate schedule, since steps that are too large lead to non-convergence, and a step size that is too small might result in very slow convergence.

Although there are no strong guarantees for optimality of stochastic gradient descent for non-convex functions, gradient descent is the most popular way to train neural networks, which can be highly non-convex. In particular, an intermediate form between full GD and SGD, mini batch gradient descent, is often used. It iteratively applies updates with gradients computed on small subsets of the dataset. The number of data points in these subsets is called the batch size.

There are several extensions to stochastic gradient descent. The first is momentum, which can accelerate convergence by applying exponential smoothing to the gradient update directions:

\[
\begin{align*}
\tilde{v}_k &= \gamma \tilde{v}_{k-1} + \eta \nabla F(\hat{x}_{k-1}) \\
\hat{x}_k &= \hat{x}_{k-1} - \tilde{v}_k.
\end{align*}
\]

There are many variants to momentum and other tricks to speed up training. Adaptive Moment Estimation (Adam) [Kingma and Ba 2014] is very popular, and will be our method of choice in this paper. In a nutshell, Adam estimates an exponentially decaying mean and variance estimate of the gradients for each network parameter and it applies updates in the direction of this mean, scaled down by its variance.

In order to apply gradient descent to a neural network, one requires the gradients of the loss with respect to each of the parameters in the network. These gradients can be computed with
the chain rule, in a procedure commonly called back-propagation [LeCun et al. 1998]. Back-propagation recursively computes gradient of the output loss $E$ with respect to all quantities in the network. In a neural network, the loss can be expressed as a chain of functions

$$E = (L \circ f_n^{\vec{w}_n} \circ f_{n-1}^{\vec{w}_{n-1}} \circ \cdots \circ f_0^{\vec{w}_0})(\vec{x}),$$  \hspace{1cm} (2.24)

where the functions $f_i$ are layers in the network. Let $\vec{y}_k$ be the partial output

$$\vec{y}_k = (f_k^{\vec{w}_k} \circ f_{k-1}^{\vec{w}_{k-1}} \circ \cdots \circ f_0^{\vec{w}_0})(\vec{x}),$$  \hspace{1cm} (2.25)

where $k \leq n$, which is the input of the layer $f_{k+1}$. The goal of backpropagation is to compute the gradient $\frac{\partial E}{\partial \vec{w}_k}$ for each network parameter $\vec{w}_k$. This is achieved by recursively applying the chain rule:

$$\frac{\partial E}{\partial \vec{y}_n} = 2(\hat{y} - \vec{y}_n), \hspace{1cm} \text{(for L2 loss)}$$
$$\frac{\partial E}{\partial \vec{y}_k} = \frac{\partial E}{\partial f_{k+1}} + \frac{\partial f_{k+1}}{\partial \vec{y}_{k+1}} \cdot \frac{\partial E}{\partial \vec{y}_{k+1}} \hspace{1cm} \forall k < n,$$  \hspace{1cm} (2.26)
$$\frac{\partial E}{\partial \vec{w}_k} = \frac{\partial E}{\partial f_{k+1}} \cdot \frac{\partial f_{k+1}}{\partial \vec{w}_{k+1}} \hspace{1cm} \forall k.$$

The computed gradients can now be used in the SGD algorithm.

### 2.3.3 Convolutional neural networks

In a fully connected layer, the number of parameters that connect the layer with the previous one is roughly the product of the number of neurons in the layers. When a color image of size $w \times h \times 3$ is the input of such a layer, and the layer has a similar number of output-neurons, the number of parameters quickly explodes and becomes infeasible.

To make neural networks for image processing more tractable, convolutional networks (CNNs) [LeCun and Bengio 1995] simplify the fully connected layer by making the connectivity of neurons between two adjacent layers sparse. Neurons are conceptually arranged into a three-dimensional structure, as illustrated in Figure 2.4, where the first two dimensions follow the spatial dimensions of an image, and the third dimension contains a number of neurons, features or channels, at each pixel location. The connectivity of the nodes in this structure is local. Each of a layer’s output neurons is connected to all input neurons in a spatial region centered around it. The size of this region, $k_x \times k_y$, is called the kernel size. The network parameters used in these regions are shared over the spatial dimensions, bringing the number of free parameters down to $d_{\text{in}} \times k_x \times k_y \times d_{\text{out}}$, where $d_{\text{in}}$ and $d_{\text{out}}$ are the number of features per pixel in the previous layer and the current layer respectively. The number $d_{\text{out}}$ is referred to as the number of channels in the layer. Convolutional neural networks are most popular for 2D images, but can be generalised to 1D data or higher-dimensional structures.
The neurons in a convolutional layer for 2D image processing are typically organized in three dimensions. Two for space, and one the number of features collected at each spatial location. Neurons have a small receptive field, $3 \times 3$ in this image, and share their parameters across the spatial dimensions. The total number of parameters for the layer in this illustration would be $d_n \times 3 \times 3 \times d_{out}$, where $d_n$ is the number of features in the layer’s input and $d_{out}$ is the number of features in the layer’s output.

Convolutional networks, consisting of a combination of convolutional layers, fully connected layers, and spatial pooling, have shown to be successful in many computer vision tasks such as object classification [Szegedy et al. 2016] and detection [Girshick et al. 2015], but have more recently also showed their potential in low-level image processing applications, such as denoising [Burger et al. 2012; Jain and Seung 2009], inpainting [Xie et al. 2012], and superresolution [Yang et al. 2016].

2.3.4 Neural networks for low-level image processing

Jain and Seung [2009] present a convolutional network trained to denoise images corrupted with Gaussian noise. They use a relatively small network (15,000 weights), consisting of 4 convolutional layers with kernel size $5 \times 5$ and 24 channels each, and train it on millions of generated training images. To generate the training set, they corrupt image databases with randomly generated noise. They find that their network performs almost as well on data with unknown noise levels, as on data with fixed noise. Their method is compared to Markov Random Fields (MRF), and they show that CNN models can have more expressive power than MRFs.

Burger et al. [2012] use networks with fully connected layers on small $16 \times 16$ patches to remove several different types of image corruption, such as salt and pepper noise, stripe noise, and JPEG quantization artifacts. They compare their performance against the popular BM3D de-
noising algorithm [Dabov et al. 2006], and find that they achieve similar quality, but their learned method extends more easily to different types of less extensively studied noise.

Xu et al. [2015] describe a successful convolutional neural network architecture for copying complex Photoshop filters. To enhance edge-preservation in their network, their network operates on spatial gradients rather than on the color directly. The final color image is retrieved using Poisson reconstruction.

Yang et al. [2016] propose a convolutional neural network design for upsamling low-resolution images. They aim to recover high frequencies in an up-sampled small image. Their design has two key elements: (1) gradient information entering the network alongside the image color, and (2) a repeated architecture, where the same block of convolutional layers is repeated three times with the same parameters. Each of these blocks predict residuals that are added to its input, in the spirit of Resnet [He et al. 2015]. The presented network is shown to outperform several state-of-the-art methods.
3 | Problem statement & challenges

3.1 The denoiser as an estimator

Traditionally, denoisers are formulated as algorithms to find local filtering kernels that reduce the variance of the MC estimates, while keeping the bias under control. We opt for a more general formulation. We use the notation $\mathcal{I} = \{I^{\text{color}}, I^{\text{feature} 1}, I^{\text{feature} 2}, \ldots \}$ to refer to an image $\mathcal{I}$ with its color and additional feature buffers. A bar is used to indicate that an image $\mathcal{I}$ is the Monte Carlo mean of a number of samples, a tilde shows that $\tilde{\mathcal{I}}$ is a converged image that would hypothetically be achieved when the number of MC samples goes to infinity, and a hat that $\hat{\mathcal{I}}$ is an estimate of $\tilde{\mathcal{I}}$.

We see a denoiser as an estimator $D(\mathcal{I})$ that takes a MC rendered image and guiding feature buffers $\mathcal{I}$. These features might include the estimated variances of the MC estimates or other information about the sample distribution. The denoiser estimates the color of the converged rendering $\tilde{\mathcal{I}}^{\text{color}}$. Its theoretical quality can be expressed as the inverse of the expected error

$$E[\ell_{\text{human}}(\tilde{\mathcal{I}}^{\text{color}}, D(\mathcal{I}))]$$

over the distribution of all MC renderings of interest $\mathcal{I}$, where $\ell_{\text{human}}$ is a hypothetical perfect human image dissimilarity metric.

Such a denoiser $D$ can be entirely designed and tuned by humans, or it may have tunable parameters $\vec{w}$. In that case it can be denoted as $D_{\vec{w}}$. In the context of machine learning, we design denoisers with potentially many parameters, and optimize these parameters to minimize the expected loss

$$L(D_{\text{train}}, \vec{w}) = \frac{1}{|D_{\text{train}}|} \sum_{(\mathcal{I}, \mathcal{L}) \in D_{\text{train}}} \ell(\mathcal{I}, D_{\vec{w}}(\mathcal{I})),$$

where $\ell$ is some differentiable cost function that ideally represents the unknown $\ell_{\text{human}}$ reasonably well.

3.2 Challenges

For the particular problem of denoising MC renderings with machine learning, there are a few key challenges that can be addressed in different ways. Chapters 4 (Single-scale Denoising) and
6 (Data) will present some potential solutions that have been explored in the context of this thesis. For clarity, this section gives an overview of the main challenges that have to be addressed.

3.2.1 Cost of generating training data

The sole reason for the existence of Monte Carlo rendering denoisers, is that converged renderings are computationally expensive to obtain. For complex scenes, visually converged images might take multiple core years to compute. Machine learning models, and neural networks in particular, shine with large amounts of training data, since the parameter space is very high dimensional. Therefore, the complexity of generating (noisy input, converged output)-pairs is problematic.

What is also important for machine learning based denoisers, is that the distribution of training examples is diverse enough and similar to the distribution of all renderings that might be of interest for denoising. This means that ideally a large variety of geometries, lighting effects and colors need to be present in the dataset. Creating a set of 3D scenes that covers all of these requirements requires much manual labor, and it would be prohibitively costly and time consuming.

Chapter 6 will describe how we deal with these problems. We describe how we use approximate ground truth images, data augmentation techniques and random scene perturbations to create a training set that is large enough to train our models on.

3.2.2 High-dynamic-range data

In contrast to the images in existing low level image processing applications of machine learning, the colors of rendered images are not bound to a fixed range like 0–255 or 0–1. Because these normal pictures are usually produced by a digital camera, the dynamic range is clipped. Everything brighter than a threshold will be white and everything lower will be black. The rendered images, however, store the raw incoming irradiance from a specific set of directions at the location of the modeled ‘camera’. These values can have a very large range. When a bright light source is directly visible on the image, these pixels can easily be multiple orders of magnitude brighter than other areas.

To add to this dynamic range, the sample distribution of light paths for a specific pixel can be very skewed. In many situations, most of the sampled light paths may not reach any light source and have no contribution, but one in a thousand samples reaches the light and is very bright. With such a sample distribution, for low sample counts, some pixels can be much too bright, also by orders of magnitude.

Due to these two effects, both the inputs and outputs of a neural network for denoising intrinsically have a high dynamic range, and have a skewed distribution. A high-dynamic-range output has the effect that many common loss functions, such as the $L_2$ distance, put extreme emphasis on ‘getting the bright regions right.’ This hurts the convergence of gradient descent algorithms.
A skewed high-dynamic-range input similarly has a negative effect on the convergence of the training procedure.

To deal with these issues, we propose the kernel-prediction architecture in Section 4.3.2, which allows a neural network to predict values in a fixed range, instead of in the full positive real number space, and in chapter 6, we propose several schemes to normalize input data to be better behaved and to speed up convergence.

3.2.3 Choosing the loss function

In Section 3.1, we described how the performance of a denoiser should be measured according to the perfect human image dissimilarity metric $\ell_{\text{human}}$. Although there has been research into perceptual image similarity [Wang et al. 2003; Wang et al. 2004], the outcomes of those studies are often ill suited for machine learning, since the proposed metrics can be expensive to compute, and often not differentiable.

Popular loss functions used to evaluate denoised images are Mean Squared Error (MSE), $\ell_{\text{MSE}}(\hat{a}, \hat{b}) = \|a - b\|_2^2$, Relative Mean Squared Error (MrSE), $\ell_{\text{MrSE}}(\hat{a}, \hat{b}) = \|\frac{\hat{a} - \hat{b}}{a}\|_2^2$, where $a$ is the reference image, and Peak Signal to Noise Ratio (PSNR), $\ell_{\text{PSNR}}(\hat{a}, \hat{b}) = 10 \cdot \log_{10}\left(\frac{\text{max}_{a} a^2}{\ell_{\text{MSE}}(\hat{a}, \hat{b})}\right)$.

All of these metrics are computed on a per-pixel basis, whereas humans care more about structural similarity than perfectly accurate colors. Wang et al. [2003] introduced the Structural Similarity Index (SSIM), to capture this. Their metric is not common as a loss function in machine learning. One reason might be that SSIM is defined on grayscale images, and straightforward application to color images might result in unwanted color shifts.

A potentially easier way to incorporate the notion that humans care about structure in the loss function, is to evaluate a common loss, like MSE, on the gradients of the image instead of on the colors directly. This approach is taken by Xu et al. [2015].

Zhao et al. [2015] investigate suitable losses for neural network learning, and find that a mix of $L_1$ loss and multi-scale SSIM loss work well for denoising images with low-variance Gaussian noise. In Section 7 we also investigate several losses, and draw conclusions about the perceptual quality of the obtained results and convergence properties of the networks.

3.2.4 Architecture

The current state-of-the-art neural networks for image classification [Szegedy et al. 2016] have millions of parameters that need to be optimized, but to train these networks millions of unique images are available. For the denoising of Monte Carlo renderings, the datasets are smaller or more redundant. To achieve the optimal denoising performance in our setting, the ideal number of weights might be lower.

It is generally considered good practice to inject knowledge about the task at hand into machine learning problems. One way to do this in the case of denoising, for example, is to realize that the
A denoiser should have the same behavior on a smaller (but equally noisy) version of an image. This can be leveraged by applying the same network on multiple scales, and then combine the results. This idea is explored in Chapter 5. The same parameters could be shared at the different scales, to reduce the total number of parameters in the model and make it easier to optimize.

Another way to reduce the number of parameters, is to repeat blocks of convolutional layers with the same parameters. The intuition for this would be to denoise iteratively, in small steps. This approach is used by Yang et al. [2016], and is evaluated in this thesis. The repeated blocks could be configured to predict residuals, that are added to the blocks input before passed to the next block, instead of just sending its output.

We also explore the idea of mimicking current denoisers more closely, and predict filter kernels instead of prediction colors directly. This architecture is described in Section 4.3.2.

There are many choices to make when creating a neural network architecture. New ideas are published weekly, and ideas can be combined into new combinations. Although we explore a couple of combinations, and attempt to find the best one, it is not possible to explore the whole space. Drawing conclusions on what architecture works best is definitely a challenge in the field of neural networks. In general, one has to find a balance between expressive power and constraining the network to use the information we already have about a problem.
Figure 4.1: Single scale denoising breakdown. The prediction function is broken down into the components pre-processing, network, and post-processing / reconstruction. Only the network component contains learnable parameters, whereas the other blocks are fixed transformations used to encode knowledge about the features and improve convergence.

4 | Single-scale denoising

The nature of our research has been experimental, and we have tried and evaluated a large number of ideas to deal with the challenges presented in Section 3.2. These ideas are modular components and can be combined in different combinations into end-to-end solutions. The aim of this chapter is to present the components that have been evaluated, and to give arguments for why they might make sense for denoising. There is a many-to-many relationship between challenges presented in Section 3.2 and solutions presented in this chapter, so we will also indicate which of those issues a certain component is meant to solve. A thorough evaluation of different combinations of components is delayed to Section 7 (Results).

In order to organize the components we investigated, we break up the denoising function $D(\tilde{I})$, as featured in Equation 3.1, into steps according to the schematic in Figure 4.1.

The first step is pre-processing, which takes raw image features and may present them to the network in a more normalized way. It might, for example, standardize the input data to have zero mean and unit variance for each feature on average over all inputs in the training set. This step might also apply whitening, a procedure that does the same, but also removes correlation between the input features. Such transformations are known to improve the convergence of a neural network trained with back-propagation [LeCun et al. 1998]. It is important to note that there are no tunable parameters in the pre-processing stage. It is a fixed transformation that allows the user to make the input data features better behaved and aid the optimization process. Pre-processing transformations do not have to be differentiable, because this first step has no trainable parameters.

Then there is a neural network, which transforms the output of the pre-processing step in a way that depends on many configurable parameters that are optimized in the training procedure. In
designing the network, choices have to be made regarding the number of neurons, connectivity patterns between them, and parameter sharing. A network with too many degrees of freedom might learn to denoise the scenes in the training set very well, but not generalize to other data (over-fitting), whereas a network with too little freedom might not be able to learn to denoise well at all.

The post-processing and reconstruction stage is optional. It can just be the identity function, in which case the network predicts a denoised image directly. Alternatively, it could apply inverse whitening to keep the network output for the different color channels uncorrelated or do a color space transformation. More generally, this stage determines what the network should predict. In Section 4.3.2, we present a novel reconstruction function that lets the network predict local filtering kernels instead of colors directly. This is a way to enforce regularity in the output and improve the network's convergence. For post-processing operations, it is important to be differentiable, so that the gradient descent algorithm can use back-propagation to compute the gradients with respect to parameters in the network step.

The last important component for a neural network denoising pipeline is optimization. There is a multitude of flavors of stochastic gradient descent algorithms, each with their own convergence properties, and they rely on manual configuration of some parameters such as the learning rate. A second component of the optimization step is the choice of a loss function. The loss function should correspond well to human perception, while at the same time lead to rapid convergence when optimized with a stochastic gradient descent algorithm.

In the next sections, each of these steps involved in the neural network denoising pipeline will be discussed, and we will outline several proposed implementations of these steps.

### 4.1 Pre-processing

In the early days of neural network research, LeCun et al. [1998] already showed that the distribution of input features affects the convergence of the stochastic gradient descent algorithm. It is important for features to be on a similar scale, and ideally also uncorrelated. In this section, we explore techniques used to make input features better behaved.

Sometimes, examples of how specific features can be normalized are given. The patterns described, however, can be easily generalized to alternative rendering features beyond the ones discussed.

#### 4.1.1 Non-linear feature transformations

To design a pixel-wise pre-processing step, we investigate the distribution of features over all pixels of all noisy training images. Sometimes, these distributions are very skewed, as is the case for the HDR color feature. There are many ways to make the feature distributions better behaved.
The color feature, for example, benefits from a logarithmic transformation. Replacing a color feature $c$ by a transformed feature $c'$ by applying

$$c' = \log(c + \varepsilon)$$

removes most of the skewness in the distribution, apart from the pixels that are perfectly black. Those pixels are visible in renderings with low sample counts, when no light path reached a light source. The position of those pixels in the resulting distribution can be controlled by an offset parameter $\varepsilon$. A suitable value for $\varepsilon$ is $10^{-4}$. The logarithmic transformation of the color feature is illustrated in Figure 4.2.

A logarithmically transformed color buffer has downsides too, unfortunately. The original color feature lives in a linear color space, where a color with twice the value is exactly twice as bright, and averages of pixel colors naturally make sense and preserve the color. Filtering an image with simple smoothing filters in the log-transformed domain would lead to color shifts and loss of energy. Therefore, we propose an alternative normalization scheme for the color feature. We
Figure 4.3: Distribution of the color feature before and after the transformations described in 4.2. The histograms at the bottom show the distributions of the features over multiple scenes, sample counts and pixels. In the histogram of ‘log of colors greater than 1’, the values that are exactly zero are excluded from the histogram. Note that there are some non-zero values up to 1200 in the left histogram that are not visible. The images at the top show the distribution of the feature visually in one scene. The images are colored to show the brightest color in white and the darkest in black.

observe that the range [0, 1] is most important for our applications because the default white-point is at 1. With that in mind, we split the color feature into two separate features: one which is clamped to the range [0, 1], and one with the bright peaks, transformed logarithmically:

\[
\begin{align*}
    c'_1 &= \begin{cases} 
        1 & c > 1, \\
        c & \text{otherwise},
    \end{cases} \\
    c'_2 &= \begin{cases} 
        0 & c < 1, \\
        \log(c) & \text{otherwise}.
    \end{cases}
\end{align*}
\]

This transformation is illustrated in Figure 4.3.

Training with the depth feature, which measures the distance from a point in the scene to the camera directly without any transformation is difficult. This is due to its large range. In outdoor scenes, depth might vary at the order of magnitude of 100’s of meters, whereas depth discontinuities at the centimeter level are already useful for reconstruction. Denoisers like NFOR [Bitterli et al. 2016] deal with this problem by normalizing the features to the range [0, 1] in small patches. We can incorporate a similar effect into the depth feature by realizing that the absolute scale of the feature is not interesting for reconstruction. It is the discontinuities in depth that matter
Figure 4.4: Distribution of the depth feature before and after applying the transformation procedure described in Equation 4.3. The images at the top show the distribution of the feature visually in one scene, with the darkest values black and the brightest white. At the bottom, a histogram of the feature’s distribution over multiple scenes and numbers of samples per pixel is shown. The transformed depth feature shows fine texture which is hidden by the large depth range in the original.

for appearance. We make use of this observation by first removing low frequencies, centering the depth at 0 everywhere, then rescaling the high frequencies to have a uniform value across scenes, and finally clipping large discontinuities with a hyperbolic tangent transformation. The intuition behind clipping is the notion that at some point, the exact distance of a depth discontinuity is not informative. Let $\tilde{d}$ represent the depth feature of an image and $g$ be a Gaussian smoothing operator with $\sigma = 1$. The proposed depth transformation follows as

$$\tilde{d}_{\text{centered}} = \tilde{d} - g(\tilde{d}),$$

$$\tilde{d}_{\text{scaled}} = \frac{\tilde{d}_{\text{centered}}}{g(|\tilde{d}_{\text{centered}}| + \varepsilon)},$$

$$\tilde{d}_{\text{normalized}} = \tanh(5 \cdot \tilde{d}_{\text{scaled}}).$$

where $\varepsilon$ is a constant that makes sure numerical quantization artifacts do not show up in the resulting image. All binary mathematical operations operate element-wise in the equation. Figure 4.4 illustrates the effect of this procedure.
Figure 4.5: Distribution of the color variance (left) and relative color variance according to Equation 4.4. The images at the top show the distribution of the feature visually in one scene, with the darkest values black and the brightest white. At the bottom, a histogram of the feature’s distributions over multiple scenes and numbers of samples per pixel is shown.

Other features that typically require treatment are the variances of the MC estimates of the features. In all the features used in our experiments, the magnitude of noise has been correlated with the mean, therefore we employ relative variances, that follow as

\[
\hat{\sigma}_{\text{rel}}^2 = \frac{\hat{\sigma}^2}{\hat{\mu}^2 + \epsilon},
\]

where \(\hat{\sigma}^2\) stands for an arbitrary variance feature, and \(\hat{\mu}\) for the mean of that same feature. \(\epsilon\) is a small constant used to avoid division by zero. It can be proven that the relative variance of a feature’s mean sample value lies in the range \([0, 1]\) when the feature is strictly non-negative. The effect of this transformation is shown for the specific case of the color feature in Figure 4.5.

The above mentioned transformations are demonstrated on a limited number of possible features. The approach of coming up with these transformations by looking at distributions and images and reasoning about the usefulness of the feature can be applied universally.
4.1.2 Standardization

After removing skewness in the distribution of features with the techniques presented in the previous section, care should be taken to make sure all features have a similar scale. This is known to improve convergence of the gradient descent optimization. Two common techniques to achieve this consider the distribution of a feature \( f \) over the whole training set, and either standardize it by the observed mean \( \mu_f \) and standard deviation \( \sigma_f \):

\[
f' = \frac{f - \mu_f}{\sigma_f}
\]  \hspace{1cm} (4.5)

or by transforming the feature’s range to \([0, 1]\) with a linear transformation:

\[
f' = \frac{f - f_{\min}}{f_{\max} - f_{\min}}
\]  \hspace{1cm} (4.6)

where \( f_{\min} \) is the minimum value of the feature \( f \) observed over the whole training set and \( f_{\max} \) is the maximum.

4.1.3 ZCA Whitening

The standardization procedure presented in Equation 4.5 normalizes the marginal distributions of all features but does not remove correlations between them. Many features are highly correlated, and this might hurt convergence too. Whitening refers to a transformation that is applied to data to make its first two moments equal to those of a standard multivariate normal distribution. It removes correlation between the features. The name ‘whitening’ comes from the fact that it produces the features that have the joint distribution of ‘white noise’.

There is no unique whitening transformation. There are several popular transformations, most notably Cholesky whitening, PCA whitening, or ZCA whitening. Each of these methods are based on the covariance \( \mathbf{M} \) of the data matrix \( \mathbf{X} \), an \( n \times d \) matrix, where \( n \) is the number of observations, and \( d \) is the number of features per observation.

ZCA whitening is popular in machine learning, since it gives whitened features that are as close as possible to the correlated input features, while removing their correlation.

To use ZCA whitening, one computes a transformation matrix \( \mathbf{A}_{\text{ZCA}} \), such that \( \mathbf{X}' = \mathbf{A}_{\text{ZCA}} \mathbf{X} \) is whitened. The first step to computing \( \mathbf{A} \) is to compute the Eigenvalue decomposition of the covariance matrix \( \mathbf{M} \) of \( \mathbf{X} \):

\[
\mathbf{M} = \mathbf{QDQ}^{-1},
\]  \hspace{1cm} (4.7)
where $\mathbf{D}$ is a diagonal matrix containing the eigenvalues of $\mathbf{M}$ and $\mathbf{Q}$ is an orthonormal matrix containing the eigenvectors of $\mathbf{M}$ as columns. The ZCA transformation matrix $\mathbf{A}_{\text{ZCA}}$ follows as

$$\mathbf{A}_{\text{ZCA}} = \mathbf{QD}^{-\frac{1}{2}}\mathbf{Q}^T,$$  \hfill (4.8)

where $\mathbf{D}^{-\frac{1}{2}}$ contains the inverse square roots of the eigenvalues of $\mathbf{M}$.

The data matrix $\mathbf{X}$ is very large in our dataset. We have 3,000 scenes with approximately 1,000,000 pixels each, resulting in a matrix with 3 billion rows. To compute the covariance $\mathbf{M}$ of that matrix, we use a two stage online algorithm that first computes the means, and then the variance of centered values.

### 4.2 Network architecture

The space of possible neural networks is infinite, and reasoning about the ‘optimal’ architecture is difficult. We explore a number of important design components that have an impact on the performance of a network and make it easy to train. First, we discuss two approaches to reusing network parameters spatially and making the network work for different size inputs in Section 4.2.1. In Section 4.2.2, architectures with repeated blocks of layers that share weights, as used in the work of Yang et al. [2016] are presented. Lastly, we go over residual connections, as popularized by the Resnet architecture [He et al. 2015].

#### 4.2.1 Patches versus ConvNets

Burger et al. [2012] and Jain and Seung [2009] organize the neural networks they use for denoising natural images in different ways. The conceptual difference is illustrated in Figure 4.6. Burger et al. [2012]’s network operates on patches. They define their denoiser from an input patch to an output patch. In the network, the spatial structure of the patch is discarded, and the input and output are connect through a series of fully connected layers. In other words, the network is a multi-layer perceptron (MLP). To denoise a full image, overlapping patches are denoised, and the full denoised image is reconstructed collaboratively by combining the pixel’s value in multiple overlapping patches. Jain and Seung [2009], on the other hand, use a convolutional architecture, depicted in the right subfigure of Figure 4.6. They use 4 layers of $5 \times 5$ convolutions, with 24 features per pixel in each layer. In this way, each pixel is computed as the same function of a region around it.

In both designs, the denoised color at an output pixel $i$ can be defined as a function $\hat{\mathbf{y}}_i = \mathbf{G}(\mathbf{R}(i))$ of a region of influence around the pixel $i$, where $\mathbf{G}$ represents the network parameters. In a patch-based architecture, the region of influence of a pixel is defined by the union of all patches it lies in. In a convolutional design, the region of influence is defined by a convolution of the kernel sizes in each layer. These regions of influence are depicted in Figure 4.7.
Figure 4.6: Patch-based design (left) and convolutional design (right). Both architectures are feed-forward neural networks. In a patch-wise architecture, the network takes patches of fixed size, and predicts full output patches. The layers in the network are typically fully connected. A full image is reconstructed by denoising overlapping patches and combining their results. In the perfectly convolutional architecture, the input and output are full images of arbitrary size. Pixels are all computed in the same way from a region of influence in the input image by convolving the region of influence repeatedly with trained kernels.

Figure 4.7: Region of influence in a patch-based architecture (left) and convolutional architecture (right). In a patch-based architecture, the region of influence of a pixel is defined by the union of all patches it lies in. In a convolutional design, the region of influence is defined by a convolution of the kernel sizes in each layer.

In all our experiments, we opt for a convolutional design. With enough features, both designs capture functions with the same input range. A convolutional design is more efficient in the number of parameters it needs and the amount of computation required to evaluate it. This efficiency comes from the fact that computation and weights are shared across the spatial dimensions at each layer.
4.2.2 Repeated architecture

Our experiments into repeated architectures were inspired by the work of Yang et al. [2016]. They observe that the quality of a convolutional denoiser with a small spatial support, such as the one presented by Jain and Seung [2009], degrades quickly as the variance of the added noise increases. A larger spatial support is required to effectively remove large amounts of noise. In the default convolutional architecture, this increases the amount of model parameters, resulting in more difficult training and a need for more training data. To tackle this issue, the authors leverage the fact that many denoising algorithms rely on denoising frequency sub-bands separately with the same algorithm, and then combining those frequencies. They propose to denoise by applying an iterative procedure $s_{i+1} = s_i + D(s_i)$, $n$ times, where $s_i$ is the input of denoising step $i$. The denoising function $D$ is the same at each step. They call this a ‘recurrent residual’ architecture. We separate two components of this idea, ‘recurrent’, and ‘residual’:

$$s_{i+1} = s_i + D(s_i).$$

In our evaluations, we will separate both components, since they can be used independently. In this section, we consider the ‘recurrent’ element of their architecture. To avoid confusion with the popular recurrent neural networks, we will call them ‘repeated’ instead. The ‘residual’ element of their design is discussed in Section 4.2.3.

Figure 4.8 shows what we consider a repeated architecture. The network starts with a number of pre-processing layers that transform the input ($x$ features per pixel) to a representation of $n$ features. This data is now repeatedly sent through a number of blocks of convolutional layers that share their parameters across blocks. The repeated block outputs $n$ features as well. Finally, the $n$ channels go through a number of post-processing layers that transform the $n$-dimensional representation to $y$ dimensions, the desired output dimensionality of the network.
4.2.3 Residual architecture

The previous section discussed how repeated blocks in a neural network architecture can help to save parameters, while extending the spatial support of a denoiser. It was motivated by the ‘recurrent residual’ design by Yang et al. [2016]. The other component of their philosophy is the residual block. A residual block in a feed-forward neural network adds a ‘skip connection’ between its input and its output, as illustrated in Figure 4.9. The block can be modeled as

\[ \tilde{y} = R(\tilde{x}) + \tilde{x}, \]

where \( \tilde{x} \) and \( \tilde{y} \) are the input and output of the block, \( R \) is the block’s function without skip connection, and \( \tilde{w} \) represents the set of model parameters of \( R \). When \( R \) is a traditional convolutional neural network, and all parameters \( \tilde{w} = 0 \), the block represents the identity function.

The idea of adding residual connections to convolutional neural networks is not new. It was popularized by the great successes of Resnet [He et al. 2015] and Highway Networks [Srivastava et al. 2015] in high level computer vision tasks. He et al. [2015] show that even though their network with residual connections lives in the exact same function space as the version without, its parametrization results in faster convergence, and achieves a significantly lower loss at the end of training.

Residual blocks for denoising can be motivated by intuition as well. Assume a network with only one block. When the noisy image is unbiased, e.g. its distribution has zero mean, the expected
required ‘update’ from a residual block is zero as well. This makes the output centered around zero across pixels.

4.3 Reconstruction

When the desired output of a neural network has very large values, and the input is normalized, this means that the weights in the network inherently have to be large, and that the randomly initialized network will be far from the optimal weights. From this observation, we conclude that it is desirable for the convergence of our model that the outputs of the network layer are small. Another desirable feature for a neural network is to have uncorrelated outputs. With independent outputs, gradients will be more normally distributed. According to Desjardins et al. [2015], this also improves convergence.

To accommodate these observations, we add an optional fixed transformation to the end of our networks. Such a transformation can make sure internal values and gradients are under control, while still producing the desired high-dynamic-range color values. They can also act as a regularizer to avoid color shifts or energy loss. In Section 4.3.1, we evaluate reconstruction blocks with which the network directly predicts colors, but potentially in a different color space. Section 4.3.2 describes the kernel-prediction architecture, which is one of our main contributions. Finally, in Section 4.3.3 we describe a decomposition trick that splits the denoising of texture (easy) and luminance (difficult) to improve preservation of texture and simplify denoising.

4.3.1 Direct prediction

The simplest reconstruction block is the identity function. A network that directly predicts high-dynamic-range colors can work, but is most likely not optimal for the reasons described in the beginning of Section 4.3.

An easy alternative is to predict tone-mapped colors, and apply the inverse gamma-correction to the network output. Let \( \hat{N} \) be the output of the learned component of our architecture. We can write the proposed transformation as:

\[
\hat{I} = \hat{N}^{3/2},
\]

where all mathematical operations are applied element-wise. This transformation will make sure the required network outputs are generally better behaved. A required HDR intensity of value 100 would now become \( 100^{3/2} = 8.11 \). A similar transformation is the inverse of the logarithmic transformation \( \log(x + \epsilon) \):

\[
\hat{I} = \exp \hat{N} - \epsilon.
\]
Kernel-prediction is one of our main contributions, and it is designed to overcome the problems of large network outputs, while preventing possible color shifts and energy loss. This is achieved by transferring traditional denoising ideas to neural networks. In all denoisers described in the related work Section 2.2, a denoised pixel is produced as the weighted sum of pixels in its neighborhood. The denoisers basically produce a local filtering kernel that does a good job at removing variance while not over-smoothing. The advantage of having such a kernel is that all color channels are filtered the same way, avoiding color shifts, and that, when the kernel is normalized, the average color intensity is preserved.

We incorporate this idea into our networks in the reconstruction phase. Utilizing filters of size \( k \times k \), we define the network’s output to be \( k^2 \) values per pixel. After normalization (discussed in the next paragraph), these weights are then applied to local neighborhoods around the pixels, constituting the predicted color. This procedure is fully differentiable, and can be back-propagated through. The design illustrated in Figure 4.10.

**Figure 4.10**: Kernel-prediction. Numbers in parentheses indicate the number of features per pixel in a convolutional architecture. In the kernel-prediction reconstruction, the network predicts a local \( k \times k \) filter kernel at each pixel. The trained output of the network is transformed to a \( k^2 \)-dimensional representation per pixel. These values are then normalized, after which they are applied to the noisy color channel of \( \tilde{I} \) by computing local dot products.

Both transformations described above have the effect of making the range of intensities better behaved. They also both suffer from the same potential issues. Because the original HDR color space is linear, a simple smoothing filter, which is naturally captured by a convolutional network, already keeps colors correct on average, and does not lose energy. In these new spaces, combining colors is less straightforward, and might not generalize as well to zero-shot inference data samples such as images with more noise than any scene in the training set. This is a potential issues when using such non-linear transformations.
The normalization procedure that is applied to the predicted weights is important. Two approaches that have produced good results for us have been softmax normalization and $L_2$-normalization. Consider the predicted weights $w_i$ used for the filter of a pixel $i$. With softmax normalization, the normalized weights follows as

$$w'_i = \frac{\exp w_i}{\sum_j \exp w_j}.$$  \hspace{1cm} (4.13)

This transformation assures that all weights are positive and add up to one. An alternative, $L_2$-normalization, normalizes by the $L_2$-norm of the weight vector:

$$w'_i = \frac{w_i}{\sqrt{\sum_j w_j^2}}.$$  \hspace{1cm} (4.14)

Although positive weights, as enforced by softmax normalization, seem sensible for filtering kernels, they can be very limited. This becomes clear with the following example: consider a rendering in which the ground truth color can be perfectly explained as an affine transformation of the texture feature, which is noise free. Denoising should be simple in this case. Consider, however, the darkest pixel in the texture, and assume that its MC estimate is too bright. There is no way for a strictly non-negative kernel to make this pixel darker, as it should be. $L_2$ normalized kernels do not have this limitation, but show to be more difficult to train.

### 4.3.3 Irradiance factorization

For diffuse surfaces, there is a multiplicative relationship between its albedo and final color of the rendering. The final color can often be decomposed as

$$C = A \cdot I,$$  \hspace{1cm} (4.15)

where $C$ stands for color, $A$ is albedo, and $I$ is irradiance (illumination). The albedo, also informally called ‘texture’ in this work, of a surface is its its local diffuse reflecting power in each color. The decomposition is illustrated in Figure 4.11. Because it is easy to extract the albedo feature from the renderer, and it has much less noise than the color feature, it makes sense to decompose the color into albedo and irradiance, and to denoise these components separately. This approach is used by Zimmer et al. [2015] to denoise renderings spatio-temporally, and it significantly improves their results.

### 4.4 Optimization & loss functions

Our formulation of the denoising problem statement was laid out in Section 3.1. In the machine learning context, a denoiser is represented as a function $D_{\theta}(\hat{I})$, parameterized by $\tilde{\theta}$, that takes
Figure 4.11: Irradiance factorization. For diffuse surfaces, the rendering’s color channel can be decomposed into the factors albedo and irradiance. The albedo is often easy to denoise, and denoising albedo and irradiance separately will make it easy to retain texture detail.

rendering features, and should estimate the color of a converged rendering $\hat{F}$. Its parameters $\hat{w}$ are optimized to minimize a certain loss function on the training set. Two assumptions lie at the basis of this approach: (1) the training is large enough and representative of the desired population, and (2) the optimized loss approximates a hypothetical perfect human dissimilarity metric well.

The second assumption means that we need to find a loss function $\ell(x,y)$, where $x$ is the predicted color and $y$ is the desired output, that accurately describes our perceived quality. At the same time, however, it is desirable for a loss function to be reasonably convex. Convex loss functions will make optimizing a neural network easier by decreasing the number and depth of local minima. These two criteria can be contradicting, and it is not straightforward to assess which loss function will give the best results. In this section, we briefly discuss a number of loss functions that can be relevant for denoising. The quality of loss functions is evaluated in the Results Section 7.

4.4.1 Per-pixel loss

The most common loss functions for denoising today are per-pixel metrics. Formulas for the $L_1$ and $L_2$ (MSE) per-pixel loss are given in Table 4.1. The first difference between $L_1$ and $L_2$ loss is that the derivative of the quadratic $L_2$ loss function goes to zero as the predicted value approaches the correct value. This is a desirable property for gradient descent algorithms, since it results in a natural convergence mechanism. Unfortunately, $L_2$ loss suffers from high sensitivity to outliers. The error blows up quadratically as the distance between two colors increases. Single pixels that are very difficult to denoise and have a high error will attract all of the network’s ‘focus’, ignoring other regions that are easier to get right.
Table 4.1: Common per-pixel loss functions. $x$ is the predicted image, $y$ is the desired output, and all operations (addition, subtraction, multiplication and division) are applied element-wise.

The final loss is the average of the losses over all pixels and color channels.

<table>
<thead>
<tr>
<th></th>
<th>L2</th>
<th>L1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Absolute MSE:</td>
<td>$\ell(x, y) = (x - y)^2$</td>
<td>L1: $\ell(x, y) =</td>
</tr>
<tr>
<td>Relative MrSE:</td>
<td>$\ell(x, y) = \frac{(x-y)^2}{y^*+\varepsilon}$</td>
<td>MAPE: $\ell(x, y) = \frac{</td>
</tr>
</tbody>
</table>

Mean Relative Squared Error (MrSE), commonly referred to as relative MSE, is a popular alternative to MSE in denoising. Large errors are often made in very bright reasons in the image, but they are visually less severe than the same error in a dark region. To correct for this, MrSE divides the squared color difference by the squared ground truth color. To avoid division by zero, there is a small constant, usually $\varepsilon = 10^{-3}$, in the denominator. Figure 4.12 shows attention heat maps of different loss functions. MrSE indeed remove some spikes in the output, but it is often still unstable in regions where the ground-truth color is dark and the predicted color light.

New to the field of denoising, we introduce Mean Average Percentage Error (MAPE) as the $L_1$ equivalent of MrSE. The formula is also shown in Table 4.1.

4.4.2 Structured loss

Next to per-pixel loss functions, the Structural Similarity Index Measure (SSIM) [Wang et al. 2003] is also popular in the field of denoising. The SSIM compares image patches (in our case of size $5 \times 5$), and evaluates them by their similarities in luminance, contrast and structure. Those are evaluated as:

$$l(x, y) = \frac{2\mu_x\mu_y + c_1}{\mu_x^2 + \mu_y^2 + c_1},$$

$$c(x, y) = \frac{2\sigma_x\sigma_y + c_2}{\sigma_x^2 + \sigma_y^2 + c_2},$$

$$s(x, y) = \frac{\sigma_{xy} + \frac{1}{2}c_2}{\sigma_x\sigma_y + \frac{1}{2}c_2}.$$  \[4.16\]

In these equations, $\mu_x$ stands for a mean, $\sigma_x$ for standard deviation, $\sigma_{xy}$ for covariances, and $c_1$ and $c_2$ are constants. Those components are combined in the similarity metric:

$$\text{SSIM}(x, y) = l(x, y) \cdot c(x, y) \cdot s(x, y).$$  \[4.17\]
Figure 4.12: Loss heatmaps. Heatmaps of different losses between denoised images and their references are shown. Heat maps are constructed such that the largest error is shown as bright yellow, and a loss of zero is purple. In all heat maps, except the one for SSIM, the losses are clipped at 50% to make them visually clearer.

The SSIM value lies between -1 and 1, where 1 indicates perfect correspondence. To use SSIM as a loss function, we use

$$\ell_{\text{SSIM}}(x, y) = 1 - \text{SSIM}(x, y).$$

As can be seen in Figure 4.12, SSIM generally corresponds better to human perceptual similarity than per pixel measurements. This comes at a cost of being more difficult to optimize for. A procedure that we will show to be helpful in this respect, is to pre-train a neural network with a stronger / more convex loss such as $L_1$, and then finetune it using the SSIM loss.

There are also multi-scale variants of SSIM [Wang et al. 2004] which capture the idea that humans perceive structure and contrast on a large range of frequencies, and not only locally. We also evaluate their MS-SSIM metric by combining the SSIM losses at multiple (3 in practice) scales.

In Section 7.5 we evaluate different loss functions for training. To evaluate the quality of our results, we provide the numbers multiple metrics.
As part of their Ray Histogram Fusion denoiser, Delbracio et al. [2014] extend their method to denoise on multiple scales. They motivate this architecture by observing that the Monte Carlo noise has a white noise distribution, and it is spread over all frequencies. When denoising with patches, low frequency noise for which the wavelength exceeds the patch size cannot be removed in this way. Instead of making the patches very large, Delbracio et al. [2014] use small patches at three scales, each of which are sub-sampled by a factor of 2. This observation is actually similar to the one made by Yang et al. [2016] when they introduce repeated modules, but the solution is different.

We propose two schemes to use neural networks in a multi-scale fashion: sequential multi-scale and parallel multi-scale. Both can be used with the identical network at every scale, sharing parameters, or the networks at each scale can have their own parameters.

5.1 Parallel design

The first scheme follows the approach by Delbracio et al. [2014]. We call it the parallel design. Let \( D \) be a Gaussian down-sampling operator that scales an image and its features down by a factor 2 after convolving with a Gaussian filter of \( \sigma = 0.96 \) to avoid aliasing. Define \( U \) as the bilinear up-sampling operation that up-scales an image by a factor 2. In the parallel design, \( n \) (typically 3 or 4) scales are filtered separately, and their contributions are combined with the following recursion:

\[
\hat{x}_n = S_n \\
\hat{x}_i = S_i + U(\hat{x}_{i+1}) - U(D(S_i)) \quad \forall i \in [1, n - 1],
\]

where \( S_i \) is the network output at scale \( i, i = n \) corresponding to the most sub-sampled image, and \( i = 1 \) to the original scale, and \( \hat{x}_i \) is the reconstruction at that scale. This method is conceptually simple, but suffers from two drawbacks. First of all, there is no control over the reconstruction. Frequencies that are slightly misaligned can lead to significant ringing artifacts. Secondly, it is not possible to add a loss to each layer separately, regularizing the network to get all scales right. This would motivate over-blurring at the largest scales. At those scales, the network would learn to filter more low frequencies than actually necessary, leading to over-blurring. The parallel architecture is illustrated in Figure 5.1.
Figure 5.1: Multi-scale: Parallel design. Numbers in parentheses represent the number of features in the layer. The notation \( /n \) stands for a spatial sub-sampling with a factor \( n \). All scales are denoised separately, and their results are combined according to Equation 5.1. Yellow circles represent substitution of the larger scale’s low frequencies by those of the smaller scale.

5.2 Sequential design

The second architecture we propose is designed to deal with these issues. We call it the sequential design, and on a high level, it feeds low-frequency corrections to the next scale, so that this scale can benefit from an improved input. The design is illustrated in Figure 5.2. Let \( I_i \) be the input of scale \( i \), and \( S_i \), the output of that scale. \( i = 1 \) corresponds to the normal size, and \( i = n \) to the smallest scale. In the sequential model, the input of a scale is given by the following recursion:

\[
\begin{align*}
I_n &= \hat{I}_n \\
I_i &= \hat{I}_i + U(S_{i+1}) - U(D(\hat{I}_i)) \\
&= \hat{I}_i + U(S_{i+1} - \hat{I}_{i+1}) \quad \forall i \in [1, n-1],
\end{align*}
\]

where \( \hat{I}_n \) are the rendering features downscaled by a factor \( 2^{n-1} \). In other words, the low frequencies of the input are corrected by the corrections from the previous layers. This gives the network at each scale the possibility to fix small frequency misalignment problems, and it also allows for the addition of a separate loss for each scale, since the output at each layer should be a good estimate of the converged image at that scale in the sequential model. A downside of this architecture is that sharing network parameters across scales becomes unintuitive. The smallest scale, which gets an input that is not pre-corrected should arguably have a different behavior than the other scales.

To correctly subsample variances, we observe that after down-sampling, the resulting pixel colors are a linear combination of independent pixel colors in the input. Therefore, the correct variance of those resulting pixels can be computed by sub-sampling the input variance with the squared weights.
The sequential multi-scale model can be trained with only a final loss, or with a loss per scale. For a network in which each scale has different parameters, this can be very helpful. It is less important when scales share layers. In the Results section, both of our multi-scale architectures are evaluated. They can be used in combination with any of the proposed single scale architectures presented in Section 4.
<table>
<thead>
<tr>
<th>Training set</th>
<th>Test set</th>
<th>Validation set</th>
</tr>
</thead>
<tbody>
<tr>
<td>15 scenes</td>
<td>3 scenes (bedroom, living room, house)</td>
<td>16 scenes</td>
</tr>
<tr>
<td>200 perturbations per scene</td>
<td>10 sampling rates</td>
<td>4 sampling rates per scene</td>
</tr>
<tr>
<td>6 sample rates per perturbation</td>
<td>100 noisy renderings per scene</td>
<td>(128, 256, 512, 1024)</td>
</tr>
<tr>
<td>100 patches per image post-rendering augmentation</td>
<td>100 patches per rendering</td>
<td>exact views and textures are not in the training set</td>
</tr>
</tbody>
</table>

6 | Data

The recent successes in machine learning rely on large datasets to learn from. Optimizing Equation 3.2 over the training set only works well if this data is representative of the all renderings of interest. This poses a challenge for the application of machine learning to Monte Carlo denoising. In contrast to natural image denoising, we cannot just corrupt existing images with a particular type of noise and train on this. We need to actually render many images in both low and high quality. This process is expensive for multiple reasons: (1) computing converged references is very expensive and can take hundreds of core hours per frame, (2) creating distinct scenes with different geometric and optical properties requires considerable manual labor, and (3) since color intensities are stored linearly in a HDR format, and additional features need to be stored as well, files tend to be large. A typical noisy scene with additional rendering features is around 100 MB.

This chapter outlines how we deal with these difficulties. We describe how we use the training scenes used by Bitterli et al. [2016], and augment this data with both pre- and post-rendering transformations. Finally, we list the additional rendering features we store to aid the denoiser in Section 6.5. Table 6.1 gives a summary of our datasets.

6.1 Scenes

To train and evaluate our denoiser, we use a number of scenes from Bitterli et al. [2016]. The scenes are illustrated in Figure 6.1. They are chosen to represent a reasonable range of geometric and optical properties. There are many reflections, indirect illumination, and different
Figure 6.1: Scenes in our dataset. The scenes are selected to cover a wide range of optical features and noise patterns.

textures. Motion blur and depth of field are not included in our data. These effects would be interesting to add in future research.

6.2 Pre-rendering augmentation

To train a network, a dataset consisting of only 16 unique images is very limited. Since there would be a limited range of colors and shapes, the generalization behavior of the learned model would be bad. Unfortunately, creating many more different scenes is an expensive undertaking. To accommodate this, we use a pre-rendering augmentation pipeline that applies random perturbations to scenes such as changing lighting, textures, materials, roughness parameters, and camera parameters. In this way, we can generate many visually very different views of the same geometry.

Our augmentation pipeline takes a scene description file and a file outlining the perturbation space. The perturbation space description file contains:

- **hotspots**: interesting points in the scene that the camera may be pointed at. One of these hotspots is sampled uniformly.
Figure 6.2: Expanded dataset. 8 of the perturbed renderings for the scenes car and bathroom are shown.

- **camera range**: one or multiple 3D boxes in which the camera position can be uniformly sampled.
- **field of view range**: an interval in which the field of view parameter of the camera can be sampled.
- **materials**: for materials in the scene description file, a set of possible material configurations that could be applied. Such a set could for example be \{wood, bright color, fabric\}. We have material distributions for wood, metal, diffuse colors, parquet, environment maps, tiles and fabric, which can be sampled from.
- **primitive transformations**: transformations that are applied to entities in the scene description, such as rotation (used for environment maps).

With this pipeline, we generate 200 perturbations of each of the scenes in Figure 6.1, and manually prune ones that are not sensible. A selection of the generated perturbations for two scenes are displayed in Figure 6.2.
6.3 Sampling and references

We render the 200 perturbations for each of these scenes at a range of sampling rates \(2^k \mid k \in [3, m]\), where \(m\) is the maximum sampling rate. The maximum sampling rate is determined visually per scene, at a point at which the renderings are easy to denoise.

As we have mentioned before, rendering reference images is expensive. To ease this problem, we work with non-perfect references. We render the ‘references’ at a sampling rate that is two times higher than the maximum sampling rate of the noisy inputs, and use the state-of-the-art NFOR denoiser [Bitterli et al. 2016] to denoise this. By providing NFOR with higher quality inputs than our training data, we leave room to improve on it. The non-perfect references are only used for training. For validation, only real references are used to be able to compare with other denoisers in a fair way.

6.4 Post-rendering augmentation

To further artificially increase the size of our training set, we have a post-rendering pipeline that transforms images pseudo-randomly.

The network input for training are image patches of size 64 x 64. Before training, these undergo the following transformations:

- **color swapping**: the RGB channels of color, color variance, albedo and albedo variance are randomly permuted.
- **mirroring**: all features are randomly mirrored horizontally or not.
- **normal swapping**: the XYZ components of the normal feature are randomly permuted.

6.5 Rendering features

We aim to make our network architecture applicable to different rendering features. The designs illustrated in Chapter 4 can have input with arbitrary numbers and types of features. In our evaluation, however, we use a specific set of 5 features. These are the same features used by Bitterli et al. [2016]. A visual overview of the features is given in Figure 6.3. The features are:

- **color**: the rendered output, the channel that should be denoised,
- **albedo**: the albedo feature captures texture colors, without lighting applied to them,
- **normal**: normal direction at each pixel,
- **depth**: the average distance from the camera to a ray’s first intersection,
Figure 6.3: Input features. For each of the features color, albedo, normal, depth and visibility, we have the mean of $n$ samples and the variance. The images show the mean.

- visibility: At the first intersection, a light source is sampled. If there is object in between the light source and the intersection point, a 1 is sampled, otherwise a 0. The visibility buffer is the mean of these values.

All of these features are means of samples. An estimated variance for this mean is also provided as separate features.
In the previous sections, we have highlighted several challenges that arise when applying machine learning to denoising (Section 3.2), presented a framework to organize solutions to these challenges (Section 4) and presented a large number of components that are designed to tackle the challenges (Sections 4 and 5). In this chapter, we evaluate the proposed components in networks trained with the dataset presented in Section 6.

We first evaluate the various proposed components to establish to what extent they contribute to denoised output quality or the convergence behavior of the network. Since the space of potential combinations of components is exponential and training a single model takes 24 hours, we are naturally unable to evaluate all configurations. Instead, we vary one component at a time and fix the rest at reasonable value. With this approach, we can never conclude with confidence that a certain component brings no benefit, since it might work in a different configuration. We can, however, conclude that a component does work in some cases if it shows to be beneficial.

After evaluating our proposed components in isolation, we construct promising network configurations and evaluate their quality against state-of-the-art denoisers. For this validation, we make use of the dataset presented by Bitterli et al. [2016]. Most of the validation scenes were used as a basis for our training set, but in the training set the textures, colors and camera parameters are different. If our denoisers works for these scenes, it can be concluded that they generalize to scenes similar to those in the training set in terms of light transport and geometric complexity.

In all experiments we use the Adam optimizer [Kingma and Ba 2014] applied with batches of 32 patches of size $64 \times 64$ with an initial learning rate of $10^{-3}$ that decays by a factor 0.96 every 10,000 iterations. We differentiate between two pre-processing blocks:

- **Full colors** The per-pixel features after pre-processing are: the gamma-corrected color features, the raw visibility feature, raw normal feature, depth feature treated as in Section 4.1.1, relative color variance, relative albedo variance, normal variance transformed as described in Section 4.1.1, and relative visibility variance.

- **Decomposed colors** The same pre-processor as ‘Full colors’, but with the color feature split into a direction and a magnitude component.

As the main building block in the body of the network, we use a block consisting of (1) 64 input channels, (2) $(1 \times 1)$-convolution to 32 channels, (3) $(3 \times 3)$-convolution to 32 channels, (4) $(3 \times 3)$-convolution to 32 channels, and (5) $(1 \times 1)$-convolution to 64 channels. Each convolution is followed by the addition of a bias-term and a ReLU-activation. Blocks like this, with a lower internal dimensionality, have been introduced in the Resnet-paper [He et al. 2015]. This main building block is repeated a variable number of times $b$ with separate parameters, consti-
tuting a multi-block. The full network-section of our architecture consists of \( r \) multi-blocks that are repeated with the same shared parameters. We refer to a \( (b, r) \)-network as a network with \( b \times r \) blocks: \( r \) repeated multi-blocks of \( b \) blocks with unique parameters each. The blocks in a network can either predict residuals, as described in Section 4.2.3, or just be linked normally. If a network has residual-blocks, we call it a residual-network.

The last important factor of variation of our networks is the reconstruction component. This can either be direct prediction or kernel-prediction. Kernel-prediction needs to be specified with a kernel size and normalization strategy, as described in Section 4.3.2. We denote a kernel-prediction reconstruction component with \( n \times n \) kernels and softmax normalization as \( (n, \text{softmax}) \)-kernel-prediction.

Note that all plots that show the evolution of test-loss during training make use of rolling window averaging with a window size of 10 to make the graphs more readable.

7.1 Whitening

LeCun et al. [1998] describe input whitening as a best-practice for training feed-forward neural networks. We can easily illustrate the benefit of whitening with a simple example. We try to train a single-layer convolutional network without activation function that should learn the identity function, i.e. reproduce its input. The network inputs and outputs are of size \( 64 \times 64 \times 3 \), so the network should predict 3 values ‘per pixel’. We generate random data to train the network by drawing values \( \tilde{x}_{ij} = y\tilde{a} + (1 - y)b, \tilde{a} \sim N^3, b \sim N \), where \( N \) is a standard normal distribution. The factor \( y \) determines the degree of correlation of the three channels, where for \( y = 1 \) the pixels are perfectly independent and for \( y = 0 \) perfectly correlated. Figure 7.1 shows that with the vanilla SGD optimizer, there is a significant drop in the convergence rate when inputs and outputs of the network are more correlated. When the Adam optimizer [Kingma and Ba 2014] is used, however, the convergence rate is quickly restored by momentum, reducing the need for whitening.

With this theoretical experiment in mind, we evaluate the effect of ZCA whitening of the input features at per-pixel level. ZCA whitening is compared to per-feature standardization on a direct-prediction \( (1, 6) \)-residual-network with full-color-pre-processing. The network is optimized by minimizing MAPE loss on the training set. Figure 7.2 shows the values of certain loss functions evaluated on the test set over the course of training for 24 hours. We observe no difference in the final quality or convergence speed between the two alternatives.
Figure 7.1: Identity-learning experiments to test the need for whitening. The left plot shows the convergence of models trained with data in which the three values per pixel have different degrees of correlation. The convergence rate quickly degrades as more correlation is added. In the right figure, vanilla SGD and the Adam optimizer are used to train with the same input data ($y = 0.9$). With superlinear convergence, Adam reduces the need for uncorrelated data.

Figure 7.2: Test-losses for two networks during training. The green line is whitened, whereas for the blue line the features are only standardized per feature. Both networks have 6 repeated convolutional blocks that predict residuals and the network directly predicts an output color in the tone-mapped color space. The networks are trained with Adam with a learning rate of $10^{-3}$. There is no noticeable difference in either output quality nor convergence.
7.2 Repeated design

We introduced the repeated design in Section 4.2.2, inspired by the repeated residual design used by Yang et al. [2016]. The motivating reason for repeated blocks is that they allow for extension of the spatial region of influence for an output pixel, while keeping the number of parameters in the model under control. A model like that is easier to train.

To test the repeated architecture, we evaluate it both in direct-output networks (Figure 7.3) and in kernel-prediction networks (Figure 7.4). First, we compare (1, 6) and (6, 1)-direct-output residual-networks with full-color-pre-processing trained on MAPE loss. Two replications of this experiment have been conducted, and in the first experiment, the non-repeated (6, 1)-residual-network reaches a lower loss than the (1, 6) equivalent. In the second replication, however, the (6, 1)-residual-network got stuck in a local minimum, and the repeated (1, 6)-residual-network was better. The converged quality of the repeated network is stable across replications. From this experiment we conclude that whereas networks with unique parameters can potentially lead to better results, repeated networks with fewer unique parameters are more robust and easier to train.

In kernel-prediction networks, which are intrinsically regularized, the repeated design is not required to stabilize convergence of the optimization procedure. This is confirmed by the experiment shown in Figure 7.4, where we compare the convergence of three (n, d)-residual-networks with (21, softmax)-kernel-prediction and decomposed colors trained on MAPE loss, where $n \cdot d = 6$. The network with 6 unique blocks consequently performs best in this setup.
Figure 7.4: Repeated design in kernel-prediction networks. We train three residual networks with \((21, \text{softmax})\)-kernel prediction. The \((6,1)\)-architecture that has unique weights per block outperforms the \((1,6)\)-network with a fully repeated architecture and the \((2,3)\) hybrid architecture.

We conclude that repeated blocks are very useful for direct-prediction architectures, but their benefit decreases for kernel-prediction networks.

7.3 Residual design

Residual connections, as discussed in Section 4.2.3, have proven successful both in high level classification tasks [He et al. 2015] and in natural image denoising [Yang et al. 2016]. To evaluate their effect in our framework, we conduct two experiments with and without residual connections. Training graphs of both experiments are shown in Figure 7.5. The first experiment evaluates residual connections in a direct-prediction context. We train two \((2,3)\)-direct-prediction networks with decomposed-colors-pre-processing and ZCA whitening that are trained on MAPE loss with the Adam optimizer and an initial learning rate of \(10^{-4}\). We can see that even with the low learning rate of \(10^{-4}\), the non-residual network does not converge. All gradients go to zero after a few iterations. The network with residual blocks does not suffer from this problem.

The second experiment we conduct is on \((2,3)\)-residual-networks with \((21, \text{softmax})\)-kernel-prediction, decomposed-colors-pre-processing and ZCA whitening that are trained on MAPE loss with the Adam optimizer and an initial learning rate of \(10^{-3}\). Also in this case, residual blocks are necessary for convergence of the networks. Our experiments illustrate that residual connections are very helpful for denoising in our framework and we will continue to use them in all our other experiments.
Figure 7.5: Test loss during training in two experiments evaluating the effect of residual connections. The left sub-figure compares the convergence of two (2,3)-direct-prediction networks with decomposed-colors-pre-processing trained on MAPE loss with an initial learning rate of $10^{-4}$. In the right sub-figure, the convergence of two (2,3)-residual-networks with (21,softmax)-kernel-prediction networks and decomposed-colors-pre-processing trained with MAPE loss is shown. In both cases, residual connections show to be essential for convergence.

7.4 Kernel-prediction

Section 4.3.2 introduced the kernel-prediction post-processing block. The goal of kernel-prediction is to regularize the denoising problem to better deal with high-dynamic-range output. In this section, we discuss experiments that show the effectiveness of the method and evaluate three normalization strategies: softmax-, $L_2$- and sum-normalization.

Figure 7.6 shows the evaluation of two (2,3)-residual-networks: one with (21,softmax)-kernel-prediction and one with direct prediction. Both networks are trained with the Adam optimizer on MAPE loss and use decomposed-colors-pre-processing and ZCA whitening. The figure shows how four different losses on the test set evolve over training time. The kernel-prediction network outperforms direct prediction both in speed of convergence and final quality.

As discussed in Section 4.3.2, normalization of the kernel weights is important, and we introduced several schemes to deal with it. We discussed softmax normalization: a scheme that generates positive weights that sum to one. Since this constraint is quite restrictive, we proposed to normalize weights such that their $L_2$-norm equals 1, or such that their sum equals 1. Figure 7.7 compares three (2,3)-residual-networks with (21,·)-kernel-prediction with each of these normalization schemes. The networks are trained with the Adam optimizer on MAPE loss and use decomposed-colors-pre-processing and ZCA whitening. From the results, we can conclude that softmax normalization is very robust and converges quickly. Sum normalization, however,
Figure 7.6: Comparison of kernel-prediction and direct-prediction networks. We evaluate (2,3)-residual-networks with (a) (21,softmax)-kernel-prediction or (b) direct prediction. Both networks are trained with Adam on MAPE loss, and use decomposed-colors-pre-processing and ZCA whitening. The kernel-prediction network outperforms direct prediction in both speed of convergence and final quality.

We choose to do most of our experiment with softmax normalization because of its robustness.

7.5 Trained loss

We evaluated multiple loss functions as training loss for our networks. In Section 3.2.3, we identified that a good loss function should both correspond well to human perceptual quality and be easy to optimize. With this in mind, we evaluate MSE, MAPE, L1 and SSIM loss as candidates for our optimization procedure. Figure 7.8 illustrates the experiment. The different loss functions were used as the trained loss in a (2,3)-residual-network with (21,softmax)-kernel-
Figure 7.7: Comparison of normalization schemes for kernel-prediction: softmax, $L_2$ and sum. We evaluate three (2,3)-residual-networks with (21,·)-kernel-prediction. All networks are trained with Adam on MAPE loss, and use decomposed-colors-pre-processing and ZCA whitening. Softmax normalization leads to the best convergence and most robust training.

prediction post-processing, decomposed-colors-pre-processing and ZCA whitening. They were trained with Adam with a learning rate of $10^{-3}$.

In this experiment, MAPE and SSIM arise as good candidates for training kernel-prediction experiments. Their merit can be intuitively explained in the sense that they do not put too much emphasis on single pixels, making the network learn from all pixels at all times. In our experiments with direct-prediction networks, however, the SSIM may result in color shifts and does not reach a good accuracy on very bright regions. Since training on MAPE loss gives similar accuracy to SSIM, even when evaluated with the SSIM metric itself, we choose for MAPE as a loss in most of our experiments.

An approach that has also shown to be useful, even with direct prediction, is fine-tuning with SSIM. When a network is first trained with a ‘stronger’ loss such as MAPE or L1, it can then be improved by continuing learning with structured loss and a smaller learning rate. This does not impact quality metrics much, but can yield visual improvements by better preserving fine details.
Figure 7.8: Comparison of different trained losses. MAPE, MSE, MrSE and SSIM loss were used as the trained loss in a $(2,3)$-residual-network with $(21,\text{softmax})$-kernel-prediction post-processing, decomposed-colors-pre-processing and ZCA whitening. They were trained with Adam with a learning rate of $10^{-3}$. The sub-figures show how different metrics on the test set evolve during training. MSE loss shows to be unsuitable for training these networks. With MSE, the parameters quickly get stuck in a local minimum. Overall, MAPE loss and SSIM give good results and rapid convergence.

7.6 Irradiance factorization

In Section 4.3.3 we proposed a post-processing block that denoises the albedo and irradiance components of the rendered color separately. This separation is easily motivated in the case of kernel-prediction networks as the required weights in the kernels for the irradiance feature can be much more aggressive in large diffuse textured areas such as wall-paper. Since the albedo feature has much less noise than the final color, it is easy to denoise as well. Although we observe no difference in the final quality of trained models with and without this decomposition, we can visually see improvements in such regions, as illustrated in Figure 7.9.
Figure 7.9: Effect of the irradiance-decomposition post-processing block. Two (1,6)-residual-networks with (21,softmax)-kernel-prediction, decomposed-colors-pre-processing and ZCA whitening have been trained with Adam on MAPE loss with an initial learning rate of $10^{-3}$. At low sample counts, the decomposition helps to reconstruct textures that are well captured by the texture feature.

The results in Figure 7.9 were generated with (1,6)-residual-networks with (21,softmax)-kernel-prediction, decomposed-colors-pre-processing and ZCA whitening that have been trained with Adam on MAPE loss with an initial learning rate of $10^{-3}$. One of the networks has the factorized reconstruction. We primarily see visual improvement of the decomposed post-processing in low sample count renderings at large textured areas.

7.7 Multi-scale reconstruction

We presented two designs for multi-scale convolutional neural networks for MC denosing in Section 5: (1) parallel design, where networks operate separately on multiple scales after which their outputs are merged and (2) sequential design, where each scale can benefit from the updates made by from the previous, smaller, scale. The parallel design was inspired by the approach taken by Delbracio et al. [2014] in their Ray Histogram Fusion denoiser. This approach can lead to problematic ringing artifacts in high-gradient regions when slight misalignments between the separate scales are created. Because of problem, we have not been able to train this model with HDR images. The parallel models we tried to train only learn to be the identity function: they do not denoise.

We successfully trained a sequential model with kernel-prediction. It is a (2,2)-residual-network on 3 scales with (11,softmax)-kernel-prediction, trained with ZCA whitening, decomposed-colors-pre-processing and optimized with Adam with an initial learning rate of $10^{-3}$. When compared to a similar single-scale network, we see visual improvements in the reduction of low-frequency residual noise (see Figure 7.10), but no improvement in metrics. There are still slight ringing artifacts left at high-gradient edges that hurt the metrics. A potential solution to this issue could be a more flexible sequential design, where the updates from the previous layer are
Figure 7.10: An example of an area where multi-scale reconstruction helps. We train a (2, 3)-residual-network with (21, softmax)-kernel-prediction on a single scale and a (2, 2)-residual-network with (11, softmax)-kernel-prediction with 3 scales using the sequential design. Both networks are trained with ZCA whitening, decomposed-colors-pre-processing and optimized with Adam with an initial learning rate of $10^{-3}$. Multi-scale reconstruction helps to remove low-frequency residual noise.

not directly added to the input of the next layer but simply provided as an extra feature. An evaluation of this proposed alternative is beyond the scope of this project.

7.8 Comparison to the state-of-the-art

The previous sections have singled out several of our proposed building blocks and evaluated their contributions to the speed of convergence and final quality. In this section, we compare four of our models to the current state-of-the-art in *a posteriori* image-space denoising for Monte Carlo renderings: NFOR [Bitterli et al. 2016] and RDFC [Rousselle et al. 2013]. We train the following models until the test loss stops improving:

- **direct**: a (6, 1)-network with direct-prediction post-processing.
- **single-scale kernel-prediction**: a (6, 1)-network with (21, softmax)-kernel-prediction post-processing.
- **decomposed kernel-prediction**: a (6, 1)-network with (21, softmax)-kernel-prediction and irradiance-factorized reconstruction with an albedo-kernel size of 5.
- **3-scale sequential kernel-prediction**: a (2, 2)-network with (11, softmax)-kernel-prediction at 3 scales with sequential design.

All models make use of residual blocks, ZCA whitening and decomposed-colors-pre-processing, and are trained with Adam and MAPE loss with an initial learning rate of $10^{-3}$ and a learning rate decay of 0.96 every 10,000 iterations.
Figure 7.11: A visual comparison of four of our denoisers against the state of the art (NFOR and RDFC) on patches of three difficult scenes in the validation set.

Figure 7.12 compares the final validation loss of those four models against the performance of NFOR and RDFC in various loss metrics. For each scene in our validation set at 64, 256 and 1024 spp, a blue line in the figure shows the ratio of the achieved loss in that scene relative to that of the best denoiser. Lower is better. Error bars show the mean and standard deviation across scenes in the test set. To compute the losses in a robust way we used a trimmed mean that removes the highest and lowest 0.1% of error values in each image. We observe that in MAPE, MrSE and SSIM loss, our kernel-prediction networks are competitive with the state of the art. The multi-scale network has high MSE values for some scenes, corresponding to artifacts at high-gradient edges. In the SSIM metric, NFOR still outperforms our networks. This corresponds to the observed visual quality. Although our learned denoisers suffer less from low-frequency residual noise, fine straight lines are better preserved by NFOR. Figure 7.11 shows visual examples of denoised quality.

We conclude that end-to-end-learned denoisers for Monte Carlo renderings can yield performance similar to that of state-of-the-art a posteriori denoisers. In future work, we expect learned denoisers to outperform classical denoisers. Potential drawbacks of end-to-end methods and limitations of our research are discussed in Section 8. In that section we also give suggestions for future work in this area.
Figure 7.12: Comparison of four of our denoisers against the state of the art (NFOR and RDFC). For each scene in our validation set at 64, 256 and 1024 spp, a blue line shows the ratio of the achieved loss in that scene relative to that of the best denoiser. Lower is better. Error bars show the mean and standard deviation across scenes in the test set.
Discussion and future work

In this work, we presented tools for a fully-learned approach to denoising Monte Carlo renderings in image space. Our main contributions are an extensible pipeline for augmenting 3D rendering data and using imperfect ground-truth renderings, the introduction of MAPE loss into the denoising community as an effective loss function for training neural networks with high-dynamic-range input and output data, and the kernel-prediction architecture which regularizes the denoising problem to make it easier to train. We consider kernel-prediction as an intermediate form between the Learning Based Filter introduced by Kalantari et al. [2015] that learns parameters of an existing filter, and a fully general network. For the dataset we worked with, kernel-prediction provided the optimal balance between model flexibility and robustness.

We found that kernel-prediction convolutional neural networks can yield competitive results to the state-of-the-art a posteriori denoisers NFOR [Bitterli et al. 2016] and RDFC [Rousselle et al. 2013]. There are cases in which the results are both visually and numerically better, but there are also cases where fine detail is better preserved by NFOR. These results are achieved by training on data that has the same optical and geometric properties as the validation set.

We believe the main advantage of denoisers optimized with machine learning is that the space of ML denoisers can include the current hand-made denoisers, but that ML denoisers can be better optimized for the particular task at hand and therefore outperform traditional techniques. Also, our kernel-prediction CNN model is very efficient to compute. With an average runtime of 20 seconds on an Intel i7-4930K CPU, it is about 4 times faster than the current NFOR code. On a Titan X GPU, the denoising time is further reduced to under 3 seconds on average.

This approach does not come without drawbacks. First of all, the process of finding an optimal architecture and tuning the optimizer requires considerable trial and error. With the current hardware, testing out new ideas is a time-consuming process which limits the speed of iteration. Lastly, perhaps the most important drawback of machine learning denoising is that there is no way yet to extend the model to use different input features without re-optimizing the whole model. Also, the models might not generalize well to optical effects and geometries that are not present in the training set.

The requirement of machine learning denoisers for training data similar to the scenes to be denoised has two sides. On the one side, ML denoisers might not generalize optimally to scenes with new light transport or unseen geometries, but on the other hand this means that they can be fine-tuned for a particular use case. In the movie industry, this could mean fine-tuning on few frames from a multi-second shot, before denoising the other frames.
8.1 Future work

We believe machine learning approaches to denoising have great potential, and we have had many more ideas than time to explore them. Therefore, in this section we list potential directions for future work that we think are promising.

8.1.1 General areas of improvement

We identify three ways in which our model is limiting and could be improved. First of all, current denoising approaches, including ours, have been local. Although the most relevant information for denoising a pixel will indeed be in its neighborhood, large spatial dependencies, such as between duplicates of the same object, could potentially be leveraged for improved quality. Secondly, we have provided variance estimates of our features as separate features, without explicitly encoding the relationship they have with the corresponding feature. We have not found a good way to make this relationship explicit, but feel this is important. Lastly, we have made use only of Monte Carlo means, and a variance estimate of these means. There is more information than that contained in the whole set of samples. Using all this information would require a tight coupling with a rendering system, since it would result in too much data to store permanently.

8.1.2 Specific extensions to our work

A first natural extension to the work in this thesis is one in the temporal domain. It should be straightforward to develop a temporal extension to our architecture by also including convolutions in the temporal dimension. An analysis of temporal stability of the denoiser should be conducted to make our methodology interesting for application in the movie industry.

Kernel-prediction was introduced as a solution to the problem of having data with a very high dynamic range. Although we show that it is an effective solution, we observe that the weight normalization schemes we described do not promote sparsity. In regions with very high contrast, however, sparsity of kernel weights is a very desirable property. We briefly looked into ways of enforcing this, but have had no success yet. Another idea is the concept of a ‘bias penalty’ on the kernel weights. We observe that the expected MSE of a the denoised images can be decomposed into a variance term and the bias squared. With kernel-prediction, bias can be extracted explicitly during training by applying the predicted kernel to the reference image. For an unbiased denoiser, this operation should yield the identity transformation in expectation. This idea can be used to weigh the variance an bias contributions of the loss separately.

Multi-scale denoising is something that makes sense intuitively. Monte Carlo noise is visible in all frequencies, and multi-scale denoising is an efficient way to deal with this problem. We discussed our proposed parallel and sequential multi-scale architectures, but have not been able to show significant numerical improvements. Even in the sequential design the denoiser suffers from artifacts generated by misaligned frequencies when two scales are combined. This problem could potentially be solved by a more general and flexible sequential multi-scale architec-
ture that does not add low frequency updates to the next scale explicitly, but provides this data as a feature instead.

8.1.3 New directions

Jampani et al. [2015] introduced the concept of a ‘bilateral neural network’ to learn sparse high dimensional filters. The authors extend convolutional neural networks by not just convolving in image-space dimensions, but by applying convolutions in a high-dimensional feature space. They make this operation computationally feasible by constructing a permutohedral lattice in the high dimensional space on which the convolutions are computed. We believe this is a very promising approach, and think bilateral neural networks could be well suited for denoising Monte Carlo renderings, since 3D coordinates and normal directions, for example, can be easily extracted from the rendering system, and would construct a better space for denoising than the image space.

We did considerable work to find and evaluate good image loss functions for our application. Our work often outperforms the state-of-the-art denoisers, but occasionally suffers from unpleasing visual artifacts that are not captured by the loss functions we employed. For generative models, adversarial training [Goodfellow et al. 2014] has gained in popularity by providing an implicitly learned loss function. Recently, Ledig et al. [2016] used such a loss metric for image upsampling. We believe it is worthwhile to investigate the use of such structures for ML denoising as well.
Conclusion

The inherent generality of Monte Carlo path tracing makes it the current method of choice for image synthesis in the movie industry but a prohibitively high number of samples per pixel is required to come to a visually converged estimate. To deal with this problem, denoising algorithms have been created to leverage spatial redundancy in images to estimate the converged image from a noisy rendering. Traditional and state-of-the-art denoisers such as NFOR [Bitterli et al. 2016] and RDFC [Rousselle et al. 2013] achieve this with human-engineered and -tuned algorithms. Kalantari et al. [2015] presented a way to optimize the parameters of existing filters based on data. In this work, we have aimed to take this approach a step further, and learn more general models for denoising Monte Carlo renderings from data.

Our first contribution has been a framework for augmenting training data that is designed to deal with the problems that (1) generating reference renderings is computationally expensive and (2) generating scenes with enough geometric and optical variety requires extensive human labor.

We evaluated several loss functions that can be used to optimize neural network models that operate on Monte Carlo renderings. The high dynamic range of such renderings makes losses such as MSE unsuitable since they put too much emphasis on getting few difficult pixels right, losing the ability to learn from all data. We introduced the MAPE loss (Mean Absolute Percentage Error) to the denoising community as a more robust alternative to the popular relative MSE metric used in denoising as a suitable loss for training models for denoising.

Our main contribution is the kernel-prediction design that can be used in combination with any convolutional neural network. Instead of predicting denoised colors directly, a kernel-prediction network predicts a local filtering kernel at each pixel, regularizing the problem. The kernel-prediction design makes our models more robust than direct prediction in exchange for generality.

We used the kernel-prediction framework in combination with repeated- and residual-blocks of convolutional layers that have been evaluated for their effect on convergence speed and contribution to the final denoising quality. We especially find that our models benefit greatly from a residual architecture.

Compared to state-of-the-art denoisers, our model shows competitive performance on validation-scenes that are similar to our training data. To denoise data with uncovered effects such as motion blur or depth of field, these effects have to be included in the training set. We see great potential in machine learning for denoising Monte Carlo renderings and provide many possible directions for future work.
Bibliography


68


Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Kernel-predicting Convolutional Neural Networks for Denoising Monte Carlo Renderings

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):

Vogels

First name(s):

Thijs

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the Citation etiquette information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date

Zürich, 15-3-2016

Signature(s)

Thijs Vogels

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.