SVBRDF Texture Synthesis with Convolutional Autoencoders

Mikhail Andrenkov

Hertford College

University of Oxford

A dissertation submitted in partial completion of the degree

Master of Science in Computer Science

Trinity Term 2020

Acknowledgements

First and foremost, I wish to express my sincerest gratitude to my project supervisor, Stefano Gogioso, for constantly challenging the ambitions of this project, providing invaluable feedback and suggestions from start to finish, and for entertaining my casual conservations during our weekly meetings. I would also like to thank the members of the OUFC, HCBC, and OUIHC for reminding me there is more to the Oxford experience than problem sheets and exams. Finally, this dissertation would not be complete without the emotional support and wonderful portrait of my dog, Sherlock.

Abstract

An outstanding problem in texture synthesis research is accurately inferring the spatially-varying bidirectional reflectance distribution function (SVBRDF) which characterizes the visual appearance of a surface for any combination of lighting and viewing conditions. Another open research question concerns modelling the underlying generative process of a texture in order to expand its spatial extent to the infinite plane. This research project explores joint solutions to both of these tasks using convolutional autoencoders. Specifically, the goal is to train a machine learning model that accepts a single fronto-parallel flash-lit photograph of a texture as input and produces an SVBRDF parameterization of the same texture as output. This is achieved by optimizing a convolutional autoencoder to minimize the perceptual difference between two renderings of a texture based on the predicted and ground-truth SVBRDF parameters, respectively. The final performance of the model is evaluated with respect to its ability to reconstruct, interpolate, and expand textures.

Contents

List of Figures

List of Tables

List of Abbreviations

BDPT . . . . Bidirectional Path Tracing

BGD . . . . . Batch Gradient Descent

BN . . . . . Batch Normalization

BRDF . . . . Bidirectional Reflectance Distribution Function

BSDF . . . . Bidirectional Scattering Distribution Function

BTDF . . . . Bidirectional Transmittance Distribution Function

DBN . . . . . Deep Belief Network

EL . . . . . Efros-Leung

ELBO . . . . Evidence Lower Bound

FNN . . . . . Feedforward Neural Network

GAN . . . . . Generative Adversarial Network

GD . . . . . Gradient Descent

GPU . . . . . Graphical Processing Unit

GTR . . . . . Generalized Trowbridge-Reitz

ICS . . . . . Internal Covariate Shift

IN . . . . . Instance Normalization

IRP . . . . . Inverse Rendering Problem

LSTM . . . . Long Short-Term Memory

MAP . . . . . Maximum A Posteriori

MGD . . . . . Mini-Batch Gradient Descent

ML . . . . . Machine Learning

MLP . . . . . Multi-Layer Perceptron

MRF . . . . . Markov Random Field

MRS . . . . . Multi-Resolution Synthesis

MSE . . . . . Mean Squared Error

MTV . . . . . Mean Total Variation

NDF . . . . . Normal Distribution Function

PBR . . . . . Physically-Based Rendering

PPM . . . . . Progressive Photon Mapping

PRNG . . . . Pseudorandom Number Generator

RBM . . . . . Restricted Boltzmann Machine

RDF . . . . . Radial Distance Field

RE . . . . . Rendering Equation

RGB . . . . . Red, Green, and Blue

RMSE . . . . Root Mean Squared Error

RNN . . . . . Recurrent Neural Network

SGD . . . . . Stochastic Gradient Descent

SMF . . . . . Shadow-Masking Function

SVBRDF . . Spatially-Varying Bidirectional Reflectance Distribution Function

TSVQ . . . . Tree-Structured Vector Quantization

VAE . . . . . Variational Autoencoder

WL . . . . . Wei-Levoy

1 Introduction

Contents

This chapter outlines the philosophical foundation of the research project. To start, Section 1.1 explains the basic premise of the research effort and motivates its undertaking by discussing potential applications. Afterwards, Section 1.2 describes the scope and evaluation criteria of the project. To conclude, Section 1.3 briefly summarizes the contents of the remaining chapters.

1.1 Motivation

Textures are ubiquitous: virtually any surface in the real world can be modelled as an instance of a texture, no matter its geometry, composition, or asymmetries. From a practical perspective, textures serve to visually communicate the physical properties of a surface and distinguish objects with a similar shape from one another. For this reason, digital textures based on real surfaces often strive to faithfully reproduce the appearance of their real counterparts under as many lighting and viewing conditions as possible lest they compromise the immersion or realism of

a virtual scene. It follows that modelling a texture as a matrix of colours (i.e. an image) is insufficient for 3D applications where a surface may appear brighter or darker depending on the lighting environment and the relative position of an observer. Instead, the appearance of a texture is typically represented using a reflectance model that describes the behaviour of light at each point on a surface. Accurately inferring the parameterization of a reflectance model from a single photograph of a texture is an open research question in the field.

While reconstructing an instance of a texture exactly is desirable in some circumstances, the more relevant challenge is usually to expand a texture sample beyond its spatial boundaries. For example, rather than capturing the appearance of a road by exhaustively measuring its reflectance, it is more practical to design a *texture synthesis* algorithm that accepts a picture of an asphalt surface as input and generates, as output, a texture instance of arbitrary size with the same aesthetic qualities as the asphalt in the input image. The earliest texture synthesis algorithms perform well on textures that satisfy certain modelling assumptions involving locality and stationarity but tend to be characterized by severe scalability issues. More recently, machine learning (ML) techniques have been employed to synthesize large textures of unprecedented quality from relatively small exemplar images. Unfortunately, these approaches either rely on trivial reflectance models or have limited capacity to generalize to new data.

This research project represents an ambitious attempt to devise a texture synthesis algorithm that is capable of producing large textures in the style of an exemplar image using a sophisticated reflectance model that is an industry standard for photorealistic rendering. If successful, this work could revolutionize the acquisition and generation of textures in the digital art industry, including films, video games, and computer-aided design tools. Specifically, the model featured in this research project has the potential to replace some applications of expensive reflectance capture devices, enhance the productivity of digital artists creating massive texture assets, and offer creative insights into the diversification of a single texture or the interpolation of multiple textures.

1.2 Scope

Ultimately, this research project is about training an ML model that predicts the macroscopic geometry and reflectance parameters corresponding to each pixel in a fronto-parallel, flash-lit image of a textured surface. It is assumed that the surface featured in the input photograph is approximately flat and positioned close enough to the camera to localize the influence of the flash to the center region of the picture. It is also assumed that the flash from the camera is the dominant source of illumination in the photograph. Note that there are no assumptions made with respect to the colours, patterns, or any other properties of the textures themselves. It is also worth mentioning that the output of the ML model is only applied to planar surfaces: volumes and curved regions fall strictly outside the scope of this project. This streamlines the training process of the model and simplifies the subsequent rendering procedure.

The performance of the final model is evaluated using a suite of experiments that involve reconstructing, interpolating, and expanding the spatial extent of textures. First, the reconstruction experiments test the ability of the model to reproduce textures under different lighting conditions and establish a baseline for the other experiments. Next, the interpolation experiments explore different strategies for blending two textures and implicitly judge the style transfer capabilities of the model. Finally, the expansion experiments exploit the internal representation of the ML model to tile an arbitrary subset of a plane with a reference texture. As a whole, the intent of these experiments is to qualitatively evaluate the fitness of the model by revealing its weaknesses and strengths at representative tasks.

1.3 Outline

The remainder of this dissertation is organized as follows:

- **Chapter 2** presents the theoretical underpinnings of the research project and serves as a reference point for the ensuing chapters.

- **Chapter 3** describes relevant aspects of the research project implementation that are necessary to reproduce the findings in Chapter 4.
- **Chapter 4** details the experiments used to evaluate the performance of the final model.
- **Chapter 5** offers conclusions from the research project and proposes some directions for future work.
- **Appendix A** displays results that fall outside the immediate scopes of Chapter 3 and Chapter 4.
- **Appendix B** documents the layout of the accompanying source code and includes an example of a YAML configuration file.
- **Appendix C** lists the sources of the training, validation, and testing datasets.

2
Background

Contents

This chapter introduces the theoretical knowledge necessary to understand and appreciate the implementation of the research project. First, Section 2.1 covers the fundamentals of physically-based rendering (PBR) and can be viewed as a selective summary of the Physically-Based Rendering module. Next, Section 2.2 presents a series of relevant machine learning concepts that should be familiar to most students who have studied the Machine Learning module. Finally, Section 2.3 combines aspects from the preceding two sections and serves as a literature review on the topic of texture synthesis. The material discussed in this final section is likely to only be familiar to experts in the field of computer graphics.

2.1 Physically-Based Rendering

2.1.1 Introduction

Central to the theme of the research project is *rendering*. Rendering is the process of converting a mathematical description of a scene into a representative image. Typically, a scene is modelled as a collection of objects, lights, and a virtual camera. The objects in a scene usually take the form of simple geometric primitives (e.g. triangles) where each vertex is associated with a set of properties (e.g. colour); however, this is a practical limitation rather than a theoretical one. The lights in a scene illuminate these objects and may take the form of point lights, area lights, or ambient lights to name a few variants. The virtual camera brings the former two components in perspective by defining the position, direction, and projection of a viewer observing the scene[1]. With this input, the renderer generates a corresponding two-dimensional (2D) array of pixels forming the desired image. While pixels are traditionally composed of red, green, and blue (RGB) colour channels, the concrete meaning of these values depends on the colour space of the image.

Rasterization

Today, there are two prevailing approaches to rendering: rasterization and ray tracing. Rasterization encodes the virtual camera of a scene as a series of linear transformations which can be applied to a vertex of a geometric primitive. The goal of these transformations is to translate a point that is defined relative to some origin in the scene to a position within the canonical view volume of the virtual camera. The vertices belonging to a primitive can then be projected, translated, and scaled onto a discrete image plane where the colour of the pixels between each vertex is computed using linear interpolation. This flavour of rendering dominates the majority of real-time graphics applications due to the hardware acceleration capabilities offered by graphical processing units (GPUs) and the support for rasterization pipelines in prominent graphics frameworks such as OpenGL [Khr].

[1] A different set of attributes may be needed to define other viewer models.

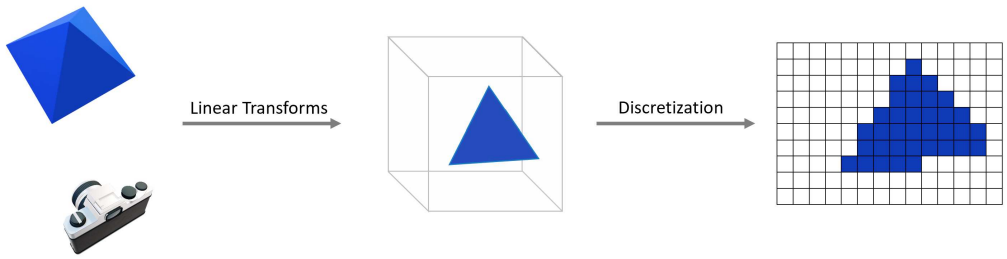


Figure 2.1: Rasterization shades pixels by transforming the scene geometry into a canonical view volume which can be projected and then discretized into an array of pixels.

Ray Tracing

In contrast, ray tracing involves casting rays from the virtual camera and tracking the nearest intersection of each ray with an object in the scene. The origin and direction of each ray are determined by the parameters of the virtual camera and the location of the pixel in the image associated with the ray, as shown in Figure 2.2. Upon colliding with an object, a ray can either be absorbed or recursively cast in a (new) direction. One key advantage of ray tracing is that it does not require objects to have an explicit geometry, thereby enabling the rendering of participating media such as fog. On the other hand, it tends to be slower than rasterization and is mostly reserved for applications where rendering latency is not a priority.

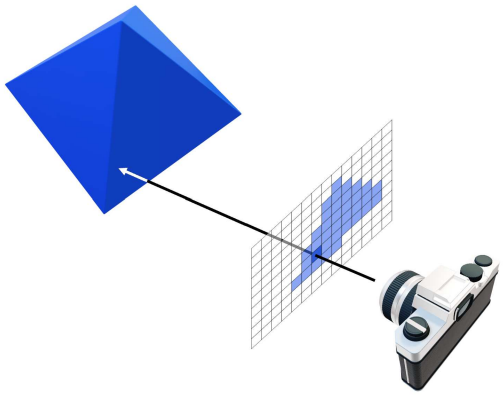


Figure 2.2: Ray tracing shades pixels by casting rays from the virtual camera through an image plane and following the paths of these rays until they are absorbed.

Physically-Based Rendering

Physically-based rendering is the application of physical light models to rendering. The aim of PBR is to improve the physical realism of a rendering process by modelling the interactions of light with surfaces and media in a way that naturally gives rise to subtle visual phenomena. Of course, more sophisticated light models, such as quantum optics, can simulate effects (e.g. fluorescence) that cannot be expressed by their simpler counterparts. In practice, many PBR methods are ray tracing variants that leverage importance sampling techniques to approximate an intractable rendering process. The remainder of this section explains the foundations of PBR with a special emphasis on components that are directly relevant to the research project.

2.1.2 Geometric Optics

The geometric optics light model captures many light interactions that are common in the real world. In this model, light is represented by a ray. When light enters a participating medium, it is absorbed or scattered with a probability proportional to the distance travelled through the medium according to the Beer-Lambert Law [Lam60]. When light reaches the boundary between two media (i.e. a surface), it is either absorbed, reflected towards the exterior of the surface, or refracted towards the interior of the surface. The ratio of reflected incident light to refracted incident light is governed by the Fresnel equations [Fre21].

Surface Interactions

Consider a scenario where a light ray hits a surface and is not absorbed. Let $\boldsymbol{n}$ denote the normal to the surface, $\boldsymbol{i}$ the opposite direction of the incident light ray, and $\boldsymbol{o}$ the direction of the outbound light ray, where each direction takes the form of a unit vector. Note that the orientation of $\boldsymbol{i}$ is a mathematical convenience that will become more clear as this chapter progresses. The possible outcomes of the collision are illustrated in Figure 2.3. Observe that all outbound rays are coplanar with the surface normal and incident ray.

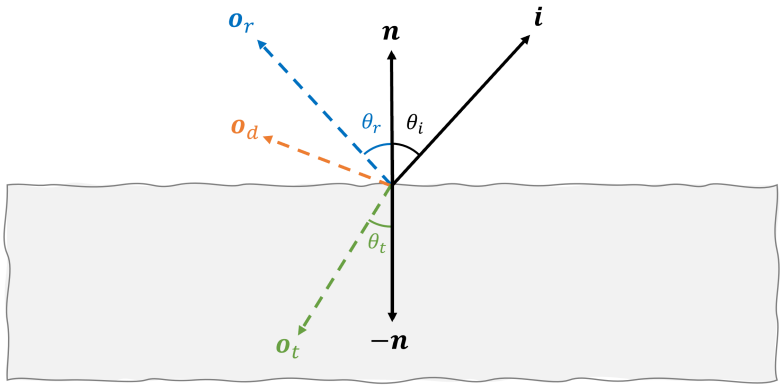


Figure 2.3: Scattering outcomes after a collision in the geometric optics model; ray $\boldsymbol{o}_r$ represents a specular reflection, ray $\boldsymbol{o}_d$ represents a diffuse reflection, and ray $\boldsymbol{o}_t$ represents a specular transmission.

One collision outcome is *specular reflection*. Here, the incident light ray is reflected across the surface normal, resulting in a reflection ray $\boldsymbol{o}_r$ where

$$\boldsymbol{o}_r = 2(\boldsymbol{i} \cdot \boldsymbol{n})\boldsymbol{n} - \boldsymbol{i} \ . \tag{2.1}$$

To understand the intuition behind Equation 2.1, recall that the dot product between two vectors is the projection of one vector onto the other. It is also worth noting the following property of specular reflections:

$$\boldsymbol{o}_r \cdot \boldsymbol{n} = \boldsymbol{i} \cdot \boldsymbol{n} \qquad \Longleftrightarrow \qquad \theta_r = \theta_i \ . \tag{2.2}$$

Another collision outcome is *diffuse reflection* whereby the incident light ray is scattered multiple times upon hitting a rough surface and leaves in an effectively random direction. As hinted by the relative magnitudes of the outbound vectors in Figure 2.3, the intensity of the incident light which is reflected in an arbitrary diffuse direction $\boldsymbol{o}_d$ is lower than that which is reflected in the specular direction $\boldsymbol{o}_r$.

The third collision outcome is *specular transmission*. In this case, the incident light ray is refracted through a surface and emerges at an angle to the negative normal that is determined by the indices of refraction of the media on either side of the collision. This angle, denoted by θ_t, is given by Snell's Law [Des37], where n_i and n_t are the indices of refraction of the media outside and inside the surface, respectively:

$$n_i \sin\theta_i = n_t \sin\theta_t \ . \tag{2.3}$$

Fresnel Equations

An important property of light that geometric optics does not explicitly model is polarization. Specifically, light is an electromagnetic wave with two transverse wave components that may be polarized (i.e. exhibit bias) in certain directions. Crucially, this polarization determines the reflectivity R and transmissivity T at the boundary of two media, or equivalently, the fraction of light that is reflected rather than refracted at the surface. Let s denote the electric polarization of light along the direction of the surface normal $\boldsymbol{n}$ and let p denote the polarization orthogonal to s (the magnetic field are ignored). The Fresnel equations [Fre21] state that the reflectivity of light polarized along the s and p directions are

$$R_s = \left| \frac{n_1 \cos\theta_i - n_2 \cos\theta_t}{n_1 \cos\theta_i + n_2 \cos\theta_t} \right|^2 \tag{2.4}$$

$$R_p = \left| \frac{n_1 \cos\theta_t - n_2 \cos\theta_i}{n_1 \cos\theta_t + n_2 \cos\theta_i} \right|^2 . \tag{2.5}$$

Furthermore, by the conversation of energy, the transmissivity coefficients are

$$T_s = 1 - R_s \tag{2.6}$$

$$T_p = 1 - R_p \ . \tag{2.7}$$

Although the derivation of the Fresnel equations is subtle, their influence on the visual perception of objects should not be underestimated. Consequently, the Fresnel equations are frequently integrated into the geometric optics model by assuming that all light is unpolarized. Under this assumption, the effective reflectivity and transmissivity of light at a media boundary is given by

$$R = \frac{R_s + R_p}{2} \tag{2.8}$$

$$T = 1 - R \ . \tag{2.9}$$

2.1.3 Radiometry

Radiometry is the area of study concerning the measurement of electromagnetic waves, including light. In the context of PBR, radiometry provides a framework

for describing the transmission of light and relating physical quantities, such as energy, to visual phenomena. Radiometry should not be confused with photometry which deals with the measurement of the *perception* of light energy.

Flux

Without delving too deeply into the physical sciences, a photon is a quantized unit of light that is characterized by a wavelength. The wavelength of a photon is inversely proportional to the energy carried by that photon through Plank's Constant. It follows that the energy of a collection of photons is the sum of the energies of the individual photons in the collection. Now, flux (Φ) is defined as the amount of energy moving through a surface per unit time. For a particular wavelength (or a small spectrum) of light, Φ is expressed in terms of Watts (Joules per second). Radiant flux is especially useful for measuring the quantity of light emitted from a light source.

Irradiance

Unfortunately, radiant flux is not particularly useful for describing the amount of power that is incident upon a surface because Φ is usually coupled with the size of the surface. Irradiance (E) overcomes this limitation by defining itself as the amount of flux Φ incident on a surface A per unit surface area:

$$E = \frac{\Phi}{A} = \frac{d\Phi}{dA} \ . \tag{2.10}$$

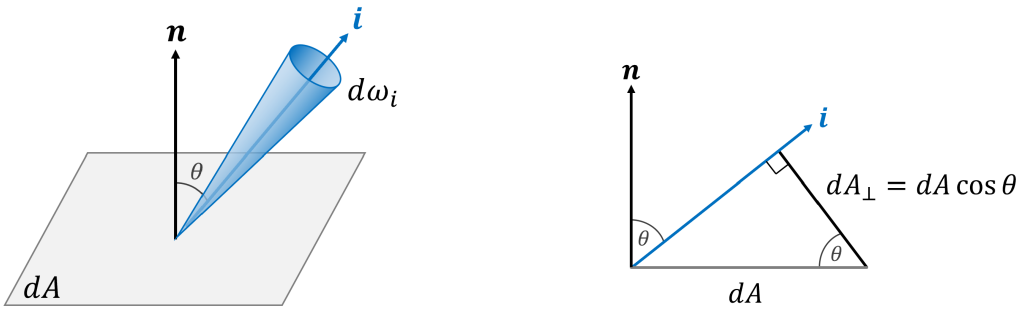
The units of irradiance are W/m^2. A similar quantity, radiosity (B), expresses the amount of emitted flux per unit surface area.

Radiance

The third radiometric quantity of interest, *radiance*, is responsible for the perceived brightness of a surface. Formally, radiance (L) is defined as the amount of radiant flux Φ per unit solid angle ω per unit projected surface area $A_\perp$:

$$L = \frac{\Phi}{\omega\, A_\perp} = \frac{d^2\Phi}{d\omega\, dA_\perp} \ . \tag{2.11}$$

The units of radiance are $\mathrm{W/sr \cdot m^2}$. Note that radiance can refer to either the flux incident on a surface or the flux exitant from a surface. To avoid confusion, the former type of radiance will be denoted by L_i and the latter type will be denoted by L_o.



(a) Radiance is defined with respect to an infinitesimal solid angle $d\omega$ and area dA.

(b) Projecting dA onto $\boldsymbol{i}$ gives $dA_\perp$.

Figure 2.4: Illustration of radiance and its associated projected area.

Notation

Throughout the PBR literature, irradiance and radiance are usually expressed as functions where $E(\boldsymbol{i})$ denotes the irradiance deposited from incident light coming from the direction of $\boldsymbol{i}$ and $L_i(\boldsymbol{i})$ denotes the incoming radiance from the direction of $\boldsymbol{i}$[2]. To this end, it is no coincidence that Equation 2.10 and Equation 2.11 are given as differential equations: putting these ideas together reveals a powerful relationship between irradiance and radiance. First, observe that from Figure 2.4b,

$$dA_\perp = dA \cos\theta = (\boldsymbol{n} \cdot \boldsymbol{i})\, dA \ . \tag{2.12}$$

Now, let Ω denote the unit hemisphere centered about the surface normal $\boldsymbol{n}$. Then,

$$L_i(\boldsymbol{i}) = \frac{d^2\Phi}{d\omega_i\, (\boldsymbol{n} \cdot \boldsymbol{i})\, dA} \tag{2.13}$$

$$\frac{d^2\Phi}{dA} = L_i(\boldsymbol{i})(\boldsymbol{n} \cdot \boldsymbol{i})\, d\omega_i \tag{2.14}$$

$$E(\boldsymbol{i}) = \int_\Omega L_i(\boldsymbol{i})(\boldsymbol{n} \cdot \boldsymbol{i})\, d\omega_i \ . \tag{2.15}$$

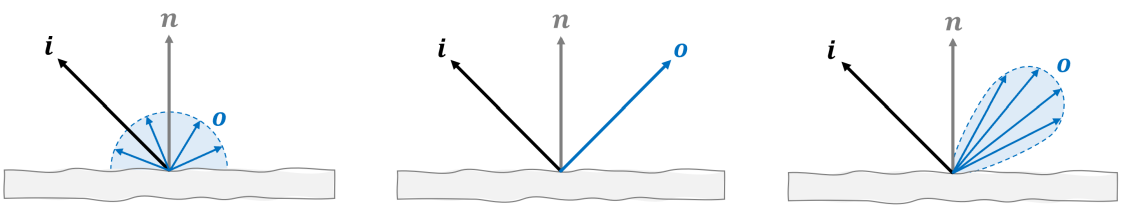
[2] An analogous pair of interpretations exist for $M(\boldsymbol{o})$ and $L_o(\boldsymbol{o})$.

2.1.4 Reflectance Models

The concept in PBR which marries radiometry to rendering is the *bidirectional reflectance distribution function* (BRDF). The BRDF (f_r) describes the proportion of incident light from an incoming direction that is reflected into an outgoing direction for any point on the surface of a given material. More concretely,

$$f_r(\boldsymbol{i}, \boldsymbol{o}) = \frac{dL_o(\boldsymbol{o})}{dE(\boldsymbol{i})} \tag{2.16}$$

where $dE(\boldsymbol{i})$ is the differential irradiance deposited from incident light in the direction of $\boldsymbol{i}$ and $dL_o(\boldsymbol{o})$ is the change in exitant radiance towards $\boldsymbol{o}$ caused by the incident light. The significance of the BRDF lies in the fact that the visual appearance of a material in a particular lighting environment can be derived from its BRDF[3]. It is worth noting that the dual of the BRDF, the *bidirectional transmittance distribution function* (BTDF), captures refractive behaviour and is essential for modelling translucent materials. Furthermore, combining the BRDF and BTDF yields the *bidirectional scattering distribution function* (BSDF), unless the material is a participating medium, in which case the term *phase function* is applied instead. Lastly, the spatially-varying BRDF (SVBRDF) is another variant of the BRDF where the BRDF of a point on a surface is parameterized by its spatial location. The units associated with a BRDF are sr^{-1}.



(a) Perfectly Diffuse BRDF **(b)** Perfectly Specular BRDF **(c)** Glossy BRDF

Figure 2.5: Sketches of a perfectly diffuse BRDF, a perfectly specular (mirror-like) BRDF, and a glossy BRDF. The blue regions and arrows indicate the values of the BRDF in a particular outbound direction $\boldsymbol{o}$ given an inbound direction $\boldsymbol{i}$.

[3]A generalization of the BRDF is needed to model translucent media

Rendering Equation

The utility of the BRDF is derived from its role in the rendering equation (RE) which relates incident radiance to exitant radiance through the BRDF. Indeed, the underlying objective of virtually all rendering programs is to calculate (or estimate) the distribution of radiance across the objects in a scene. The RE facilitates this task by combining Equation 2.15 and Equation 2.16 in the following manner:

$$f_r(\boldsymbol{i}, \boldsymbol{o}) = \frac{dL_o(\boldsymbol{o})}{dE(\boldsymbol{i})} \tag{2.17}$$

$$dL_o(\boldsymbol{o}) = f_r(\boldsymbol{i}, \boldsymbol{o})\, dE(\boldsymbol{i}) \tag{2.18}$$

$$\int_\Omega dL_o(\boldsymbol{o}) = \int_\Omega f_r(\boldsymbol{i}, \boldsymbol{o})\, dE(\boldsymbol{i}) \tag{2.19}$$

$$L_o(\boldsymbol{o}) = \int_\Omega f_r(\boldsymbol{i}, \boldsymbol{o}) L_i(\boldsymbol{i})(\boldsymbol{n} \cdot \boldsymbol{i})\, d\omega_i \ . \tag{2.20}$$

Note that the full RE includes an emissive radiance term $L_e(\boldsymbol{o})$ to account for surfaces that naturally radiate light:

$$L_o(\boldsymbol{o}) = L_e(\boldsymbol{o}) + \int_\Omega f_r(\boldsymbol{i}, \boldsymbol{o}) L_i(\boldsymbol{i})(\boldsymbol{n} \cdot \boldsymbol{i})\, d\omega_i \ . \tag{2.21}$$

By convention, the dependence of the exitant radiance on the surface normal $\boldsymbol{n}$ and light wavelength λ is omitted from the notation in the RE. Written explicitly,

$$L_o(\lambda, \boldsymbol{n}, \boldsymbol{o}) = \int_{\lambda \in \mathbb{R}} L_e(\lambda, \boldsymbol{o}) + \int_\Omega f_r(\lambda, \boldsymbol{i}, \boldsymbol{o}) L_i(\lambda, \boldsymbol{i})(\boldsymbol{n} \cdot \boldsymbol{i})\, d\omega_i\, d\lambda \ . \tag{2.22}$$

For practical reasons, the wavelengths featured in the RE are restricted to three canonical values representing red, green, and blue light respectively.

Physical Plausibility

Given the nature of the subject, it is desirable for BRDFs to be plausible in the physical sense. Such BRDFs must satisfy the following criteria [DBB18]:

1. **Range**: The range of the BRDF must be a subset of $\mathbb{R}^+$:

$$\forall \boldsymbol{i}, \boldsymbol{o} \in \Omega : \ f_r(\boldsymbol{i}, \boldsymbol{o}) \geq 0 \ . \tag{2.23}$$

2. **Reciprocity**: The BRDF must exhibit *Helmholtz reciprocity*. That is,

$$\forall \boldsymbol{i}, \boldsymbol{o} \in \Omega : \; f_r(\boldsymbol{i}, \boldsymbol{o}) = f_r(\boldsymbol{o}, \boldsymbol{i}) \; . \tag{2.24}$$

3. **Energy Conservation**: The BRDF must never allow the reflected radiosity to exceed the irradiance for any given distribution of incident light:

$$\int_\Omega L_o(\boldsymbol{o})(\boldsymbol{n} \cdot \boldsymbol{o}) \, d\omega_o \leq \int_\Omega L_i(\boldsymbol{i})(\boldsymbol{n} \cdot \boldsymbol{i}) \, d\omega_i \; . \tag{2.25}$$

This property applies to any distribution of incident light, including the one where $L_i(\omega_i) = \delta(\boldsymbol{n})$. Combined with Equation 2.20,

$$\int_\Omega L_o(\boldsymbol{o})(\boldsymbol{n} \cdot \boldsymbol{o}) \, d\omega_o \leq \int_\Omega L_i(\boldsymbol{i})(\boldsymbol{n} \cdot \boldsymbol{i}) \, d\omega_i \tag{2.26}$$

$$\delta(\boldsymbol{n})(\boldsymbol{n} \cdot \boldsymbol{n}) \int_\Omega f_r(\boldsymbol{n}, \boldsymbol{o})(\boldsymbol{n} \cdot \boldsymbol{o}) \, d\omega_o \leq \delta(\boldsymbol{n})(\boldsymbol{n} \cdot \boldsymbol{n}) \tag{2.27}$$

$$\int_\Omega f_r(\boldsymbol{n}, \boldsymbol{o})(\boldsymbol{n} \cdot \boldsymbol{o}) \, d\omega_o \leq 1 \; . \tag{2.28}$$

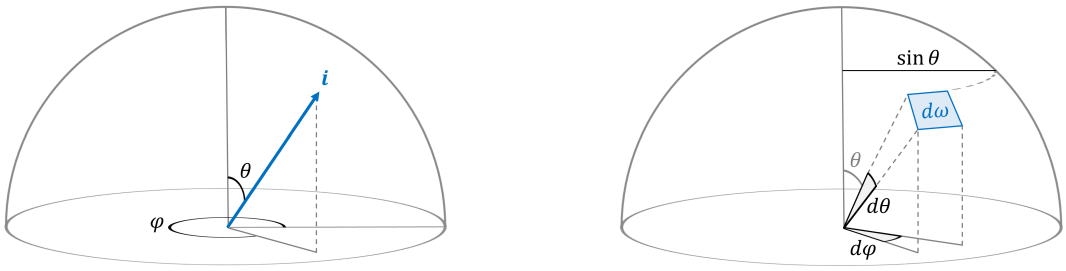
Lambertian BRDF

The simplest BRDF of any practical value is the *Lambertian* BRDF [Lam60] pictured in Figure 2.5a. The Lambertian BRDF represents materials with perfectly diffuse reflectivity and can be modelled using a constant value:

$$f_r(\boldsymbol{i}, \boldsymbol{o}) = \frac{\rho_d}{\pi} \; . \tag{2.29}$$

Above, ρ_d is called the *diffuse albedo* and models the ratio of light that is not absorbed by the material at a particular wavelength. While there are no ideal diffuse reflectors in the real world, the Lambertian BRDF is nevertheless adequate for capturing the appearance of matte surfaces like carpet or soil. The appearance of π in the BRDF is a consequence of the energy conservation property of physically-plausible BRDFs given by Equation 2.28. Before diving into the proof, observe that changing the variable of integration from a solid angle to spherical coordinates yields

$$\int_\Omega d\omega = \int_0^{2\pi} \int_0^{\frac{\pi}{2}} \sin\theta \, d\theta \, d\varphi \tag{2.30}$$



(a) Absolute Spherical Coordinates **(b)** Differential Spherical Coordinates

Figure 2.6: Sketch of the zenith angle θ and azimuth angle φ which constitute the spherical coordinate system, along with an intuition for Equation 2.30.

Now, replacing $f_r(\boldsymbol{n}, \boldsymbol{o})$ with a normalization constant C in Equation 2.28 and applying Equation 2.30 gives

$$\int_\Omega f_r(\boldsymbol{n}, \boldsymbol{o})(\boldsymbol{n} \cdot \boldsymbol{o})\, d\omega_o \leq 1 \tag{2.31}$$

$$\int_0^{2\pi} \int_0^{\frac{\pi}{2}} C \cos\theta \sin\theta \, d\theta \, d\phi \leq 1 \tag{2.32}$$

$$2\pi C \int_0^{\frac{\pi}{2}} \cos\theta \sin\theta \, d\theta \leq 1 \tag{2.33}$$

$$\pi C \int_0^{\frac{\pi}{2}} \sin(2\theta) \, d\theta \leq 1 \tag{2.34}$$

$$\pi C[0 - (-1)] \leq 1 \tag{2.35}$$

$$C \leq 1/\pi \ . \tag{2.36}$$

Hence, the role of π in Equation 2.29 is to scale the albedo $\rho_d \in [0, 1]$ to ensure it is always the case that $f_r(\boldsymbol{i}, \boldsymbol{o}) \in [0, 1/\pi]$.

Blinn-Phong BRDF

Another early BRDF model is the Blinn-Phong BRDF [Bli77] which takes a similar form to Figure 2.5c. In contrast to the Lambertian model, the Blinn-Phong BRDF approximates the appearance of shiny surfaces like smooth plastic or metal. The definition of the Blinn-Phong BRDF is usually expressed in terms of the Rusinkiewicz coordinate system [Rus98] displayed in Figure 2.7. In this system, the incoming direction $\boldsymbol{i}$ and outgoing direction $\boldsymbol{o}$ are used to derive a *halfway vector* $\boldsymbol{h}$ where

$$\boldsymbol{h} = \frac{\boldsymbol{i} + \boldsymbol{o}}{\|\boldsymbol{i} + \boldsymbol{o}\|} \ . \tag{2.37}$$

Additionally, a *difference vector* $\boldsymbol{d}$ is defined as the position of $\boldsymbol{i}$ after rotating the sphere such that $\boldsymbol{h}$ is aligned with the original position of $\boldsymbol{n}$. One advantage of Rusinkiewicz coordinates is that isotropic BRDFs (i.e. those which are invariant to rotations about $\boldsymbol{n}$) can be naturally expressed in terms of three of the coordinates.

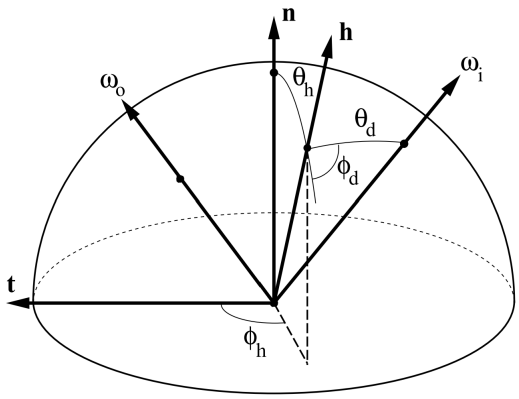


Figure 2.7: The Rusinkiewicz coordinate system is based on the halfway vector $\boldsymbol{h}$ and difference vector $\boldsymbol{d}$. Image is courtesy of [Rus98].

With this in mind, the unnormalized Blinn-Phong BRDF is given by

$$f_r(\boldsymbol{i}, \boldsymbol{o}) = \rho_d + \rho_s \frac{(\boldsymbol{n} \cdot \boldsymbol{h})^\gamma}{\boldsymbol{n} \cdot \boldsymbol{i}} \tag{2.38}$$

where ρ_d is the diffuse albedo, ρ_s is the specular albedo, and γ is the glossiness exponent which controls the sharpness of the specular highlight. Unfortunately, the $(\boldsymbol{n} \cdot \boldsymbol{i})$ factor in the specular term denominator prevents the BRDF from satisfying Helmholtz reciprocity (see Equation 2.24). For this reason, the *modified* Blinn-Phong BRDF [Lew94] is often used instead:

$$f_r(\boldsymbol{i}, \boldsymbol{o}) = \frac{\rho_d}{\pi} + \rho_s \frac{\gamma + 8}{8\pi} (\boldsymbol{n} \cdot \boldsymbol{h})^\gamma \ . \tag{2.39}$$

Observe that Equation 2.39 is a physically-plausible BRDF if the sum of ρ_d and ρ_s is constrained to fall in the range $[0, 1]$. Interestingly, the $\frac{\gamma+8}{8\pi}$ normalization factor is a consequence of Equation 2.28 but is not a tight bound; however, it is still favoured in many rendering applications due to its computational efficiency.

2.1.5 Microfacet Framework

The weak generalizability of both the Lambertian and Blinn-Phong BRDF models stems from their idealized assumptions about surface reflectance. The *microfacet framework* [CT82] partially overcomes this limitation by implicitly modelling the microscopic geometry of a surface. Specifically, the framework models a surface as a composition of a macrosurface and a microsurface consisting of flat *microfacets*.

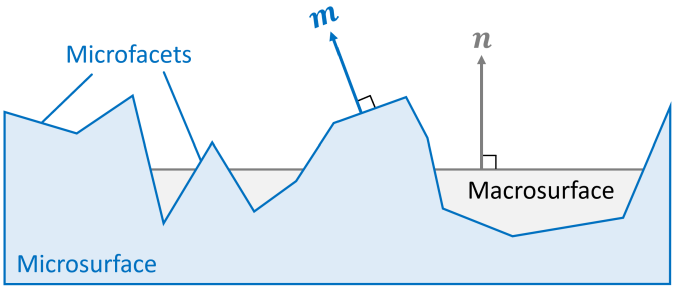


Figure 2.8: Visualization of the surface geometry in the microfacet framework.

To avoid the computational burden associated with modelling each microfacet explicitly, the microfacet framework uses a normal distribution function (NDF), designated by D, and a shadow-masking function (SMF), designated by G, to describe the detailed surface geometry. Briefly put, $D(\boldsymbol{m})$ gives the density of microfacet normals oriented in the direction of $\boldsymbol{m}$ while $G(\boldsymbol{i}, \boldsymbol{m}, \boldsymbol{o})$ gives the fraction of microfacets with normals oriented towards $\boldsymbol{m}$ which are visible from the incident light direction $\boldsymbol{i}$ and exitant light direction $\boldsymbol{o}$. These functions are further discussed later in this section.

Let f_n denote the macrosurface BRDF and f_m the microfacet BRDF. In the following derivation of f_n, any term that takes a microfacet direction $\boldsymbol{m}$ as input (e.g. $dA(\boldsymbol{m})$) pertains only to the microsurface with the corresponding normal direction. To begin, observe that from Equation 2.11 and Equation 2.12,

$$f_n(\boldsymbol{i}, \boldsymbol{o}) = \frac{1}{E(\boldsymbol{i})} \frac{d^2\Phi}{d\omega_o \, (\boldsymbol{n} \cdot \boldsymbol{o}) \, dA} \, . \tag{2.40}$$

Given that the flux radiated from the macrosurface is the sum of the flux radiated from the visible microsurface, applying Equation 2.11 and the definition of G yields

$$d^2\Phi = \int_\Omega L_o(\boldsymbol{o}, \boldsymbol{m}) [d\omega_o \, (\boldsymbol{m} \cdot \boldsymbol{o}) \, dA(\boldsymbol{m})] G(\boldsymbol{i}, \boldsymbol{m}, \boldsymbol{o}) \, d\omega_m \, . \tag{2.41}$$

Substituting Equation 2.41 into Equation 2.40 and simplifying the result gives

$$f_n(\boldsymbol{i}, \boldsymbol{o}) = \frac{1}{E(\boldsymbol{i})} \int_\Omega \frac{\boldsymbol{m} \cdot \boldsymbol{o}}{\boldsymbol{n} \cdot \boldsymbol{o}} L_o(\boldsymbol{o}, \boldsymbol{m}) \frac{dA(\boldsymbol{m})}{dA} G(\boldsymbol{i}, \boldsymbol{m}, \boldsymbol{o})\, d\omega_m \ . \tag{2.42}$$

By definition $D(\boldsymbol{m}) = dA(\boldsymbol{m})/dA$ and so

$$f_n(\boldsymbol{i}, \boldsymbol{o}) = \frac{1}{E(\boldsymbol{i})} \int_\Omega \frac{\boldsymbol{m} \cdot \boldsymbol{o}}{\boldsymbol{n} \cdot \boldsymbol{o}} L_o(\boldsymbol{o}, \boldsymbol{m}) D(\boldsymbol{m})\, G(\boldsymbol{i}, \boldsymbol{m}, \boldsymbol{o})\, d\omega_m \ . \tag{2.43}$$

Now, applying Equation 2.16 to the microfacet radiance term $L_o(\boldsymbol{i}, \boldsymbol{m}, \boldsymbol{o})$ yields

$$f_n(\boldsymbol{i}, \boldsymbol{o}) = \int_\Omega \frac{\boldsymbol{m} \cdot \boldsymbol{o}}{\boldsymbol{n} \cdot \boldsymbol{o}} \frac{E(\boldsymbol{i}, \boldsymbol{m})}{E(\boldsymbol{i})} f_m(\boldsymbol{i}, \boldsymbol{m}, \boldsymbol{o}) D(\boldsymbol{m})\, G(\boldsymbol{i}, \boldsymbol{m}, \boldsymbol{o})\, d\omega_m \ . \tag{2.44}$$

Lastly, expanding and cancelling the irradiance terms $E(\boldsymbol{i})$ and $E(\boldsymbol{i}, \boldsymbol{m})$ gives

$$f_n(\boldsymbol{i}, \boldsymbol{o}) = \int_\Omega \frac{\boldsymbol{m} \cdot \boldsymbol{i}}{\boldsymbol{n} \cdot \boldsymbol{i}} \frac{\boldsymbol{m} \cdot \boldsymbol{o}}{\boldsymbol{n} \cdot \boldsymbol{o}} f_m(\boldsymbol{i}, \boldsymbol{m}, \boldsymbol{o}) D(\boldsymbol{m})\, G(\boldsymbol{i}, \boldsymbol{m}, \boldsymbol{o})\, d\omega_m \ . \tag{2.45}$$

Note that the final microfacet BRDF also includes a Fresnel correction term F:

$$f_n(\boldsymbol{i}, \boldsymbol{o}) = \int_\Omega \frac{\boldsymbol{m} \cdot \boldsymbol{i}}{\boldsymbol{n} \cdot \boldsymbol{i}} \frac{\boldsymbol{m} \cdot \boldsymbol{o}}{\boldsymbol{n} \cdot \boldsymbol{o}} f_m(\boldsymbol{i}, \boldsymbol{m}, \boldsymbol{o}) F(\boldsymbol{m}, \boldsymbol{i}) D(\boldsymbol{m})\, G(\boldsymbol{i}, \boldsymbol{m}, \boldsymbol{o})\, d\omega_m \ . \tag{2.46}$$

Specular Microfacet BRDF

Although Equation 2.46 is theoretically sound, the integration is problematic since there is no guarantee that an arbitrary triplet of D, F, and G functions can be integrated analytically (not to mention the microfacet BRDF). Fortunately, a clean solution exists: choosing the microsurface BRDF to be perfectly specular implies that the only microfacets that reflect light towards ω_o from ω_i are those microfacets with a normal $\boldsymbol{m}$ that lies in the direction of the halfway vector $\boldsymbol{h}$:

$$f_m(\boldsymbol{i}, \boldsymbol{m}, \boldsymbol{o}) = \delta(\boldsymbol{m} - \boldsymbol{h}), \text{ where } \boldsymbol{h} = \frac{\boldsymbol{i} + \boldsymbol{o}}{\|\boldsymbol{i} + \boldsymbol{o}\|} \ . \tag{2.47}$$

Similar to the situation with the Lambertian and Blinn-Phong BRDFs, f_m must conserve energy to be plausible in a physical sense. For any fixed microfacet normal direction $\boldsymbol{m}$, define a normalization constant C where

$$f_{\boldsymbol{m}}(\boldsymbol{i}, \boldsymbol{o}) = C \delta_{\boldsymbol{m}}(\boldsymbol{h}) = C\, \delta(\boldsymbol{m} - \boldsymbol{h}) \ . \tag{2.48}$$

Before proceeding with an examination of Equation 2.28, it is instructive to compute the solid angle ratio $\frac{\omega_o}{\omega_h}$. This can be done by plotting $\boldsymbol{i}$, ω_o, and ω_h on a unit sphere and then rotating the sphere such that $\boldsymbol{i}$ is aligned with the north pole.

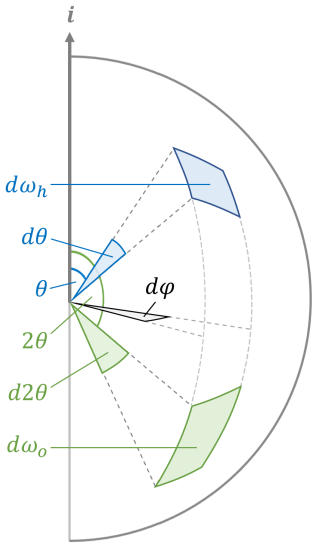


Figure 2.9: Solid angles of the halfway direction $\boldsymbol{h}$ and outbound direction $\boldsymbol{o}$.

Now, suppose that $\boldsymbol{h}$ is located at spherical coordinates (θ, φ) and, consequently, $\boldsymbol{o}$ is at $(2\theta, \varphi)$. Then,

$$d\omega_o = \sin(2\theta)\, d(2\theta)\, d\varphi = 4\cos\theta \sin\theta\, d\theta\, d\varphi \ . \tag{2.49}$$

It follows that

$$\frac{\omega_o}{\omega_h} = \frac{4\cos\theta \sin\theta\, d\theta\, d\varphi}{\sin\theta\, d\theta\, d\phi} = 4(\boldsymbol{h} \cdot \boldsymbol{i}) \ . \tag{2.50}$$

With that, substituting Equation 2.48 and Equation 2.50 into Equation 2.28 gives

$$\int_\Omega f_{\boldsymbol{m}}(\boldsymbol{i}, \boldsymbol{o})\, (\boldsymbol{m} \cdot \boldsymbol{o})\, d\omega_o \leq 1 \tag{2.51}$$

$$\int_\Omega f_{\boldsymbol{m}}(\boldsymbol{i}, \boldsymbol{o})\, (\boldsymbol{m} \cdot \boldsymbol{o})\, \frac{d\omega_o}{d\omega_m}\, d\omega_m \leq 1 \tag{2.52}$$

$$\int_\Omega C\, \delta_{\boldsymbol{m}}(\boldsymbol{h})\, (\boldsymbol{m} \cdot \boldsymbol{o})\, 4(\boldsymbol{m} \cdot \boldsymbol{i})\, d\omega_m \leq 1 \tag{2.53}$$

$$C \cdot 4\, (\boldsymbol{m} \cdot \boldsymbol{o})\, (\boldsymbol{m} \cdot \boldsymbol{i}) \leq 1 \tag{2.54}$$

$$C \leq \frac{1}{4\, (\boldsymbol{m} \cdot \boldsymbol{o})\, (\boldsymbol{m} \cdot \boldsymbol{i})} \ . \tag{2.55}$$

As a result, the normalized microsurface BRDF is

$$f_m(\boldsymbol{i}, \boldsymbol{m}, \boldsymbol{o}) = \frac{\delta(\boldsymbol{m} - \boldsymbol{h})}{4\,(\boldsymbol{m} \cdot \boldsymbol{o})\,(\boldsymbol{m} \cdot \boldsymbol{i})}\ . \tag{2.56}$$

Recall that the purpose of using a specular microsurface BRDF is to manage the integral in Equation 2.46. Plugging Equation 2.56 into Equation 2.46,

$$f_n(\boldsymbol{i}, \boldsymbol{o}) = \frac{F(\boldsymbol{i}, \boldsymbol{h}, \boldsymbol{o}) D(\boldsymbol{h})\, G(\boldsymbol{i}, \boldsymbol{h}, \boldsymbol{o})}{4(\boldsymbol{n} \cdot \boldsymbol{i})(\boldsymbol{n} \cdot \boldsymbol{o})}\ . \tag{2.57}$$

Normal Distribution Functions

The NDF D models the distribution of microfacet normals over a microsurface. Statistically, the total microsurface area occupied by microfacets with normals oriented in the direction of $\boldsymbol{m}$ is given by $D(\boldsymbol{m})\,\omega_m\,dA$ where dA is the area of the macrosurface corresponding to the microsurface. This means that setting the NDF to $D(\boldsymbol{m}) = \delta(\boldsymbol{m} - \boldsymbol{n})$ would represent a completely smooth surface. As with BRDFs, there are several properties that an NDF must support to be plausible:

1. **Range**: The range of the NDF must be a subset of $\mathbb{R}^+$:

$$\forall \boldsymbol{m} \in \Omega :\ D(\boldsymbol{m}) \geq 0\ . \tag{2.58}$$

2. **Projected Area**: The projected area of the microsurface modelled by the NDF must equal the projected area of the corresponding macrosurface:

$$\forall \boldsymbol{v} \in \Omega :\ \int_\Omega D(\boldsymbol{m})\,(\boldsymbol{m} \cdot \boldsymbol{v})\,d\omega_m = \boldsymbol{n} \cdot \boldsymbol{v}\ . \tag{2.59}$$

The most popular NDF in modern PBR applications is the GGX[4] distribution [WMLT07]. The isotropic GGX distribution is an instantiation of the generalized Trowbridge-Reitz (GTR) distribution [BS12] with $\gamma = 2$ where

$$D_{\mathrm{GTR}}(\boldsymbol{m}) = \frac{(\gamma - 1)(\alpha^2 - 1)}{\pi(1 - (\alpha^2)^{1-\gamma})(1 + (\alpha^2 - 1)(\boldsymbol{m} \cdot \boldsymbol{h})^2)^\gamma}\ . \tag{2.60}$$

[4]The origin of this acronym is not specified in the original paper.

Here, α is an isotropic roughness parameter which is shared across all microfacet normal directions. The isotropic GGX distribution is effectively a clamped eclipse [HDCD15] and can be defined as follows:

$$D_{\text{GGX,Isotropic}}(\boldsymbol{m}) = \frac{\alpha^2}{\pi(1 + (\alpha^2 - 1)(\boldsymbol{m} \cdot \boldsymbol{h})^2)^2} \ . \tag{2.61}$$

The anisotropic GGX distribution [BS12] extends the isotropic GGX distribution by replacing the roughness parameter α with a pair of roughness parameters, α_x and α_y, for two orthogonal directions $\boldsymbol{x}$ and $\boldsymbol{y}$ that lie tangent to a surface. The absolute orientations of $\boldsymbol{x}$ and $\boldsymbol{y}$ are controlled by an anisotropy parameter φ where

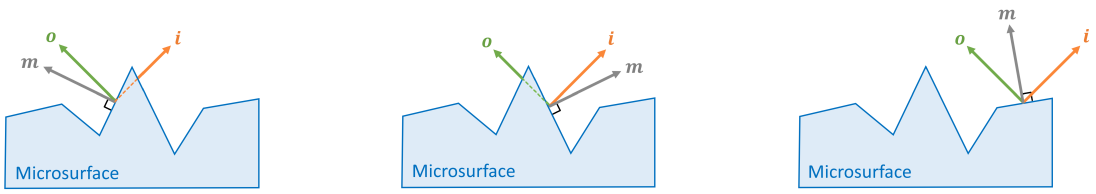
$$\frac{1}{\alpha^2} = \frac{\cos^2 \varphi}{\alpha_x^2} + \frac{\sin^2 \varphi}{\alpha_y^2}. \tag{2.62}$$

Bearing this in mind, the anisotropic GGX distribution is given by

$$D_{\text{GGX,Anisotropic}}(\boldsymbol{m}) = \frac{1}{\pi \alpha_x \alpha_y ((\boldsymbol{h} \cdot \boldsymbol{x})^2 / \alpha_x^2 + (\boldsymbol{h} \cdot \boldsymbol{y})^2 / \alpha_y^2 + (\boldsymbol{h} \cdot \boldsymbol{m})^2)^2} \ . \tag{2.63}$$

Shadow-Masking Function

The SMF G models the geometric occlusion of the microsurface. Ideally, G should account for both *shadowing*, where the path from the incoming radiance to a microfacet is blocked by another fragment of the microsurface, and *masking*, where the path from a microfacet to the outbound radiance destination is similarly blocked.



(a) Shadowed Microfacet **(b)** Masked Microfacet **(c)** Visible Microfacet

Figure 2.10: Microfacets may be hidden from the incoming radiance direction $\boldsymbol{i}$ or outgoing radiance direction $\boldsymbol{o}$, or they may be visible from both directions.

The influence of G becomes especially noticeable as the incident or exitant radiance directions approach grazing angles. A physically-plausible G function must satisfy the following properties:

1. **Range**: The range of the SMF must fall in the closed interval $[0, 1]$:

$$\forall \boldsymbol{i}, \boldsymbol{m}, \boldsymbol{o} \in \Omega : \; G(\boldsymbol{i}, \boldsymbol{m}, \boldsymbol{o}) \in [0, 1] \; . \tag{2.64}$$

2. **Symmetry**: The value of the SMF should not depend on the order of the incident and exitant radiance directions:

$$\forall \boldsymbol{i}, \boldsymbol{m}, \boldsymbol{o} \in \Omega : \; G(\boldsymbol{i}, \boldsymbol{m}, \boldsymbol{o}) = G(\boldsymbol{o}, \boldsymbol{m}, \boldsymbol{i}) \; . \tag{2.65}$$

3. **Sidedness**: The support of the SMF must not include microfacets with normals that are oriented away from the macrosurface:

$$\begin{aligned} \forall \boldsymbol{i}, \boldsymbol{m}, \boldsymbol{n}, \boldsymbol{o} \in \Omega : [(\boldsymbol{i} \cdot \boldsymbol{m})(\boldsymbol{i} \cdot \boldsymbol{n}) \leq 0] \vee \\ [(\boldsymbol{o} \cdot \boldsymbol{m})(\boldsymbol{o} \cdot \boldsymbol{n}) \leq 0] \rightarrow G(\boldsymbol{i}, \boldsymbol{m}, \boldsymbol{o}) = 0 \; . \end{aligned} \tag{2.66}$$

A common approach to deriving a suitable G is to use the Smith shadow-masking approximation [Smi67] which expresses the bidirectional shadow-masking function G_2 as the product of two evaluations of a unidirectional shadow-masking function G_1:

$$G(\boldsymbol{i}, \boldsymbol{m}, \boldsymbol{o}) = G_2(\boldsymbol{i}, \boldsymbol{m}, \boldsymbol{o}) \approx G_1(\boldsymbol{m}, \boldsymbol{i}) \, G_1(\boldsymbol{m}, \boldsymbol{o}) \; . \tag{2.67}$$

In this formulation, G_1 is calculated directly from an NDF D with the assumption that the height and orientation of the microsurface are independent. Now, let $\mathbb{I}$ denote the indicator function. The G_1 function for the isotropic GGX distribution is

$$G_{1,\mathrm{GGX}}(\boldsymbol{v}, \boldsymbol{m}) = \mathbb{I}\Big(\frac{\boldsymbol{v} \cdot \boldsymbol{m}}{\boldsymbol{v} \cdot \boldsymbol{n}} > 0\Big) \frac{2}{1 + \sqrt{1 + \alpha^2 \tan^2(\cos^{-1}(\boldsymbol{v} \cdot \boldsymbol{n}))}} \; . \tag{2.68}$$

Similarly, the G_1 function for the anisotropic GGX distribution is the same as Equation 2.68 except

$$\alpha = \alpha_x^2 \cos^2 \varphi + \alpha_y^2 \sin^2 \varphi \; . \tag{2.69}$$

Fresnel Function

The Fresnel term F models the fraction of exitant light intensity that is lost due to specular transmission at the microsurface. The exact proportion of scattered light that is reflected or refracted is governed by the Fresnel equations detailed in Section 2.1.2; however, the Fresnel equations are rarely implemented in PBR application due to their computational burden and awkward parameterization[5]. For these reasons, the Schlick Fresnel approximation [Sch94] is favoured instead:

$$F_{\text{Schlick}}(\boldsymbol{i}, \boldsymbol{m}) = F_0 + (1 - F_0)(1 - (\boldsymbol{m} \cdot \boldsymbol{i}))^5 \ . \tag{2.70}$$

Above, F_0 represents the reflectivity of the material when $\boldsymbol{i} = \boldsymbol{m}$. A spherical Gaussian approximation [Lag12] to the Schlick Fresnel approximation can also be employed to optimize certain rendering pipelines:

$$F_{\text{Schlick,SG}}(\boldsymbol{i}, \boldsymbol{m}) = F_0 + (1 - F_0)\, 2^{(-5.55473(\boldsymbol{m}\cdot\boldsymbol{i}) - 6.98316)(\boldsymbol{m}\cdot\boldsymbol{i})} \ . \tag{2.71}$$

2.1.6 Illumination

Conceptually, the goal of an illumination algorithm is to simulate light transport. This boils down to finding a solution to the rendering equation that faithfully captures the reflectance, emission, and occlusion of light between the various surfaces in a scene. Notably, there are two classes of illumination algorithms: direct illumination algorithms and global illumination algorithms. Algorithms that belong to the former category are relatively inexpensive to compute but are limited in their capacity to model light paths that are the product of multiple scattering events. Consequently, rendering applications that use direct lighting often resort to non-physical models of light transport to capture second-order lighting effects like ambient occlusion. On the other hand, global illumination algorithms offer efficient approximations of the integral in Equation 2.21 that eventually converge to solutions which successfully model subtle visual phenomena. These algorithms serve a vital role in photorealistic rendering and form one of the main pillars of PBR.

[5] Artists generally do not have a strong intuition for indices of refraction.

(a) Direct Illumination **(b)** Global Illumination

Figure 2.11: Direct illumination only light surfaces that are directly visible to light sources; however, global illumination can reproduce indirect lighting effects. Images are courtesy of [Rei12].

Direct Illumination

Regardless of how the pixels in an image are assigned to points on a surface, direct illumination algorithms operate by evaluating the RE for every point of interest from the direction of each light source into the direction of the virtual camera and accumulating the result. To determine whether a surface point is visible to a particular light source, a *shadow ray* is cast from the surface point in the direction of the light: if the shadow ray collides with an object in the scene before reaching the intended light source, then the surface point does not receive any radiance contribution from the light source. A Monte Carlo extension of this method can be employed to handle light sources that occupy entire surfaces by estimating the percentage of the light source that is visible to a surface point.

Sometimes, it is desirable to render a scene under environmental illumination where the radiance contributed to each surface point is independent of its position. In this case, the light source is effectively an inverted sphere whose radiance values are typically specified using an environment map that is indexed using spherical coordinates. Importance sampling the incident light directions from the BRDF of each surface point is often the tool of choice to manage the inherent computational complexity associated with this approach.

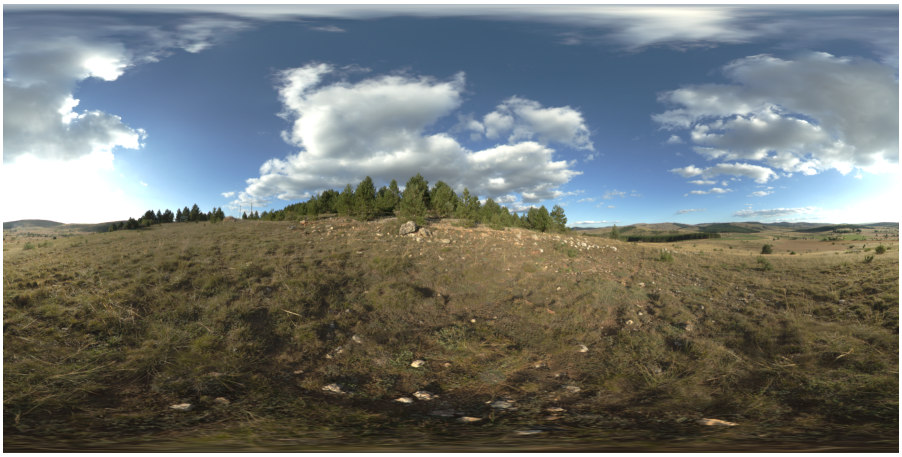


Figure 2.12: Environment maps specify lighting conditions that are shared between all objects in a scene. Image is courtesy of [Adob].

Radiosity

One strategy for tackling the global illumination problem draws inspiration from the physics literature and is called the Radiosity method [GTGB84] (not to be confused with the radiosity quantity B from Section 2.1.3). Radiosity is a finite-element method that approximates surfaces as finite compositions of geometric primitives in an attempt to discretize the integral in Equation 2.21. The original formulation of the Radiosity method assumes Lambertian reflectance so that $B = \pi L(\boldsymbol{v})$. To get started with the Radiosity method, the RE must be expressed in terms of radiosity (instead than radiance) and surface area (instead of solid angle). This amounts to

$$B(\boldsymbol{x})\,dA(\boldsymbol{x}) = E(\boldsymbol{x})\,dA(\boldsymbol{x}) + \rho_d(\boldsymbol{x})\,dA(\boldsymbol{x}) \int_{\mathcal{S}} G(\boldsymbol{x},\boldsymbol{y})\,V(\boldsymbol{x},\boldsymbol{y})\,B(\boldsymbol{y})\,dA(\boldsymbol{y}) \ . \quad (2.72)$$

Observe that the dependence of each term on the points $\boldsymbol{x}$ and $\boldsymbol{y}$ is explicitly stated. In Equation 2.72, $B(\boldsymbol{x})$ is the total radiosity emitted from the differential area $dA(\boldsymbol{x})$ around $\boldsymbol{x}$, $E(\boldsymbol{x})$ is the emitted radiosity from $dA(\boldsymbol{x})$, $\rho_d(\boldsymbol{x})$ is the diffuse albedo at $\boldsymbol{x}$, $\mathcal{S}$ is the surface geometry of the entire scene, $G(\boldsymbol{x},\boldsymbol{y})$ is the geometric attenuation between the incident and exitant radiance at $\boldsymbol{x}$ and $\boldsymbol{y}$, and $V(\boldsymbol{x},\boldsymbol{y})$ is the fraction of $dA(\boldsymbol{y})$ that is visible to $dA(\boldsymbol{x})$. Now, let $N(\boldsymbol{x})$ be the surface normal at $\boldsymbol{x}$. The value of G is given by

$$G(\boldsymbol{x},\boldsymbol{y}) = \frac{1}{\pi \left\|\boldsymbol{x}-\boldsymbol{y}\right\|_2^2} \left|N(\boldsymbol{x})\cdot(\boldsymbol{y}-\boldsymbol{x})\right| \left|N(\boldsymbol{y})\cdot(\boldsymbol{x}-\boldsymbol{y})\right| \ . \quad (2.73)$$

If the differential areas are expanded to an appreciable size, then the integration in Equation 2.72 can be transformed into the following summation:

$$B(\boldsymbol{x}) = E(\boldsymbol{x}) + \rho_d(\boldsymbol{x}) \sum_{\mathcal{S}} G(\boldsymbol{x}, \boldsymbol{y}) V(\boldsymbol{x}, \boldsymbol{y})\, B(\boldsymbol{y}) \ . \tag{2.74}$$

Hence, the radiosity emitted from each discretized surface in a scene can be computed by solving a system of linear equations. At first glance, it may appear that solving for $\boldsymbol{B}$ exactly is always the right move; however, in practice, the number of variables of this system necessitates the use of numerical methods with lower computational complexity. The final step in the Radiosity algorithm is to smoothly interpolate the computed radiosity values across the surfaces in the scene.

Bidirectional Path Tracing

A popular alternative to the Radiosity method is *bidirectional path tracing* (BDPT) [LW93]. This global illumination algorithm is compatible with all physically-plausible BRDFs (provided they have an efficient sampling procedure) and can be trivially extended to model participating media and subsurface scattering behaviour. The basic premise of BDPT is to leverage importance sampling to estimate the value of the RE by tracing the path of a *light ray* and an *eye ray* through the scene and computing *shadow rays* between the collisions of these rays. The initial direction of the light ray is selected with a probability that is proportional to its radiosity contribution while the initial direction of the eye ray is computed directly from the positions of the virtual camera and target pixel in the image plane. The direction of a ray following each collision is chosen through an importance sampling procedure where each direction is weighted by its radiosity contribution in the BRDF. The path of a light or eye ray terminates during a collision if a sample drawn from the uniform distribution between 0 and 1 exceeds the integrated probability of the cosine-weighted BRDF. This termination rule, called Russian Roulette, is often used in conjunction with another rule that indiscriminately terminates a light or eye path after a certain number of bounces.

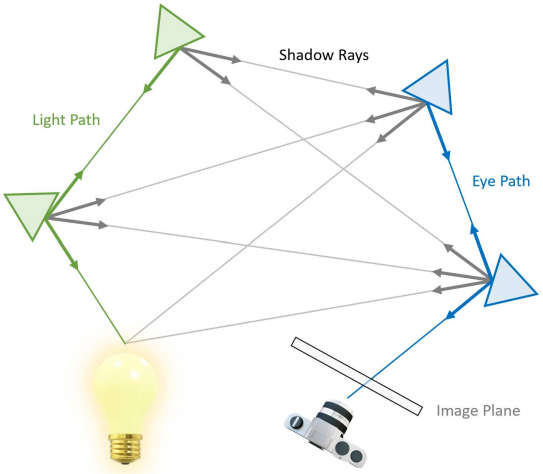


Figure 2.13: Bidirectional path tracing accumulates contributions from the shadow rays induced between the collisions of a light path and eye path.

The radiance contribution of each shadow ray is the product of a geometric attenuation factor and two BRDF evaluations at the light and eye path collisions. These contributions are weighted in accordance with a flux conservation condition to produce one radiance sample for a pixel. As the number of radiance samples grow, the average radiance value converges to an exact solution of Equation 2.21. For this reason, BDPT is an industry standard for photorealistic rendering.

Progressive Photon Mapping

Any comprehensive overview of global illumination algorithms would not be complete without mention of progressive photon mapping (PPM) [HOJ08]. In a nutshell, PPM uses a pair of *photon maps* to accelerate the rendering of independent RE components using a combination of ray casting and ray tracing techniques[6]. To begin, a fixed number of photons are emitted from a light source and traced around the scene using the same importance sampling procedure featured in BDPT. When a photon collides with a surface, its position, direction, and intensity are recorded in a tree-like data structure called a photon map. As before, the absorption of a photon is determined by the Russian Roulette criterion. When a photon is reflected off a surface, a corresponding *shadow photon* is created and emitted in

[6]Ray casting is a form of ray tracing where rays are always absorbed upon collision.

the original direction of the incoming photon to aid in the importance sampling of direct illumination shadows. In this way, a caustic photon map and global photon map are constructed. The difference between these maps is that the caustic photon map ignores diffuse BRDF components, omits the casting of shadow photons, and only populates photons at their point of absorption. From here, the emissive and direct illumination components are rendered directly with ray casting, the specular and diffuse indirect illumination components are rendering using ray casting with the global and caustic photons maps respectively, and the remaining specular reflection component is rendered using ray tracing with both the global and caustic photon maps. The role of the photon maps is to aid importance sampling and approximate the incident radiance on a surface. Unfortunately, this approach may experience bias due to the limited number of photons that are initially released in the scene. To overcome this defect, the rendering and photon mapping stages are swapped such that the rendering pass simply determines the collision points of interest while the photon mapping pass iteratively refines the photon maps until the scene converges to the RE solution. A comparison of how PPM and BDPT (along with several other global illumination algorithms) perform in a scene with challenging lighting is pictured in Figure 2.14.

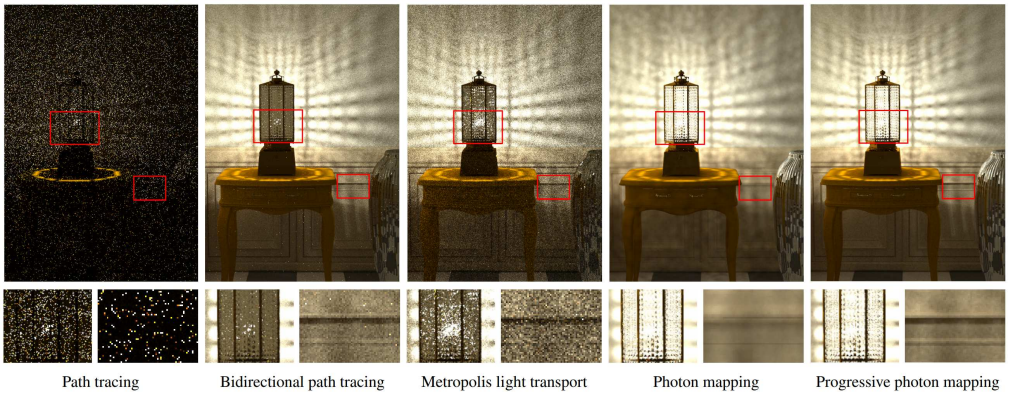


Figure 2.14: Comparison of the renderings produced by different global illumination algorithms after 22 hours of processing time. Image is courtesy of [HOJ08].

2.2 Machine Learning

2.2.1 Introduction

The overwhelming investment of resources into artificial intelligence research over the past decade can be attributed to the unparalleled success of machine learning. While there is no singular, universal definition of machine learning, the following passage from [Mit97] eloquently summarizes the essence of the field:

> A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P if its performance at tasks in T, as measured by P, improves with experience E.

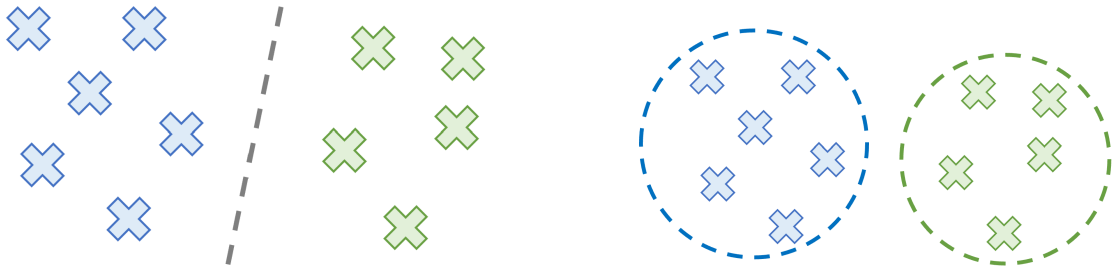
The vast majority of ML systems are composed of a parametric model and an algorithm which estimates the values (or distribution of values) of the model parameters that minimize some loss function $\mathcal{L}$ over a dataset $\mathcal{D}$. For example, consider the task of performing a linear regression over the points in a scatter plot. This is an instantiation of machine learning where the parametric model is a line of the form $f(x, a, b) = ax + b$ and the algorithm is one that calculates the slope a and intercept b which minimize the residual sum of squares $\mathcal{L}$ across the points $\mathcal{D}$ in the scatter plot. For the sake of mathematical convenience, ML algorithms that are designed to operate on parametric models are usually tasked with estimating the parameters of probabilistic models where the goal is to maximize the likelihood that some underlying generative process yields a given dataset. Other ML algorithms, such as principal component analysis or k-means clustering, operate on models whose behaviour is governed exclusively by hyperparameters.

Discriminative and Generative Models

Virtually all parametric ML models belong to one of two groups: discriminative models or generative models. Discriminative models are applicable to both regression and classification tasks[7] and model a dataset $\mathcal{D}$ as a conditional distribution of the form $p(\boldsymbol{y} \mid \boldsymbol{X}, \boldsymbol{\theta})$ where $\boldsymbol{\theta}$ is a vector of parameters. To clarify, a dataset $\mathcal{D}$ is usually composed of a design matrix $\boldsymbol{X} \in \mathbb{R}^{N \times D}$ consisting of N training examples

[7] Regression models predict real-valued quantities while classification models predict class labels.

and a label vector $\boldsymbol{y} \in \mathbb{R}^N$ indicating the target output of each training example. Generative models provide a means of sampling from the distribution of a dataset by modelling it as a joint distribution of the form $p(\boldsymbol{X}, \boldsymbol{y} \mid \boldsymbol{\theta})$. Both discriminative and generative models are viable candidates for classification tasks.



(a) Discriminative Model **(b)** Generative Model

Figure 2.15: Discriminative models try to predict outputs directly while generative models try to explain the data generation process.

Bayesian Machine Learning

Another distinguishing factor between ML approaches lies in their interpretation of probability and statistics. Machine learning systems based on the frequentist framework tend to view probabilities as asymptotic frequencies over large number of experiments and the parameters of a model as fixed constants that can be repeatedly sampled to form a dataset. On the other hand, Bayesian machine learning systems treat probabilities as degrees of belief and ultimately model the parameters of a model as probability distributions over a fixed dataset. In the latter setting, a *prior* distribution $p(\boldsymbol{\theta})$ is assumed over the parameters of a model which expresses a set of initial beliefs about the distributions of the parameter values. Bayes' theorem [Bay63] is then used to combine the prior with information gleaned from the dataset $\mathcal{D}$ to yield an informed *posterior* distribution $p(\boldsymbol{\theta} \mid \mathcal{D})$:

$$p(\boldsymbol{\theta} \mid \mathcal{D}) = \frac{p(\mathcal{D} \mid \boldsymbol{\theta})\, p(\boldsymbol{\theta})}{p(\mathcal{D})} \propto p(\mathcal{D} \mid \boldsymbol{\theta})\, p(\boldsymbol{\theta}) \ . \tag{2.75}$$

Crucially, the frequentist approach only gives a single estimate for $\boldsymbol{\theta}$ while the Bayesian approach yields a probability distribution over all possible values of $\boldsymbol{\theta}$. When a prediction on a new input is desired, the *maximum a posteriori* (MAP)

estimate (a mode of the posterior distribution) is often supplied to the model; the alternative is to integrate the model predictions over the entire domain of $\boldsymbol{\theta}$.

Supervised and Unsupervised Learning

The final distinction of note is that of supervised learning and unsupervised learning. In a supervised ML context, the dataset $\mathcal{D}$ is composed of a design matrix $\boldsymbol{X}$ and a label vector $\boldsymbol{y}$. This setup is especially appropriate for predictive models where precisely one correct output exists for each input to the model. The dataset in an unsupervised ML setting contains a design matrix $\boldsymbol{X}$ but no label vector. Consequently, unsupervised ML models are primarily used for inferring patterns or structures within a dataset that are useful for tasks like clustering or encoding inputs in low-dimensional spaces. More recently, a hybrid of supervised learning and unsupervised learning called semi-supervised learning has emerged where the label vector $\boldsymbol{y}$ is missing entries for some training examples.

2.2.2 Artificial Neural Networks

Artificial neural networks (ANNs) are a family of ML models that have gained immense popularity with advancements in GPU technology. The architectures of these models are heavily inspired by the neural circuitry found inside the human brain but are fundamentally based on a simple ML model called the *perceptron* [Ros58]. Perceptrons accept a vector $\boldsymbol{x} \in \mathbb{R}^D$ as input and produce a binary class $f(\boldsymbol{x}, \boldsymbol{w})$ as output using a weight vector $\boldsymbol{w} \in \mathbb{R}^D$:

$$f(\boldsymbol{x}, \boldsymbol{w}) = \begin{cases} 1 & \boldsymbol{w}^\intercal \boldsymbol{x} > 0 \\ 0 & \boldsymbol{w}^\intercal \boldsymbol{x} \leq 0 \end{cases} . \tag{2.76}$$

Note that a common notational trick is to fix $x_0 = 1$ so that w_0 encodes a bias term. Hence, a perception can be interpreted as a linear classifier which assigns binary labels based on which side of the hyperplane $H \equiv \boldsymbol{w}^\intercal \boldsymbol{x} = 0$ an input occupies. This means that a perceptron can only achieve a perfect classification score if all of the items in a dataset are linearly separable (i.e. can be separated by a hyperplane).

(a) Linearly Separable Dataset **(b)** Linearly Inseparable Dataset

Figure 2.16: It is impossible for a perceptron to perfectly classify data that cannot be separated by a hyperplane.

For a training example $(\boldsymbol{x}_i, y_i)$, the weights of a perceptron are updated as follows:

$$\boldsymbol{w}' = \boldsymbol{w} + \eta \cdot (y_i - f(\boldsymbol{x}_i, \boldsymbol{w}))\, \boldsymbol{x}_i \ . \tag{2.77}$$

Here, η denotes the *learning rate* and controls the magnitude of the influence of each training example over $\boldsymbol{w}$. Observe that the weight vector does not change when the perceptron correctly classifies an example; however, the hyperplane encoded by the weight vector is shifted and rotated to accommodate incorrect classifications.

Neurons

The defining feature of a neural network is the *neuron*. Neurons can be viewed as generalizations of perceptrons where the product of an input vector $\boldsymbol{x} \in \mathbb{R}^D$ and weight vector $\boldsymbol{w} \in \mathbb{R}^D$ is added to an explicit bias $b \in \mathbb{R}$ and then passed through an *activation function* $f : \mathbb{R} \to \mathbb{R}$.

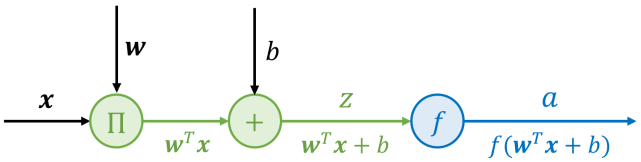


Figure 2.17: Block diagram of a neuron.

The input to f is the *preactivation* z and the output of f is the *activation* a.

$$z = \boldsymbol{w}^\intercal \boldsymbol{x} + b \tag{2.78}$$

$$a = f(z) \ . \tag{2.79}$$

Activation Functions

In practice, activation functions are non-linear functions that are differentiable across their domain. The non-linearity property allows chains of connected neurons to approximate highly non-linear functions while the differentiability property is a prerequisite for many optimization algorithms. Some common activation functions are described below and sketched in Figure 2.18.

- **Sigmoid**: The sigmoid function collapses inputs into the open interval $(0, 1)$:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \ . \tag{2.80}$$

  The output of a sigmoid activation is often used to represent a probability, as is the case in a *logistic regression*. Crucially, the derivative $\sigma'(x)$ of the sigmoid is relatively inexpensive to compute if $\sigma(x)$ is known:

$$\sigma'(x) = \frac{e^{-x}}{(1 + e^{-x})^2} = \frac{1}{1 + e^{-x}} \frac{e^{-x}}{1 + e^{-x}} = \sigma(x)(1 - \sigma(x)) \ . \tag{2.81}$$

- **Tanh**: The hyperbolic tangent collapses inputs into the open interval $(-1, 1)$:

$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \ . \tag{2.82}$$

  As depicted in Figure 2.18a and Figure 2.18b, tanh may be preferred over σ if a steeper derivative is desired. Note that the derivative $\tanh'(x)$ is similarly inexpensive to compute from $\tanh(x)$:

$$\tanh'(x) = \frac{(e^{x} + e^{-x})^2 - (e^{x} - e^{-x})^2}{(e^{x} + e^{-x})^2} = 1 - \tanh^2(x) \ . \tag{2.83}$$

- **ReLU**: The ReLU activation discards negative inputs and keeps positive ones:

$$\mathrm{ReLU}(x) = \max(0, x) \ . \tag{2.84}$$

  Two key advantages of the ReLU are its simplicity and computational efficiency. Also, unlike σ' and $\tanh'$, the derivative of ReLU is undefined at $x = 0$ and does not approach 0 as the magnitude of x increases towards infinity:

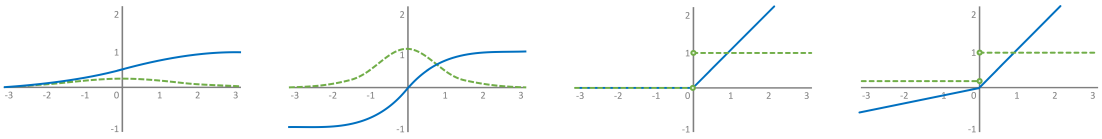
$$\mathrm{ReLU}'(x) = \begin{cases} 0 & x < 0 \\ 1 & x > 0 \end{cases} \ . \tag{2.85}$$

This partially mitigates the *vanishing gradient* problem where the derivative of a series of neurons rapidly declines to 0 due to the multiplication of small derivative values. A neuron is said to *saturate* when its weights and bias are predisposed to derivatives near zero since most gradient-based optimization methods will struggle to make significant changes to these parameters.

- **Leaky ReLU**: The leaky ReLU corrects a shortcoming of the ReLU function by introducing a small gradient (typically 0.01) to the negative domain:

$$\text{LeakyReLU}(x) = \begin{cases} 0.01x & x < 0 \\ x & x \geq 0 \end{cases} . \tag{2.86}$$

Leaky ReLU neurons do not saturate and thus avoid the *dying ReLU* problem.



(a) Sigmoid **(b)** Tanh **(c)** ReLU **(d)** Leaky ReLU

Figure 2.18: Activation functions (blue, solid) and their derivatives (green, dashed).

Feedforward Neural Networks

The most basic type of neural network is the feedforward neural network (FNN). An FNN is composed of an *input layer*, followed by zero or more *hidden layers* of neurons, and then a final *output layer* of neurons. The distinguishing property of an FNN is its acyclic computational graph which defines the flow of data through the network[8]. Typically, the neurons within a single layer of an FNN operate independently and only have direct connectivity to neurons in adjacent layers[9]. A notable exception to this first rule is the softmax layer where the preactivations of neurons within a layer are shared to form a probability distribution over several outputs. Figure 2.19 showcases an FNN with $L = 4$ layers.

[8]The most common type of ANN which is not an FNN is the recurrent neural network (RNN).
[9]Some FFN architectures have *skip connections* that connect neurons lying several layers apart.

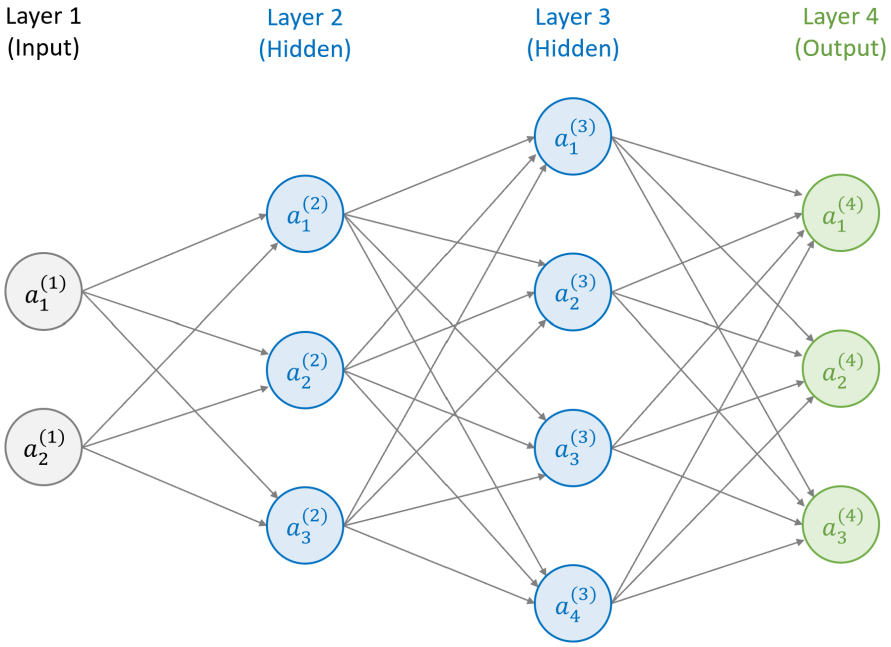


Figure 2.19: FNN with two fully-connected hidden layers and a fully-connected output layer. The symbol on each neuron denotes its activation; the activation of the input layer is just the input provided to the FNN.

A multi-layer perceptron (MLP) can be defined as an FNN consisting only of fully-connected layers. In other words, each neuron in an MLP receives input from every neuron in its previous layer. Let $\boldsymbol{w}_i^{(\ell)}$, $b_i^{(\ell)}$, $z_i^{(\ell)}$, and $a_i^{(\ell)}$ denote the weights, bias, preactivation, and activation of the i^{th} neuron in layer ℓ, respectively. Furthermore, let $f^{(\ell)}$ denote the activation function of the neurons in layer ℓ and let $n^{(\ell)}$ denote the number of neurons in layer ℓ. The following matrix notation is assumed:

$$\boldsymbol{W}^{(\ell)} = \begin{bmatrix} \left(\boldsymbol{w}_1^{(\ell)}\right)^\mathsf{T} \\ \vdots \\ \left(\boldsymbol{w}_{n^{(\ell)}}^{(\ell)}\right)^\mathsf{T} \end{bmatrix}, \quad \boldsymbol{b}^{(\ell)} = \begin{bmatrix} b_1^{(\ell)} \\ \vdots \\ b_{n^{(\ell)}}^{(\ell)} \end{bmatrix}, \quad \boldsymbol{z}^{(\ell)} = \begin{bmatrix} z_1^{(\ell)} \\ \vdots \\ z_{n^{(\ell)}}^{(\ell)} \end{bmatrix}, \quad \boldsymbol{a}^{(\ell)} = \begin{bmatrix} a_1^{(\ell)} \\ \vdots \\ a_{n^{(\ell)}}^{(\ell)} \end{bmatrix}. \tag{2.87}$$

Observe that the output $\boldsymbol{a}^{(L)}$ of an MLP for some input $\boldsymbol{x}$ can be derived from Equation 2.78 and Equation 2.79 as follows:

$$\boldsymbol{z}^{(\ell)} = \boldsymbol{W}^{(\ell)}\boldsymbol{a}^{(\ell-1)} + \boldsymbol{b}^{(\ell)} \tag{2.88}$$

$$\boldsymbol{a}^{(\ell)} = f^{(\ell)}(\boldsymbol{z}^{(\ell)}) \tag{2.89}$$

$$\boldsymbol{a}^{(1)} = \boldsymbol{x}\ . \tag{2.90}$$

Computing $\boldsymbol{a}^{(L)}$ from $\boldsymbol{x}$ and then applying a loss function $\mathcal{L}$ is a process known as *forward propagation*. Likewise, computing the gradient of $\mathcal{L}$ with respect to each parameter in the network is known as *backpropagation*. Backpropagation is simply a clever application of the chain rule that proceeds through the network in the reverse ordering of the layers. Given that the loss and activation functions are differentiable,

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{z}^{(L)}} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{a}^{(L)}} \cdot \frac{\partial \boldsymbol{a}^{(L)}}{\partial \boldsymbol{z}^{(L)}} \tag{2.91}$$

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{z}^{(\ell)}} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}^{(\ell+1)}} \cdot \boldsymbol{W}^{(\ell+1)} \cdot \frac{\partial \boldsymbol{a}^{(\ell)}}{\partial \boldsymbol{z}^{(\ell)}} \tag{2.92}$$

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}^{(\ell)}} = \left(\boldsymbol{a}^{(\ell-1)} \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}^{(\ell)}} \right)^{\mathsf{T}} \tag{2.93}$$

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{b}^{(\ell)}} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{a}^{(\ell)}} \ . \tag{2.94}$$

Note that the gradients in Equation 2.91 through Equation 2.94 are expressed as row vectors rather than column vectors for the sake of notational convenience. The overwhelming majority of ANN optimization algorithms on the market today rely on gradients to update ANN parameters and thus make heavy use of backpropagation.

2.2.3 Deep Learning

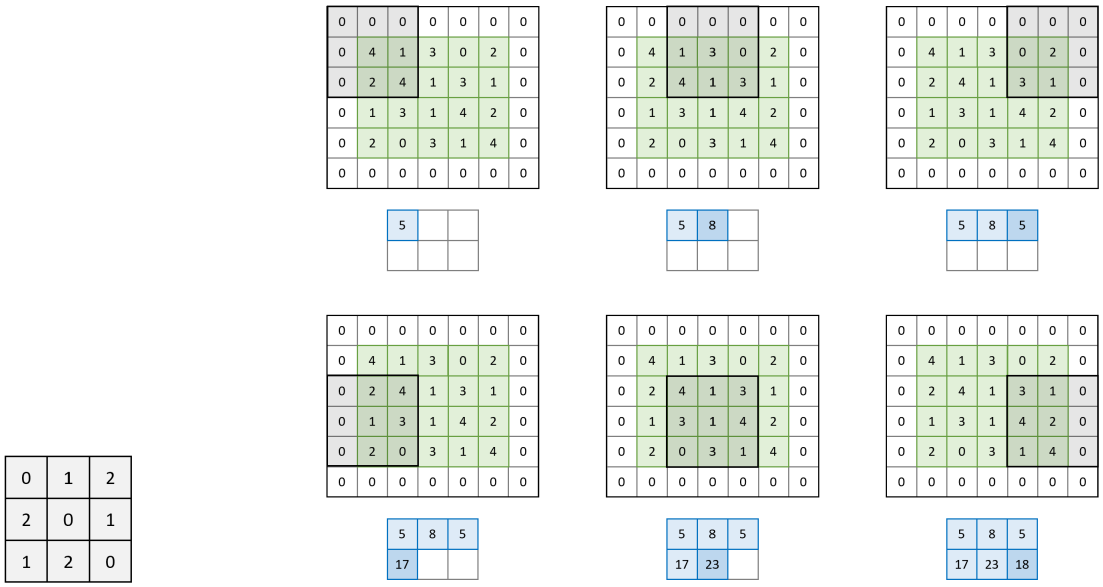
One of the key discoveries leading to the modern AI summer is *deep learning*, or more specifically, the advent of deep neural networks (DNNs). Informally, a DNN is an ANN with many layers. As the number of layers in DNN grows, so does the level of abstraction represented by features near the output layer of the DNN. Thus, DNNs are capable of learning rich feature hierarchies that are absent in their shallow counterparts. Of course, deep learning also has some drawbacks. For example, DNNs tend to have a large number of learnable parameters which slows down training and places a high demand on GPU memory. They are also more vulnerable to certain training pitfalls such as vanishing or exploding gradients. On the whole, the advantages and exciting promises of deep learning are widely considered to outweigh its training difficulties and lack of rigorous mathematical justification.

Convolutional Neural Networks

Convolutional neural networks (CNNs) are a subclass of DNNs that specialize in learning features of an input that are invariant under translation. CNNs are especially popular in computer vision applications where the goal is to detect some spatially invariant feature of an input image. Technically, a DNN is a CNN if it contains at least one convolutional layer. A 2D convolutional layer operates by sliding a 2D *kernel* over a set of input feature maps which collectively take the form a 3D tensor[10] $\mathbb{R}^{I_h \times I_w \times I_c}$. A 2D kernel is simply a matrix of size $k \times k$ representing one (broadcasted) operand of a dot product operation with an input tensor. The manner in which a kernel, also called a *filter*, is applied to an input tensor determines the spatial extent of the convolutional layer and is governed by a set of hyperparameters, namely the zero padding $p \in \mathbb{Z}^+$ and stride $s \in \mathbb{R}^+$. Note that convolutional layers typically have several filters, all of which share the same kernel hyperparameters. The output of each filter is concatenated to form an output tensor $\mathbb{R}^{O_h \times O_w \times O_c}$. An example of a convolution is pictured in Figure 2.20.

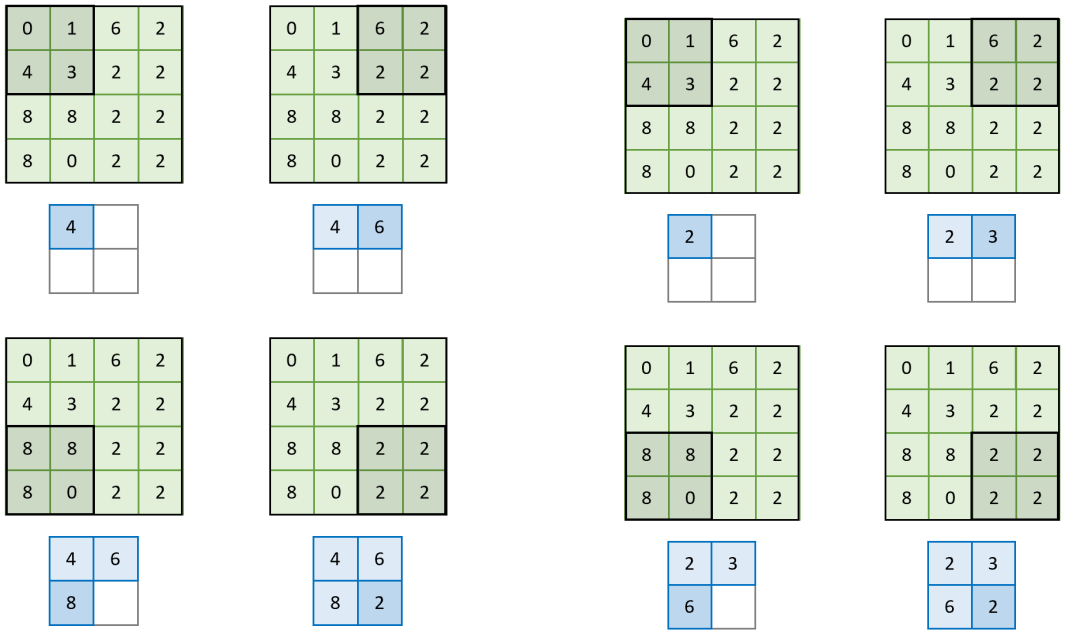
Usually, the stride of a convolution is chosen to be a strictly positive integer and represents the spatial downsampling factor between the input layer and the convolutional layer. Similarly, the zero padding applied to an input tensor is used to fine-tune the spatial extent of the convolutional layer but is otherwise undesirable. Another way to reduce the size of an input tensor is to apply a convolutional layer where $s = 1$ followed immediately by a *pooling layer*. Pooling layers can be viewed as convolutional layers where the kernel is replaced with an arbitrary summary function, such as one that returns the average value of a collection of numbers. A demonstration of two common pooling layers is displayed in Figure 2.22.

[10] This is only the usual case but will be assumed for the remainder of the discussion. Furthermore, the kernel size k, zero padding p, and stride s could vary by axis in the general case, but this fact will also be ignored to avoid cluttering the notation.



(a) Kernel **(b)** Convolution

Figure 2.20: Example of a convolution where a single kernel of size $k = 3$ is applied with padding $p = 1$ and stride $s = 2$ to an input of shape $\{I_h = 4, I_w = 5, I_c = 1\}$, resulting in an output of shape $\{O_h = 2, O_w = 3, O_c = 1\}$. Some convolutional layers implement affine transformations by associating a learnable bias $b \in \mathbb{R}$ with each filter.



(a) Maximum Pooling **(b)** Average Pooling

Figure 2.22: Example of maximum pooling and average pooling where a 2D pooling operator of size $k = 2$ is applied with no padding (i.e. $p = 0$) and stride $s = 2$ to an input of shape $\{I_h = 4, I_w = 4, I_c = 1\}$, resulting in an output of shape $\{O_h = 2, O_w = 2, O_c = 1\}$.

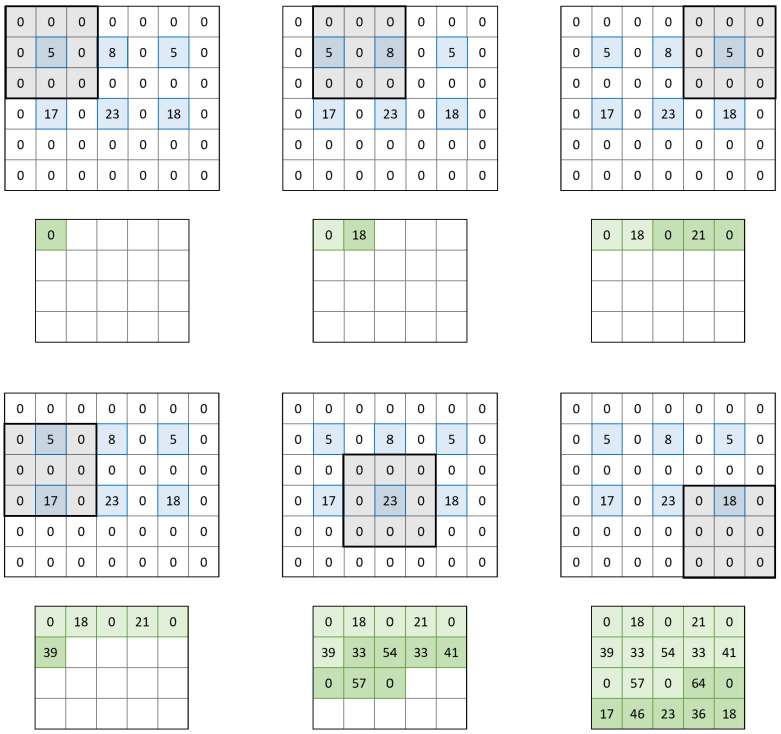


Figure 2.21: Example of a fractionally-strided convolution that inverts the spatial transformation performed in Figure 2.20. The kernel is the same as the one in Figure 2.20a although the weights need not match in the general case. Note that $s = \frac{1}{2}$.

Of course, there are also established methods for increasing the spatial extent of a convolutional layer relative to the input tensor (beyond the addition of trivial zero-padding). The simplest approach is to leverage a sampling technique like nearest-neighbour sampling or bilinear sampling to upscale the input tensor before applying a traditional convolution where $s = 1$. Alternatively, a fractionally-strided convolution[11] can be used to merge the upsampling and convolutional steps into a single operation. Ultimately, a fractionally-strided convolution with stride $s = \frac{1}{s'}$ seeks to invert the change in dimensionality caused by an ordinary convolution with stride s'. This is done by dilating the input tensor (i.e. inserting zero-padding between the elements of the tensor) and then appending the necessary padding to ensure that the *receptive field* of the fractionally-strided convolutional layer matches the *projective field* of a convolutional layer of reciprocal stride [DV16]. Here, the receptive field of a neuron is defined to be the region of the (padded) input tensor that

[11] Fractionally-strided convolutions are also called deconvolutions or transpose convolutions.

directly influences the computation of the value of that neuron; the projective field models the opposite relationship. An example of a fractionally-strided convolution which spatially deconvolves the convolution in Figure 2.20 is provided in Figure 2.21.

Latent Variable Models

Latent variable models (LVMs) are a subset of DNNs that try to learn the relationship between a set of observable variables and a set of latent variables. In the statistical sense of the word, a latent variable is any variable which is not directly observed but whose value can be inferred from one or more observable variables. The latent variables in a DNN are typically the activations of the neurons in the hidden layer with the smallest dimensionality.

The *autoencoder* is a classic example of a DNN with the spirit of an LVM. The goal of an autoencoder is to learn an invertible mapping from inputs to latent variables. To this end, autoencoders compress inputs into latent variables through an encoder DNN and then attempt to decompress these latents back into their original inputs through a decoder DNN. Note that this reconstruction process is likely to be imperfect by design: the latent variables are meant to encode meaningful features of the input rather than irrelevant, high-frequency content. One subclass of autoencoders that is worth drawing attention to is the variational autoencoder (VAE) [KW13]. A VAE is composed of an inference network and generator network which function as an encoder and decoder, respectively. Unlike traditional autoencoders, the output of a VAE encoder describes a latent variable distribution[12] which must be sampled before proceeding through to the decoder. Similarly, the VAE decoder also produces a distribution which must be sampled to obtain a concrete output. Trained with the evidence lower bound (ELBO) loss function, VAEs can produce latent variable distributions whose parameters can be linearly interpolated to represent the blending of latent variables from different inputs. As a result, they can smoothly morph one input into another and generate

[12] A popular choice for the latent variable distribution is the factorised Gaussian distribution.

novel outputs by randomly sampling from a latent variable prior. An overview of the VAE architecture is shown in Figure 2.23.

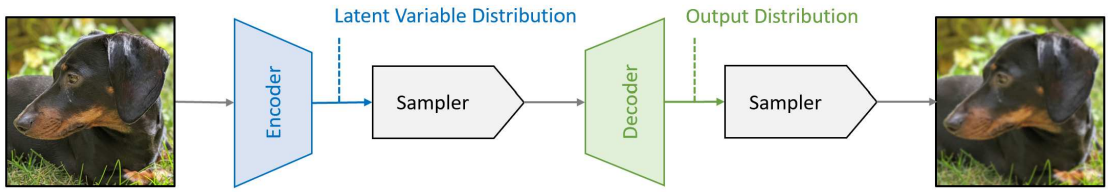


Figure 2.23: Overview of the VAE architecture. Observe that high-frequency details in the input image tend to be lost during the encoding of the latent variable distribution.

A different take on the autoencoder is the deep belief network (DBN), or equivalently, a stack of restricted Boltzmann machines (RBMs). Conceptually, an RBM is a shallow instance of a VAE with the same weights in the encoder and decoder networks. Practically, an RBM is an ANN with one visible layer and one bidirectional fully-connected hidden layer. Both layers are equipped with sigmoid activations and only take on binary values. When an RBM is supplied with an input, the activations of each neuron in the hidden layer are computed in the usual way and then replaced with a sample from a Bernoulli distribution. The parameter of each Bernoulli distribution is simply the activation of the corresponding neuron. The sampled latent variables are then processed symmetrically in the opposite direction to yield a binary output sample.

The final LVM of relevance is the generative adversarial network (GAN) [GPAM+14] which approaches generative modelling from a game theory perspective. Specifically, a GAN is composed of two competing DNNs: a generator and a discriminator. The aim of a discriminator network is to determine whether an input image is the product of the generator network or a genuine sample from a dataset; the aim of a generator network is to synthesize inputs for the discriminator network which are believed to come from the dataset. The input to the generator network itself is a uniform random sample from a latent space. Although both DNNs are vital to the training process, the discriminator network is usually discarded once training is over since all of the generative behaviour is encapsulated in the generator network. Despite their

apparent simplicity, GANs are notoriously difficult to train in large part due to *mode collapse*: the tendency of the generator network to produce a limited diversity of outputs. An overview of the GAN architecture is displayed in Figure 2.24.

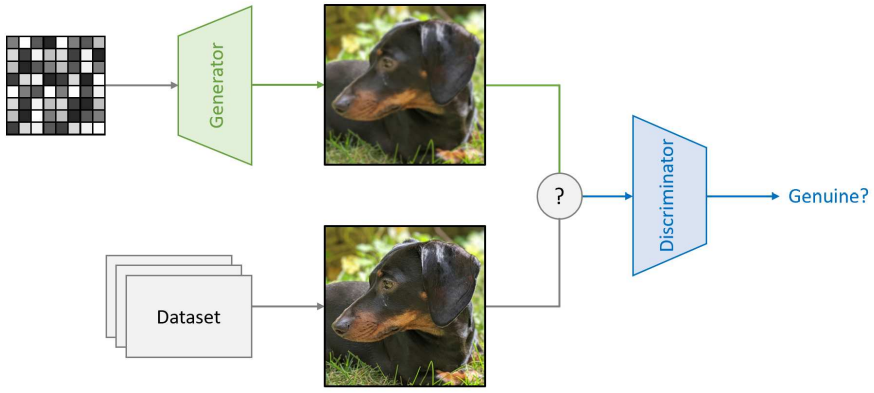


Figure 2.24: Overview of the GAN architecture. Note the similarity between the GAN generator network and the VAE decoder network.

2.2.4 Optimization

Mathematically, optimization refers to the process of minimizing or maximizing the value of an objective function $f(\boldsymbol{x})$ by selecting an appropriate value for $\boldsymbol{x}$ within an optional set of constraints. Thus, optimizing a parametric ML model corresponds to finding an assignment for the model parameters $\boldsymbol{\theta}$ that minimize a loss function $\mathcal{L}$ over a dataset $\mathcal{D}$. Unfortunately, some parametric ML models, like the ANN, do not lend themselves to efficient optimization in the global sense. In these cases, an iterative optimization algorithm can be used to approximate the optimal solution that is otherwise computationally intractable. The best optimization algorithm for a particular ML instance depends on the structure of the ML model, the convexity and differentiability of the loss function, and the size of the dataset under investigation.

Convexity

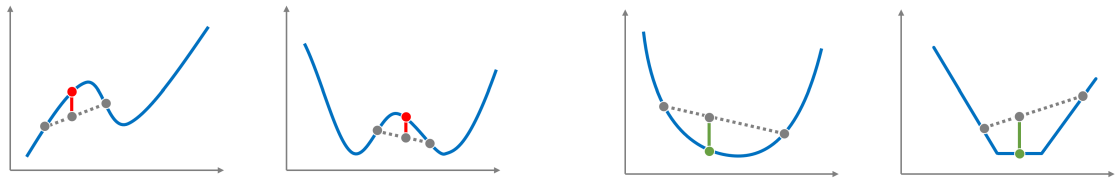
Optimization algorithms that operate over convex objective functions are among the most efficient of their kind. Formally, a set $\mathcal{C} \subseteq \mathbb{R}^D$ is convex if and only if

$$\forall \boldsymbol{x}_1 \in \mathcal{C}, \boldsymbol{x}_2 \in \mathcal{C}, \lambda \in [0,1] : \ \lambda \boldsymbol{x}_1 + (1-\lambda)\boldsymbol{x}_2 \in \mathcal{C} \ . \tag{2.95}$$

Examples of convex sets include the set of positive semi-definite matrices in $\mathbb{R}^{D\times D}$, polyhedra, intersections of convex sets, and $\mathbb{R}^D$ itself. A function $f : \mathcal{C} \to \mathbb{R}$ defined over a convex domain $\mathcal{C}$ is said to be convex if and only if

$$\forall \boldsymbol{x}_1 \in \mathcal{C}, \boldsymbol{x}_2 \in \mathcal{C}, \lambda \in [0,1] : \; f(\lambda\, \boldsymbol{x}_1 + (1-\lambda)\, \boldsymbol{x}_2) \leq \lambda\, f(\boldsymbol{x}_1) + (1-\lambda)\, f(\boldsymbol{x}_2)\;. \quad (2.96)$$

Note that affine functions, quadratic functions, and positive-weighted sums of convex functions all satisfy Equation 2.96 and therefore are convex. Intuitively, a convex function is one that never exceeds a secant passing through two points on its curve.



(a) Non-Convex Functions **(b)** Convex Functions

Figure 2.25: Convex functions do not lie above the linear interpolation between two points on their curve.

The most relevant property of convex functions to machine learning is that local optima are global optima. This means that an expression for the optimal parameterization of an ML model can be derived by setting the gradient of the loss function with respect to the model parameters to zero. If the resulting expression has no analytic solution or is simply too expensive to evaluate, then an iterative optimization algorithm that converges to the global optimum by following the first-order or second-order derivative of the loss function can be employed instead.

Gradient Descent

Gradient descent (GD) is a ubiquitous optimization algorithm that is suitable for both convex and non-convex optimization scenarios. At its core, GD iteratively refines an initial guess at the optimal parameter values of a model by moving the point estimate along the negative gradient of the objective function. The distance covered in each move is determined by the *learning rate* η. Increasing η tends to accelerate convergence but also raises the susceptibility of the GD algorithm to

oscillatory behaviour near local minima. Conversely, decreasing η stabilizes the optimization procedure at the cost of convergence speed.

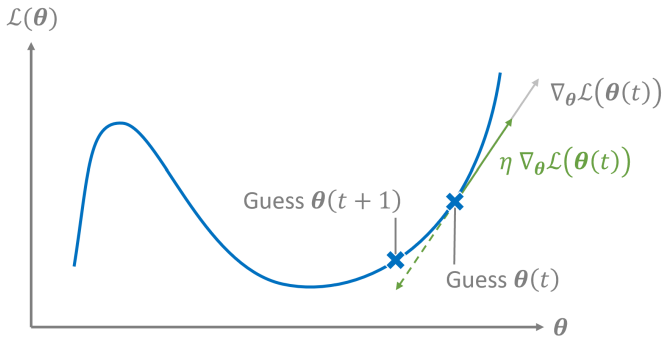


Figure 2.26: Gradient descent finds local minima by shifting parameter values in the opposite direction of the loss function gradient.

Let $\boldsymbol{\theta}(t)$ denote the values of the model parameters at step t in the GD algorithm and let $\mathcal{L}$ denote the loss function. The GD update rule can be stated as follows:

$$\boldsymbol{\theta}(t+1) = \boldsymbol{\theta}(t) - \eta \cdot \nabla_{\boldsymbol{\theta}}\, \mathcal{L}(\boldsymbol{\theta}(t)) \ . \tag{2.97}$$

Most ANN models are optimized using a specialization of GD, namely a variant of batch gradient descent (BGD), mini-batch gradient descent (MGD), or stochastic gradient descent (SGD). The derivative of each parameter in the ANN with respect to the loss function can be computed using the backpropagation equations in Equation 2.91 through Equation 2.94. Let $\mathcal{D} = \langle (\boldsymbol{x}_i, y_i) \rangle_{i=1}^{N}$ be a training dataset with N examples and $\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{x}_i, y_i)$ the value of the loss function value for $(\boldsymbol{x}_i, y_i)$ under the $\boldsymbol{\theta}$ parameterization. Observe that BGD is similar to the formulation in Equation 2.97 and performs GD over the entire dataset:

$$\boldsymbol{\theta}(t+1) = \boldsymbol{\theta}(t) - \frac{\eta}{N} \sum_{i=1}^{N} \nabla_{\boldsymbol{\theta}}\, \mathcal{L}(\boldsymbol{\theta}(t), \boldsymbol{x}_i, y_i)) \ . \tag{2.98}$$

Clearly, the feasibility of BGD may be limited on large datasets due to computational time and memory constraints. To this end, MGD represents a scalable version of BGD that performs GD over a random batch $\mathcal{B}$ from the training dataset where the contents of $\mathcal{B}$ change during each iteration:

$$\boldsymbol{\theta}(t+1) = \boldsymbol{\theta}(t) - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \nabla_{\boldsymbol{\theta}}\, \mathcal{L}(\boldsymbol{\theta}(t), \boldsymbol{x}_i, y_i) \ . \tag{2.99}$$

SGD is an extreme version of MGD where $|\mathcal{B}| = 1$. The expectation of the BGD and SBD update rules match if the training example $(\boldsymbol{x}_k, y_k)$ is chosen uniformly at random during each iteration:

$$\boldsymbol{\theta}(t+1) = \boldsymbol{\theta}(t) - \eta \cdot \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}(t), \boldsymbol{x}_k, y_k) \ . \tag{2.100}$$

An important caveat to performing GD in the context of an ANN is that the converged solution is likely to represent a local minimum of the loss function rather than a global minimum[13]. Consequently, performing GD from different parameter initializations can give different solutions.

Adam

One of the most robust gradient-based optimization algorithm on the market today is *Adam* [KB14]. Adam is a variant of GD that incorporates information about the first and second-order moments of the loss function gradient to accelerate or decelerate the effective learning rate of each individual parameter to be optimized. To simplify notation, let $\boldsymbol{g}(t)$ denote the gradient of the loss function $\mathcal{L}$ with respect to the model parameters $\boldsymbol{\theta}$ at step t:

$$\boldsymbol{g}(t) = \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}(t)) \ . \tag{2.101}$$

The first-order moment $\boldsymbol{m}$ of the gradient is estimated by an exponentially-weighted average of gradient calculations which decays at rate $\beta_1 \in [0, 1)$:

$$\boldsymbol{m}(t+1) = \beta_1 \, \boldsymbol{m}(t) + (1 - \beta_1) \, \boldsymbol{g}(t) \ . \tag{2.102}$$

Likewise, the second-order moment $\boldsymbol{v}$ of the gradient is estimated with an exponentially-weighted average of the element-wise square of the gradient. This rate of exponential decay is controlled by another hyperparameter $\beta_2 \in [0, 1)$:

$$\boldsymbol{v}(t+1) = \beta_2 \, \boldsymbol{v}(t) + (1 - \beta_2) \, \boldsymbol{g}(t) \cdot \boldsymbol{g}(t) \ . \tag{2.103}$$

[13]Virtually all ANN models have a non-convex optimization landscape.

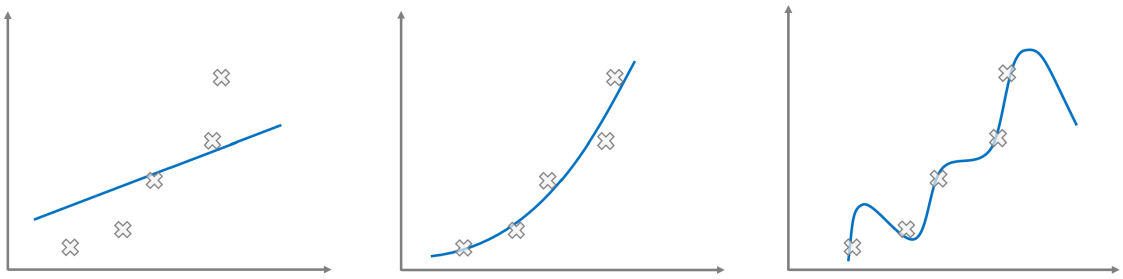
These moments are corrected for the bias present during earlier iterations of the optimization procedure (since Adam initializes the moments to zero) and then integrated into the GD update rule as follows:

$$\boldsymbol{\theta}(t+1) = \boldsymbol{\theta}(t) - \eta \frac{\hat{\boldsymbol{m}}(t)}{\sqrt{\hat{\boldsymbol{v}}(t)} + \epsilon}, \quad \hat{\boldsymbol{m}}(t) = \frac{\boldsymbol{m}(t)}{1-\beta_1^t}, \quad \hat{\boldsymbol{v}}(t) = \frac{\boldsymbol{v}(t)}{1-\beta_2^t} \ . \tag{2.104}$$

Based on Equation 2.104, the first-order moment estimate $\boldsymbol{m}$ ensures that the gradient of a parameter varies smoothly while the second-order moment estimate $\boldsymbol{v}$ attempts to normalize the learning rate η by diminishing the adjustment of parameters with large gradients and emphasizing the adjustment of parameters with small gradients.

2.2.5 Regularization

Training a DNN is not easy: there are innumerable ways in which the training process can fail to produce a useful parameterization of a model despite using a suitable loss function, optimization algorithm, and dataset. That said, most failures can be attributed to either underfitting or overfitting. Underfitting occurs when the *representational capacity* of an ML model is insufficient to capture important nuance in a dataset. Note that representational capacity refers to the family of functions that can be instantiated by different parameterizations of an ML model. Underfitting may also refer to a situation where the parameters selected by an optimization algorithm could be improved (with respect to the loss function) by undergoing more training. The opposite phenomenon, overfitting, occurs when the representational capacity of an ML model is superfluous to a fault. Here, the model chooses to learn the generative process of the samples specific to a training dataset (rather than the whole population) including any noise that is present in the data. Generally speaking, an underfitting model exhibits a large bias while an overfitting model is characterized by excessive variance.



(a) Underfitting Model **(b)** True Model **(c)** Overfitting Model

Figure 2.27: Underfitting models have high bias; overfitting models have high variance.

A telltale sign of overfitting is when a machine learning model begins to perform progressively worse on a testing dataset while experiencing steady improvements over the contents of a training dataset. Usually, the performance of a model over the testing dataset decreases during the earlier phases of training and then increases after training has progressed too far, forming the U-shaped *learning curve* in Figure 2.28.

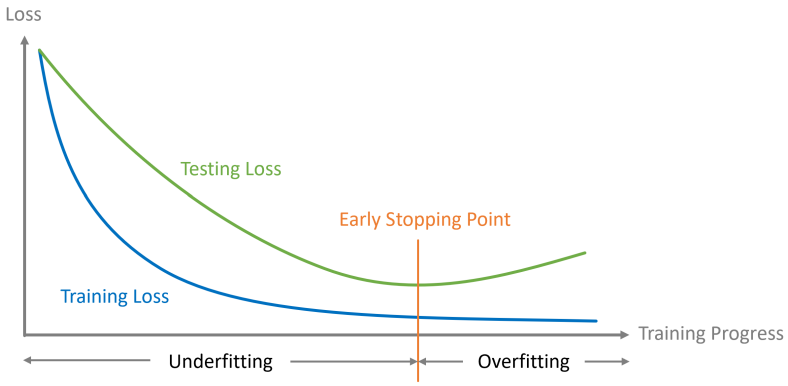


Figure 2.28: Training a model with too few or too many steps of an iterative optimization negatively influences the ability of the model to generalize to novel data.

One strategy for minimizing the risk of overfitting is *regularization*. Regularization refers to any modification of the training process that is intended to reduce generalization error (i.e. the error on the testing dataset) at the potential cost of training error. For instance, *early stopping* is a regularization technique which involves terminating the training process after observing an upward trend in generalization error. An illustration of early stopping is given in Figure 2.28.

Loss Functions

Another approach to regularization is to augment the original loss function $\mathcal{L}$ with an L1 (*Lasso*) [Tib96] or L2 (*Ridge*) [Wil79] penalty term over the values of the model parameters. The intuition here is that model complexity tends to be correlated with high parameter values when the design matrix $\boldsymbol{X}$ and label vector $\boldsymbol{y}$ are normalized appropriately. The magnitude of the penalties are scaled by a hyperparameter λ:

$$\mathcal{L}_{\text{Ridge}}(\boldsymbol{X}, \boldsymbol{y}, \boldsymbol{\theta}) = \mathcal{L}(\boldsymbol{X}, \boldsymbol{y}, \boldsymbol{\theta}) + \lambda \sum_i \theta_i^2 \tag{2.105}$$

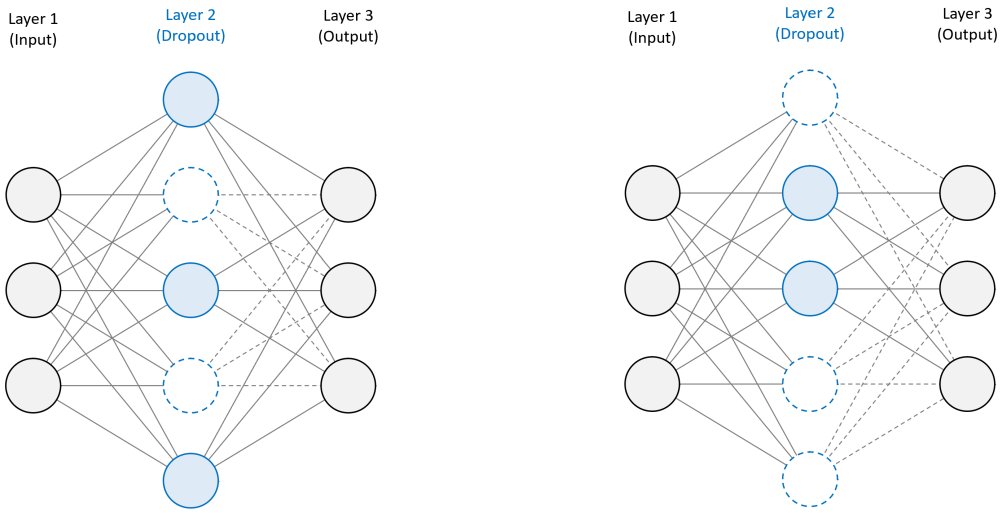
$$\mathcal{L}_{\text{Lasso}}(\boldsymbol{X}, \boldsymbol{y}, \boldsymbol{\theta}) = \mathcal{L}(\boldsymbol{X}, \boldsymbol{y}, \boldsymbol{\theta}) + \lambda \sum_i |\theta_i| \ . \tag{2.106}$$

Some loss functions naturally give rise to regularization. For example, taking the MAP estimate of a Bayesian linear regression with a spherical Gaussian prior over its weights is equivalent to minimizing the squared error of a standard linear regression model using Ridge regularization. Additionally, the ELBO objectives in most VAE formulations implicitly regularize the distribution of the latent variables to a unit spherical Gaussian prior. This encourages the latent variables of each training example to reside near the origin, enabling smooth interpolation between latent samples as well as the ability to generate interesting outputs by sampling from the unit spherical Gaussian distribution.

Neural Architecture

Regularization can also be achieved by taking advantage of the flexibility afforded by the DNN framework. Specifically, an existing component of a DNN can be replaced with one that has more desirable regularization properties or a new component can be inserted into a DNN with the express purpose of regularization. In fact, one DNN regularization technique that belongs to the former category was previously discussed in Section 2.2.3: the convolutional layer. Convolutional layers implement regularization through a form of *parameter sharing* where the preactivation of each neuron in the convolutional layer is derived from the same kernel. While this significantly limits the representational capacity of a convolutional layer relative to a fully-connected one, it also reduces the risk of overfitting and tends to accelerate

training. Another option, termed *dropout* [SHK+14], involves randomly disabling neurons in a fully-connected or convolutional layer to discourage the DNN from learning mistake-correction dependencies that are unlikely to generalize to new data.



(a) Training Iteration 1 **(b)** Training Iteration 2

Figure 2.29: Dropout regularization randomly discards activations in a hidden layer by fixing them to 0. Usually, neurons are deactivated with probability $p = 0.5$. Deactivated neurons are indicated by dashed lines in Figure 2.29a and Figure 2.29b.

A third option is to simply change all the ReLU activations in a DNN to a self-normalizing variant called the SELU [KUMH17]:

$$\text{SELU}(x) = \lambda \begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases} . \tag{2.107}$$

The primary advantage of the SELU activation is that a preactivation distribution with zero mean and unit variance will produce an activation distribution of zero mean and unit variance. This concept is further explored in the next passage.

Batch Normalization

A different approach to DNN regularization is *batch normalization* (BN) [IS15]. BN refers to the insertion of BN layers which normalize the activations of the preceding layer across a batch of inputs. The target mean β and variance γ of the normalization are learnable parameters[14] of a BN layer. To understand the BN

[14]The mean and variance can optionally be fixed, namely to $\beta = 0$ and $\gamma = 1$.

algorithm, consider a batch of neuron activations $\boldsymbol{x} = \{x_1, \ldots, x_n\}$. The output $\boldsymbol{y}$ of the BN layer is given by standardizing the activations by their mean μ and variance σ^2 before scaling and shifting the standardized result $\hat{\boldsymbol{x}}$ by γ and β, respectively:

$$\mu = \frac{1}{n}\sum_{i=1}^{n} x_i \tag{2.108}$$

$$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2 \tag{2.109}$$

$$\hat{x}_i = \frac{x_i - \mu}{\sqrt{\sigma^2 + \epsilon}} \tag{2.110}$$

$$y_i = \gamma\hat{x}_i + \beta \ . \tag{2.111}$$

The ϵ above is added for numeric stability. Now, suppose that the input to a BN layer is a batch of tensors $\boldsymbol{X} \in \mathbb{R}^{B\times H\times W\times C}$ with height H, width W, and C feature maps. Then, a total of HWC independent γ and β parameters must be learned; however, if $\boldsymbol{X}$ represents the activations of a 2D convolutional layer, then only C independent γ and β parameters must be learned. Assuming the latter case and taking $\boldsymbol{Y} \in \mathbb{R}^{B\times H\times W\times C}$ to be the output of the BN layer,

$$\mu_c = \frac{1}{BHW}\sum_{b=1}^{B}\sum_{h=1}^{H}\sum_{w=1}^{W} X_{bhwc} \tag{2.112}$$

$$\sigma_c^2 = \frac{1}{BHW}\sum_{b=1}^{B}\sum_{h=1}^{H}\sum_{w=1}^{W}(X_{bhwc} - \mu_c)^2 \tag{2.113}$$

$$\hat{X}_{bhwc} = \frac{X_{bhwc} - \mu_c}{\sqrt{\sigma_c^2 + \epsilon}} \tag{2.114}$$

$$Y_{bhwc} = \gamma_c\hat{X}_{bhwc} + \beta_c \ . \tag{2.115}$$

An alternative to BN for convolutional layers is *instance normalization* (IN) [UVL16]. As the name suggests, C independent γ and β parameters are learned for each tensor in $\boldsymbol{X}$:

$$\mu_{bc} = \frac{1}{HW}\sum_{h=1}^{H}\sum_{w=1}^{W} X_{bhwc} \tag{2.116}$$

$$\sigma_{bc}^2 = \frac{1}{HW}\sum_{h=1}^{H}\sum_{w=1}^{W}(X_{bhwc} - \mu_{bc})^2 \tag{2.117}$$

$$\hat{X}_{bhwc} = \frac{X_{bhwc} - \mu_{bc}}{\sqrt{\sigma_{bc}^2 + \epsilon}} \tag{2.118}$$

$$Y_{bhwc} = \gamma_c\hat{X}_{bhwc} + \beta_c \ . \tag{2.119}$$

The purpose of inserting a BN or IN layer into a DNN is to reduce internal covariate shift (ICS). ICS refers to the change in the input distribution of a layer resulting from optimizations in previous layers. The primary issue with ICS is that most ANN optimization algorithms do not consider how parameters in one layer influence the parameters of deeper layers. Hence, standardizing the output distribution of each layer has the potential to stabilize and accelerate training. Furthermore, adapting the standardization to each individual neuron (or convolutional channel) through the learnable β and γ parameters ensures that the interpretation of certain activation functions, like the sigmoid function, are not accidentally normalized away.

Dataset Partitioning

Finally, partitioning a corpus of data into a training dataset and testing dataset is often necessary to reduce overfitting. The training dataset is used to estimate the optimal parameter values of an ML model while the testing dataset is used to evaluate the performance of the model once training is complete. Note that it is absolutely essential to leave the testing dataset out of the training feedback loop to avoid introducing bias into the final evaluation. Hence, any hyperparameter tuning and early stopping must be constrained to the training dataset. For this reason, the training dataset is usually further subdivided into a validation dataset and a smaller training dataset. As a rule of thumb, about 20% of the original data corpus should be donated to the testing dataset with 20% of the remainder destined for the validation dataset.

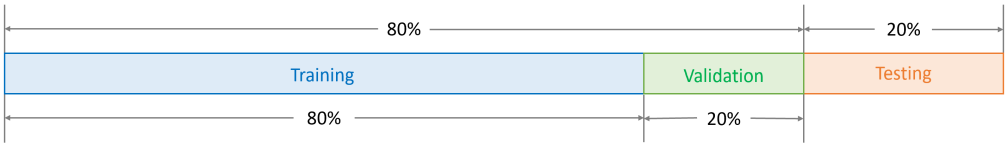


Figure 2.30: Cardinality guidelines for the training, validation, and testing datasets.

Under this scheme, training commences by tuning hyperparameters over the validation dataset. The implementation of this procedure depends on the semantics on the hyperparameters being tuned and the number of available computational resources. The outcome of the hyperparameter tuning process is then applied to

the training of the model over the smaller training dataset; the validation dataset is reused to estimate generalization error (i.e. for early stopping). Afterwards, the performance of the model is analyzed using the untouched testing dataset.

2.3 Texture Synthesis

2.3.1 Introduction

Texture synthesis lies at the crossroads of computer graphics and machine learning. Put simply, a texture is a characterization of the visual appearance of a surface. This includes homogenous materials, like plastic or paint, as well as composite materials with distinctive patterns, like bricks or wood. The goal of texture synthesis is to *analyze* the underlying generative process of a texture and construct a model that can be efficiently sampled to *synthesize* new instances of that texture. To illustrate this process, consider the task of rendering an aerial view of a large grass field. A relevant texture synthesis program might accept an exemplar image of a grass patch as input, derive a procedural model which implicitly captures the height, orientation, and colour distributions of the grass blades within the patch, and then proceed to sample this model at each location in the field to generate an output image of the desired size. Ideally, the output of the texture synthesis program should preserve the perceptual qualities of the exemplar image while expanding its dimensions in a way that appears natural and free of artefacts such as misaligned edges or visible seams.

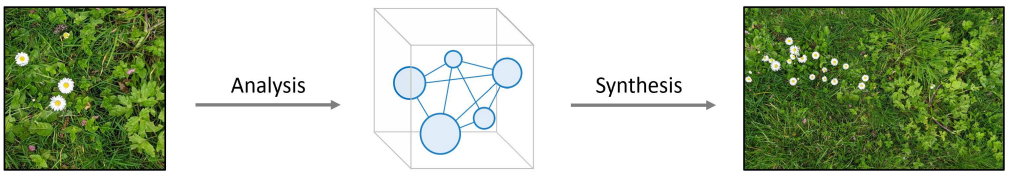


Figure 2.31: Example-based texture synthesis is performed by inferring the parameters of a generative model from an input texture (analysis) and then sampling that model to produce an output texture (synthesis).

Paradigms

There are no formal restrictions on the inputs and outputs of a texture synthesizer. This lack of standardization gives rise to a number of texture synthesis paradigms, the

most common of which is *example-based texture synthesis*. Here, a texture synthesis program accepts an image of a texture as input and produces a representation of the same texture as output. Traditionally, these algorithms generate RGB images as output and thus lack the fidelity required for 3D applications that need to reproduce textures under different lighting conditions. For this reason, commercial texture synthesizers tend to output SVBRDF textures which capture the reflectance properties of a material at each point on a surface. *Solid texture synthesis* programs extend the notion of 2D texture synthesis into the volumetric domain by producing output in the form of voxels (i.e. volumetric pixels). The primary aim of solid texture synthesis is to improve the continuity of textures along warped surfaces and objects with exposed cross-sections. Lastly, *dynamic texture synthesis* programs attempt to animate static textures into videos or identify opportunities for spatially-localized loops within existing videos. This research project focuses exclusively on example-based texture synthesis with SVBRDF textures.

Applications

The main consumer of texture synthesis technology is the digital entertainment industry. For example, material authoring systems like Substance Designer [Adob] allow environment artists to simultaneously customize the appearance of large textures by simply changing the parameter values of a texture synthesis model. Moreover, rendering engines take advantage of procedural texture synthesis models to circumvent the memory limitations present in many devices. More recently, deep learning techniques have been successfully applied to a variety of challenging problems in texture synthesis. Specifically, DNN models have been trained to enhance the resolution of photographs [JAFF16], inpaint missing regions of images [TB15], and interpolate between the styles of different textures [BJV17, LFY$^+$17, YBS$^+$19]. Some of the more ambitious research efforts in this area have included the training of a deep LVM that can remove the presence of windows from a room [RMC15] and the training of a DNN that can estimate the appearance of chairs from angles that are partially hidden from view [DSTB14].

2.3.2 Procedural Noise

Fundamentally, noise is an unstructured random process defined by its power spectrum [LLC+10]. White noise has a power spectrum that contains all frequencies in equal proportion [PH89] and is generally undesirable for visual applications; however, limiting the support of white noise to particular frequencies, octaves, or bands can yield discernible patterns. This controlled form of noise is often simulated and evaluated using a *procedural noise function* to model textures described by stochastic generative processes. Most procedural noise functions strive to be aperiodic, continuous, efficient, parameterized, and randomly accessible[15] [LLC+10]. Naturally, there are several categories of procedural noise functions, including lattice gradient noise, explicit noise, and sparse convolution noise. The first category encompasses all procedural noise functions that generate noise by interpolating between random values on an integer lattice and is examined more closely in this section.

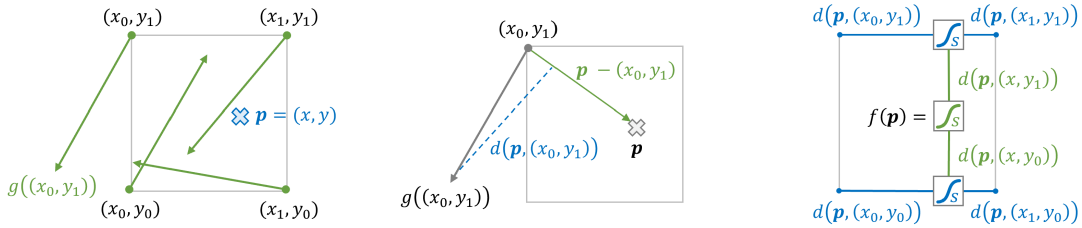
Perlin Noise

Perlin noise [Per85] is a lattice gradient noise that has become an industry standard for the procedural generation of natural phenomena like clouds or landscapes. Ultimately, Perlin noise describes a function $f : \mathbb{R}^D \to \mathbb{R}$ that maps points from a real coordinate space to a number in the closed interval $[-1, 1]$. The procedure for evaluating this function is best explained by considering the two-dimensional case where the goal is to find the value of f at some point $\boldsymbol{p} = (x, y) \in \mathbb{R}^2$. To begin, let $\mathcal{N}(\boldsymbol{p})$ denote the set of points on the integer lattice that are neighbours of $\boldsymbol{p}$:

$$\mathcal{N}(\boldsymbol{p}) = \{(x_0, y_0), (x_0, y_1), (x_1, y_0), (x_1, y_1)\} \ . \tag{2.120}$$

Above, $x_0 = \lfloor x \rfloor$, $x_1 = x_0 + 1$, $y_0 = \lfloor y \rfloor$, and $y_1 = y_0 + 1$. An overview of the Perlin noise algorithm is pictured in Figure 2.32 and described in greater detail below.

[15] This is by no means an exhaustive list of desirable noise function properties.



(a) Gradient Step **(b)** Projection Step **(c)** Interpolation Step

Figure 2.32: Perlin noise describes a function f which assigns a value to a point $\boldsymbol{p} \in \mathbb{R}^D$ by finding the gradients of $\boldsymbol{p}$'s neighbours in an integer lattice, projecting the gradients in a direction towards $\boldsymbol{p}$, and then smoothly interpolating the resulting projections at $\boldsymbol{p}$. The computational complexity of Perlin noise scales exponentially with D.

1. **Gradients**: The first step in the Perlin noise algorithm is to assign a gradient $g(\boldsymbol{n}) \in \mathbb{R}^D$ to each neighbour $\boldsymbol{n} \in \mathcal{N}(\boldsymbol{p})$. The gradient at a lattice point $\boldsymbol{n}$ is a unit vector that is typically computed by seeding a pseudorandom number generator (PRNG) with a hash $h(\boldsymbol{n})$ and then using a Monte Carlo method[16] to sample a vector uniformly distributed along the unit D-sphere. Some Perlin noise implementations optimize this step by precomputing a list of possible gradient directions and then using $h(\boldsymbol{n})$ to index into this list [Per02].

2. **Projection**: The second step in the Perlin noise algorithm is to compute the dot product between the gradient $g(\boldsymbol{n})$ and distance vector $(\boldsymbol{p} - \boldsymbol{n})$ for each $\boldsymbol{n} \in \mathcal{N}(\boldsymbol{p})$. The resulting projection is denoted by $d(\boldsymbol{p}, \boldsymbol{n}) = (\boldsymbol{p} - \boldsymbol{n}) \cdot g(\boldsymbol{n})$.

3. **Interpolation**: The final step in the Perlin noise algorithm is to smoothly interpolate between the values of $d(\boldsymbol{p}, \boldsymbol{n})$ at $\boldsymbol{p}$. This is analogous to bilinear interpolation where the linear weight w between two points is passed through a smoothing function s. A viable candidate for s is any function that is non-decreasing over the interval $[0, 1]$ and satisfies $s(0) = 0$ and $s(1) = 1$. In practice, imposing the additional constraint that $s'(0) = s'(1) = s''(0) = s''(1) = 0$ helps avoid visual discontinuities. A common choice for $s(t)$ is

$$s(t) = 6t^5 - 15t^4 + 10t^3 \ . \tag{2.121}$$

[16] A popular Monte Carlo method for this task is *rejection sampling*. Here, points are sampled uniformly at random within the unit D-cube until a sample falls inside the unit D-ball. This sample is then projected onto the unit D-sphere to form the gradient.

Now, let $s_x = s(x - x_0)$ and $s_y = s(y - y_0)$. The value of $f(\boldsymbol{p})$ is then

$$f(\boldsymbol{p}) = (1 - s_y)\, d(\boldsymbol{p}, (x, y_0)) + s_y\, d(\boldsymbol{p}, (x, y_1)) \tag{2.122}$$

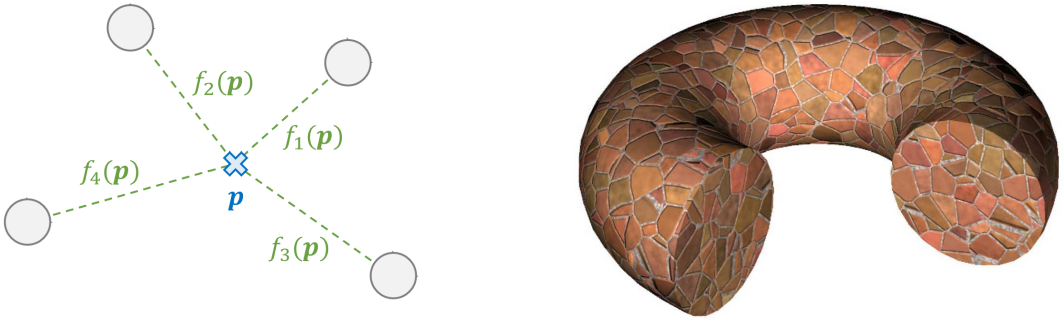
$$d(\boldsymbol{p}, (x, y_0)) = (1 - s_x)\, d(\boldsymbol{p}, (x_0, y_0)) + s_x\, d(\boldsymbol{p}, (x_1, y_0)) \tag{2.123}$$

$$d(\boldsymbol{p}, (x, y_1)) = (1 - s_x)\, d(\boldsymbol{p}, (x_0, y_1)) + s_x\, d(\boldsymbol{p}, (x_1, y_1))\ . \tag{2.124}$$

It is worth noting that the frequency of Perlin noise can be adjusted along each dimension by multiplying the coordinate associated with that dimension by a scalar value before invoking f. As a result, multiple layers of Perlin noise can be stacked on top of one another to simulate patterns in a texture or structure that occur at different scales.

Worley Noise

A different approach to lattice gradient noise is captured by Worley noise [Wor96]. The idea behind this method is to place a random number of *feature points* within each cell of an integer lattice and define a cellular texture basis function $f_n(\boldsymbol{p})$ which calculates the distance to the n^{th} nearest feature point from $\boldsymbol{p}$. The number of feature points in a particular lattice cell is computed by initializing a PRNG with the hash of the coordinates of the cell and then sampling from a predefined Poisson distribution. The precise location of each feature point within a cell is chosen uniformly at random. Interesting patterns can be obtained by combining basis functions (e.g. $f(\boldsymbol{p}) = f_2(\boldsymbol{p}) - f_1(\boldsymbol{p})$) or swapping distance metrics (e.g. using Manhattan distance instead of Euclidean distance). In practice, Worley noise is proficient at generating textures with cellular patterns like scales or rocks.



(a) Cellular Texture Basis Functions **(b)** Solid Synthesis Example

Figure 2.33: Worley noise describes a cellular texture basis function $f_n(\boldsymbol{p})$ given by the distance of a point $\boldsymbol{p}$ to its n^{th} nearest feature point. Figure 2.33b is courtesy of [Wor96].

2.3.3 Classical Algorithms

The earliest example-based texture synthesis algorithms generated images by making the neighbourhood of each pixel in the output texture similar to a neighbourhood in the input texture. Implicitly, these algorithms modelled textures as Markov random fields (MRFs) where the colour of each pixel is determined by its local neighbourhood and is independent of its absolute position in the image. While this approach was surprisingly effective, the algorithms produced output images one pixel at a time and were usually too slow to gain widespread adoption. One strategy for improving the efficiency and global coordination of these algorithms involves using a multi-resolution representation of an image called an image pyramid. The Gaussian pyramid is a popular image pyramid which iteratively halves the width and height of a source image at each layer by applying a Gaussian filter and downsampling the image stored in the previous layer. An alternative to the Gaussian pyramid is the Laplacian pyramid where the image stored in layer ℓ is the difference between the original and blurred images from layer ℓ of the equivalent Gaussian pyramid. This latter representation forms the basis of LAPGAN which is an ensemble of GANs that synthesize images by progressively upsampling and refining a seed image using the structure of a Laplacian pyramid [DCF+15].

(a) Gaussian Pyramid **(b)** Laplacian Pyramid

Figure 2.34: Each layer in a Gaussian pyramid represents a blurred and downsampled version of the previous layer; the Laplacian pyramid is similar except only the difference between the original and blurred images is stored in each layer. Note that the brightness and contrast of the first three levels in Figure 2.34b are modified to improve visibility.

Efros-Leung Algorithm

One of the simplest texture synthesis algorithms based on the MRF model is the Efros-Leung (EL) algorithm [EL99]. The EL algorithm begins by seeding the center of the output texture with a random crop from the input texture. The size of this crop is controlled by a hyperparameter and is expected to be no larger than the size of a pixel neighbourhood. Incidentally, the size of a pixel neighbourhood is another hyperparameter and is often set to the dimensions of the largest feature in the input texture. Regardless, the EL algorithm proceeds in an outward spiral from the center of the output texture, colouring each pixel by comparing its filled neighbourhood to the pixel neighbourhoods in the input texture. Specifically, for a pixel p in the output texture with neighbourhood $\mathcal{N}(p)$, the algorithm finds the pixel in the input texture with a neighbourhood $\mathcal{N}(q)$ that has the minimum distance d to $\mathcal{N}(p)$. The distance between two neighbourhoods is simply the weighted mean squared error (MSE) of their pixel values where the weight of a pixel is given by a 2D Gaussian kernel over the center of a neighbourhood. Afterwards, the algorithm assigns the colour of p to the center of a random neighbourhood in the input texture with a distance score d' where $d' < (1 + \varepsilon)\, d$ for some small hyperparameter $\varepsilon \in \mathbb{R}^+$. The primary drawback of the EL algorithm is that the shape of the filled neighbourhood changes frequently, making it difficult to implement the usual precomputation strategies for optimizing the performance of the algorithm.

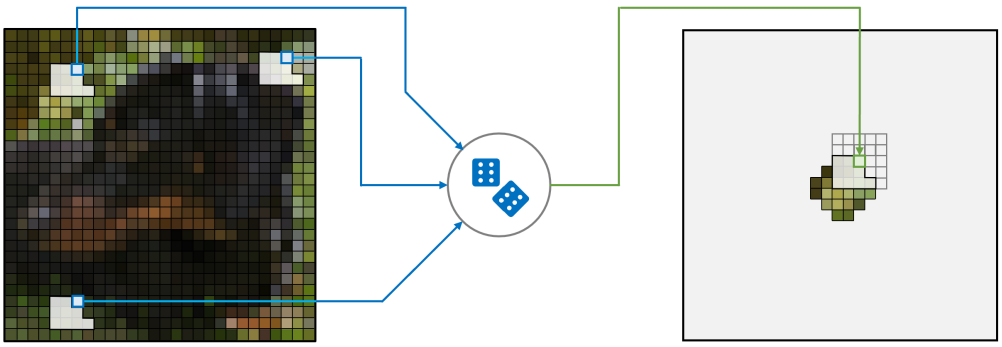


Figure 2.35: The EL algorithm sets the colour of an output pixel to the colour of a random input pixel with a similar neighbourhood. Here, the neighbourhood of a pixel in the output texture is the filled subset of a 5×5 rectangular window centered on the pixel.

Wei-Levoy Algorithm

The Wei-Levoy (WL) algorithm [WL00] is an extension of the EL algorithm that addresses some of the performance issues plaguing the original formulation. In the WL algorithm, the output texture is initialized with random noise, preferably one that matches the colour histogram of the input texture. The algorithm then iterates across the output texture in raster order[17], selecting a new colour for each pixel based on its causal neighbourhood. To be precise, the colour of each output pixel is assigned to the colour of the pixel in the input texture associated with a causal neighbourhood that has the smallest distance score to the causal neighbourhood of the output pixel. The causal neighbourhood of a pixel p consists of pixels that lie either strictly above p or share the same row as p but are located to the left of p, forming an "L". Note that the shape of the causal neighbourhood is independent of the position of the reference pixel or the progress of the algorithm. Neighbourhoods that extend past the border of the output image are wrapped toroidally to promote the synthesis of tileable textures. Using a causal neighbourhood ensures that the colour of most output pixels will not be directly influenced by neighbouring pixels with random colours that have yet to be refined by the algorithm. A notable exception are the pixels in the first few rows of the output texture which may be cropped for good measure.

[17] Raster order refers to iterating over pixels in a row-by-row, left-to-right fashion.

(a) Input **(b)** 3×3 Window **(c)** 7×7 Window **(d)** 11×11 Window

Figure 2.36: Increasing the size of the neighbourhood in the EL and WL algorithms improves coherence at the cost of diversity. Note that Figure 2.36a measures 64×64 pixels while the remaining figures measure 256×128 pixels.

Tree-Structured Vector Quantization

The main advantage the WL algorithm holds over the EL algorithm is that all of the output pixel neighbourhoods have the same shape. This opens the door to several opportunities for optimization, one of which is *tree-structured vector quantization* (TSVQ) . The basic premise of TSVQ is to arrange the causal neighbourhoods of the input texture in the form of a tree to quickly find the neighbourhood in the input texture with (approximately) the smallest distance score to an arbitrary neighbourhood in the output texture. The tree is constructed by clustering all the neighbourhoods in the input texture using the standard k-means algorithm with the distance score metric. Afterwards, the center of each cluster is replaced with a node denoting the average neighbourhood of its constituents and is connected to its parent (initially the root node). The same procedure is then recursively performed over the set of neighbourhoods belonging to each cluster unless the cluster is empty. The branching factor of the tree (i.e. the number of clusters taken at each level) is governed by a hyperparameter and controls how accurately the tree can answer neighbourhood queries. A neighbourhood query is processed by traversing the tree from the root node using a greedy selection criterion whereby the child of a node with the smallest distance score to the given output neighbourhood is chosen for traversal. The search terminates when a leaf node of the TSVQ tree is reached (containing a single neighbourhood from the input texture). This strategy has the potential to decrease the running time of the WL algorithm by several orders of magnitude, although using a more practical (i.e. conservative) branching factor yields a running time improvement closer to a single order of magnitude.

Multi-Resolution Synthesis

Another way to accelerate the performance of the WL algorithm (and the EL algorithm) is through multi-resolution synthesis (MRS). The purpose of this method is to enable the synthesis of textures with larger neighbourhoods without significantly compromising the speed of the algorithm. As illustrated in Figure 2.36, increasing the size of a neighbourhood can drastically improve the quality of the output texture when the input texture has large features. MRS is usually implemented by synthesizing a series of output textures in the reverse order of a Gaussian pyramid. Taking the WL algorithm as example, the first step of MRS is to store the input texture and randomly-initialized output texture as a pair of Gaussian pyramids with L layers. Then, the refinement portion of the WL algorithm is invoked at level L of the output pyramid, followed by level $L-1$, $L-2$, and so on. The only difference in the invocations is that the causal neighbourhood of a pixel includes both the "L"-shaped neighbourhood in its current layer in addition to the full neighbourhood at its equivalent position in the previous layer[18]. The intuition behind this approach is that each layer in the output pyramid guides the synthesis of the layer immediately below it and thus mitigates the need for a large neighbourhood. Note that MRS can be used in conjunction with TSVQ although a new tree must be constructed for each pair of adjacent layers in the pyramid.

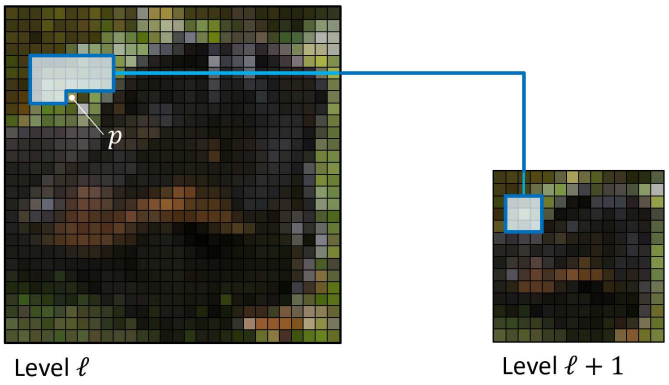


Figure 2.37: The causal neighbourhood of pixel $p = (x, y)$ in layer ℓ of a Gaussian pyramid includes the full (but often narrower) neighbourhood of pixel $p' = (\lfloor x/2 \rfloor, \lfloor y/2 \rfloor)$ in layer $\ell + 1$ of the Gaussian pyramid.

[18]The size of a pixel neighbourhood should be inversely proportional to its level in the pyramid.

Ashikhmin Algorithm

One weakness present in many WL algorithm implementations is that the distance score is taken to be the pixel-wise MSE of two neighbourhoods. Although intuitive and efficient, this metric is not always a good indication of perceptual similarity. In particular, using the MSE to synthesize textures often results in the blurring of shape boundaries and the loss of fine detail [Ash01]. The Ashikhmin algorithm [Ash01] attempts to correct this deficiency by modifying the WL algorithm in a way that encourages verbatim copying of patches from the input texture to the output texture. To this end, the algorithm annotates each pixel in the output texture with the location of the pixel in the input texture whose colour was assigned to the output pixel. To begin, the output image is initialized by copying random pixels from the input texture to the output texture, taking care to update the annotations accordingly. The algorithm then proceeds to iterate over the pixels of the output texture in raster order, copying pixels from the input texture based on their causal neighbourhood. Unlike the WL algorithm, only a handful of neighbourhoods from the input texture as considered for each output pixel p. As shown in Figure 2.38, this candidate set of neighbourhoods is given by the shifted neighbourhoods of the input pixels associated with the output pixels in the neighbourhood of p. The input pixel corresponding to the neighbourhood with the smallest distance score to the neighbourhood of p is copied into the location of p.

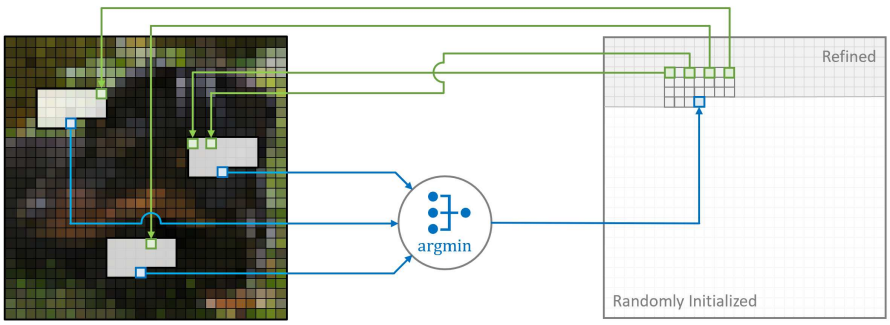


Figure 2.38: The Ashikhmin algorithm selects the pixel to be copied from the input texture to a location in the output texture by choosing among the input texture neighbourhoods associated with the output texture neighbourhood.

A variant of the Ashikhmin algorithm, called k-coherence [TZL+02], tries to restore diversity in the output of the Ashikhmin algorithm by mapping each neighbourhood in the input texture to its k nearest neighbourhoods with respect to distance score. This way, when an input pixel is included in the candidate set for an output pixel, the k nearest neighbourhoods of the input pixel neighbourhood are also added to the candidate set.

2.3.4 Inverse Rendering

Synthesizing textures from photographs of objects in the real world is partially a matter of solving the inverse rendering problem (IRP). The IRP describes the challenge of deriving the geometry, reflectance, and lighting of an object from one or more images of that object. This disambiguation is necessary to understand how the object may appear from different angles or in various lighting environments. Unfortunately, many geometry, reflectance, and lighting configurations are likely to explain the radiance distribution of any particular scene and so a unique solution to an instance of the IRP may not exist. Hence, the goal of most approaches that tackle the IRP is to deduce a plausible decomposition of a scene with the aid of a few simplifying assumptions. For example, some methods assume that one of the scene components, such as geometry, is known beforehand, while other methods assume that the lighting of a scene is strictly environmental (i.e. invariant to translation) or does not feature any shadows. A number of basic strategies for inferring the BRDF of a surface are described in this section; more advanced techniques are reserved for Section 2.1.6.

Measurement Devices

The gonioreflectometer is a measurement device that can densely sample the BRDF of a surface using a movable light source and sensor. The samples obtained from this device are typically the most accurate of their kind and can be interpolated to yield a ground-truth BRDF. Gonioreflectometers produce excellent results but suffer from two significant drawbacks. One problem is logistics: gonioreflectometers tend

to be slow, expensive, and immobile. The other issue is that gonioreflectometers, by design, represent a BRDF using millions of raw samples. This has negative implications with regards to memory consumption, rendering performance, and integration with global illumination algorithms. A lightweight alternative to the gonioreflectometer is the camera. Here, the burden of the IRP rests primarily with the algorithms tasked with processing the images from the camera. Of course, the exposure, resolution, and colour depth of the photographs also influence the accuracy of a BRDF reconstruction. Among the many advantages of using a camera to measure radiance are portability, accessibility, and the capacity to estimate the environment lighting in a scene by taking a photograph of a spherical mirror. One interesting derivative of the digital camera is the light-field camera. Light-field cameras, also known as plenoptic cameras, employ an array of micro-lenses to capture the intensity and direction of incoming light. Recently, [AS17] showed that a CNN can be successfully trained to reconstruct BRDFs at oblique viewing angles given (simulated) light-field data from viewing angles that are relatively close to the surface normal. Combining traditional camera images with depth information from a commodity depth sensor can further enhance the proficiency of IRP techniques. Specifically, [KGT$^+$17] demonstrated that two different DNN architectures can be trained to accurately reconstruct the macroscopic geometry and reflectance properties of an object given a set of RGBD images (i.e. RGB images with a depth channel) with sufficient coverage over the voxels composing the object. The depth channel of the images is used to determine both the voxel associated with the RGB values in the other channels as well as the approximate surface normal of that voxel.

MERL 100 Dataset

The MERL 100 dataset [MPBM03] is an established collection of BRDF measurements obtained from spherical specimens of 100 isotropic materials. Recall that an isotropic BRDF f is one that is invariant to changes in the azimuth angle φ_h of the halfway vector. Combined with the Helmholtz reciprocity criterion in Equation 2.24,

$$\forall \varphi_d \in [0, \pi] : \; f(\theta_h, \theta_d, \varphi_d) = f(\theta_h, \theta_d, \varphi_d + \pi) \; . \tag{2.125}$$

Thus, the BRDF measurements in the MERL 100 dataset are discretized into $90 \times 90 \times 180 = 1\,458\,000$ uniformly-distributed bins for each material. A common way to visualize the MERL 100 dataset is through a BRDF slice. BRDF slices are images which encode the values of an isotropic BRDF by varying θ_h over the horizontal axis and θ_d over the vertical axis while keeping φ_d fixed to $\pi/2$. As shown in Figure 2.39, BRDF slices offer valuable insight into the behaviour of a BRDF near specular and grazing angles.

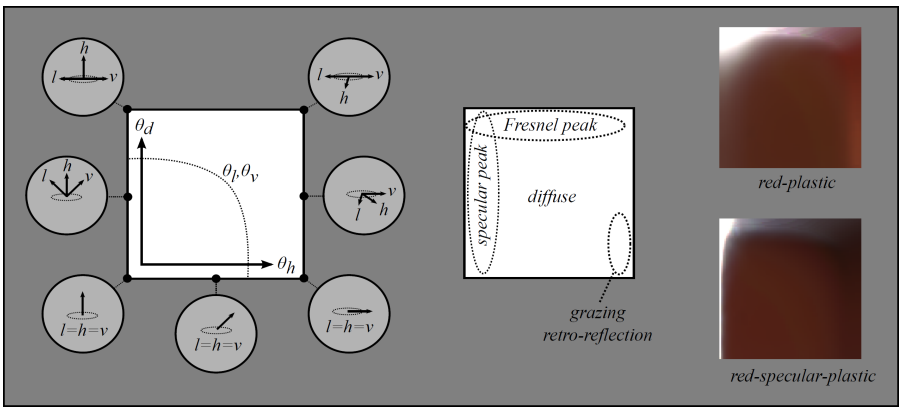


Figure 2.39: The BRDF slice summarizes key properties of a BRDF, including its tendencies near specular, grazing, and diffuse angles. Image is courtesy of [BS12].

It is worth noting that the original purpose of the MERL 100 dataset was to explore strategies for reducing the number of measurements needed to faithfully represent isotropic BRDFs. To this end, the researchers applied a wavelet analysis to the BRDF measurements and discovered that each BRDF can be approximated as a linear combination of about 69 000 wavelets and match the precision of the experimental apparatus. The researchers also found that isotropic materials outside the dataset could be accurately represented as linear combinations of the 100 BRDFs inside the dataset, provided that 800 BRDF measurements of the material are available.

Pocket Reflectometry

Another data-driven solution to an instance of the IRP is pocket reflectometry [RWS+11]. At its core, pocket reflectometry is a procedure for reconstructing the SVBRDF of a planar surface using a video camera, linear light source, and

a BRDF chart. A BRDF chart is simply a card-sized palette of representative BRDF materials, including a sample of Spectralon[19] [Lab] which serves as a diffuse reference. The reconstruction process begins by placing the BRDF chart next to the target sample and recording a 30-second video of the linear light source moving over the sample and chart. This setup is illustrated in Figure 2.40.

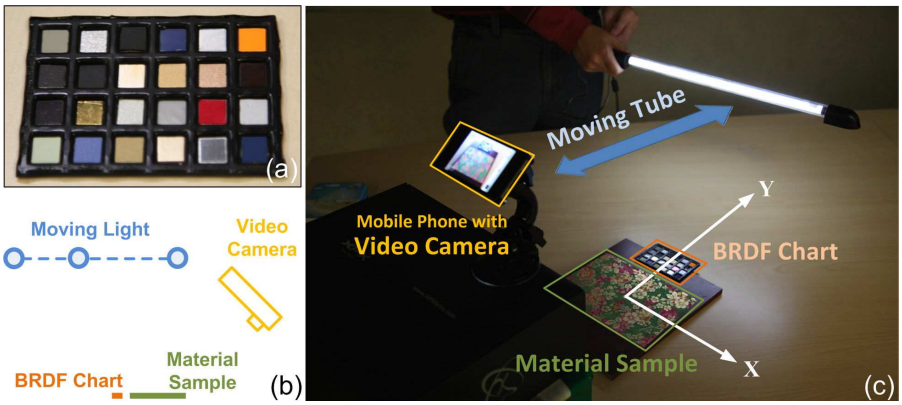


Figure 2.40: Pocket reflectometry is conducted by waving a linear light source over a material sample and BRDF chart, as shown in figures (b) and (c). Figure (a) depicts the BRDF chart used in the experiment. Image is courtesy of [RWS+11].

The BRDF at each point on the target surface is modelled by the summation of a diffuse albedo term and a linear combination of specular terms corresponding to the BRDFs of the materials in the BRDF chart. Prior to computing these terms, it is essential to calibrate the *reflectance vectors* at each point on the surface. A reflectance vector of a point is a vector $\boldsymbol{r}$ where $\boldsymbol{r}_i$ denotes the intensity of light (with respect to one colour channel) that is reflected from that point during frame i of the video. Now, calibration is performed by subtracting an unlit image of the scene across each reflectance vector to give $\check{\boldsymbol{r}}$, dividing each element of $\check{\boldsymbol{r}}$ by the corresponding value of the Spectralon reflectance vector, and then linearly normalizing the result to the range $[0, 1]$. From here, dynamic time warping is used to align the reflectance vectors of each point to that of a single specular tile with the highest unsaturated reflectance. The final step of the SVBRDF recovery process is use quadratic programming to find the values of the albedo and specular BRDF weights at each point that jointly minimize the difference between the reconstructed

[19]Spectralon is an industry-standard near-perfect diffuse reflector over the visible light spectrum.

BRDF and calibrated reflectance vector across each colour channel. Note that this method can be extended to recover surface normal information (through calibration) by adding a pair of diffuse and specular spherical caps to the BRDF chart.

2.3.5 Perceptual Metrics

Virtually all texture synthesis programs can be viewed as optimization algorithms where the objective is to minimize the perceptual difference between two textures with respect to certain constraints. Unfortunately, developing a metric that scales proportionally to the perceptual similarity of two images is still an open problem in the field. To understand the difficulty of this task, consider the fact that an ideal perceptual metric must (implicitly or explicitly) account for imperceptible misalignments between images, distinguish between foreground and background discrepancies, and compare semantic content. Consequently, a variety of different perceptual loss functions have been proposed, each with their own performance characteristics and inevitable bias. Some notable omissions from the following passages are perceptual metrics from the domain of signal processing, such as the structural similarity index measure or the peak signal-to-noise ratio, which have consistently proved to be poor indicators of perceptual quality [JAFF16].

Spatial Metrics

The most basic perceptual loss function is based on the norm of the pixel-wise difference between two images. Specifically, let $\mathcal{I}^{(1)}$ and $\mathcal{I}^{(2)}$ denote two RGB images of height H and width W and let $\mathcal{I}_{ijk}$ denote the value of the k^{th} colour channel in row i column j of the image $\mathcal{I}$. Then,

$$\mathcal{L}_{\text{L1}}(\mathcal{I}^{(1)}, \mathcal{I}^{(2)}) = \frac{1}{3HW} \sum_{i=1}^{H} \sum_{j=1}^{W} \sum_{k=1}^{3} |\mathcal{I}^{(1)}_{ijk} - \mathcal{I}^{(2)}_{ijk}| \tag{2.126}$$

$$\mathcal{L}_{\text{L2}}(\mathcal{I}^{(1)}, \mathcal{I}^{(2)}) = \frac{1}{3HW} \sum_{i=1}^{H} \sum_{j=1}^{W} \sum_{k=1}^{3} (\mathcal{I}^{(1)}_{ijk} - \mathcal{I}^{(2)}_{ijk})^2 \ . \tag{2.127}$$

While the L1 and L2 loss functions are simple, flexible, and relatively inexpensive to compute, they are also poor at recognizing the similarity between different instances of the same texture or images that differ by a mere scaling, rotation, or

translation transform. Furthermore, note that the L1 and L2 losses can be trivially extended to compare SVBRDF textures by taking the norm of the differences between the SVBRDF parameter values at each position in the two textures. In contrast to simply rendering a pair of SVBRDF textures using an arbitrary light and virtual camera and then applying an image-based L1 or L2 loss function, this approach compares textures at a higher level of abstraction but introduces the issue of inferring the significance of each BRDF parameter in the perception of an image rendered from the SVBRDF texture. Another variant of the L2 loss is the root mean squared error (RMSE) loss which normalizes the units of the L2 loss and penalizes[20] residuals more strongly than that of either the L1 or L2 loss functions:

$$\mathcal{L}_{\text{RMSE}}(\mathcal{I}^{(1)}, \mathcal{I}^{(2)}) = \sqrt{\frac{1}{3HW} \sum_{i=1}^{H} \sum_{j=1}^{W} \sum_{k=1}^{3} (\mathcal{I}_{ijk}^{(1)} - \mathcal{I}_{ijk}^{(2)})^2} \ . \tag{2.128}$$

Lastly, it is worth mentioning that the reliability of these pixel-wise losses can be improved by changing the encoding of the $\mathcal{I}^{(1)}$ and $\mathcal{I}^{(2)}$ images. For example, converting $\mathcal{I}^{(1)}$ and $\mathcal{I}^{(2)}$ into the CIELAB colour space is likely to give residuals that are better correlated with perceived differences in colours [GB20, VCGLM19].

Histogram Metrics

Another way to quantitatively describe the similarity of two images is by comparing their *colour histograms*. A colour histogram H describes the distribution of pixel frequencies for a particular colour channel in an image. Let $H_{ij}^{(k)}$ be the proportion of pixels in colour channel i of image $\mathcal{I}^{(k)}$ that belong to intensity bin $1 \leq j \leq B$. The similarity between two RGB images can then be measured using the mean total variation (MTV) distance:

$$\mathcal{L}_{\text{MTV}}(\mathcal{I}^{(1)}, \mathcal{I}^{(2)}) = \frac{1}{6B} \sum_{i=1}^{3} \sum_{j=1}^{B} |H_{ij}^{(1)} - H_{ij}^{(2)}| \ . \tag{2.129}$$

Observe that the MTV loss function does not model the spatial distribution of the pixel intensities. As a result, the MTV metric may erroneously conclude that two images are the same even if their differ on each pixel. A different kind of histogram

[20] Assuming the values of each pixel component fall in the range $[0, 1]$.

metric is based on the notion of a co-occurrence matrix. Here, the pixel values in an image are clustered into a set of groups before a symmetric matrix C is constructed where entry C_{ij} represents a measure of how closely the pixels in group i are located to the pixels in group j [DLT+20]. Note that co-occurrences matrices have been successfully applied to the training of conditional GAN models to synthesize plausible variants of a texture while giving users the ability to manually control the second-order statistics (i.e. co-occurrences) of the generated results [DLT+20].

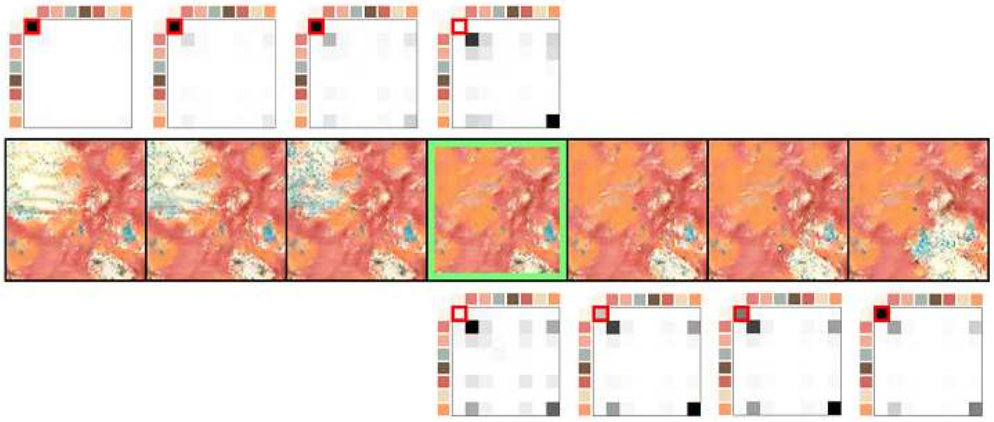


Figure 2.41: Training a DNN to synthesize textures from a set of localized co-occurrence matrices allows intuitive control over the texture synthesis process. Above, the top and bottom rows depict the co-occurrence matrices controlling the top-left and bottom-right regions, respectively, of the texture in the middle row. Image is courtesy of [DLT+20].

Neural Metrics

The most influential discovery in the field of texture synthesis was published in a seminal work by [GEB15] where it was demonstrated that object detection DNNs can be repurposed as perceptual loss functions. In particular, the researchers showed that the activations of feature maps in certain layers of the VGG [SZ14] family of CNNs can be summarized in the form of a Gram matrix which captures the spatially-invariant style of an image. These Gram matrices may then be compared across different inputs to the DNN using the standard L1 or L2 loss functions. Formally, the Gram matrix $\boldsymbol{G} \in \mathbb{R}^{C \times C}$ of a set of feature maps $\boldsymbol{F} \in \mathbb{R}^{H \times W \times C}$ in one layer is the mean inner product of the flattened versions of the feature maps:

$$G_{c_1 c_2} = \frac{1}{HW} \sum_{i=1}^{H} \sum_{j=1}^{W} F_{ijc_1} F_{ijc_2} \ . \tag{2.130}$$

Observe that $\boldsymbol{G}$ can be interpreted as the uncentered covariance matrix over each sample $\boldsymbol{F}_{ij} \in \mathbb{R}^C$ in the feature maps. A more recent publication offered empirical evidence that subtracting the mean activation of the feature maps from $\boldsymbol{F}$ can mitigate training bias related to the scale of the Gram matrices in DNNs that learn to synthesize a diverse range of textures [LFY+17]:

$$G'_{c_1 c_2} = \frac{1}{HW} \sum_{i=1}^{H} \sum_{j=1}^{W} (F_{ijc_1} - \mu)(F_{ijc_2} - \mu), \quad \mu = \frac{1}{HWC} \sum_{i=1}^{H} \sum_{j=1}^{W} \sum_{c=1}^{C} F_{ijc} \; . \quad (2.131)$$

Typically, the preactivations or activations of the `conv1_1`, `conv2_1`, `conv3_1`, `conv4_1`, and `conv5_1` layers of a VGG network are used to construct the Gram matrices [LFY+17, ULVL16, YBS+19]. The preactivations or activations of the `conv4_2` layer are also sometimes extracted to compute a content (rather than style) loss [LW16, LFY+17, ULVL16]. In this case, the feature maps of the chosen layer are directly compared using an L1 or L2 loss without any further complications. Interestingly, the style and content losses can be employed to train a DNN to perform a style transfer task where the goal is to synthesize an image with the content of one image and the style of another image.

2.3.6 Deep Texture Synthesis

The advent of deep learning has accelerated the progress of texture synthesis algorithms far beyond the limits of its procedural ancestors. Most generative DNN models in texture synthesis research take the form of deep LVMs that are trained to reconstruct images from their latent representation. When paired with an appropriate DNN-based perceptual loss function from Section 2.3.5, these models are often able to synthesize an impressive variety of images in the style of the same reference texture. Additionally, interpolating between the latent representations of two textures can yield images that represent a natural hybrid of the two original textures. Of course, the type of input a DNN consumes, the architecture of the DNN, and the output representation of the DNN, not to mention its training regime, all have important consequences for the utility of the final model.

Single-Image SVBRDF Acquisition

The category of texture synthesis research that strikes closest to the heart of this research project is the reproduction of SVBRDF textures from a single image. The pioneering work in this domain used an L-BFGS optimizer to find the Blinn-Phong SVBRDF parameters of a 256×256 SVBRDF tile that can accurately reconstruct a flash-lit image of an MRF texture [AAL16]. During each iteration of the optimization algorithm, 15 random 256×256 crops of the input texture are rendered[21] using a preconditioned estimate of the SVBRDF tile parameters and then compared using a style loss derived from the VGG-19 network. Additionally, a set of stationarity priors which penalize large changes in brightness, contrast, or other higher-order statistics across the SVBRDF parameters are also factored into the overall loss function. This approach performs well on homogenous, stochastic textures but is relatively slow and has difficulty capturing the essence of inhomogeneous or regular textures. Another research effort involved combining a convolutional autoencoder with a classifier ANN and a set of densely-connected continuous conditional random fields (DCRFs) to infer the microfacet SVBRDF parameters of a flash-lit input texture [LSC18]. Notably, the input to the convolutional autoencoder includes a radial coordinate field to overcome the spatial invariance that would normally be implied by a convolutional architecture. It is also worth mentioning that the predicted SVBRDF parameters are rendered within the network (under a random point light) to enable the backpropagation of gradients from the reconstruction loss to the CNN.

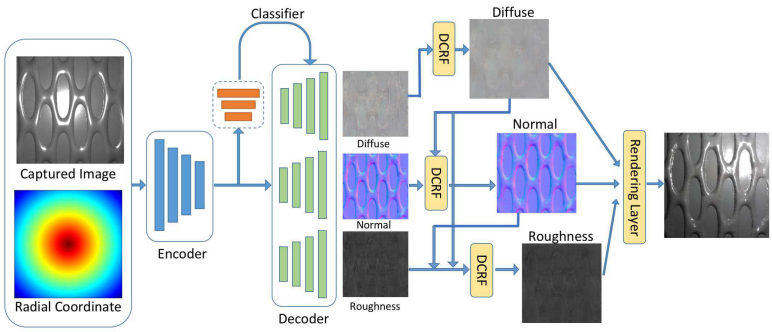


Figure 2.42: Architecture of [LSC18]. Image is courtesy of [LSC18].

[21] The directions of the halfway vectors are inferred from the position and angular distribution of the flash, as well as the field-of-view of the camera which took the photograph.

Lastly, [DAD+18] also attempted to train a DNN to infer the microfacet SVBRDF parameters of a single flash-lit image. The distinguishing feature of this work is the pairing of an autoencoder CNN based on the U-Net architecture [RFB15] with a parallel fully-connected DNN that integrates global features into the CNN that may lie outside the receptive fields of the individual neurons. The model also makes use of the SELU activations described in Section 2.2.5 and renders the SVBRDF parameters under a random point light and viewing position or a mirror configuration that is designed to highlight purely specular reflections.

Architectures

Virtually all the generative DNN models described throughout the texture synthesis literature are unique; however, there are a few architectures that deserve a special mention. First, [MMZ+18] leverages a combination of five specialized DNNs to extract the parameters of a Blinn-Phong BRDF from an input image. As illustrated in Figure 2.43, the *SegmentationNet* is the first DNN encountered during forward propagation and identifies the region of an image corresponding to the object of interest. Afterwards, *SpecularNet* and *MirrorNet* respectively predict the soft and hard specular reflections of the object. Finally, *AlbedoNet* and *ExponentNet* regress the Blinn-Phong BRDF parameters of the material comprising the object. Interestingly, the DNNs are initially trained independently on a synthetic data corpus to accelerate convergence during the later end-to-end training phase.

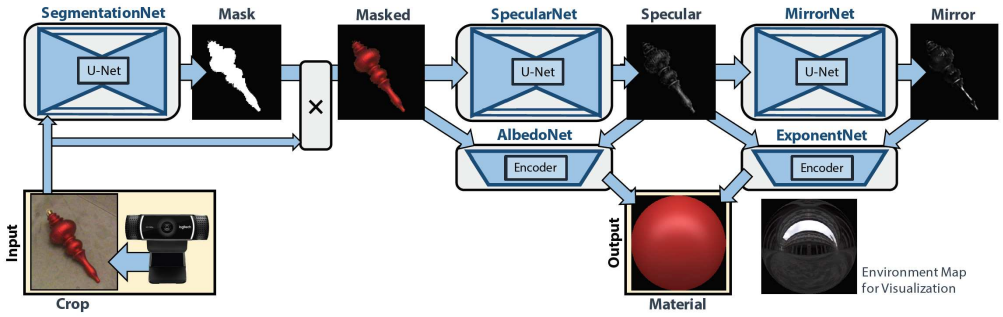


Figure 2.43: Architecture of [MMZ+18]. Image is courtesy of [MMZ+18].

Next, [VCGLM19] proposes an architecture that accepts two *homographied* images of a homogenous planar surface as input, one aligned with the surface normal and

another at an angle of 60° to the surface normal, and predicts the BRDF parameters of a modified Ashikhmin-Shirley model as output. Here, a homography refers to a projection that maps points from a plane inside an image to the image plane, thereby correcting the perspective of the embedded plane. The key innovation of this work is the use of a nested learning architecture where the parameters of the BRDF are predicted sequentially such that the output of the earlier predictions are passed as input to the later predictions. Finally, [ULVL16] introduced the notion of using a multi-scale DNN architecture to synthesize textures in the style of some reference texture. The multi-scale generator DNN accepts a series of random noise tensors as input, each of which is processed by a convolutional subnetwork unique to its size. These outputs are then iteratively reduced by upsampling the smallest output by a factor of two, concatenating it with the second-smallest output (now of the same spatial dimensions), and then passing the result through another convolutional subnetwork. The output of the final convolutional subnetwork is the output texture.

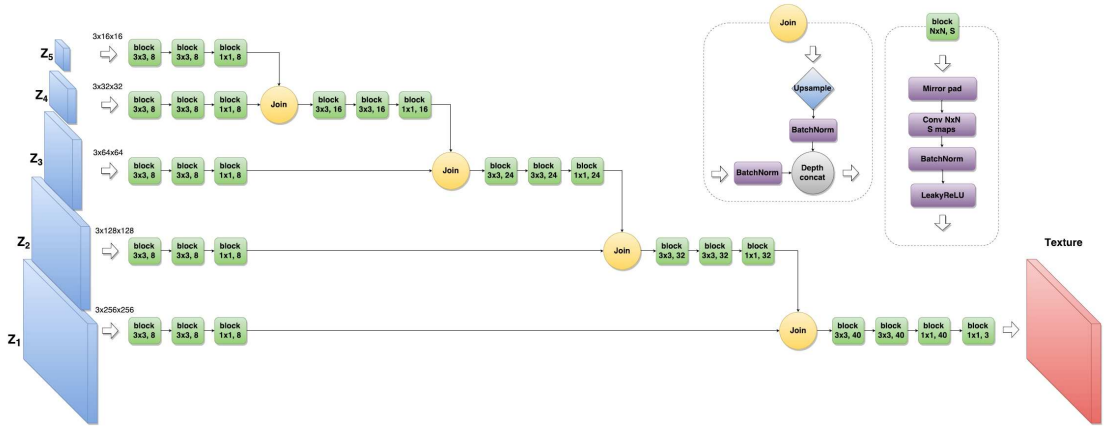


Figure 2.44: Architecture of [ULVL16]. Image is courtesy of [ULVL16].

Training

Another area that is ripe with creative opportunities is the training process of texture synthesis DNNs. For example, [LDPT17] applied the concept of self-augmentation to compensate for the lack of supervised training data in SVBRDF reconstruction tasks. The idea behind this approach is to automatically generate labelled training data by rendering the output of a partially-trained SVBRDF

reconstruction network. This technique is not effective for untrained, randomly-initialized models since the generated SVBRDF parameter maps are unlikely to resemble the distribution of the training set. Next, [YBS+19] developed a strategy for optimizing the interpolation performance of a texture synthesis autoencoder DNN by concurrently training the DNN with two textures such that each training iteration yields two reconstruction losses and one interpolation loss. The interpolation loss is computed by randomly shuffling, tiling, and cropping the blended latent representations of the two input textures and then feeding the result through a weighted neural perceptual loss function. Moreover, [KWKT15] successfully disentangled the latent variable representation of a VAE by grouping related images into the same training batch and then controlling which latent variables could be modified to account for the variation across a batch. This involves feeding a batch through the inference network of the VAE, replacing the activations of each latent variable (except for the one meant to encode the variation) with its mean activation over the batch, and then invoking the generator network of the VAE in the usual manner, backpropagating the reconstruction losses as required. Lastly, [LFY+17] explored the advantages of *incremental training* for the reconstruction of RGB textures with a generative DNN model. Incremental training is a form of curriculum learning where the contents of the training dataset are slowly revealed to an ML model as it achieves good performance on the visible subset. In this case, a new input texture is introduced to the training process after a fixed number of training iterations.

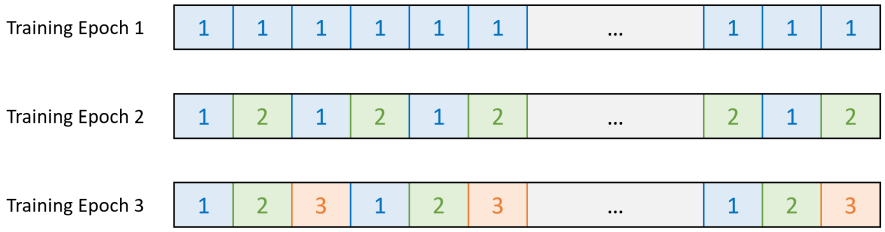


Figure 2.45: Incremental training exposes a new element from the training dataset every epoch. The number on each square represents the ID of the texture used for the corresponding training iteration.

Expansion

One of the most exciting applications of deep learning in the realm of texture synthesis is, broadly speaking, the expansion of texture images. A recent publication by [ZZB+18] showed that a GAN can be trained to double the spatial extent of an arbitrary input texture without changing the scale of any features. This is achieved using a deep convolutional generator network with residual blocks that modelled the synthesis of large-scale structures from the input texture. Surprisingly, the adversarial loss [GPAM+14] component of the GAN loss function is responsible for most of the training progress of the GAN: the L1 loss and neural style loss only serve to stabilize training and diminish artefacts. Despite the simplicity of the generator and discriminator architectures, the GAN is remarkably proficient at expanding stationary, regular, and stochastic textures with challenging symmetries.

Figure 2.46: Results from [ZZB+18]. The middle column is the input to the GAN while the side columns are the output of the GAN. Image is courtesy of [ZZB+18].

A different perspective on image expansion is presented in [JAFF16]. Here, the researchers designed a convolutional DNN that can enhance the resolution of photographs by a 4x or 8x scale factor. As expected, the DNN approach qualitatively

and quantitatively outperforms bicubic interpolation; however, this is only the case when the DNN is trained using a neural content loss function based on the VGG-16 network. Specifically, training the DNN with a traditional L1 loss function yields performance that is inferior to bicubic interpolation.

Tiling

The quest to synthesize a single tileable texture from an input image was partially abandoned when it was discovered that the human perceptual system is obnoxiously proficient at detecting seamless patterns. Nonetheless, generating a *variety* of compatible textures from one input image is a powerful idea that can be realized through the expressive power of deep LVMs. To this end, [JBV16] observed that a fully-convolutional GAN can be coerced into synthesizing tileable textures by enforcing a periodic boundary condition on its noise tensor input. As along as the receptive fields of the neurons lining the edges of the output texture match their counterparts on the opposite side, the texture will be tileable by virtue of the localized nature of CNNs. A subsequent work divided the noise tensor into a set of local, global, and periodic components with the aim of improving the interpolation performance of the original GAN as well as its ability to reconstruct regular textures [BJV17]. The researchers encoded the semantics of the global and periodic partitions by keeping the value of the global noise tensor the same at each spatial location and varying the value of the periodic noise tensor according to one or more wave functions. Unfortunately, this modification complicates the synthesis of tileable textures since the frequencies of the periodic noise almost certainly do not align with the dimensions of the output texture. [FAW19] presents a different approach to tileable texture synthesis where the task is to generate a large output image under the guidance of a content image using the styles present throughout a corpus of small reference textures. In this work, the output texture is synthesized by partitioning the content image into a grid of tiles, finding the reference texture that most closely resembles each of these tiles, and then merging the latent representations of the corresponding reference textures before feeding

the result through a generator DNN. Note that the burden of creating natural boundaries between adjacent latent tiles rests primarily with the generator network.

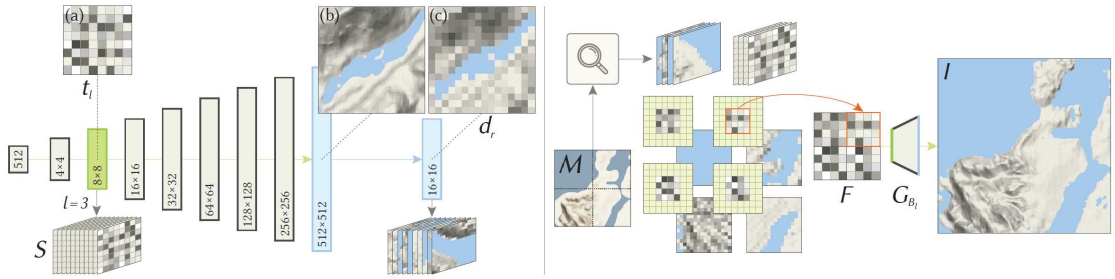


Figure 2.47: Overview of the synthesis pipeline from [FAW19]. On the left, the generator of a GAN is trained to produce textures in the style of a dataset. Afterwards, the generator is sampled to form a corpus of reference textures. On the right, a guidance map M is subdivided into a grid of tiles which are replaced by the latent representations of their closest reference textures. The latents are then merged and propagated through the generator to form the desired output image I. Image is courtesy of [FAW19].

Miscellaneous

To conclude this survey of applied deep learning methods, it is appropriate to consider a few research publications that do not fit under one of the aforementioned categories. For example, [DSTB14] trained a DNN to synthesize images of chairs and tables from inputs that independently describe the identity of the object to be synthesized, the zenith and azimuth angles of the viewer relative to the object in the output image, and a set of transformations conveying changes in the rotation, scale, brightness, etc. of the synthesized object. The researchers demonstrated that such a DNN is capable of transferring knowledge about transformations between different classes of objects (e.g. tables and chairs) and can even infer the appearance of an object from viewpoints that are not disclosed by the training data. Another noteworthy publication documented the results of training a DBN to encode the Lambertian reflectance of a face [TSH12]. In this model, a rendering layer is inserted between the visible layer and first hidden layer of the DBN which computes the intensity of each pixel in the image by multiplying an albedo by the dot product of a surface normal and a global light vector. Otherwise, [TB15] experimented with using a long short-term memory (LSTM) RNN to synthesize and inpaint textures

by introducing a total ordering to the pixels. The textures obtained by both the DBN and LSTM networks in [TSH12] and [TB15] are substandard in terms of both size and quality compared to the usual feedforward DNN approaches.

3
Approach

Contents

This chapter is devoted to the implementation of the research project. First, Section 3.1 presents an overview of the ML system to lay the context of the ensuing sections. Then, Section 3.2 explains the architecture of the chosen ML model while Section 3.3 delves into the SVBRDF reflectance models leveraged by the system. Afterwards, Section 3.4, Section 3.5, and Section 3.6 respectively describe the loss functions, dataset, and training regime employed to derive the parameters of the final model. Finally, Section 3.7 touches upon the hardware and software environments that played a role in the development and execution of the ML system.

3.1 Overview

The original intent of this research project was to train a generative ML model that accepts an image of a texture as input and produces an SVBRDF texture of arbitrary size as output that captures the style and reflectance properties of the input texture. After the background reading phase of the project drew to a close, this evolved into the following technical specification: train a deep convolutional autoencoder that predicts the surface normals and Disney BRDF[1] parameters at each point of a flash-lit image of an input texture.

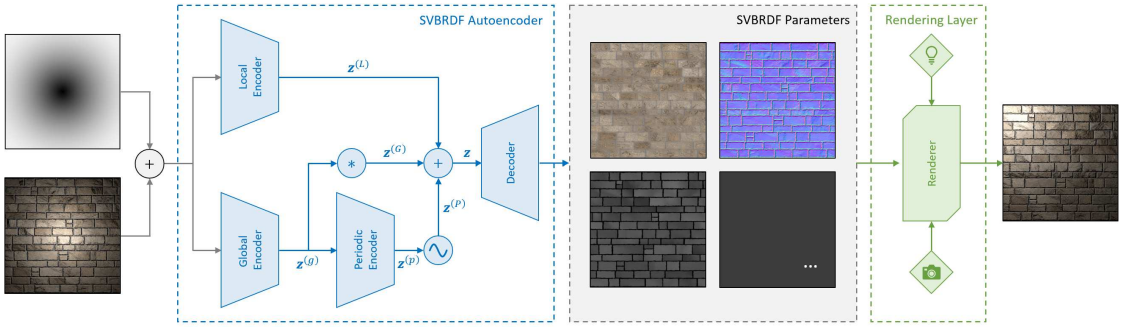


Figure 3.1: Overview of the proposed ML system. The two + nodes denote concatenation along the channel dimension, the $*$ node denotes spatial duplication, and the $\sim$ node denotes the vectorized sine function. See the text for more details.

3.1.1 Input

The input to the ML model is a concatenation of a 256×256 linear RGB image[2] and a corresponding *radial distance field* (RDF). The image is expected to be a fronto-parallel photograph of a stationary, regular, or stochastic texture where the dominant light source is approximately co-located with the viewing position. This is a common setup among texture synthesis models in the deep learning literature: the flash dominates environment shadows and permits the DNN to infer the high-frequency, specular behaviour of the texture. The RDF encodes the normalized distance of each pixel to the center of the image, similar to [LSC18]. This enables the first convolutional layer of the ML model to process pixels according to their saturation from the flash. It follows that the concatenated input is a $256 \times 256 \times 4$ tensor.

[1] The Disney BRDF is explained in Section 3.3.
[2] Images are usually stored in the sRGB colour space.

3.1.2 Model

The ML model, dubbed the *SVBRDF Autoencoder*, is composed of three encoder ANNs and one decoder DNN. Conceptually, the *Local Encoder* and *Global Encoder* can be viewed as deep CNNs whose role is to transform the input of the model into the noise tensors expected by the generator DNN in [BJV17]. More concretely, the Local Encoder is a fully-convolutional DNN that converts its input into a local latent field $\boldsymbol{z}^{(L)} \in \mathbb{R}^{32\times32\times256}$. Intuitively, the local latent field represents the structural variation that distinguishes instances of a texture from one another. Otherwise, the Global Encoder is a convolutional DNN with a fully-connected final layer that produces a global latent vector $\boldsymbol{z}^{(g)} \in \mathbb{R}^{128}$ as output. In contrast to the local latent field, $\boldsymbol{z}^{(g)}$ identifies the invariant characteristics of a texture. As such, $\boldsymbol{z}^{(g)}$ is duplicated along the spatial extent of $\boldsymbol{z}^{(L)}$ to yield a global latent field $\boldsymbol{z}^{(G)} \in \mathbb{R}^{32\times32\times128}$. Lastly, the Periodic Encoder is a shallow MLP that translates $\boldsymbol{z}^{(g)}$ into the parameters $\boldsymbol{z}^{(p)} \in \mathbb{R}^{6}$ of two sinusoidal plane waves. These waves are then evaluated at each spatial index of $\boldsymbol{z}^{(L)}$ and $\boldsymbol{z}^{(G)}$ to give the periodic latent field $\boldsymbol{z}^{(P)} \in \mathbb{R}^{32\times32\times2}$. Each latent field is then concatenated along its third dimension to produce the final latent field $\boldsymbol{z} \in \mathbb{R}^{32\times32\times386}$. Naturally, $\boldsymbol{z}$ is then fed into the fully-convolutional *Decoder* network to form an SVBRDF output texture of size $256 \times 256 \times 16$. Note that the first two channels of the Decoder output describe the zenith and azimuth angles of a surface normal while the remaining fourteen channels convey the parameters of the Disney BRDF.

3.1.3 Rendering

The parameters of the SVBRDF Autoencoder are optimized by comparing the renderings of the predicted and ground-truth SVBRDF textures under a random point light and viewer location. The positions of both the light and viewer are independently sampled from a cosine-weighted hemisphere centered on a 256×256 texture patch of unit size, as proposed by [DAD+18]. Weighing the hemisphere

in this fashion simplifies the sampling process[3] and favours common light and viewer positions. Next, the rendering equation is computed at each location on the texture patch using both SVBRDF textures: this is effectively a homographied perspective rendering of the texture patch. Note that evaluating Equation 2.20 boils down to multiplying the radiance at each location on the texture patch by the cosine of the surface normal at that point. Furthermore, observe that the SVBRDF reflectance models need not match between the predicted and ground-truth textures. Indeed, the predicted SVBRDF parameters are rendered using the Disney BRDF while the ground-truth parameters are rendered using a BRDF from Substance Designer [Adob].

3.1.4 Optimization

The output of the differentiable rendering layer is used to update the SVBRDF Autoencoder parameters by evaluating a reconstruction loss function $\mathcal{L}$ which is positively correlated with the difference between the ground-truth and predicted renderings. In practice, $\mathcal{L}$ is a weighted sum of three independent loss functions:

$$\mathcal{L}(\mathcal{I}^{(1)}, \mathcal{I}^{(2)}) = \lambda_T \, \mathcal{L}_T(\mathcal{I}^{(1)}, \mathcal{I}^{(2)}) + \lambda_S \, \mathcal{L}_S(\mathcal{I}^{(1)}, \mathcal{I}^{(2)}) + \lambda_C \, \mathcal{L}_C(\mathcal{I}^{(1)}, \mathcal{I}^{(2)}) \; . \qquad (3.1)$$

Above, $\mathcal{L}_T$ is the *texel loss function*, $\mathcal{L}_S$ is the *style loss function*, and $\mathcal{L}_C$ is the *content loss function*. Additionally, $\lambda_T, \lambda_S, \lambda_C \in \mathbb{R}^+$ are the hyperparameter weights used to correct the scale and importance of each loss function. To begin, the $\mathcal{L}_T$ loss function computes the L1 loss between the renderings and serves primarily to reduce noise artefacts [ZZB+18]. In contrast, the $\mathcal{L}_C$ and $\mathcal{L}_S$ loss functions are based on the VGG-19 network and exert a more profound influence on the learning of the SVBRDF Autoencoder. Like most other texture synthesis DNNs, the SVBRDF Autoencoder is trained in an end-to-end manner using the Adam optimizer with MGD.

[3] Sampling is performed by taking two independent samples ξ_1 and ξ_2 from the uniform distribution $\mathcal{U}[0, 1]$ and computing the spherical coordinates $\varphi = 2\pi\xi_1$ and $\theta = \cos^{-1}(\sqrt{\xi_2})$.

3.2 Architecture

Mathematically, the SVBRDF Autoencoder is a function $f : \mathbb{R}^{256\times256\times4} \rightarrow \mathbb{R}^{256\times256\times16}$ that is implemented using a convolutional autoencoder DNN. The convolutional aspect of the SVBRDF Autoencoder takes advantage of the translational invariance expected from the encoder and decoder processes to significantly reduce the number of tunable parameters in the DNN while simultaneously improving the convergence and speed of the training procedure. Moreover, the autoencoder characterization of the SVBRDF Autoencoder allows the receptive fields of the output neurons to span most of the input neurons and provides an effective way to interpolate between known textures and synthesize novel textures. Conveniently, a set of guidelines for such DNN models is documented in [RMC15]. The key highlights of this publication include the promotion of strided convolutions for downsampling and upsampling operations, the denouncement of fully-connected hidden layers, and the endorsement of batch normalization and ReLU or Leaky ReLU activations between hidden layers. The SVBRDF Autoencoder architecture obeys all of these points.

3.2.1 Local Encoder

The Local Encoder is a fully-convolutional DNN that expands the number of channels in an input tensor before iteratively halving its spatial extent and doubling its depth. Each convolutional layer of the DNN is equipped with a $k = 5 \times 5$ kernel and an $s = 2$ stride (except for the first layer) followed by a ReLU activation and a BN layer, except for the last layer which is succeeded by a tanh activation to facilitate sampling of the latent field. Note that the ratio of spatial contraction to channel expansion in the DNN is commonplace in the texture synthesis literature [BJV17, DLT+20, GRR+18, JAFF16, LDPT17, RFB15, SDSY16, VCGLM19, ZZB+18], as is the size of the kernel [BJV17, DLT+20, KWKT15, VCGLM19]. Furthermore, the ReLU activation is used in favour of the Leaky RELU or SELU activations because these ReLU variants did not qualitatively improve the reconstruction performance of the SVBRDF Autoencoder in preliminary experiments[4]. A different experiment also

[4]See Section 3.6 for more information about the preliminary experiments.

revealed that appending another convolutional block to downsample the input to a $16 \times 16 \times 512$ representation did not significantly alter reconstruction performance.

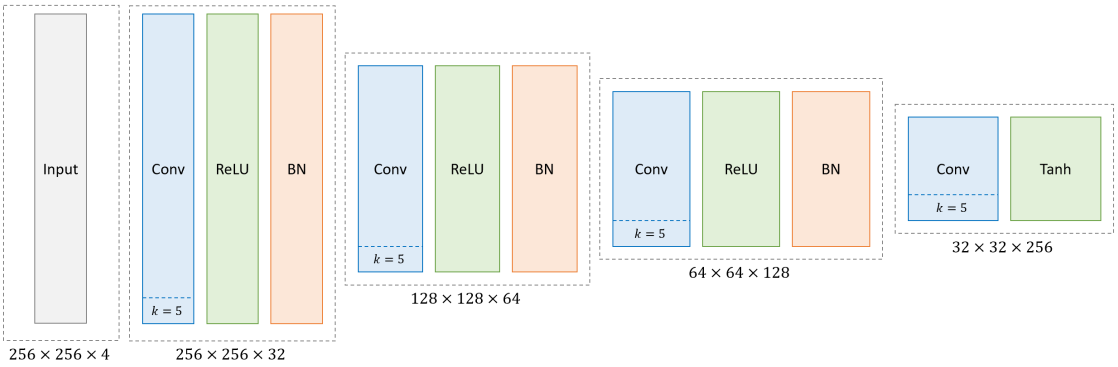


Figure 3.2: Architecture of the Local Encoder. Here, `Input`, `Conv`, `ReLU`, `Tanh`, and `BN` denote input, convolutional, ReLU activation, tanh activation, and BN layers, respectively. The dimensions below a dashed block indicate the shape of each layer inside that block.

3.2.2 Global Encoder

The Global Encoder can be viewed as a modification of the Local Encoder where the spatial downsampling is more aggressive (performed using a stride of $s = 4$) and the last convolutional layer is followed by a fully-connected layer, terminating in a vector with no spatial dimensions. The size of the fully-connected layer is, perhaps counter-intuitively, smaller than the number of channels in the preceding layer in order to streamline training and reduce the memory footprint of the SVBRDF Autoencoder. Observe that each neuron in the first convolutional layer of the Decoder network has direct access to the output of the Global Encoder since the output vector is duplicated across the spatial dimensions of the Local Encoder.

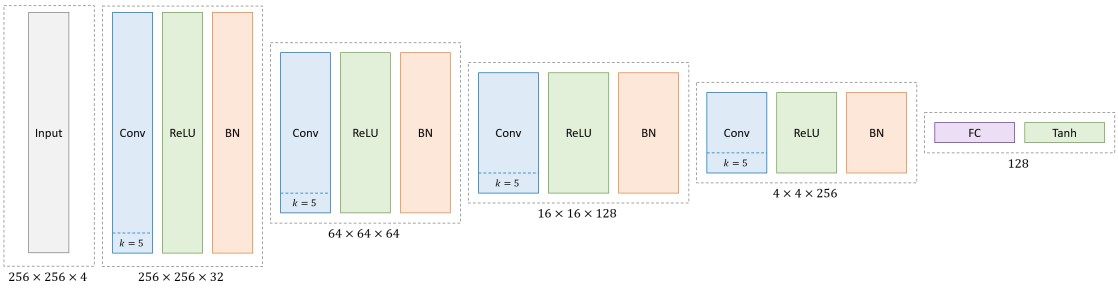


Figure 3.3: Architecture of the Global Encoder. Note that `FC` denotes a fully-connected layer and the rest matches the caption of Figure 3.2.

3.2.3 Periodic Encoder

The Periodic Encoder is a shallow MLP that contains one fully-connected hidden layer with 256 ReLU-activated neurons and a fully-connected output layer with 6 neurons. Unlike the Local Encoder or the Global Encoder, the Periodic Encoder does not directly operate over the input to the SVBRDF Autoencoder. Instead, the Periodic Encoder accepts input directly from the Global Encoder. Observe that the architecture of the Periodic Encoder as well its dependency on the Global Encoder mirrors that of [BJV17]. The output of the Periodic Encoder must be a multiple of 3 because each sinusoidal plane wave has three parameters: the row frequency r_k, the column frequency c_k, and the phase ϕ_k. Row i column j channel k of the periodic latent field $\boldsymbol{z}^{(P)}$ is given by

$$\boldsymbol{z}^{(P)}_{ijk} = \sin(i \cdot r_k + j \cdot c_k + \phi_k) \ . \tag{3.2}$$

The decision to model the periodic elements of the generative model with only two sinusoidal plane waves is motivated by the risk of aliasing described in [BJV17] as well as the results from an experiment which demonstrated that using three waves in the Periodic Encoder increases the reconstruction loss over the validation dataset.

3.2.4 Decoder

The Decoder is a fully-convolutional DNN with an architecture that is essentially the inverse of the Local Encoder. The Decoder first reduces the number of channels in the latent field using a convolutional layer of unit stride $s = 1$ and then proceeds to upsample the spatial extent of each succeeding layer using a fractionally-strided convolution where $s = \frac{1}{2}$. As before, each convolution in the DNN is defined by a $k = 5$ kernel and is followed by a ReLU activation and a BN layer. The only exception is the final convolutional layer which shapes the output of the Decoder to a $256 \times 256 \times 16$ tensor using a sigmoid activation.

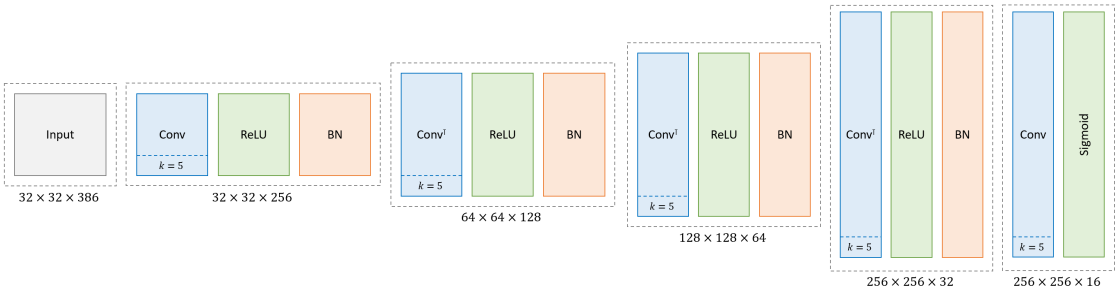


Figure 3.4: Architecture of the Decoder. Note that `Conv⊺` and `Sigmoid` denote transpose convolutional and sigmoid activation layers; the remainder is described in the caption of Figure 3.2.

The significance of the sigmoid activation is that it perfectly matches the range of the Disney BRDF parameters and is sufficient to communicate the direction of the surface normals. In particular, the surface normals can be parameterized using a linearly scaled version of the spherical coordinates (θ, φ) such that both θ and φ fall in the interval $[0, 1)$. Under this scheme, the Cartesian coordinates of a surface normal $\boldsymbol{n} \in \mathbb{R}^3$ are given by

$$\boldsymbol{n} = \begin{bmatrix} \cos(\varphi')\sin(\theta') & \sin(\varphi')\sin(\theta') & \cos(\theta') \end{bmatrix}^\intercal, \quad \theta' = \frac{\pi}{2}\theta, \quad \varphi' = 2\pi\varphi \; . \tag{3.3}$$

Note that a popular alternative to fractionally-strided convolutions are regular convolutions that are preceded by an explicit upsampling procedure which increases the spatial extent of a tensor using nearest-neighbour upsampling or bilinear upsampling [ODO16]. Theoretically, the fractionally-strided convolution is the superior option because it is a generalization of the latter approach; however, fractionally-strided convolutions are also prone to checkerboard-like artefacts that can be avoided with a traditional upsampling method. To this end, a set of preliminary experiments were conducted which offered empirical evidence for the conclusion that bilinear upsampling is inferior to nearest-neighbour upsampling, and both nearest-neighbour and bilinear upsampling resulted in unnatural, oversmoothed output textures.

3.3 Reflectance Models

There are two BRDF models that are relevant to this research project: the Disney BRDF [BS12] and the Substance BRDF [Adob]. The Disney BRDF is a physically-inspired BRDF model developed at Walt Disney Animation Studios and is the parameterization of choice for the SVBRDF Autoencoder output. The Substance BRDF is a microfacet BRDF featured in Substance Designer[5] and is used to render the input to the SVBRDF Autoencoder as well as the ground-truth output textures. Using two different BRDF models emphasizes the flexibility of the proposed approach and decouples the SVBRDF Autoencoder from the dataset. Part of the reason why the Disney BRDF and Substance BRDF were selected is that both of their implementations are publicly available, albeit it in a shader language called GLSL. Nevertheless, their counterparts in the research project implementation were verified by comparing their outputs to the results from the ground-truth shaders for random surface normals, point lights, viewing positions, and parameter values.

3.3.1 Disney BRDF

The Disney BRDF is a principled BRDF model designed for digital artists working on high production value films. The BRDF borrows many concepts from the PBR literature, although the model itself is not physically correct. According to [BS12], the Disney BRDF was founded on several principles, including:

1. The BRDF should favour intuitive parameters over physical ones.
2. The number of BRDF parameters should be kept to a minimum.
3. The range of each BRDF parameter should be $[0, 1]$.
4. All combinations of BRDF parameters should give plausible materials.

Note that principles (2) and (3) make the Disney BRDF an ideal candidate for the SVBRDF Autoencoder. In practice, the BRDF consists of 11 parameters: `baseColour` (RGB), `subsurface`, `metallic`, `specular`, `specularTint`, `roughness`,

[5]Substance Designer is an industry-standard material authoring application.

`anisotropic` (level, angle), `sheen`, `sheenTint`, `clearCoat`, and `clearcoatGloss`, where 2 and 3 values are required to specify `anisotropic` and `baseColour`, respectively. The semantic interpretation of each parameter is illustrated in Figure 3.5.

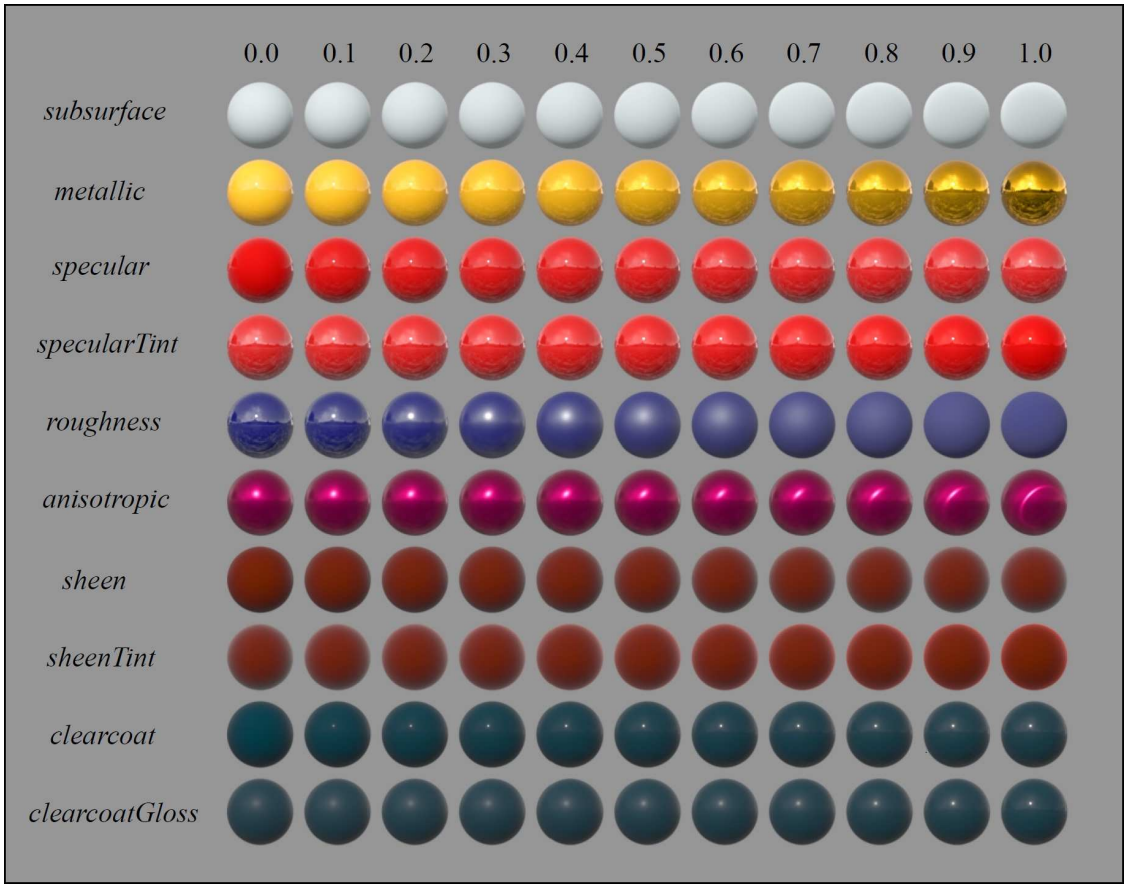


Figure 3.5: Influence of each parameter in the Disney BRDF model. The values of the parameters are varied from 0 (left) to 1 (right) across each row. Image is courtesy of [BS12].

At its core, the Disney BRDF is a composition of five independent BRDF models, each of which is intended to capture a different visual phenomenon. The first of these BRDF models, f_L, is a diffuse Lambertian model that is augmented by a pair of Fresnel refraction terms (since reflected light arising from refraction is attenuated twice at the surface of a medium). Let $\boldsymbol{n}$ denote the surface normal, $\boldsymbol{h}$ the halfway vector between the incident light direction $\boldsymbol{i}$ and outbound light

direction $\boldsymbol{o}$, and let $\boldsymbol{v} \in \{\boldsymbol{i}, \boldsymbol{o}\}$. Furthermore, define

$$F_{D90} = 0.5 + 2\,(\boldsymbol{h} \cdot \boldsymbol{v})^2\,\texttt{roughness} \tag{3.4}$$

$$F_L(\boldsymbol{v}) = 1 + (F_{D90} - 1)(1 - \boldsymbol{n} \cdot \boldsymbol{v})^5\ . \tag{3.5}$$

Observe that $F_{D90} \in [0.5, 2.0]$ is proportional to the intensity of light emitted at grazing angles and scales with the `roughness` parameter. It follows that

$$f_L(\boldsymbol{i}, \boldsymbol{o}) = \frac{\texttt{baseColour}}{\pi}\, F_L(\boldsymbol{i})\, F_L(\boldsymbol{o})\ . \tag{3.6}$$

The second BRDF, f_U, is an approximation of the Hanharan-Kreuger BRDF [HK93] and models the subsurface scattering[6] behaviour responsible for giving materials like wax, jade, and skin their distinctive look near interface boundaries. Let

$$F_U(\boldsymbol{v}) = 1 + ((F_{D90} - 0.5)/2 - 1)(1 - \boldsymbol{n} \cdot \boldsymbol{v})^5\ . \tag{3.7}$$

The subsurface scattering BRDF is then given by

$$f_U(\boldsymbol{i}, \boldsymbol{o}) = \frac{5\,\texttt{baseColour}}{8\pi}\left(F_U(\boldsymbol{i})\, F_U(\boldsymbol{o})\left(\frac{2}{\boldsymbol{n} \cdot \boldsymbol{i} + \boldsymbol{n} \cdot \boldsymbol{o}} - 1\right) + 1\right)\ . \tag{3.8}$$

The third BRDF, f_H, models the sheen of cloth materials using a single Fresnel term. After deriving `sheenColour` from `baseColour`, `sheen`, and `sheenTint`,

$$f_H(\boldsymbol{i}, \boldsymbol{o}) = \texttt{sheenColour}\,(1 - \boldsymbol{h} \cdot \boldsymbol{i})^5\ . \tag{3.9}$$

The two remaining BRDFs are both instances of the microfacet BRDF framework. One of these BRDFs, f_S, represents the base specular contribution of the material and is defined with respect to the $D_{\text{GGX,Anisotropic}}$ and $G_{\text{GGX,Anisotropic}}$ distributions, as well as the Schlick Fresnel term F_{Schlick}. The other BRDF, f_C, represents the clearcoat specular reflection of the material and is defined using the D_{GTR} distribution with $\gamma = 1$, the Schlick Fresnel term F_{Schlick}, and the $G_{\text{GGX,Isotropic}}$ distribution. Now, the full Disney BRDF f_{Disney} is expressed as follows:

$$f_{L,U}(\boldsymbol{i}, \boldsymbol{o}) = f_L(\boldsymbol{i}, \boldsymbol{o}) + \texttt{subsurface}\,(f_U(\boldsymbol{i}, \boldsymbol{o}) - f_L(\boldsymbol{i}, \boldsymbol{o})) \tag{3.10}$$

$$f_{L,U,H}(\boldsymbol{i}, \boldsymbol{o}) = (f_{L,U}(\boldsymbol{i}, \boldsymbol{o}) + f_H(\boldsymbol{i}, \boldsymbol{o}))(1 - \texttt{metallic}) \tag{3.11}$$

$$f_{\text{Disney}}(\boldsymbol{i}, \boldsymbol{o}) = f_{L,U,H}(\boldsymbol{i}, \boldsymbol{o}) + f_S(\boldsymbol{i}, \boldsymbol{o}) + 0.25\, f_C(\boldsymbol{i}, \boldsymbol{o})\,\texttt{clearcoat}\ . \tag{3.12}$$

[6] Subsurface scattering occurs when light is reflected off a surface in such a way that the location where the light entered the surface differs from the location where the light exited the surface.

Note that there are several magic constants sprinkled throughout the reference Disney BRDF implementation which do not appear to be grounded in PBR principles, as hinted by Equation 3.8 and Equation 3.12.

3.3.2 Substance BRDF

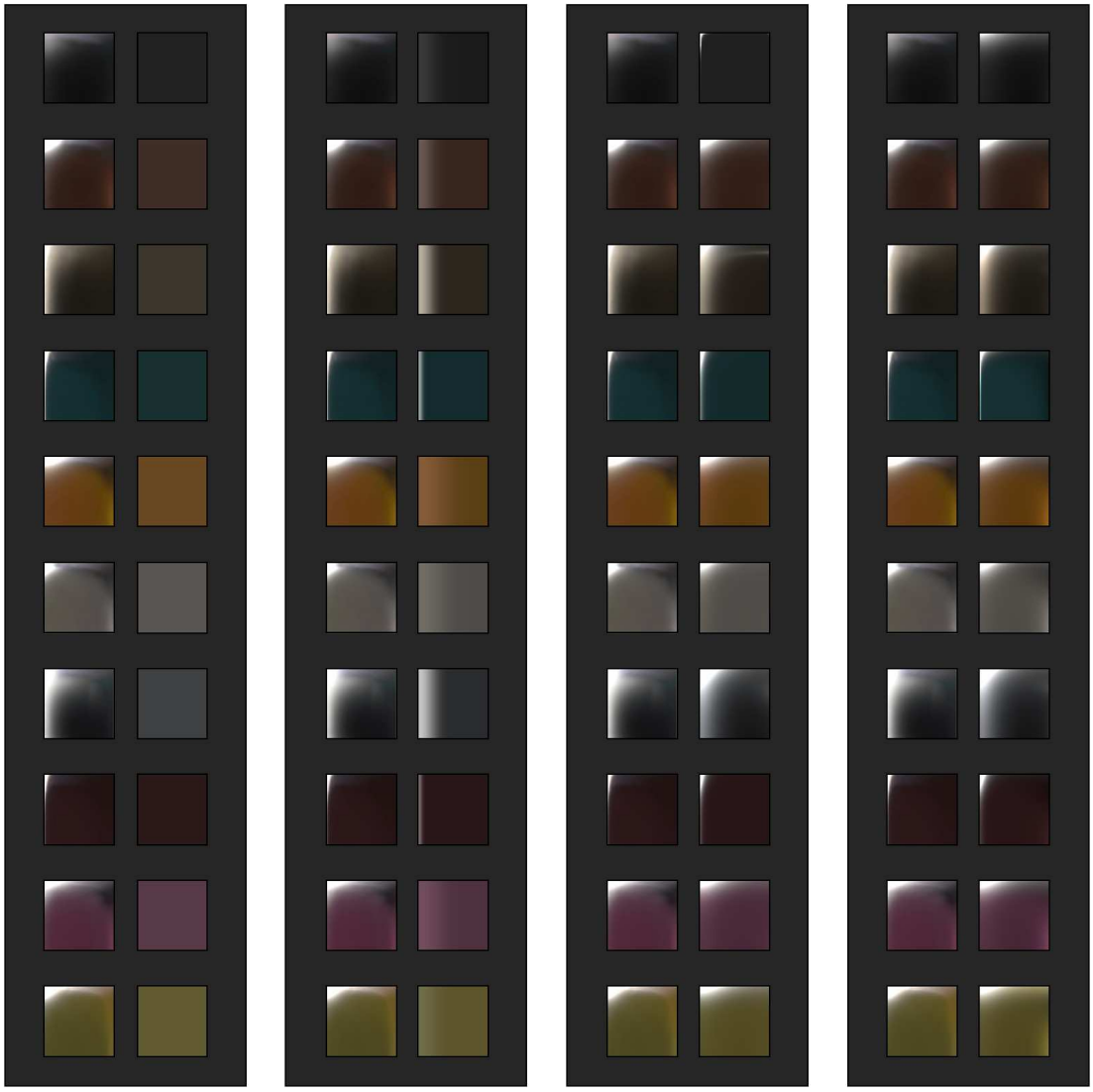
The Substance BRDF is based on the anisotropic *physically_specular_glossiness* shader from Substance Designer. Unlike the Disney BRDF, the Substance BRDF is a physically plausible BRDF and was primarily chosen because of its tight integration with the dataset described in Section 3.5. The Substance BRDF is a combination of a Lambertian BRDF and a microfacet BRDF that is parameterized by 9 values denoting the `diffuseColour` (RGB), `specularColour` (RGB), `glossiness`, `anisotropyLevel`, and `anisotropyAngle` of the material. As usual, the `diffuseColour` and `specularColour` parameters just scale the contributions of the Lambertian BRDF and microfacet BRDF, respectively. Note that the microfacet BRDF is instantiated using the $D_{\text{GGX,Anisotropic}}$, $F_{\text{Schlick,SG}}$, and $G_{\text{GGX,Anisotropic}}$ distributions. The roughness parameters passed to the anisotropic GGX distributions are derived as follows:

$$\alpha_x = 1 - \texttt{glossiness}, \qquad \alpha_y = \frac{\alpha_x}{\sqrt{1 - \texttt{anisotropyLevel}}} \ . \tag{3.13}$$

To avoid numeric instability, the `glossiness` and `anisotropyLevel` parameters are clamped to the range $[\varepsilon, 1 - \varepsilon]$ for some small ε.

Qualitative Comparison

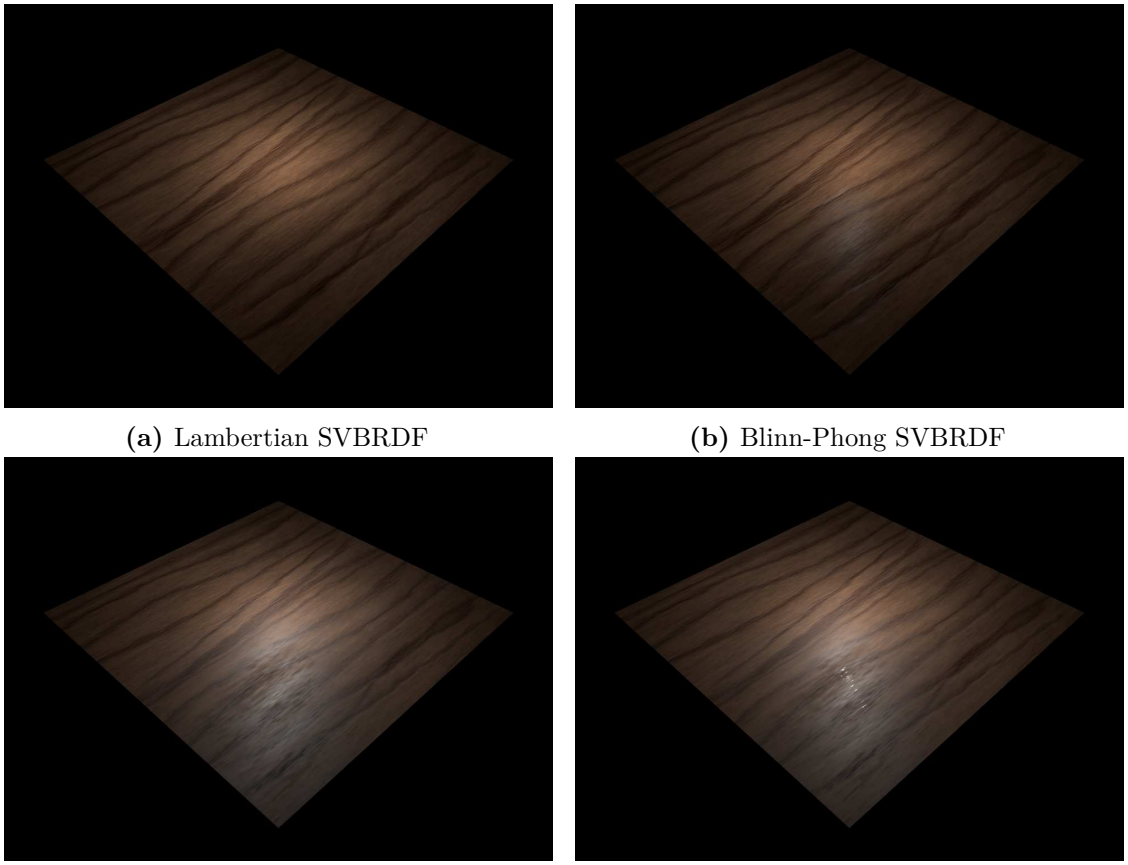
To appreciate the expressive power of the Substance and Disney reflectance models, consider Figure 3.6. Observe that the Lambertian BRDF only captures the average diffuse colour of a material while the Blinn-Phong BRDF fails to model the Fresnel effect due to its limited parameterization. In contrast, the Substance BRDF competently matches the ground-truth BRDF near $\theta_h = 0$ but loses precision as the halfway vector moves away from the normal. Finally, the Disney BRDF offers a slight improvement over the Substance BRDF by accounting for the subtle *grazing retroreflection* effect near the bottom-right corner of each BRDF slice.



(a) Lambertian BRDF **(b)** Blinn-Phong BRDF **(c)** Substance BRDF **(d)** Disney BRDF

Figure 3.6: BRDF slices obtained by fitting Lambertian, Blinn-Phong, Substance, and Disney reflectance models to 10 materials from the MERL 100 dataset using the L-BFGS-B optimization algorithm [BLNZ95]. The left column in each subfigure depicts the ground-truth BRDF slice while the right column displays the best fit of the corresponding model. See Appendix A.1 for the optimization results over the entire dataset.

The representational capacities of these models are further distinguished in Figure 3.7. Here, the Lambertian SVBRDF does not convey any specular highlights and the Blinn-Phong SVBRDF gives the impression of an extremely smooth surface. On the other hand, the Substance SVBRDF and Disney SVBRDF depict an authentic roughness along the grains of the wood and are generally quite similar with the subtle exception of the clearcoat gloss.



(a) Lambertian SVBRDF **(b)** Blinn-Phong SVBRDF

(c) Substance SVBRDF **(d)** Disney SVBRDF

Figure 3.7: Portrayals of a wood texture under the Lambertian, Blinn-Phong, Substance, and Disney reflectance models. The Disney rendering was generated using improvised parameters where ground-truth values were not available. Texture is courtesy of [All15b].

3.4 Loss Functions

The final revision of the SVBRDF Autoencoder was optimized with respect to the reconstruction loss function displayed in Equation 3.1; however, earlier iterations also featured a *diversity loss* intended to expand the effective domain of the Decoder. Specifically, the diversity loss aimed to train the Decoder to synthesize a plausible texture in the style encoded by a global latent field from an arbitrary local latent field. In theory, this would improve the interpolation performance of the SVBRDF Autoencoder and enable the synthesis of diverse textures by randomly sampling the local latent field. Unfortunately, an extensive suite of preliminary experiments revealed that the diversity loss and reconstruction loss were mutually incompatible: one usually dominated the other and the compromises tended to satisfy neither

objective. As a result, the diversity loss was discarded from the training of the final model. For the remainder of this section, let $\mathcal{I}^{(1)} \in \mathbb{R}^{256\times256\times3}$ be a rendering of a predicted SVBRDF texture and let $\mathcal{I}^{(2)} \in \mathbb{R}^{256\times256\times3}$ be the rendering of the associated ground-truth SVBRDF texture.

3.4.1 Texel Loss

The texel loss $\mathcal{L}_T$ is taken to be the L1 loss between two textures:

$$\mathcal{L}_T(\mathcal{I}^{(1)}, \mathcal{I}^{(2)}) = \frac{1}{256 \cdot 256 \cdot 3} \sum_{i=1}^{256} \sum_{j=1}^{256} \sum_{k=1}^{3} |\mathcal{I}^{(1)}_{ijk} - \mathcal{I}^{(2)}_{ijk}| \ . \tag{3.14}$$

The L2 loss was also considered, but a preliminary experiment showed that the SVBRDF Autoencoder converged more quickly on a validation texture using $\mathcal{L} = \mathcal{L}_{\text{L1}}$ in comparison to $\mathcal{L} = \mathcal{L}_{\text{L2}}$. In fact, the same results were obtained for the style and content loss functions which is the reason they are also implemented in terms of an L1 loss rather than an L2 loss. Moreover, another preliminary experiment determined that converting $\mathcal{I}^{(1)}$ and $\mathcal{I}^{(2)}$ into the sRGB colour space prior to evaluating the texel loss did not have a significant impact on the convergence of the model.

3.4.2 Style Loss

The style loss $\mathcal{L}_S$ is based on the L1 loss between the normalized Gram matrices of the `relu1_1`, `relu2_1`, `relu3_1`, `relu4_1`, and `relu5_1` layers in the VGG-19 network. To be precise, if $G^{(k)}(\ell)$ is the normalized Gram matrix constructed by applying Equation 2.131 to the activations of layer ℓ for an input image $\mathcal{I}^{(k)}$, then

$$\mathcal{L}_T(\mathcal{I}^{(1)}, \mathcal{I}^{(2)}) = \frac{1}{|L|} \sum_{\ell \in L} \mathcal{L}_{\text{L1}}(G^{(1)}(\ell), G^{(2)}(\ell)) \ . \tag{3.15}$$

Above, $L = \{1, 6, 11, 20, 29\}$ is the set of indices in the VGG-19 network corresponding to the ReLU layers listed in the previous paragraph. This choice of L ranks among the most popular for neural style loss functions integrated with the VGG-19 network [LFY+17, ULVL16, YBS+19]; replacing these layers with their equivalent (pre-activation) convolutional layers did not enhance the performance of the SVBRDF Autoencoder in preliminary experiments. Note that the instantiation

of the VGG-19 network used to compute the style loss is pre-trained on the ImageNet dataset [DDS+09]. Consequently, the VGG-19 network expects its inputs to be pre-processed in a certain way, namely by expressing each pixel in terms of its standardized difference from the mean of the corresponding colour channel in the ImageNet dataset. Strangely, applying this transformation to the $\mathcal{I}^{(1)}$ and $\mathcal{I}^{(2)}$ images before feeding them through the VGG-19 network caused the SVBRDF Autoencoder to produce incoherent swirling textures, as shown in Figure 3.8. It follows that this pre-processing step was subsequently ignored in the training of SVBRDF Autoencoder models.

(a) Input Texture **(b)** Expected Output **(c)** Predicted Output

Figure 3.8: Pre-processing the input to the VGG-19 network degenerates the performance of the SVBRDF Autoencoder. Figure 3.8c is the output after 4000 training iterations.

3.4.3 Content Loss

The content loss $\mathcal{L}_C$ is calculated by taking the L1 loss directly over the feature maps in the `relu4_2` layer of the VGG-19 network. Let $\boldsymbol{X}^{(n)} \in \mathbb{R}^{H \times W \times C}$ denote the activations in this layer for an image $\mathcal{I}^{(n)}$. Then,

$$\mathcal{L}_C(\mathcal{I}^{(1)}, \mathcal{I}^{(2)}) = \frac{1}{HWC} \sum_{i=1}^{H} \sum_{j=1}^{W} \sum_{k=1}^{C} |\boldsymbol{X}_{ijk}^{(1)} - \boldsymbol{X}_{ijk}^{(2)}| \ . \tag{3.16}$$

Akin to the style loss, moving the targeted activations from the `relu4_2` layer to the `conv4_2` layer did not yield promising results in the preliminary experiments.

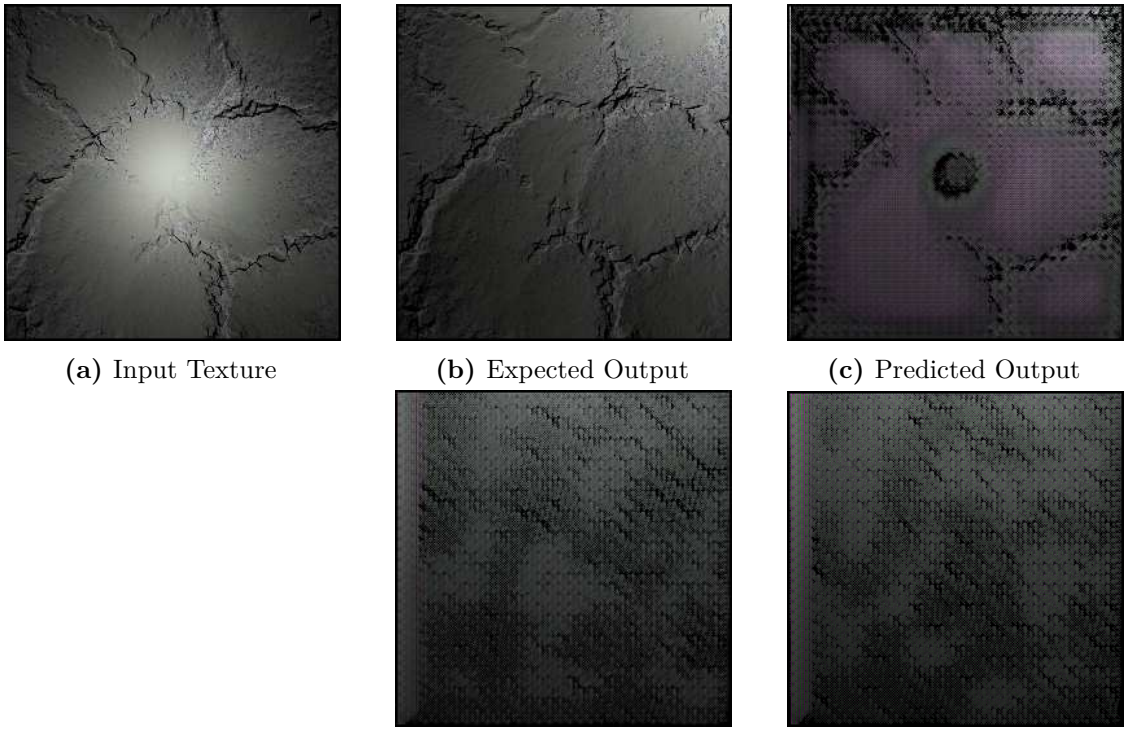
3.4.4 Diversity Loss

The idea of integrating a diversity loss into the training regime of the SVBRDF Autoencoder came from [LFY+17]. In this work, a GAN was trained to generate

variations of an input texture by feeding batches of the same input texture with different noise tensors through the generator DNN and then applying a loss function that encouraged the outputs to have a high neural content loss between themselves. Similarly, most variants of the diversity loss function for the SVBRDF Autoencoder operated by feeding an input image through the Encoder part of the DNN to obtain a latent field, making several copies of this latent field and replacing the local latent field in each copy with a sample from a uniform distribution, and then running the modified latent fields through the Decoder. The outputs of the batch could then be compared among themselves and the original ground-truth SVBRDF texture.

Unsuccessful Formulations

Initially, the diversity loss function included a positively-weighted style loss between each predicted texture and the ground-truth texture along with a negatively-weighted content loss between each predicted texture and the next one in the batch (wrapping as needed). When that approach failed, the style loss was swapped with a content loss which effectively modelled the ground-truth texture as a positively-charged particle and each of the predicted textures as a negatively-charged particle. Afterwards, the diversity loss took the form of a negatively-weighted reconstruction loss between each of the predicted SVBRDF textures. No matter how the hyperparameters were tuned, the predicted textures in the preliminary experiments depicted either a bland, featureless texture or a corrupted image. Changing the diversity loss to one that promotes the reconstruction of the ground-truth texture but demotes similarity between the surface normals of the predicted textures did not yield any promising results. Finally, in an attempt to improve the coherence of the transitions between adjacent latent tiles, the local latent field was split into quadrants and permuted (rather than randomly sampled anew) and the resulting SVBRDF Autoencoder output was judged by its style in comparison to the ground-truth texture. Needless to say, this did not significantly change the tiling performance of the SVBRDF Autoencoder relative to a model trained without a diversity loss term.



(a) Input Texture **(b)** Expected Output **(c)** Predicted Output

(d) Derived Texture 1 **(e)** Derived Texture 2

Figure 3.9: A diversity loss enables the sampling of interesting textures at the expense of reconstruction performance. Figure 3.9a through Figure 3.9c demonstrate the reconstruction capabilities of the SVBRDF Autoencoder after 4000 iterations while Figure 3.9d and Figure 3.9e show two textures derived from the global latent field of Figure 3.9a.

3.5 Dataset

The SVBRDF Autoencoder was trained using a curated dataset of 35 textures from the Substance Share[7] platform [Adob]. Substance Share offers a free selection of over 1500 procedural SVBRDF textures that are catalogued by type and supported by a community review system. Crucially, Substance Share also has a precedent in the texture synthesis literature [DAD+18, HDR19]. To reduce bias and standardize the quality of the textures used to train the SVBRDF Autoencoder, only the materials authored by Allegorithmic (i.e. the original developers of Substance Share) were considered for the dataset. Overall, textures from seven categories were collected: Earth, Fabric, Industry, Metal, Rock, Urban, and Wood. Each category contains precisely five textures; three textures from each category formed part of

[7]Substance Share is an online marketplace for sharing digital assets, such as SVBRDF textures, created in Substance Designer or Substance Painter.

the training dataset while the remaining two were donated to the validation and testing datasets, resulting in a 60:20:20 split among the training, validation, and testing datasets. The textures were partitioned such that each dataset contained at least one stochastic, regular, diffuse, and specular texture.

Figure 3.10: Random sample from each texture in the training, validation, and testing datasets. The texture in the top row of the validation dataset is the *Earth - Dry Mud* texture discussed in Section 3.6.1. Links to each texture can be found in Appendix C.

Observe from Figure 3.11 that a texture is represented by a set of 1024×1024 images denoting the surface normal, diffuse colour, specular colour, and glossiness at each point of a texture. The anisotropy level and angle required to complete the Substance BRDF parameterization are not included because they are not considered

to be part of the core material definition in Substance Designer and are therefore missing from the designs on Substance Share.

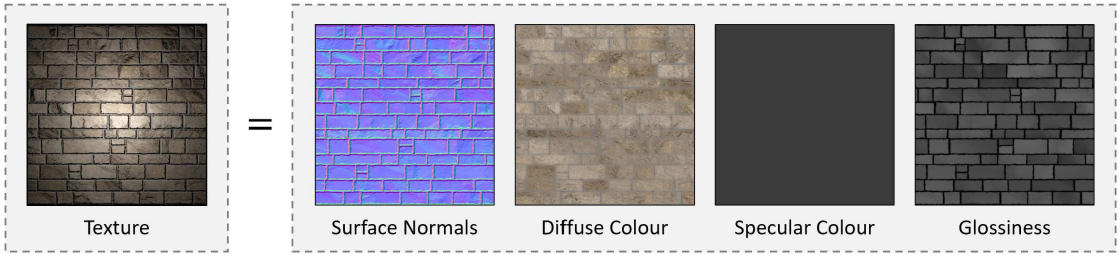
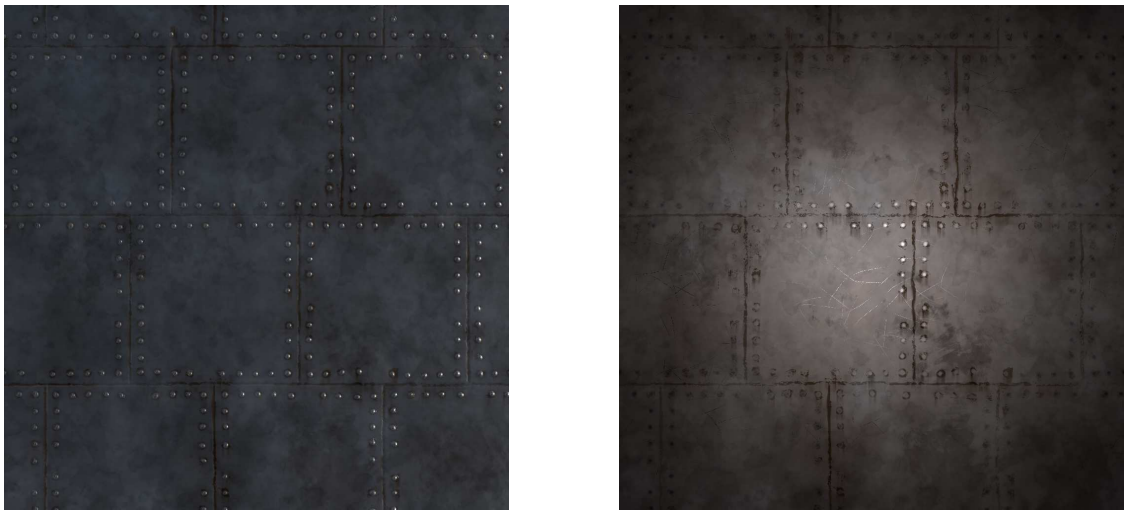


Figure 3.11: Substance Designer textures are composed of spatially-varying normal, diffuse colour, specular colour, and glossiness maps. Texture is courtesy of [All15a].

3.5.1 Data Augmentation

Recall that the SVBRDF Autoencoder accepts an input of size $256 \times 256 \times 4$ while each dataset texture has a spatial extent of 1024×1024. To make the dimensions of the dataset textures compatible with those expected by the DNN, the textures are cropped to a random 256×256 region each time they are accessed. This is preferable to simply downscaling the textures since it increases the diversity of the training inputs and prevents the SVBRDF Autoencoder from simply memorizing the training dataset. To this end, each 256×256 crop is, with probability $\frac{1}{4}$, rotated by 90, 180, or 270 degrees, and with the same probability, reflected horizontally or vertically (or both). Clearly, these data augmentation procedures assume that the input textures are valid regardless of their orientation. Furthermore, a random anisotropy level and anisotropy angle are applied in a uniform manner across each SVBRDF texture crop to expose the DNN to anisotropic highlights. Note that the $256 \times 256 \times 3$ textures that are spliced with an RDF to form an SVBRDF Autoencoder input are generated by rendering their corresponding (cropped, rotated, reflected, and angled) SVBRDF texture under a point light that is placed above the center of the texture with respect a virtual camera near the same position as the light. The elevation and intensity of the point light are varied between each training iteration to discourage the DNN from assuming that the spatial influence of a flash is constant. The

SVBRDF textures are not rendered under environmental illumination since this may introduce a bias in shiny materials like brushed aluminum.



(a) Environmental Lighting **(b)** Distant Point Light

Figure 3.12: Environmental lighting can bias the rendering of surfaces with high specularity. Observe that Figure 3.12a exhibits a blue tint relative to Figure 3.12a from the sky in the environment map depicted in Figure 2.12.

3.6 Training

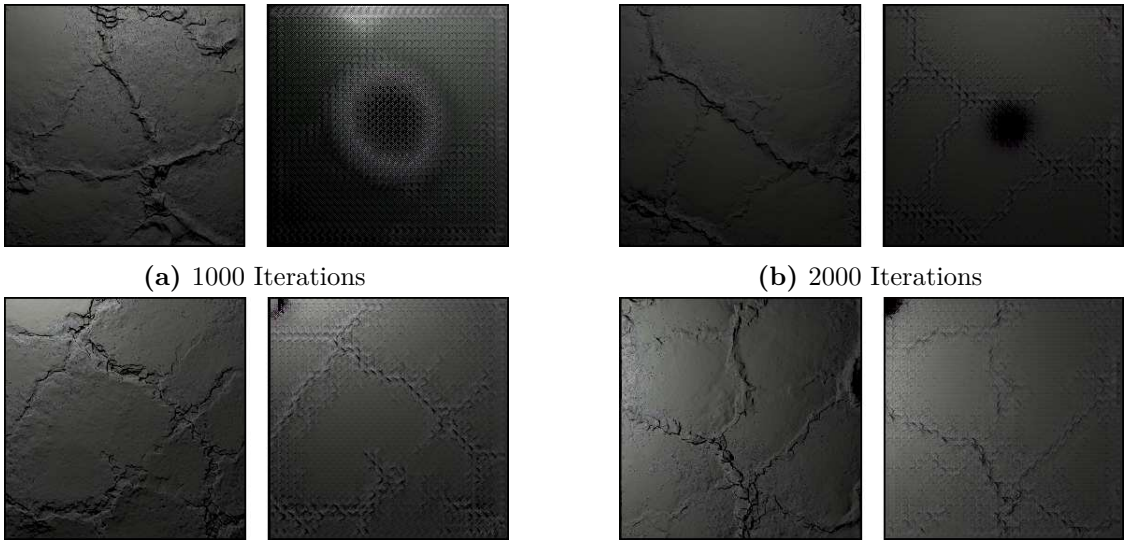
Before training is officially under way, the parameters of a new SVBRDF Autoencoder instance are initialized by sampling its weights from the normal distribution $\mathcal{N}(0, (0.02)^2)$ and fixing its biases to a constant 0.01, as proposed by [BJV17, JBV16, RMC15, ZZB+18]. The values of these parameters are then optimized using Adam with the incremental training approach described in Section 2.3.6. Here, a new input texture is introduced to the SVBRDF Autoencoder every epoch (1000 iterations) until the entire training corpus is in circulation. An iteration of the training loop consists of selecting an (active) texture in a round-robin fashion, constructing a mini-batch from several samples of that texture, evaluating the loss function over the mini-batch, computing the gradient of each SVBRDF Autoencoder parameter using the average loss, and then updating the parameters with the Adam optimizer. After each training step, the value of the loss function is recorded in a log file and monitored on a dashboard. Occasionally, the state of the

SVBRDF Autoencoder is also saved to disk for later investigation. Early stopping is implemented but not used due to the high variance in the training loss observed for each texture. The learning rate of the Adam optimizer is held constant throughout the entire training regime since there does not appear to be a consensus on the parameters of learning rate decay in texture synthesis applications of deep learning.

3.6.1 Preliminary Experiments

Limited computational resources were available to tune the hyperparameters of both the SVBRDF Autoencoder and the training process. This included selecting an appropriate architecture for the SVBRDF Autoencoder, picking the right combination of loss functions to shape the optimization of the model, and finding the best training parameters in terms of both convergence speed and quality. To cope with this demand in a principled way, most hyperparameters were tuned based on their performance after 4000 iterations of training on the *Earth - Dry Mud* texture from the validation dataset. This texture was chosen based on its membership to the validation dataset[8] and the intricate structure of its surface normals. Furthermore, the training only proceeds for 4000 iterations because the performance of the SVBRDF Autoencoder has usually stabilized by that point. As illustrated in Figure 3.13, the SVBRDF Autoencoder learns to compensate for the flash in the center of the input texture over the course of about 2000 iterations before devoting modelling resources to finer texture details. Experiments performed under the conditions described in this passage are termed *perliminary experiments* and were used to derive an initial estimate for the hyperparameter values.

[8]Picking a texture from the training dataset biases the training and picking a texture from the testing dataset biases the final performance evaluation.



(a) 1000 Iterations **(b)** 2000 Iterations

(c) 3000 Iterations **(d)** 4000 Iterations

Figure 3.13: Convergence of the *Earth - Dry Mud* texture. The left image of each subfigure pair is the expected output while the right image is the predicted output.

3.6.2 Validation Experiments

After the preliminary experiments concluded, twelve *validation experiments* were conducted to explore how deviations from the initial hyperparameter estimates would affect performance over the validation dataset. The first two validation experiments involved training an SVBRDF Autoencoder with the base hyperparameter values over 16000 iterations. The difference between the experiments was the number of training iterations between successive introductions of new textures: 1000 or 2000. Comparing the (exponentially-averaged) loss function values at the conclusion of the two runs demonstrated that, for optimal convergence, an epoch should encompass 1000 training iterations. From there, six validation experiments were performed by independently decreasing or increasing each of the λ_T, λ_S, and λ_C loss function weights. The learning rate applied to the Adam optimizer was also raised or lowered in two separate validation experiments. The final two validation experiments investigated the impact of using an explicit upsampling layer in the Decoder rather than a fractionally-strided convolution, as well as widening the output of the Periodic Encoder to model three waves. Any modifications to the hyperparameters

that improved the performance of the SVBRDF Autoencoder with respect to the unscaled loss functions was integrated into the training of the final model.

3.7 Implementation

The bulk of this research project is implemented in Python 3. The decision to use Python 3 was partially motivated by its reputation as a portable and productive language; however, the most compelling argument was its extensive ecosystem of scientific computing libraries, including PyTorch [PGM+19] and SciPy [JOP+]. PyTorch is a machine learning framework that offers a useful set of abstractions for accelerating linear algebra operations and optimizing machine learning models. One particularly convenient feature of PyTorch is automatic differentiation which trivializes the backpropagation of gradients through tensor operations, such as those found in the SVBRDF Autoencoder and the rendering layer. PyTorch also boasts a close integration with TensorBoard [AAB+15]: a dashboard that is handy for visualizing the training progress of a machine learning model. On the other hand, the scope of SciPy was primarily limited to the experiment shown in Figure 3.6, although it plays a more central role in the implementation of the classical texture synthesis algorithms described in Section 2.3.3 included in appendix B.X. Note that all of the accompanying Python 3 source code is meticulously documented and annotated with types where possible; the accompanying source code includes HTML documentation generated by pdoc [Pdo].

3.7.1 Usage

All of the training and experimental flows featured throughout this research project are accessible through a simple command-line interface. For example, to train an SVBRDF Autoencoder using the `configs/training.yaml` configuration with CUDA (i.e. GPU) acceleration, it suffices to run

```
python main.py -flow training -config configs/training.yaml -cuda
```

Each execution flow is supported by a YAML configuration file that, depending on the flow, may specify anything from the lighting conditions of a texture to the contents of a dataset to the architecture of the SVBRDF Autoencoder itself. This simplifies the process of making adjustments to hyperparameters, facilitates automation, and avoids bloating the command-line interface. An example of a configuration file is given in Appendix B.2.

3.7.2 Environment

The SVBRDF Autoencoder was trained using an i7-6700 CPU and an NVIDIA GTX 1070 GPU equipped with 8GB of virtual GPU memory. Although the GPU was able to accelerate the training progress of the DNN by an order of magnitude, it also limited the maximum size of the mini-batches to 8 samples. On average, 30 minutes were required to cycle through 1000 training iterations (i.e. one epoch). The training of the final SVBRDF Autoencoder model, consisting of 210 000 training iterations, took approximately 100 hours to complete.

4
Results

Contents

This chapter discusses the findings of the research project. First, Section 4.1 highlights the results of the validation experiments and documents the training progress of the final SVBRDF Autoencoder model. Afterwards, Section 4.2, Section 4.3, and Section 4.4 explore the texture reconstruction, interpolation, and expansion performance of the trained model, respectively. The goal of these experiments is to evaluate the utility of the final model and gain insight into the advantages and shortcomings of the proposed approach.

4.1 Training

The preliminary experiments conducted on the *Earth - Dry Mud* texture moulded the architecture of the SVBRDF Autoencoder discussed in Section 3.2 and suggested that $\lambda_C = 300$, $\lambda_S = 1$, and $\lambda_T = 200$ may be an optimal combination of loss function weights. The experiments also revealed that the Adam optimizer performs

well using a learning rate of $\lambda_{LR} = 2 \times 10^{-4}$ and moment hyperparameters of $\beta_1 = 0.9$ and $\beta_2 = 0.999$. Note that this particular Adam parameterization is similar to many other works in the field [DLT+20, DAD+18, LW16]. Altogether, these hyperparameter estimates constitute the base SVBRDF Autoencoder model and form a baseline for the validation experiments.

4.1.1 Validation Experiments

The results of the validation experiments are listed in Table 4.1. Unless explicitly stated otherwise, each model was trained for 16 000 iterations using an incremental training scheme where a new texture from the validation dataset is introduced every 1000 iterations. The hyperparameter values chosen for λ_C, λ_S, and λ_T (in addition to λ_{LR}) are based on their relative influence over the qualitative performance of the SVBRDF Autoencoder during preliminary experiments.

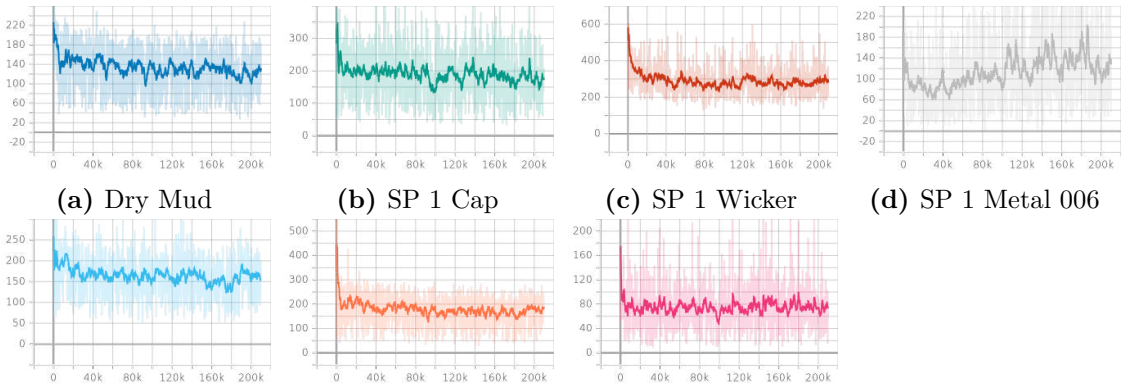
Model Variation	Average Loss			Accepted
	Content	Style	Texel	
Base (1000 steps/epoch)	0.1434	56.84	0.03904	✓
Base (2000 steps/epoch)	0.1594	68.90	0.04174	-
$\lambda_C = 400$	0.1387	55.90	0.03877	✓
$\lambda_C = 200$	0.1508	58.28	0.03744	-
$\lambda_S = 2$	0.1555	58.91	0.03959	-
$\lambda_S = 0.5$	**0.1339**	56.53	0.03632	✓
$\lambda_T = 400$	0.1421	**55.07**	0.02969	✓
$\lambda_T = 100$	0.1424	55.46	0.03775	-
$\lambda_{LR} = 1 \times 10^{-3}$	0.1531	61.49	**0.02747**	-
$\lambda_{LR} = 1 \times 10^{-5}$	0.1880	81.27	0.06044	-
Three Periodic Waves	0.1456	57.02	0.03680	-
Explicit Upsampling	0.1690	69.79	0.02952	-

Table 4.1: Results of the validation experiments. The *Average Loss* column denotes the final loss over the validation dataset; the *Accepted* column indicates whether the change to the base SVBRDF Autoencoder model was accepted into the final model. Furthermore, underlined values are improvements over the corresponding loss of the accepted base model; **bold** values are the smallest values observed for a loss function.

Observe that a change to the base model is only accepted if it improves at least two of the three reconstruction losses; however, it is possible for two mutually-exclusive modifications to yield desirable outcomes (i.e. decreasing or increasing λ_T). Fortunately, the performance of the $\lambda_T = 400$ model is strictly better than the $\lambda_T = 100$ model, so a formalization of the selection criterion in such circumstances is not necessary. The overall loss function is not included in the decision rule because its value depends on the scale of the loss function weights. Consequently, the results in Table 4.1 imply that the final SVBRDF Autoencoder model should set $\lambda_C = 400$, $\lambda_S = 0.5$, and $\lambda_T = 400$, and adopt an incremental training scheme with 1000 iterations between introductions of new training textures. One surprising outcome of the validation experiments is that increasing the weight of one loss function may leave the greatest impact on a different loss function.

4.1.2 Final Model

The final SVBRDF Autoencoder model was trained for 210 000 iterations over the course of four days. The number of training iterations was derived by setting the number of training epochs to $10\,|\mathcal{D}^{(T)}|$ where $|\mathcal{D}^{(T)}|$ is the number of textures in the training dataset. Early stopping conditions were ignored due to the large variance in the generalization error displayed in Figure 4.1.



(a) Dry Mud **(b)** SP 1 Cap **(c)** SP 1 Wicker **(d)** SP 1 Metal 006

(e) Stone Tiles 001 **(f)** Concrete 085 **(g)** Wenge

Figure 4.1: Generalization error of the SVBRDF Autoencoder over the validation dataset. In each subfigure, the horizontal axis is the training iteration, the vertical axis is the value of total loss function (note the differing scales), and the dark trend line is the exponentially-weighted average loss function value (where $\alpha = 0.9$).

The erratic behaviour of the generalization error can partially be explained by Table 4.2 and Table 4.3 which display the weighted loss function values over the training and validation datasets, respectively. Clearly, the content loss dominates its style and texel loss counterparts. Consequently, the extreme variance of the total loss may be caused by instability in the content loss calculation.

Training Dataset Texture	**Weighted Average Loss**			
	Content	**Style**	**Texel**	**Total**
Earth - SP 1 Mud	49.99	3.26	14.24	67.48
Earth - Forest Ground 001	143.58	15.80	13.88	173.26
Earth - SP 1 Fur Cow Long	77.24	6.56	10.99	94.80
Fabric - Camouflage 001	55.72	3.99	9.65	69.37
Fabric - Fabric Suit Vintage	120.50	19.58	12.05	152.13
Fabric - SP 1 Fabric Silk	39.58	3.67	11.49	54.73
Industry - Carbon Fiber	42.27	3.28	12.35	57.91
Industry - SP 1 Cardboard	61.21	4.84	11.41	77.46
Industry - SP 1 Jeans	138.32	25.28	16.88	180.48
Metal - Metal Panels 001	36.87	3.59	16.99	57.45
Metal - SP 1 Metal Plate 011	56.47	5.69	19.74	81.90
Metal - SP 1 Metal Steel Brushed	66.74	7.79	12.82	87.36
Rock - Granite 001	122.21	10.48	20.36	153.05
Rock - Rock 012 Bitmap	144.81	15.69	21.45	181.95
Rock - Stacked Rectangular Stones	134.25	12.77	13.01	160.02
Urban - Classic Brown Concrete	107.84	11.11	11.99	130.93
Urban - Garden Tiles 001 Bitmap	121.50	11.58	27.39	160.46
Urban - SP 1 Concrete 011	154.98	17.61	24.33	196.92
Wood - SP 1 Bark Black Pine	110.40	8.89	10.62	129.91
Wood - SP 1 Old Painted Planks	133.27	15.61	12.06	160.94
Wood - Wood 024 Walnut	27.60	2.13	9.92	39.65

Table 4.2: Weighted loss values of the SVBRDF Autoencoder model over the training dataset. Each loss is averaged over 100 random samples of the corresponding texture.

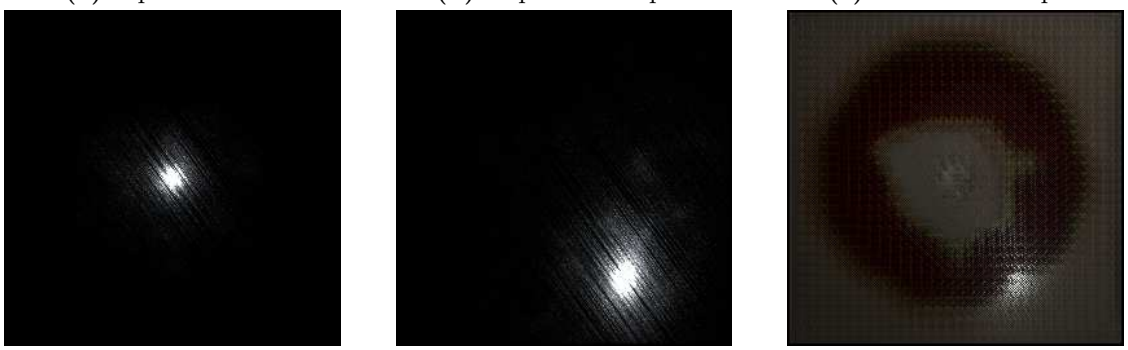
Interestingly, the *Fabric - SP 1 Fabric Silk* texture and all the metal textures are characterized by relatively small total loss values in the training dataset despite having the worst qualitative reconstruction performance. This implies

that weaknesses in the final SVBRDF Autoencoder model are more likely to be the product of suboptimal training procedures or inadequate perceptual loss functions rather than deficiencies in the ML model itself. Sample reconstructions of these textures are pictured in Figure 4.2.

Validation Dataset Texture	Weighted Average Loss			
	Content	Style	Texel	Total
Earth - Dry Mud	81.78	5.41	8.02	95.21
Fabric - SP 1 Cap	84.61	9.36	11.83	105.80
Industry - SP 1 Wicker	121.78	16.90	23.39	162.07
Metal - SP 1 Metal 006	42.73	5.12	17.69	65.54
Rock - Stone Tiles 001	101.19	6.34	10.62	118.15
Urban - Concrete 085	80.94	7.52	22.93	111.39
Wood - Wenge	34.81	2.79	13.29	50.89

Table 4.3: Weighted loss values of the SVBRDF Autoencoder model over the validation dataset. For more details about the loss value computations, see the caption of Table 4.2.

(a) Input Texture **(b)** Expected Output **(c)** Predicted Output



(d) Input Texture **(e)** Expected Output **(f)** Predicted Output

Figure 4.2: Reconstructions of the *Fabric - SP 1 Fabric Silk* and *Metal - SP 1 Metal Steel Brushed* textures after 210 000 training iterations. Textures that lack distinctive patterns are not faithfully represented by the content or style loss functions.

Testing Dataset Texture	Weighted Average Loss			
	Content	**Style**	**Texel**	**Total**
Earth - SP 1 Fur Cow Short	67.85	5.63	13.07	86.55
Fabric - SP 1 Fabric Wool Fluffy	42.52	2.62	7.26	52.39
Industry - SP 1 Backpack Padding	49.24	3.44	13.74	66.43
Metal - SP 1 Metal Dumpster	25.02	1.94	10.91	37.87
Rock - Ice 001 Bitmap	67.83	5.78	13.87	87.48
Urban - Concrete 063	101.57	10.95	11.93	124.45
Wood - SP 1 Wood Board 1	62.83	5.04	9.43	77.30

Table 4.4: Weighted loss values of the SVBRDF Autoencoder model over the testing dataset. For more details about the loss value computations, see the caption of Table 4.2.

Finally, it is worth pointing out that the distribution of the total loss function values is greater, on average, in the training dataset compared to the validation or testing datasets, although the corresponding standard deviations imply that this difference is not significant. Therefore, if the final SVBRDF Autoencoder model does not generalize well to new data, the root issue is unlikely to be associated with overfitting on the training dataset.

Dataset	**Mean**	**Variance**
Training	117.53	49.83^2
Validation	101.29	33.71^2
Testing	76.07	25.90^2

Table 4.5: Total loss statistics across Table 4.2, Table 4.3, and Table 4.4.

4.2 Reconstruction Experiments

The main objective of the SVBRDF Autoencoder is to reconstruct an input texture under novel lighting conditions. It follows that the collective goal of the *reconstruction experiments* is to analyze the proficiency of the final SVBRDF Autoencoder model at precisely this task.

4.2.1 Training Textures

The first suite of reconstruction experiments aims to relight textures from the training dataset. It is reasonable to expect the final SVBRDF Autoencoder model to excel here given that the loss functions model this setup exactly. The best reconstruction results belong to the *Wood - SP 1 Old Painted Planks* texture and are showcased in Figure 4.3. Observe that the major patterns of the flaking wood in the input texture are preserved in the predicted texture although the base colour is lacking some contrast.

(a) Input Texture **(b)** Reconstructed Textures

Figure 4.3: Reconstruction results for the *Wood - SP 1 Old Painted Planks* texture. Figure 4.3b renders the predicted texture under different point lights.

This discolouration is strongly emphasized in many of the other training textures. As depicted in Figure 4.4, the final SVBRDF Autoencoder model usually captures the structure of the input texture but fails to restore its diffuse colour.

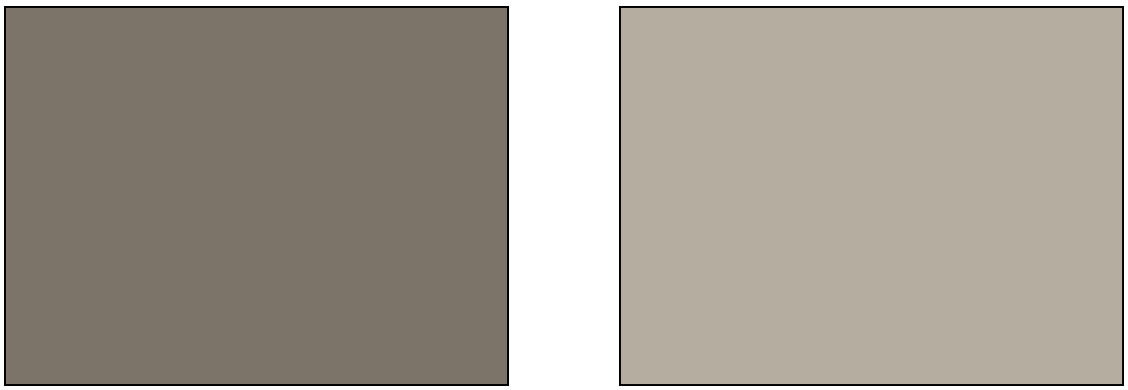
(a) Forest Ground 001 **(b)** Stacked Rectangular Stones **(c)** Garden Tiles 001 Bitmap

Figure 4.4: Reconstruction results for the *Earth - Forest Ground 001*, *Rock - Stacked Rectangular Stones*, and *Urban - Garden Tiles 001 Bitmap* textures. The right texture in each pair is a reconstruction of the left texture under the same lighting conditions. Most reconstructions of textures from the training dataset exhibit a passive brown hue.

In hindsight, the discolouration defect is likely to be related to the decision to forgo the pre-processing transformation expected by the VGG-19 network. This transformation amounts to normalizing the colour of each pixel in the input image by the mean μ_{VGG} and standard deviation σ_{VGG} of each colour channel in the ImageNet dataset. Specifically, an RGB pixel represented by the triplet (R, G, B) is expected to be transformed into the triplet (R', G', B') where

$$R' = \frac{R - 0.485}{0.229}, \qquad G' = \frac{G - 0.456}{0.224}, \qquad B' = \frac{B - 0.406}{0.225}. \tag{4.1}$$

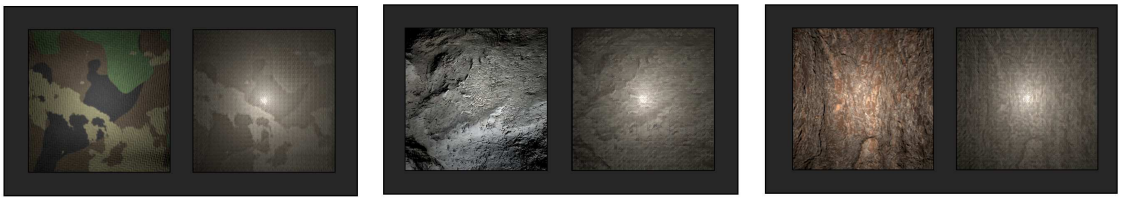
It follows that a standard RGB image is interpreted by the VGG-19 network as having a colour palette that is restricted between Figure 4.5a (i.e. μ_{VGG}) and Figure 4.5b (i.e. $\mu_{\text{VGG}} + \sigma_{\text{VGG}}$). Hence, the SVBRDF Autoencoder is likely to make predictions near $(R, G, B) \approx \mu_{\text{VGG}}$ to balance the RGB interpretation of the texel loss against the VGG interpretation of the content and style losses.



(a) VGG interpretation of $(0, 0, 0)$. **(b)** VGG interpretation of $(1, 1, 1)$.

Figure 4.5: Colours assumed by the VGG-19 network for a black and white input in the RGB colour space without undergoing the expected pre-processing transformation.

Another property shared among many training textures is the bias of the final SVBRDF Autoencoder model towards specular predictions. This behaviour is showcased in Figure 4.6. One possible explanation for this bias is that the model might inadvertently learn a uniform prior over the base colour of the Disney SVBRDF from monochrome textures and subsequently interpret variations in the brightness of other textures as specular highlights rather than base colour fluctuations.



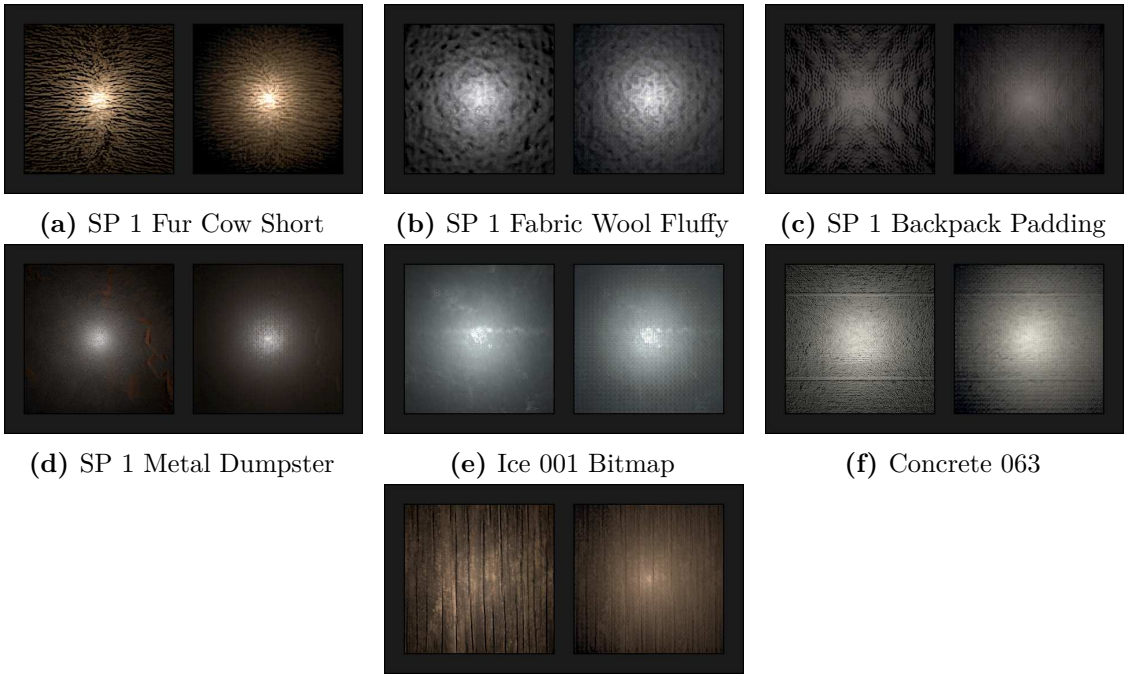
(a) Camouflage 001 **(b)** Rock 012 Bitmap **(c)** SP 1 Bark Black Pine

Figure 4.6: Reconstruction results for the *Fabric - Camouflage 001*, *Rock - Rock 012 Bitmap*, and *Wood - SP 1 Bark Black Pine* textures. Observe that the predicted textures have a strong specular component even though the input textures are mostly diffuse.

Fortunately, the deficiencies exhibited in both Figure 4.4 and Figure 4.6 can be mitigated by matching the RGB colour histograms of the output textures to the input textures using the procedure in [HB95]. This post-processing step is applied to all of the remaining reconstructions in this chapter as well as those featured in Appendix A.

4.2.2 Test Textures

Overall, the qualitative performance of the SVBRDF Autoencoder on the testing dataset mirrors that of the training dataset. On one hand, this means the model did not simply memorize the training dataset and instead learned some aspects about SVBRDF texture synthesis that can be generalized to novel data. Of course, the caveat is that the shortcomings of the model over the training dataset are also evident in the testing dataset. A summary of the reconstruction performance over the testing dataset is displayed in Figure 4.7.



(a) SP 1 Fur Cow Short **(b)** SP 1 Fabric Wool Fluffy **(c)** SP 1 Backpack Padding

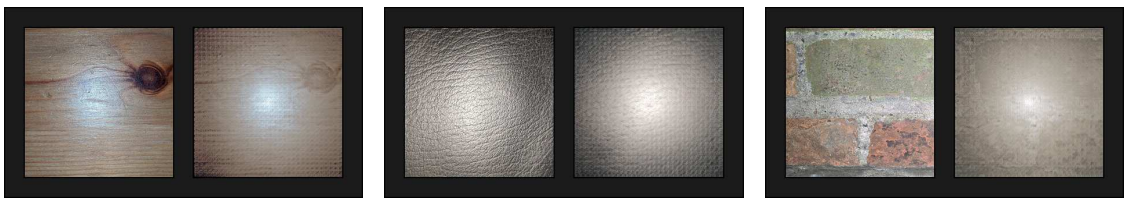
(d) SP 1 Metal Dumpster **(e)** Ice 001 Bitmap **(f)** Concrete 063

(g) SP 1 Wood Board 1

Figure 4.7: Reconstruction results for random samples from the testing dataset.

4.2.3 Real Textures

The reconstruction quality of real photographs with the SVBRDF Autoencoder is arguably worse than that of the training or testing textures. Specifically, the reconstructions suffer from the same deformations as the dataset textures but are also substantially noisier. This is likely a consequence of several factors. For instance, one potential issue is that the surfaces in Figure 4.8b and Figure 4.8c are not perfectly flat on the macrosurface scale. Another conspiring variable is that the flash of the camera has a considerably larger angular distribution than what is exposed through the training dataset.



(a) Wooden Table **(b)** Leather Sofa **(c)** Brick Wall

Figure 4.8: Reconstruction results for real flash photographs. Each picture was taken with a Google Pixel 3 XL mobile phone in an environment with negligible ambient lighting.

4.2.4 Feedback Experiment

The *feedback experiment* originates from [KNL+15] and serves to measure the robustness of the SVBRDF Autoencoder. In this experiment, an input texture is combined with an RDF, processed by the SVBRDF Autoencoder model, and then rendered with a point light in a flash position. This process is then repeated several times, using the previous output of the rendering layer as the input texture for the next cycle. Ideally, the rendering at the end of the chain should resemble the original input texture; however, if the model is unstable, the rendering may degenerate into a different texture.

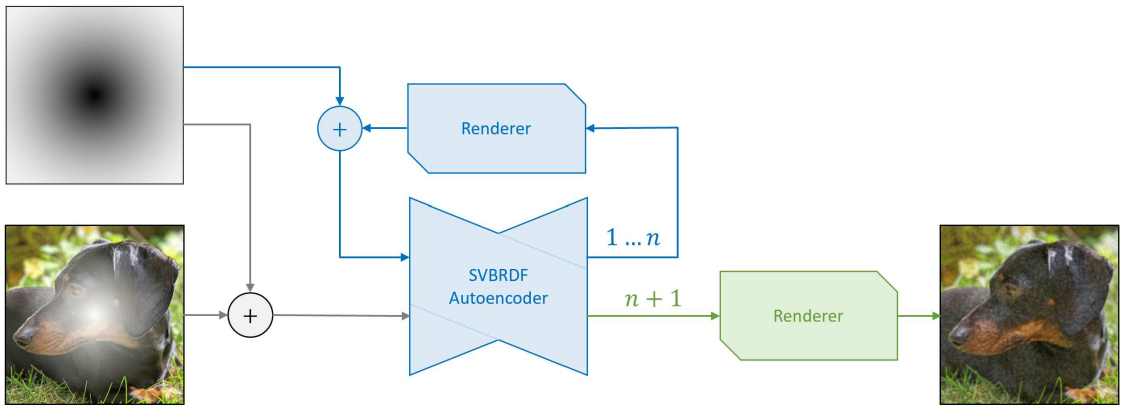
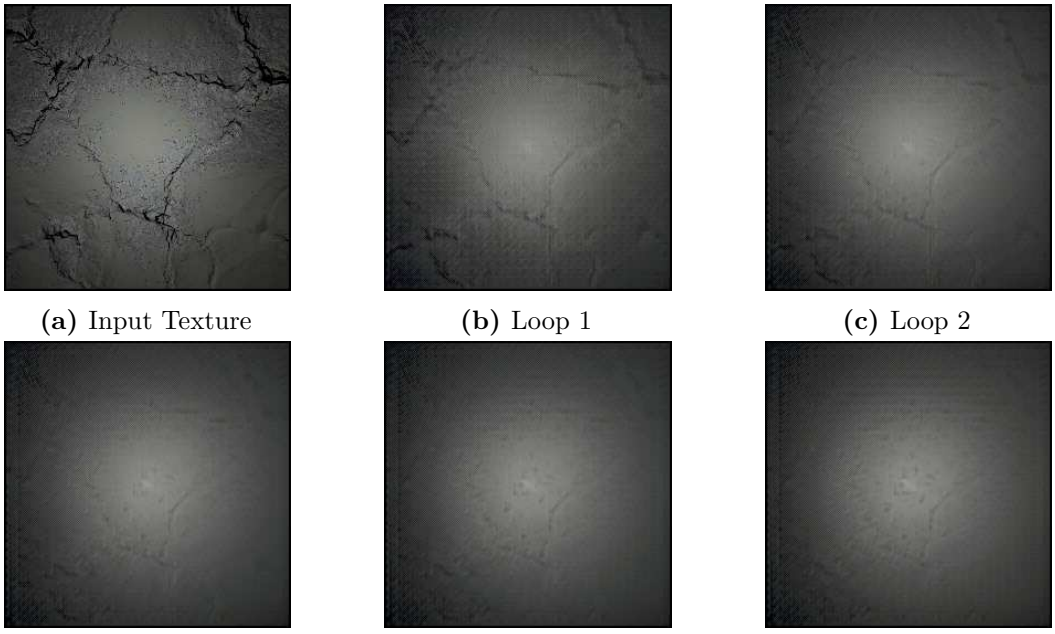


Figure 4.9: The feedback experiment simulates a closed-loop system where an SVBRDF Autoencoder predicts the parameterization of an input texture, renders the predicted texture under a centered point light, and then feeds the rendering back into the model.

Below, Figure 4.10 demonstrates that the final SVBRDF Autoencoder model is not particularly stable. Observe that the rendering of the input texture loses most of its definition by the fourth feedback iteration. Virtually all of the dataset textures degenerate into an approximately-uniform SVBRDF parameter field before 10 feedback iterations have transpired.



(a) Input Texture **(b)** Loop 1 **(c)** Loop 2

(d) Loop 3 **(e)** Loop 4 **(f)** Output Texture

Figure 4.10: Results of the feedback experiment for the *Earth - Dry Mud* texture with 5 feedback iterations.

4.2.5 Partition Experiment

The *partition experiment* represents an attempt to generalize the capabilities of the SVBRDF Autoencoder to reconstruct large images with dimensions that are integer multiples of 256. To do so, the large image and its corresponding RDF field are spatially partitioned into a set of constituent $256 \times 256 \times 4$ tensors which are individually encoded by the SVBRDF Autoencoder. The resulting latent fields are then merged together (usually with some blended overlap) and then reconstructed using the fully-convolutional Decoder.

Figure 4.11: The split node (left) spatially partitions a tensor into multiple tensors; the merge node (right) spatially blends multiple tensors into a single tensor.

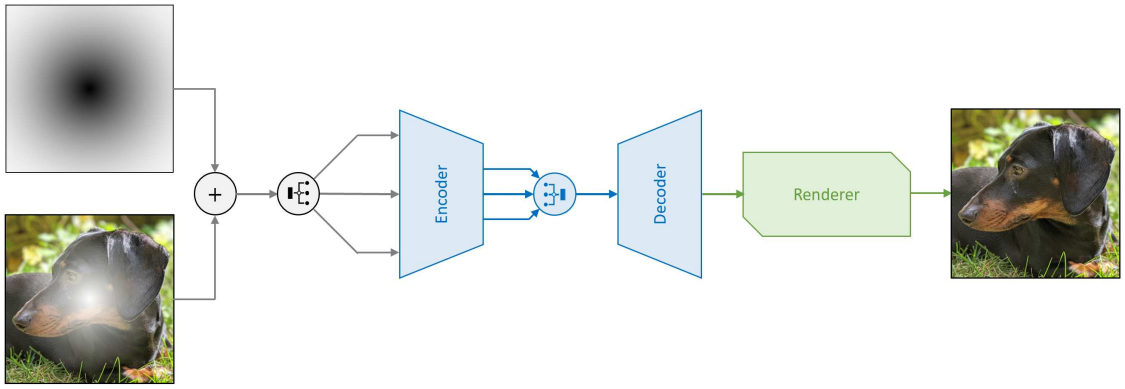


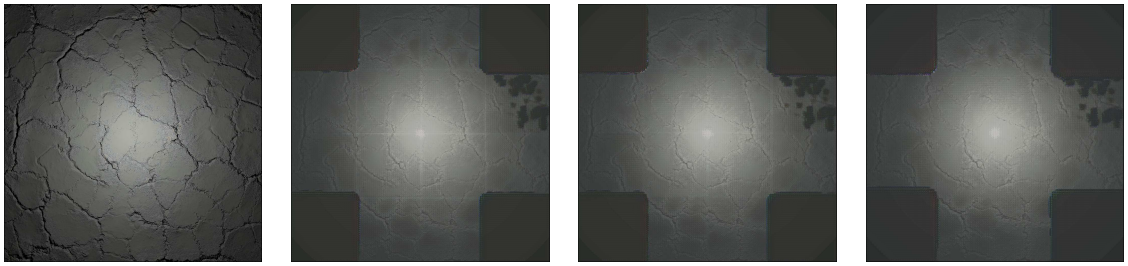
Figure 4.12: The partition experiment uses the SVBRDF Autoencoder to reconstruct a large image by dividing the image into a set of 256×256 images, encoding each small image into a latent field, merging the latent fields together, and then decoding the result.

Unfortunately, decomposing and reconstructing a large image in this way is not well-supported by the SVBRDF Autoencoder. In particular, the model struggles with the interpretation of the partitioned RDF in certain regions (which vary depending on the texture) and decodes a uniform field in its place.

(a) SP 1 Wicker **(b)** Stone Tiles 001

Figure 4.13: Results of the partition experiment for 1024×1024 renderings of the *Industry - SP 1 Wicker* and *Rock - Stone Tiles 001* textures with a latent tile overlap of 5 units.

As a side note, naïvely concatenating latent fields may produce seams in the output texture, as shown in Figure 4.14b. To overcome this issue, it suffices to blend neighbouring latent fields using bilinear interpolation prior to invoking the Decoder. It is also worth mentioning that, regardless of the splicing technique, the periodic latent field must be recomputed after the latent fields are merged in order to smoothly transition the indices and parameters of the sinusoidal plane waves.



(a) Input Texture **(b)** Overlap 0 **(c)** Overlap 1 **(d)** Overlap 5

Figure 4.14: Increasing the overlap between adjacent latent tiles in the *Earth - Dry Mud* texture reduces boundary artefacts but also the size of the final image.

4.3 Interpolation Experiments

A natural extension to the set of reconstruction experiments is the set of *interpolation experiments* which seek to evaluate the quality of the latent field representations of the SVBRDF Autoencoder. The three interpolation experiments present different ways of blending two textures depending on the desired level of overlap.

4.3.1 Blend Experiment

The *blend experiment* combines two textures by independently weighing the local, global, and periodic components of their latent fields. To be precise, given two textures with latent fields $\boldsymbol{z}^{(1)}$ and $\boldsymbol{z}^{(2)}$, the blend experiment applies an alpha vector $\boldsymbol{\alpha} = \begin{bmatrix}\alpha_L & \alpha_G & \alpha_P\end{bmatrix}$ such that the latent field $\boldsymbol{z}$ of the blended texture is

$$\boldsymbol{z}_L = (1 - \alpha_L)\, \boldsymbol{z}_L^{(1)} + \alpha_L\, \boldsymbol{z}_L^{(2)} \tag{4.2}$$

$$\boldsymbol{z}_G = (1 - \alpha_G)\, \boldsymbol{z}_G^{(1)} + \alpha_G\, \boldsymbol{z}_G^{(2)} \tag{4.3}$$

$$\boldsymbol{z}_P = (1 - \alpha_P)\, \boldsymbol{z}_P^{(1)} + \alpha_P\, \boldsymbol{z}_P^{(2)}\ . \tag{4.4}$$

Observe that the first texture can be recovered by setting $\boldsymbol{\alpha} = \begin{bmatrix}0 & 0 & 0\end{bmatrix}$ while the second texture corresponds to $\boldsymbol{\alpha} = \begin{bmatrix}1 & 1 & 1\end{bmatrix}$. Assuming that the local, global, and periodic latent fields of the SVBRDF Autoencoder are correctly decomposed, this approach can transfer the periodic properties of one texture onto another texture or combine the local structure of one texture with the style of a different texture. The output of this experiment is a texture with the same size as the input textures.

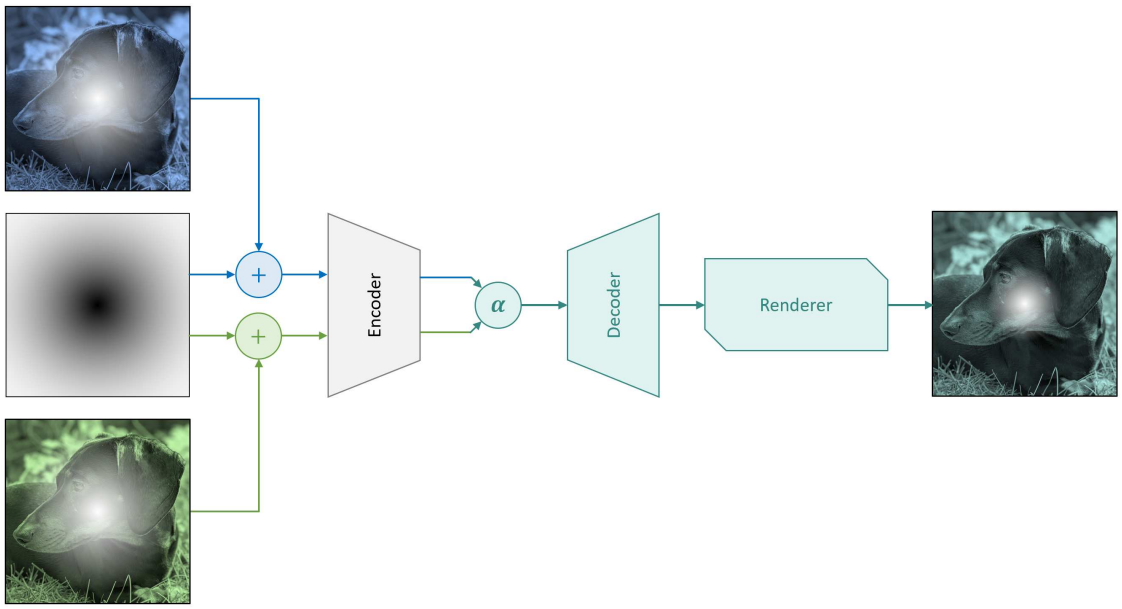
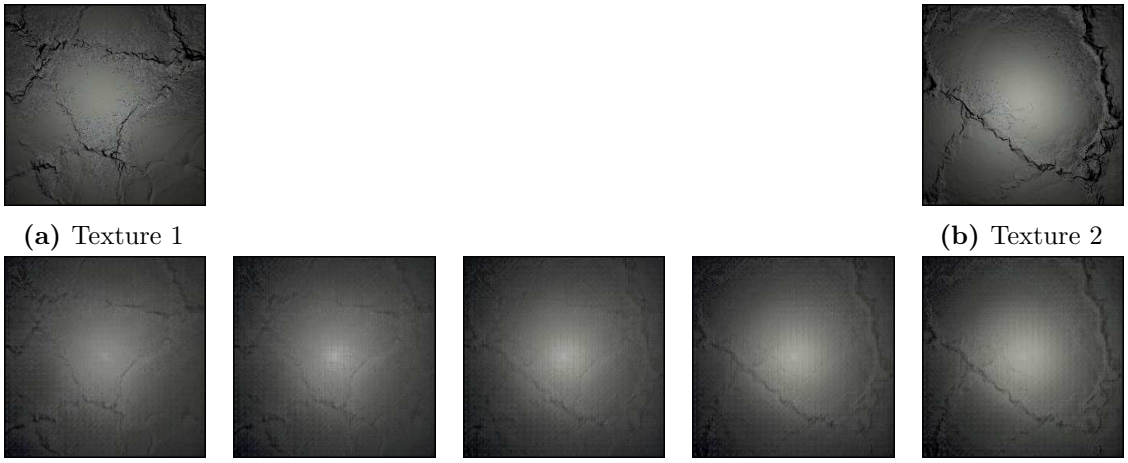


Figure 4.15: The blend experiment interpolates between two textures by independently weighing the local, global, and periodic latent fields of each texture.

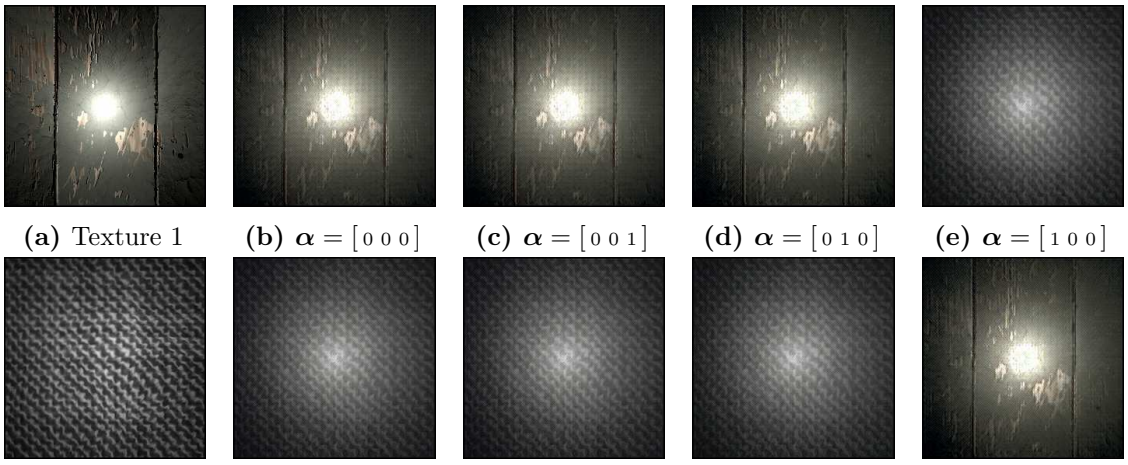
The SVBRDF Autoencoder is partially successful at combining textures in the manner specified by the experiment. For example, Figure 4.16 demonstrates that linearly interpolating between two different samples of the same texture can yield plausible intermediate textures that shows hints of both extremes. On the other hand, Figure 4.17 suggests that the model is unable to independently blend qualitative aspects of different latent fields. In fact, the latter experiment instance offers evidence of a serious flaw in the encoding of the latent fields: the influence of the local latent field dominates that of the global and periodic latent fields.



(a) Texture 1 **(b)** Texture 2

(c) $\alpha = 0.00$ **(d)** $\alpha = 0.25$ **(e)** $\alpha = 0.50$ **(f)** $\alpha = 0.75$ **(g)** $\alpha = 1.00$

Figure 4.16: Results of a blend experiment for two samples of the *Earth - Dry Mud* texture. Here, the same α is applied to each of the local, global, and periodic latent fields.



(a) Texture 1 **(b)** $\boldsymbol{\alpha} = [0\ 0\ 0]$ **(c)** $\boldsymbol{\alpha} = [0\ 0\ 1]$ **(d)** $\boldsymbol{\alpha} = [0\ 1\ 0]$ **(e)** $\boldsymbol{\alpha} = [1\ 0\ 0]$

(f) Texture 2 **(g)** $\boldsymbol{\alpha} = [1\ 1\ 1]$ **(h)** $\boldsymbol{\alpha} = [1\ 1\ 0]$ **(i)** $\boldsymbol{\alpha} = [1\ 0\ 1]$ **(j)** $\boldsymbol{\alpha} = [0\ 1\ 1]$

Figure 4.17: Results of a blend experiment for the *Wood - SP 1 Old Painted Planks* and *Fabric - Fabric Suit Vintage* textures using staggered $\boldsymbol{\alpha}$ weights.

4.3.2 Merge Experiment

The *merge experiment* splices two textures by linearly interpolating their latent fields across a specified overlap. For instance, if $\boldsymbol{z}^{(1)}$ and $\boldsymbol{z}^{(2)}$ are 3×5 latent fields that need to be merged with an overlap of 3 units, the blending matrix $\boldsymbol{A}$ denoting the weight of each latent field in the merged latent field is given by

$$\boldsymbol{A} = \begin{bmatrix} 0 & 0 & 0.25 & 0.50 & 0.75 & 1 & 1 \\ 0 & 0 & 0.25 & 0.50 & 0.75 & 1 & 1 \\ 0 & 0 & 0.25 & 0.50 & 0.75 & 1 & 1 \end{bmatrix} . \tag{4.5}$$

Note that the local and global latent fields are interpolated in an identical manner and thus share the same blending matrix. It follows that the width of the SVBRDF Autoencoder output decreases as the size of the overlap grows.

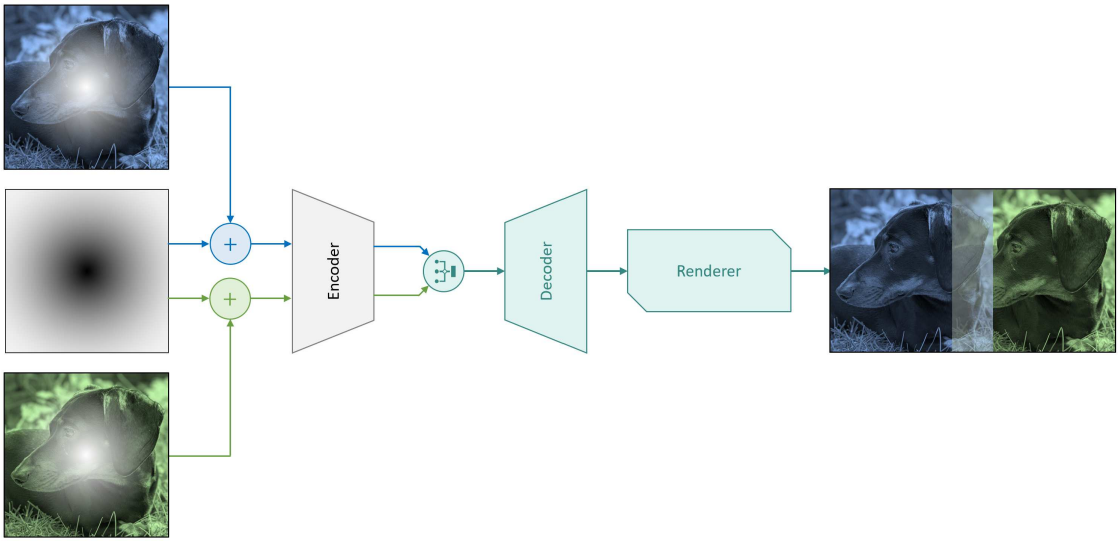
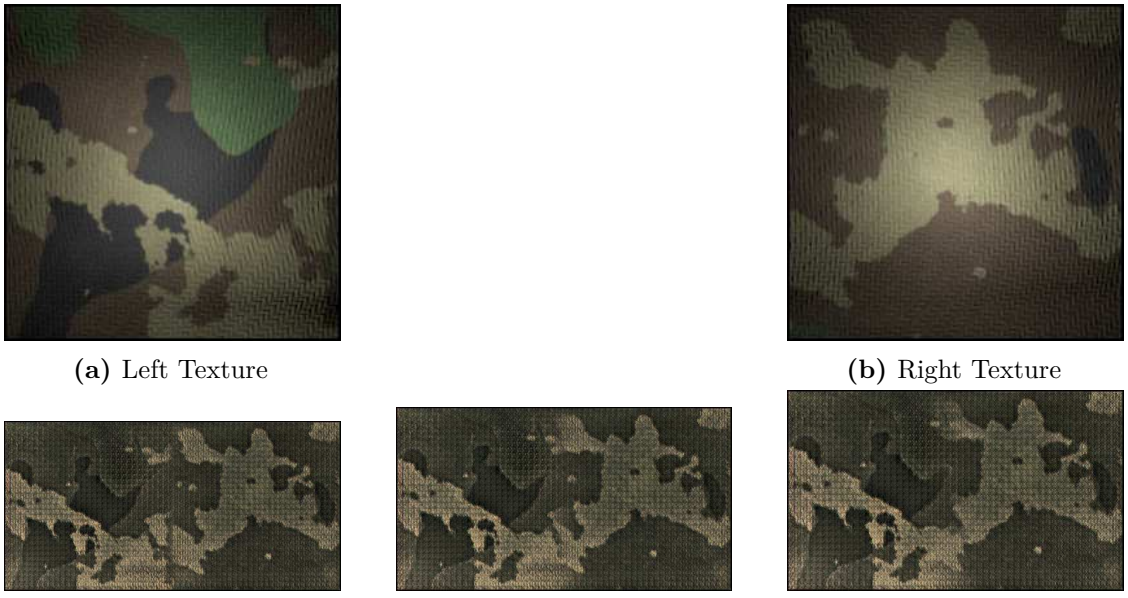


Figure 4.18: The merge experiment combines two textures by concatenating them with an overlap that gradually transitions from one latent field to the other.

The result pictured in Figure 4.19 supports the conclusion that the final SVBRDF Autoencoder model is proficient at merging textures. For instance, the black cusp lying in the center-right area of Figure 4.19a is smoothed into a round shape in Figure 4.19e, indicating that the latent representation of the model is capable of morphing patterns in the process of interpolation. That said, a defect is also visible near the top-center of Figure 4.19e where the transition between a pair of light and dark features is implemented with a blur rather than a natural deformation.



(a) Left Texture **(b)** Right Texture

(c) Overlap 0 **(d)** Overlap 5 **(e)** Overlap 10

Figure 4.19: Results of a merge experiment for two samples of the *Fabric - Camouflage 001* texture. Increasing the size of the overlap decreases the aspect ratio of the output.

4.3.3 Morph Experiment

The *morph* experiment hyperbolizes the spatial interpolation of two textures by converting one into the other through a series of $(n-1)$ intermediate textures. In this experiment, the output texture is constructed by arranging the intermediate textures in a row (with a source texture on either end) and assigning, from left to right, a local latent field alpha α_i to each texture such that

$$\alpha_i = \frac{i}{n} \ . \tag{4.6}$$

Thus, if $\boldsymbol{z}_L^{(0)}$ and $\boldsymbol{z}_L^{(n)}$ are the local latent fields of the left and right source textures, the local latent field $\boldsymbol{z}_L^{(i)}$ of texture $i \in \{1, \ldots, n-1\}$ is given by

$$\boldsymbol{z}_L^{(i)} = (1 - \alpha_i)\, \boldsymbol{z}_L^{(0)} + \alpha_i\, \boldsymbol{z}_L^{(n)} \ . \tag{4.7}$$

In contrast, the global latent field varies between the columns of each intermediate texture and is computed analogously to Equation 4.5 where the size of the overlap is now the width of a source texture multiplied by $(n-1)$. Note that the interpolation performance of several texture synthesis techniques are measured in [YBS+19] using a similar approach.

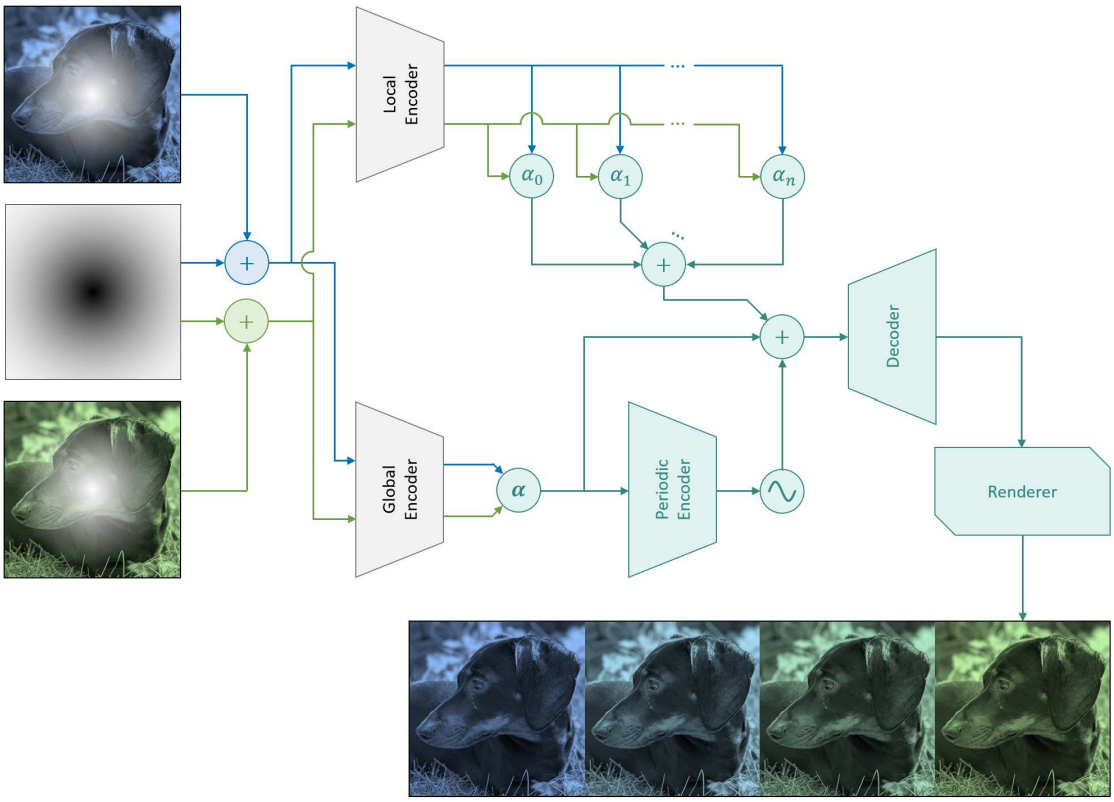


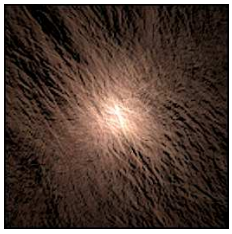
Figure 4.20: The morph experiment transforms one texture into another texture over $(n-1)$ intermediate textures. Here, the global latent field is interpolated smoothly across the intermediate textures whereas the local latent field is interpolated in discrete chunks.

(a) Left Texture **(b)** Right Texture

(c) Morphing Result

Figure 4.21: Results of a morph experiment for two samples of the *Earth - Dry Mud* texture with 4 intermediate textures. The blotted appearance of Figure 4.21c is a consequence of matching the colour histogram from the directional light in Figure 4.21c to the blended colour histogram from the point lights in Figure 4.21a and Figure 4.21b.

Given the performance of the SVBRDF Autoencoder in the merge experiment, it should come as no surprise that the model also shines in the morph experiment. Figure 4.21 and Figure 4.22 show a gradual transition between two source textures using intermediate textures that look their part. Note that the boundaries between adjacent textures could be improved by blending the edges of the (blended) local latent fields.



(a) Left Texture

(b) Right Texture

(c) Morphing Result

Figure 4.22: Results of a morph experiment for the *Earth - SP 1 Cow Fur Long* and *Rock - Rock 012 Bitmap* textures with 3 intermediate textures.

4.4 Expansion Experiments

The final collection of experiments address the original purpose of the SVBRDF Autoencoder: extending a texture to the infinite plane. Most of these *expansion experiments* operate on a single instance of a texture and leverage the latent field representation of the SVBRDF Autoencoder to generate seamless boundaries between latent tiles. Note that the procedures for expanding a single texture can usually be generalized to multiple textures in a trivial fashion.

4.4.1 Tile Experiment

The *tile experiment* synthesizes a tileable variant of an input texture by blending a 3×3 grid of its latent field with a predetermined amount of overlap. The resulting

latent field is then passed through the Decoder and cropped around its center to extract the tileable texture. Of course, a larger latent field (e.g. 5×5) can also be used although the small receptive fields of the output neurons in the Decoder make this unnecessary. Increasing the extent of the blended edges gives smoother results at the risk of smoothing away distinctive texture features.

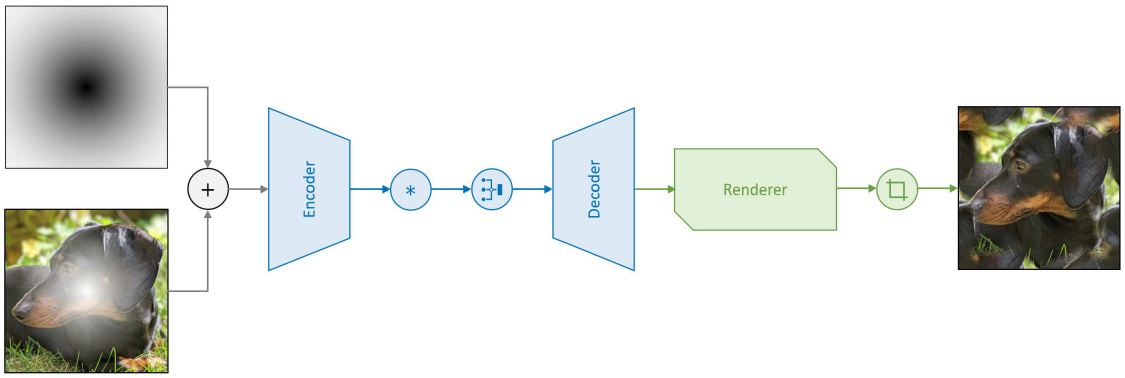
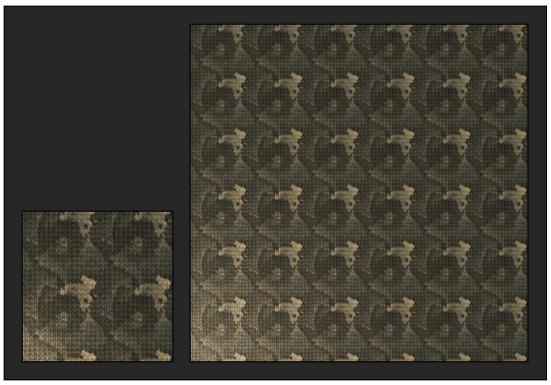


Figure 4.23: The tile experiment derives a tileable version of a texture by duplicating its latent field, blending the edges of the duplicates, and then cropping the reconstruction of the blended latent field.

(a) Input Texture

(b) Overlap 6



(c) Overlap 16

Figure 4.24: Results of a tile experiment for a sample of the *Fabric - Camouflage 001* texture. The left image in Figure 4.24b and Figure 4.24c represents the output of the SVBRDF Autoencoder while the right image is a 3×3 tiling of that texture.

As mentioned in Section 2.3.6, tileable textures are not practical for most applications due to their repetitive structure. This is especially evident in Figure 4.24. In practice, the SVBRDF Autoencoder begins to synthesize plausible tileable textures when the overlap between the duplicated latent fields is close to half the extent of the fields, as demonstrated by the lack of visual artefacts in Figure 4.24c compared to Figure 4.24b.

4.4.2 Local Experiment

The *local experiment* randomizes the structure of an input texture by setting each entry of its local latent field to a random sample from the uniform distribution $\mathcal{U}[-1, 1]$. Since the global and periodic latent fields remain unchanged, the output texture should, ideally, represent the style of the input texture. Textures of arbitrary size can be synthesized by sampling a local latent field with the desired spatial extent and then expanding the global and periodic latent fields in the usual way.

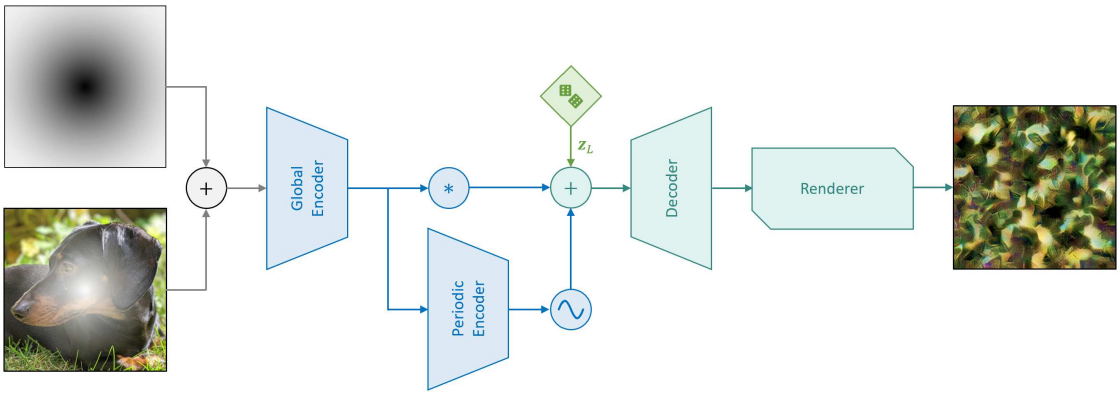


Figure 4.25: The local experiment replaces the local latent field of a texture with a sample from a uniform distribution, resulting in an output texture with arbitrary structure. The output image in this diagram was generated using [Lun].

Figure 4.26 demonstrates that the SVBRDF Autoencoder has trouble with this task. Clearly, the Decoder lacks the flexibility to generate an interesting set of SVBRDF parameters from a random local latent field. This failure can be viewed as a consequence of the curse of dimensionality coupled with a lack of regularization to facilitate the sampling of the random latent variables (as is done in a VAE).

(a) SP 1 Cap **(b)** Stone Tiles 001 **(c)** Wood 024 Walnut

Figure 4.26: Local experiment results for samples from the *Fabric - SP 1 Cap*, *Rock - Stone Tiles 001*, and *Wood - Wood 024 Walnut* textures. All the textures in the dataset exhibit the same checkerboard output shown in Figure 4.26a through Figure 4.26b; the checkerboard is caused by the fractionally-strided convolutional layers coupled with colour histogram matching.

4.4.3 Shuffle Experiment

The *shuffle experiment* constructs a derivative of an input texture by spatially concatenating samples of the local latent field associated with that texture. The samples (i.e. latent tiles) are drawn uniformly at random with replacement and are placed beside one another without blending. Both the size of the latent tiles and the dimensions of the grid are configurable hyperparameters.

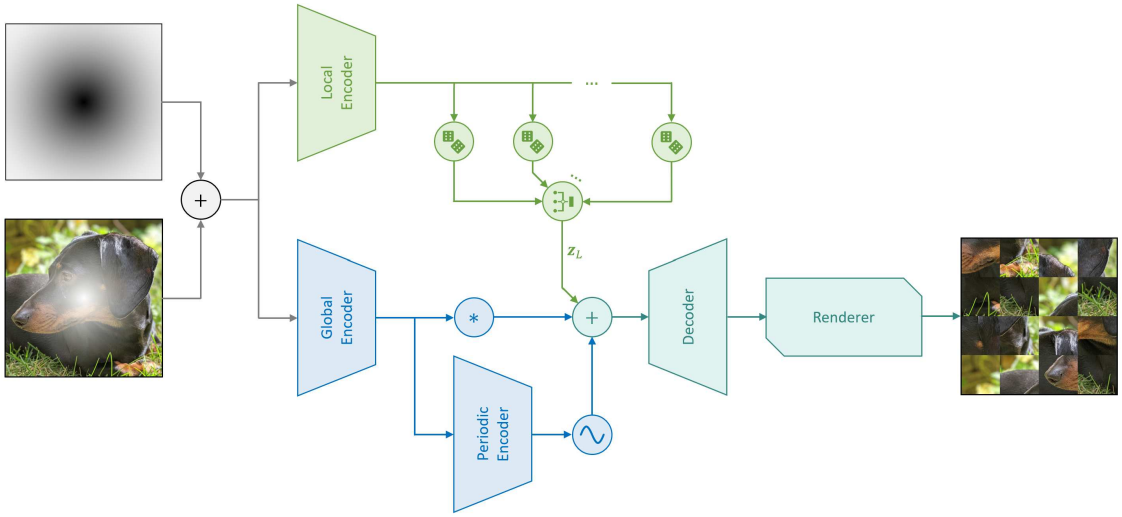
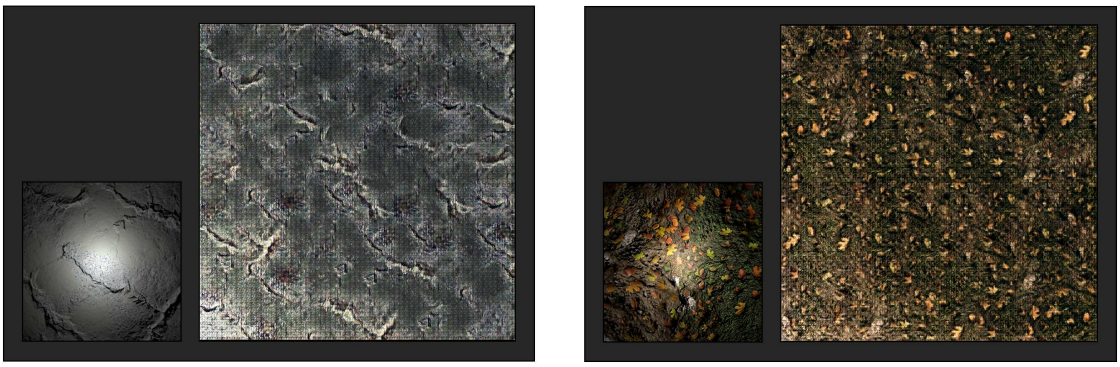


Figure 4.27: The shuffle experiment samples random crops from the local latent field of an input texture and arranges them in a grid to be decoded into an output texture.

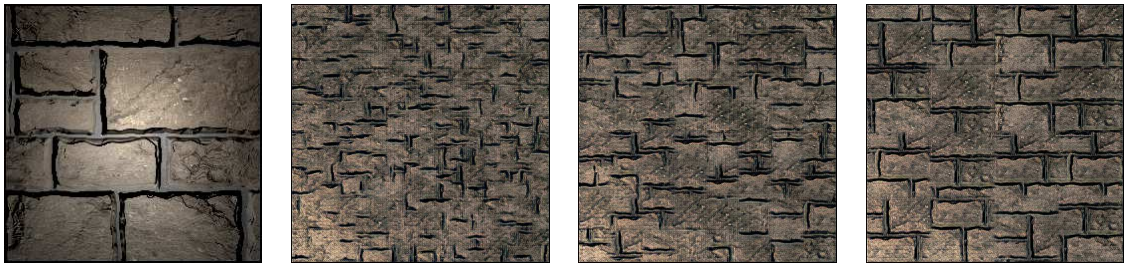
The SVBRDF Autoencoder displays an impressive degree of competence when it comes to synthesizing stochastic textures in the manner described by the experiment. Specifically, Figure 4.28 provides evidence that the model can synthesize natural tile boundaries that are imperceptible to the average human observer.



(a) Dry Mud **(b)** Forest Ground 001

Figure 4.28: Shuffle experiment results for samples from the *Earth - Dry Mud* and *Earth - Forest Ground 001* textures with latent tiles of size $8 \times 8 \times 256$.

Similar to the texture synthesis algorithms surveyed in Section 2.3.3, the dimensions of the latent tiles impact the size of the features that are coherently reproduced in the output texture. This dependence is illustrated in Figure 4.29.



(a) Input Texture **(b)** Size $4 \times 4 \times 256$ **(c)** Size $8 \times 8 \times 256$ **(d)** Size $16 \times 16 \times 256$

Figure 4.29: Shuffle experiment results for a sample from the *Rock - Stacked Rectangular Stones* texture with different latent tile sizes.

4.4.4 Quilt Experiment

The *quilt experiment* draws inspiration from [BJV17] to produce a *texture quilt* from an arbitrary number of input textures. The quilt is assembled in the latent domain by randomly concatenating and blending the latent fields of the input textures in the structure of a grid. Unlike the shuffle experiment, the quilt experiment can leverage samples from multiple textures to create an output texture that features a diverse composition of styles.

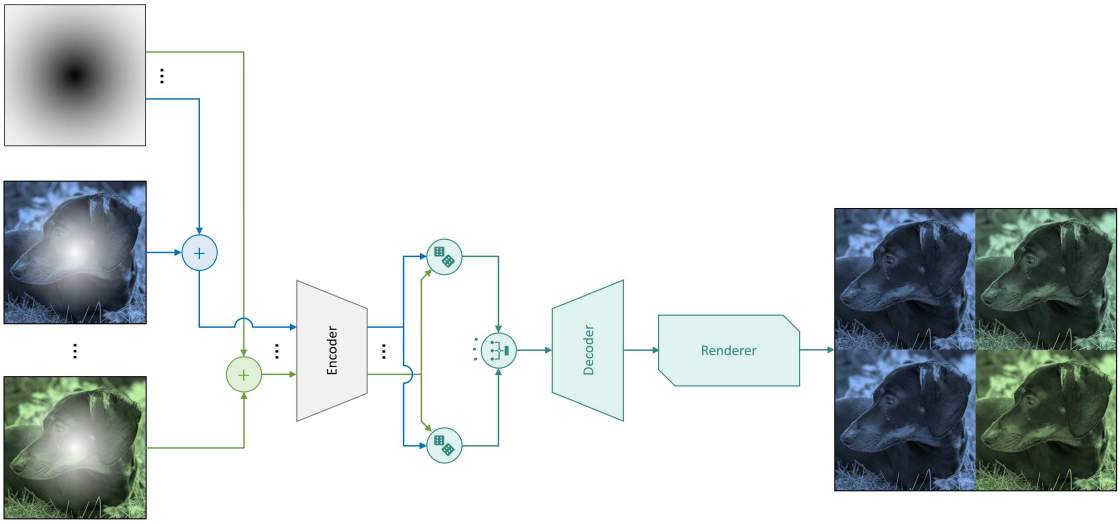


Figure 4.30: The quilt experiment blends a grid of latent fields derived from a set of input textures to produce a composite output texture. One aspect that is omitted from this diagram is that the periodic latent field must be updated after blending the global latent fields to use the indices of the blended texture.

(a) Stone Tiles 001

(b) Stacked Rectangular Stones

(c) Output Texture

Figure 4.31: Quilt experiment results based on samples from the *Rock - Stone Tiles 001* and *Rock - Stacked Rectangular Stones* textures with an overlap size of 12.

Needless to say, executing the quilt experiment with samples from the same input texture gives results that closely resemble Figure 4.28. Compare this to Figure 4.31 where samples from two distinct tile textures are combined into a single rendering.

Observe that most of the latent field boundaries in the grid are betrayed by a noticeable blur artefact; however, the SVBRDF Autoencoder did manage to merge the four latent fields in the center of Figure 4.31c in a satisfying way.

5 Conclusions

Contents

This chapter places the key highlights from Chapter 3 and Chapter 4 in the context of the original research question and the current state of the texture synthesis field. First, Section 5.1 recaps the objectives, approach, and findings of the research project. Thereafter, Section 5.2 proposes a set of extensions to the research project intended to guide future efforts based on this work.

5.1 Review

Fundamentally, the goal of this research project is to train an ML model to infer the reflectance properties of a material and tile an arbitrary surface with the same pattern of reflectance behaviour. As input, the model accepts a fronto-parallel flash-lit photograph of a surface featuring the material of interest. As output, the model generates a parameterization of a reflectance model for each point on the desired surface. Prior works in this area have successfully reproduced the exact reflectance of a surface while others have managed to extrapolate the RGB content of an image

beyond its borders; however, there do not appear to be any attempts that combine the precision of the first task with the creativity of the second task in a scalable way.

The ML model adopted by this research project is a deep convolutional autoencoder and is composed of three encoders and one decoder. Two of the encoders are fully-convolutional DNNs that extract the local and global reflectance properties of a texture, respectively. The third encoder is a shallow MLP that derives a few sinusoidal plane waves from the global encoding of an input texture. From here, the output of each encoder is concatenated to form a latent field which is consumed by a fully-convolutional decoder DNN to yield the surface normals and Disney BRDF parameters at each point on the input surface. This model, called the SVBRDF Autoencoder, is trained in an end-to-end fashion using an incremental learning strategy over a dataset of synthetic materials from the Substance Share repository. The parameters of the model are optimized with the Adam optimizer using a reconstruction loss that renders the output of the decoder for a random light and viewing position before applying an L2 loss along with a pair of loss functions based on the activations of the VGG-19 network.

Ultimately, the reconstruction performance of the SVBRDF Autoencoder is unsatisfying due to the obvious discolouration of its output. It is hypothesized that this defect is a consequence of ignoring the pre-processing step expected by the VGG-19 loss network but further investigation is required before reaching a definite conclusion. The model also performs poorly on highly specular textures, such as metals, although this is likely to be a flaw in the structure of the style and content loss functions, as indicated by the relatively low loss values of these textures. That said, the SVBRDF Autoencoder does appear to successfully replicate the dominant structures and patterns featured in an input texture. Furthermore, the performance of the model is similar between the training, validation, and testing datasets.

In addition to simply reconstructing a texture verbatim, the performance of the SVBRDF Autoencoder was explored through a series of experiments which address the original purpose of the model. One crucial finding from these experiments is that the SVBRDF Autoencoder is not particularly stable: chaining several applications

of the model causes a rapid degradation in image synthesis quality. An interesting failure mode of the SVBRDF Autoencoder is also observed when the model is tasked with reconstructing a large image by partitioning it into small regions and encoding each region independently. With respect to interpolation, the SVBRDF Autoencoder can blend two textures in its latent space to produce a result that is slightly beyond what can be achieved by alpha compositing in the RGB image space. That said, the local, global, and periodic representations of a texture are not properly disentangled in the latent space which limits the capacity of the model to perform style (or frequency) transfer. Fortunately, the latent representation of the SVBRDF Autoencoder *is* sufficient for generating a tileable variant of a texture or arbitrarily increasing the spatial extent of an existing texture by tiling small regions of its latent field; however, expanding a texture by naïvely sampling its local representation from a uniform distribution gives rise to textures that are devoid of interesting structure.

5.2 Future Work

One avenue for future work is to improve upon the proposed model and training procedure with the same research objective in mind. For instance, integrating the pre-processing transformation required by the VGG-19 network and then retuning the hyperparameters of the model would be a useful step towards validating the approach of this research project. It would also be instructive to design an experiment that tests how different BRDF models, ranging from the Blinn-Phong BRDF to the Unreal Engine 4 BRDF [KG13], affect the convergence and performance of the SVBRDF Autoencoder. Regarding the model itself, it would be interesting to see if transforming the SVBRDF Autoencoder into a VAE overcomes the curse of dimensionality that is faced when randomly sampling from the local latent field. Alternatively, the existing SVBRDF Autoencoder decoder can be directly trained as part of a GAN before introducing the encoders as a means of mapping input textures to noise tensors. Lastly, it could be meaningful to train the SVBRDF Autoencoder with a different dataset, such as Adobe Stock 3D [Adoa], or limit the diversity of the textures in the dataset to ease the generalization burden of the model.

The other flavour of future work involves broadening the scope of the research project and lifting assumptions from the texture synthesis model. One such option is to integrate a diversity loss function into the training process that encourages the SVBRDF Autoencoder to reconstruct variants, rather than exact clones, of input textures, as done in [LFY+17]. Similarly, a loss function can also be used to explicitly enforce the semantics of the global and local latent fields (e.g. by penalizing differences in the global latent vectors from different samples of the same texture) to enable style transfer applications. A more ambitious extension of this research project involves treating surfaces as volumes and taking advantage of the microflake framework [JAM+10], along with a corresponding BSDF such as the SGGX microflake distribution [HDCD15], to accurately model anisotropic fibrous materials such as cloth or hair. Finally, there is room for innovation in the input format of the SVBRDF Autoencoder. For example, using a homography in conjunction with some signal processing techniques could alleviate the need for the input textures to be taken from a front-parallel view with a co-located light source. Taking this idea one step further, the input medium itself can be swapped from a photograph to a video in order to glean information about the reflectance of a surface from multiple viewing positions.

Appendices

A
Results

Contents

This appendix complements the results showcased in Chapter 3 and Chapter 4.

A.1 MERL 100 Optimization

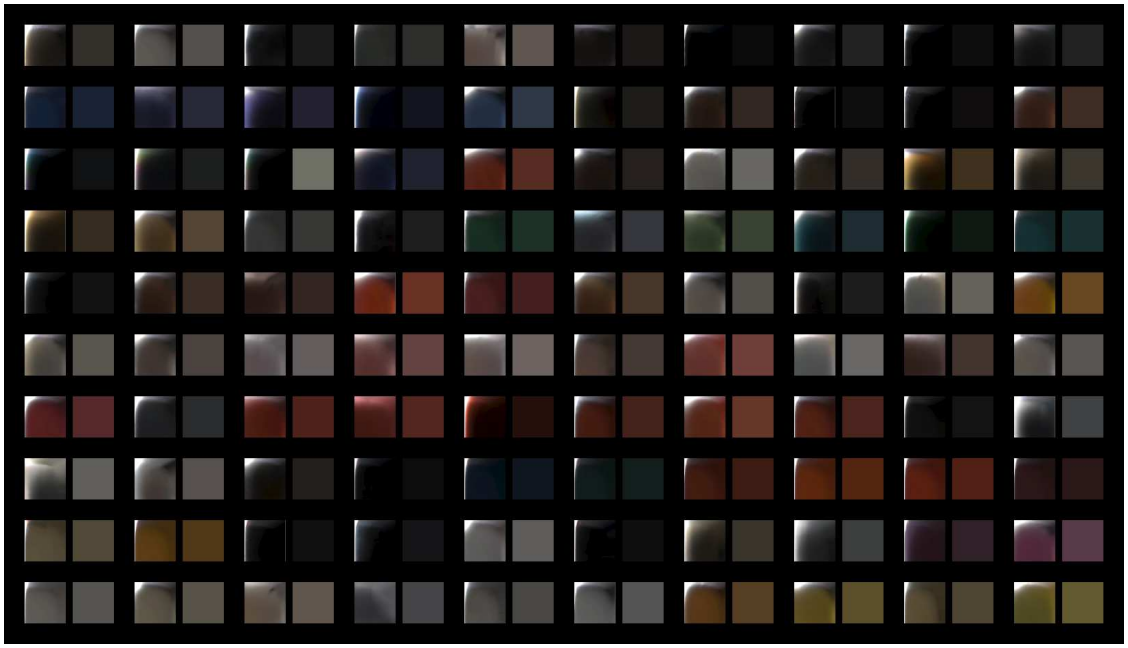


Figure A.1: BRDF slices obtained by fitting the Lambertian BRDF to the MERL 100 dataset using the L-BFGS-B optimizer. The left image in each pair depicts the ground-truth BRDF slice while the right image is the best fit of the Lambertian BRDF.

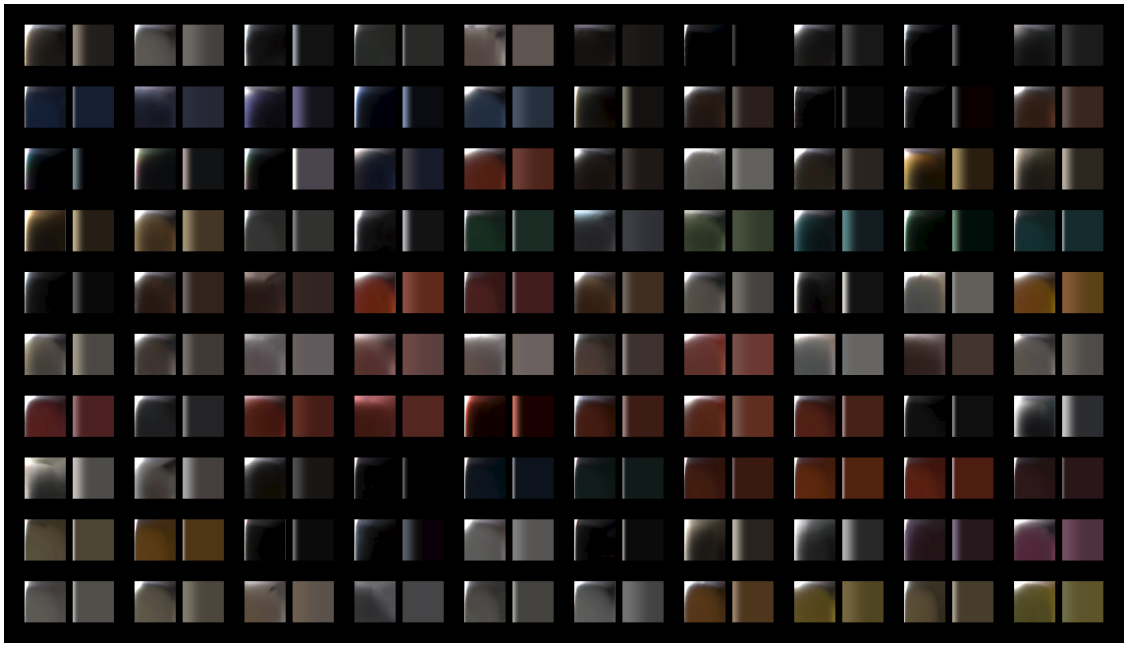


Figure A.2: BRDF slices obtained by fitting the Blinn-Phong BRDF to the MERL 100 dataset. For more information, see the caption of Figure A.1.

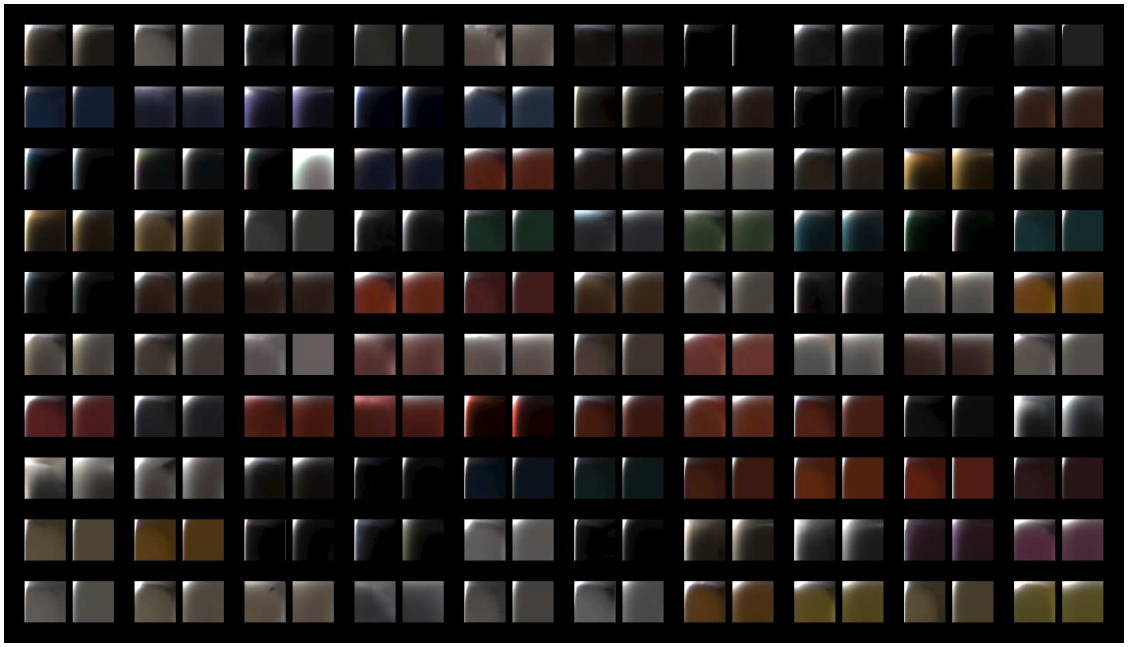


Figure A.3: BRDF slices obtained by fitting the Substance BRDF to the MERL 100 dataset. For more information, see the caption of Figure A.1.

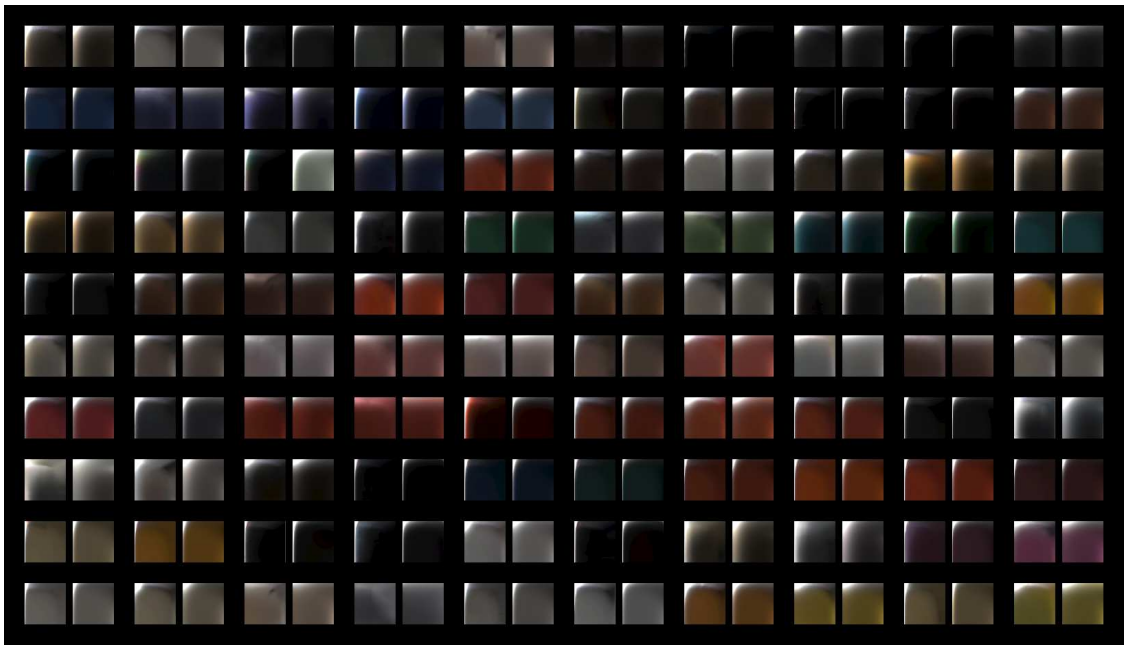
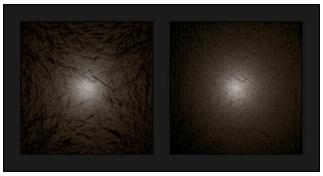
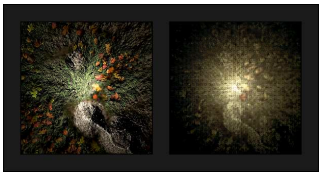


Figure A.4: BRDF slices obtained by fitting the Disney BRDF to the MERL 100 dataset. For more information, see the caption of Figure A.1.

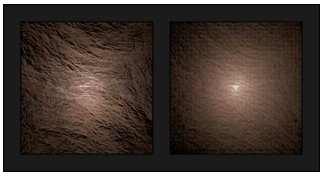
A.2 Training Dataset



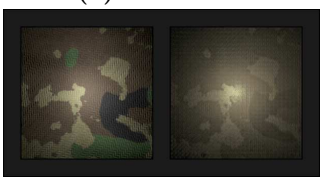
(a) SP 1 Mud



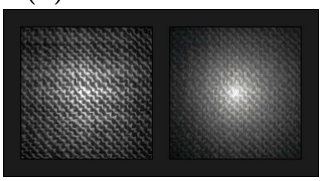
(b) Forest Ground 001



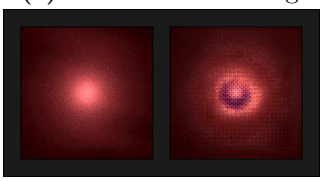
(c) SP 1 Fur Cow Long



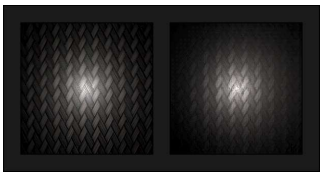
(d) Camouflage 001



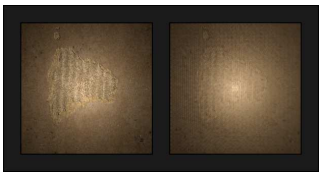
(e) Fabric Suit Vintage



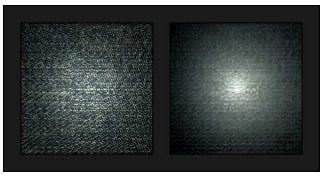
(f) SP 1 Fabric Silk



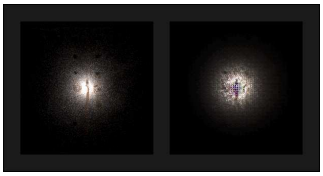
(g) Carbon Fiber



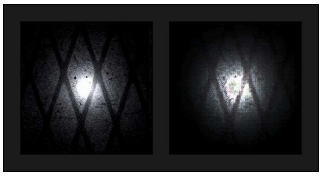
(h) SP 1 Cardboard



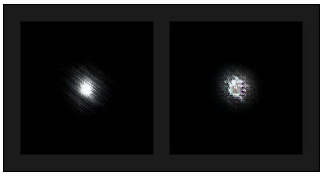
(i) SP 1 Jeans



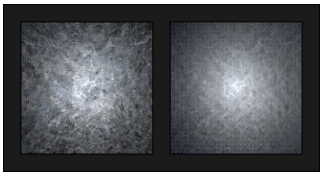
(j) Metal Panels 001



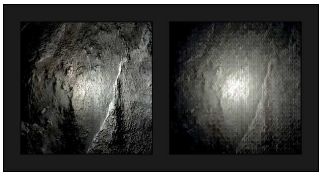
(k) SP 1 Metal Plate 011



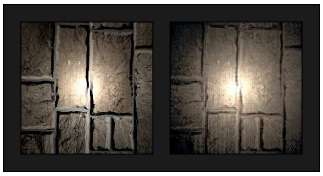
(l) SP 1 Metal Steel Brushed



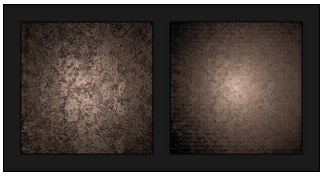
(m) Granite 001



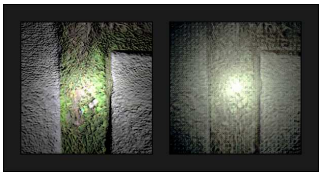
(n) Rock 012 Bitmap



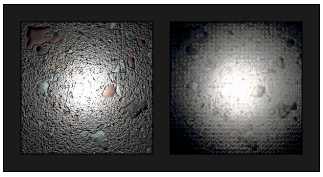
(o) Stacked Rectangular Stones



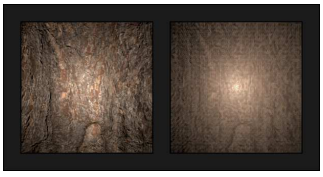
(p) Classic Brown Concrete



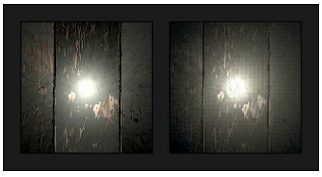
(q) Garden Tiles 001 Bitmap



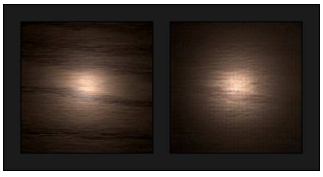
(r) SP 1 Concrete 011



(s) SP 1 Bark Black Pine



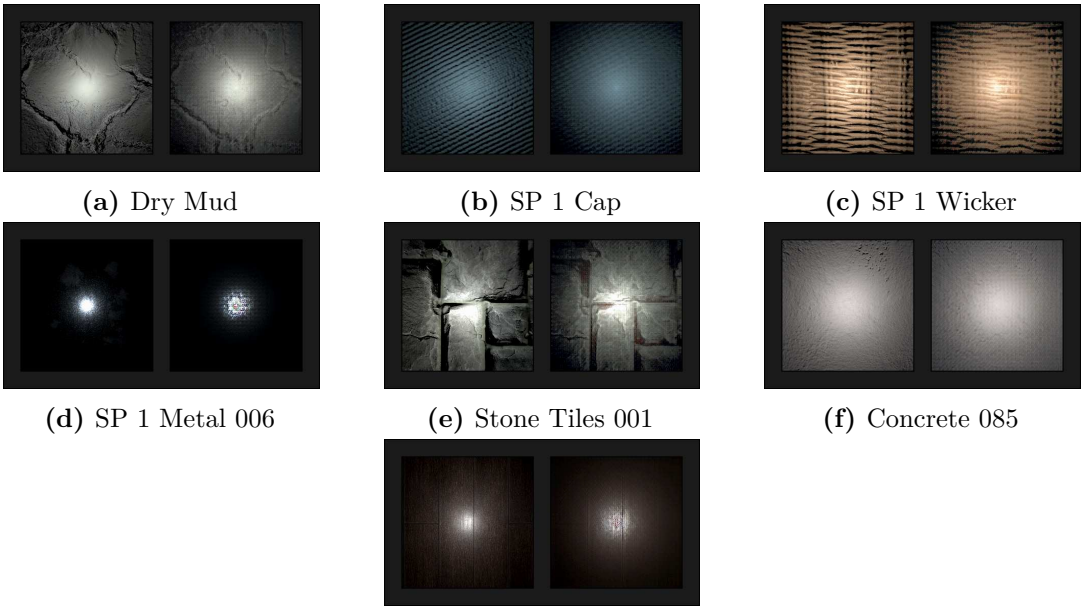
(t) SP 1 Old Painted Planks



(u) Wood 024 Walnut

Figure A.5: Reconstruction results for the training dataset. The left image in Figure A.5b through Figure A.5u represents the input to the SVBRDF Autoencoder while the right image is the colour-matched output under the same lighting and viewing conditions.

A.3 Validation Dataset



(a) Dry Mud **(b)** SP 1 Cap **(c)** SP 1 Wicker

(d) SP 1 Metal 006 **(e)** Stone Tiles 001 **(f)** Concrete 085

(g) Wenge

Figure A.6: Reconstruction results for the validation dataset. For more information, see the caption of Figure A.5.

B
Source Code

Contents

This appendix details instructions for executing the accompanying source code and gives an example of a commented YAML configuration file.

B.1 Execution Guide

The accompanying source code is partitioned across two directories: `project` and `classic`. As their names suggest, `project` contains the implementation of the research project while `classic` holds the implementation of the texture synthesis algorithms discussed in Section 2.3.3. The execution entry point in both repositories is `main.py`. To run the research project[1], it suffices to download PyTorch [PGM+19], install the dependencies listed in `requirements.txt`, and then invoke `main.py`. Note that the project dependencies can be trivially installed with `pip` using

```
pip install -r requirements.txt
```

[1]The same instructions apply to the `classic` module (except PyTorch is not required).

Furthermore, the command-line arguments to either module is given by running

```
python main.py -h
```

Note that the accompanying source code has been tested on Windows 10 and Ubuntu 18.04 with Python versions 3.7.5 and 3.8.0. For the sake of convenience, the `project` module includes the parameters of the final SVBRDF Autoencoder model as well as the *Earth - Dry Mud* texture[2]. Lastly, HTML documentation for both the `classic` and `project` modules can be found in the `doc` directory.

B.2 Example Configuration

The following YAML configuration is compatible with the `relight` flow which reconstructs an input texture under arbitrary lighting and viewing conditions with respect to a virtual camera.

```yaml
# The Flow value specifies parameters that control the flow of an experiment.
Flow:
    Input Path: data/Earth - Dry Mud/Sample 0.png
    Output Path: results/relight.png

# The Network value specifies the path to the SVBRDF Autoencoder configuration.
Network: configs/network.yaml

# The SVBRDF value specifies how to interpret the SVBRDF Autoencoder output.
SVBRDF:
    Type: Disney

# The Lights, Viewer, and Camera values specify the context of the rendering.
Lights:
    - Type: Punctual
      Position: [1, 1, 1]
      Lumens: [2, 2, 2]
Viewer:
    Type: Perspective
    Position: [-1.3, -0.8, 1.2]
Camera:
    Type: Perspective
    Position: [-1.3, -0.8, 1.2]
    Direction: [1.5, 1.0, -1.6]
    Field of View: [64, 36]
    Resolution: [1280, 720]
    Exposure: 1
```

[2]The remainder of the dataset can be retrieved from the links provided in Appendix C.

C Dataset

This appendix lists the sources of the textures that compose the testing, validation, and training datasets in Table C.1, Table C.2, and Table C.3, respectively.

Texture	**Source**
Earth - SP 1 Fur Cow Short	`https://share.substance3d.com/libraries/1748`
Fabric - SP 1 Fabric Wool Fluffy	`https://share.substance3d.com/libraries/1754`
Industry - SP 1 Backpack Padding	`https://share.substance3d.com/libraries/1775`
Metal - SP 1 Metal Dumpster	`https://share.substance3d.com/libraries/1786`
Rock - Ice 001 Bitmap	`https://share.substance3d.com/libraries/714`
Urban - Concrete 063	`https://share.substance3d.com/libraries/1380`
Wood - SP 1 Wood Board 1	`https://share.substance3d.com/libraries/1772`

Table C.1: Sources of the textures in the testing dataset.

Texture	**Source**
Earth - Dry Mud	`https://share.substance3d.com/libraries/1101`
Fabric - SP 1 Cap	`https://share.substance3d.com/libraries/1756`
Industry - SP 1 Wicker	`https://share.substance3d.com/libraries/1763`
Metal - SP 1 Metal 006	`https://share.substance3d.com/libraries/1785`
Rock - Stone Tiles 001	`https://share.substance3d.com/libraries/705`
Urban - Concrete 085	`https://share.substance3d.com/libraries/1381`
Wood - Wenge	`https://share.substance3d.com/libraries/1168`

Table C.2: Sources of the textures in the validation dataset.

Texture	**Source**
Earth - Forest Ground 001	`https://share.substance3d.com/libraries/702`
Earth - SP 1 Fur Cow Long	`https://share.substance3d.com/libraries/1747`
Earth - SP 1 Mud	`https://share.substance3d.com/libraries/1679`
Fabric - Camouflage 001	`https://share.substance3d.com/libraries/710`
Fabric - Fabric Suit Vintage	`https://share.substance3d.com/libraries/1100`
Fabric - SP 1 Fabric Silk	`https://share.substance3d.com/libraries/1758`
Industry - Carbon Fiber	`https://share.substance3d.com/libraries/578`
Industry - SP 1 Cardboard	`https://share.substance3d.com/libraries/1777`
Industry - SP 1 Jeans	`https://share.substance3d.com/libraries/1782`
Metal - Metal Panels 001	`https://share.substance3d.com/libraries/703`
Metal - SP 1 Metal Plate 011	`https://share.substance3d.com/libraries/1787`
Metal - SP 1 Metal Steel Brushed	`https://share.substance3d.com/libraries/1803`
Rock - Granite 001	`https://share.substance3d.com/libraries/709`
Rock - Rock 012 Bitmap	`https://share.substance3d.com/libraries/708`
Rock - Stacked Rectangular Stones	`https://share.substance3d.com/libraries/1169`
Urban - Classic Brown Concrete	`https://share.substance3d.com/libraries/1378`
Urban - Garden Tiles 001 Bitmap	`https://share.substance3d.com/libraries/725`
Urban - SP 1 Concrete 011	`https://share.substance3d.com/libraries/1751`
Wood - SP 1 Bark Black Pine	`https://share.substance3d.com/libraries/1776`
Wood - SP 1 Old Painted Planks	`https://share.substance3d.com/libraries/1771`
Wood - Wood 024 Walnut	`https://share.substance3d.com/libraries/731`

Table C.3: Sources of the textures in the training dataset.

Bibliography

[AAB+15] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[AAL16] Miika Aittala, Timo Aila, and Jaakko Lehtinen. Reflectance modeling by neural texture synthesis. *ACM Trans. Graph.*, 35(4), July 2016.

[Adoa] Adobe. Adobe Stock 3D Assets. https://stock.adobe.com/3d-assets.

[Adob] Adobe. Substance. https://www.substance3d.com/.

[All15a] Allegorithmic. Stacked rectangular stones, December 2015. https://share.substance3d.com/libraries/1169.

[All15b] Allegorithmic. Wood 024 Walnut, July 2015. https://share.substance3d.com/libraries/731.

[AS17] Doris Antensteiner and Svorad Stolc. Full brdf reconstruction using cnns from partial photometric stereo-light field data. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 13–21, 2017.

[Ash01] Michael Ashikhmin. Synthesizing natural textures. In *Proceedings of the 2001 symposium on Interactive 3D graphics*, pages 217–226, 2001.

[Bay63] Thomas Bayes. Lii. an essay towards solving a problem in the doctrine of chances. by the late rev. mr. bayes, frs communicated by mr. price, in a letter to john canton, amfr s. *Philosophical transactions of the Royal Society of London*, (53):370–418, 1763.

[BJV17] Urs Bergmann, Nikolay Jetchev, and Roland Vollgraf. Learning texture manifolds with the periodic spatial gan, 2017.

[Bli77] James F. Blinn. Models of light reflection for computer synthesized pictures. In *Proceedings of the 4th Annual Conference on Computer Graphics and*

Interactive Techniques, SIGGRAPH '77, page 192–198, New York, NY, USA, 1977. Association for Computing Machinery.

[BLNZ95] Richard H. Byrd, Peihuang Lu, Jorge Nocedal, and Ciyou Zhu. A limited memory algorithm for bound constrained optimization. *SIAM J. Sci. Comput.*, 16(5):1190–1208, September 1995.

[BS12] Brent Burley and Walt Disney Animation Studios. Physically-based shading at disney. In *ACM SIGGRAPH*, volume 2012, pages 1–7, 2012.

[CT82] Robert L. Cook and Kenneth E. Torrance. A reflectance model for computer graphics. *ACM Trans. Graph.*, 1(1):7–24, January 1982.

[DAD+18] Valentin Deschaintre, Miika Aittala, Fredo Durand, George Drettakis, and Adrien Bousseau. Single-image svbrdf capture with a rendering-aware deep network. *ACM Transactions on Graphics*, 37(4):1–15, Aug 2018.

[DBB18] Philip Dutre, Philippe Bekaert, and Kavita Bala. *Advanced Global Illumination*. CRC Press, 2018.

[DCF+15] Emily L Denton, Soumith Chintala, Rob Fergus, et al. Deep generative image models using a laplacian pyramid of adversarial networks. In *Advances in neural information processing systems*, pages 1486–1494, 2015.

[DDS+09] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.

[Des37] Réne Descartes. *Discours de la méthode pour bien conduire sa raison et chercher la vérité dans les sciences. Plus la Dioptrique. Les Meteores. Et la Geometrie qui sont des essais de cete [sic] methode [par Descartes].* Ian Maire, 1637.

[DLT+20] Anna Darzi, Itai Lang, Ashutosh Taklikar, Hadar Averbuch-Elor, and Shai Avidan. Co-occurrence based texture synthesis, 2020.

[DSTB14] Alexey Dosovitskiy, Jost Tobias Springenberg, Maxim Tatarchenko, and Thomas Brox. Learning to generate chairs, tables and cars with convolutional networks, 2014.

[DV16] Vincent Dumoulin and Francesco Visin. A guide to convolution arithmetic for deep learning, 2016.

[EL99] Alexei A Efros and Thomas K Leung. Texture synthesis by non-parametric sampling. In *Proceedings of the seventh IEEE international conference on computer vision*, volume 2, pages 1033–1038. IEEE, 1999.

[FAW19] Anna Frühstück, Ibraheem Alhashim, and Peter Wonka. Tilegan: Synthesis of large-scale non-homogeneous textures. *ACM Trans. Graph.*, 38(4), July 2019.

[Fre21] Augustin Fresnel. Note sur le calcul des teintes que la polarisation développe dans les lames cristallisées. In *Annales de Chimie et Physique*, volume 17, pages 101–112, 1821.

[GB20] Rémi Giraud and Yannick Berthoumieu. Texture superpixel clustering from patch-based nearest neighbor matching, 2020.

[GEB15] Leon A. Gatys, Alexander S. Ecker, and Matthias Bethge. Texture synthesis using convolutional neural networks. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'15, page 262–270, Cambridge, MA, USA, 2015. MIT Press.

[GPAM+14] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks, 2014.

[GRR+18] Stamatios Georgoulis, Konstantinos Rematas, Tobias Ritschel, Efstratios Gavves, Mario Fritz, Luc Van Gool, and Tinne Tuytelaars. Reflectance and natural illumination from single-material specular objects using deep learning. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 40(8):1932–1947, 2018.

[GTGB84] Cindy M. Goral, Kenneth E. Torrance, Donald P. Greenberg, and Bennett Battaile. Modeling the interaction of light between diffuse surfaces. *SIGGRAPH Comput. Graph.*, 18(3):213–222, January 1984.

[HB95] David J. Heeger and James R. Bergen. Pyramid-based texture analysis/synthesis. In *Proceedings of the 22nd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '95, page 229–238, New York, NY, USA, 1995. Association for Computing Machinery.

[HDCD15] Eric Heitz, Jonathan Dupuy, Cyril Crassin, and Carsten Dachsbacher. The sggx microflake distribution. *ACM Trans. Graph.*, 34(4), July 2015.

[HDR19] Yiwei Hu, Julie Dorsey, and Holly Rushmeier. A novel framework for inverse procedural texture modeling. *ACM Trans. Graph.*, 38(6), November 2019.

[HK93] Pat Hanrahan and Wolfgang Krueger. Reflection from layered surfaces due to subsurface scattering. In *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '93, page 165–174, New York, NY, USA, 1993. Association for Computing Machinery.

[HOJ08] Toshiya Hachisuka, Shinji Ogaki, and Henrik Wann Jensen. Progressive photon mapping. In *ACM SIGGRAPH Asia 2008 papers*, pages 1–8. 2008.

[IS15] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift, 2015.

[JAFF16] Justin Johnson, Alexandre Alahi, and Li Fei-Fei. Perceptual losses for real-time style transfer and super-resolution, 2016.

[JAM+10] Wenzel Jakob, Adam Arbree, Jonathan T. Moon, Kavita Bala, and Steve Marschner. A radiative transfer framework for rendering materials with anisotropic structure. *ACM Trans. Graph.*, 29(4), July 2010.

[JBV16] Nikolay Jetchev, Urs Bergmann, and Roland Vollgraf. Texture synthesis with spatial generative adversarial networks, 2016.

[JOP+] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2001–.

[KB14] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014.

[KG13] Brian Karis and Epic Games. Real shading in unreal engine 4. *Proc. Physically Based Shading Theory Practice*, 4:3, 2013.

[KGT+17] Kihwan Kim, Jinwei Gu, Stephen Tyree, Pavlo Molchanov, Matthias Nießner, and Jan Kautz. A lightweight approach for on-the-fly reflectance estimation. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 20–28, 2017.

[Khr] Khronos Group. OpenGL - The Industry's Foundation for High Performance Graphics. https://www.khronos.org/opengl/.

[KNL+15] Alexandre Kaspar, Boris Neubert, Dani Lischinski, Mark Pauly, and Johannes Kopf. Self tuning texture optimization. *Comput. Graph. Forum*, 34(2):349–359, May 2015.

[KUMH17] Günter Klambauer, Thomas Unterthiner, Andreas Mayr, and Sepp Hochreiter. Self-normalizing neural networks, 2017.

[KW13] Diederik P Kingma and Max Welling. Auto-encoding variational bayes, 2013.

[KWKT15] Tejas D Kulkarni, William F. Whitney, Pushmeet Kohli, and Josh Tenenbaum. Deep convolutional inverse graphics network. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems 28*, pages 2539–2547. Curran Associates, Inc., 2015.

[Lab] Labsphere. Spectralon Diffuse Reflectance Material. https://www.labsphere.com/labsphere-products-solutions/materials-coatings-2/coatings-materials/spectralon/.

[Lag12] Lagarde, Sébastien. Spherical Gaussian approximation for Blinn-Phong, Phong and Fresnel, June 2012. https://seblagarde.wordpress.com/2012/06/03/spherical-gaussien-approximation-for-blinn-phong-phong-and-fresnel/.

[Lam60] Johann Heinrich Lambert. *Photometria sive de mensura et gradibus luminis, colorum et umbrae*. Klett, 1760.

[LDPT17] Xiao Li, Yue Dong, Pieter Peers, and Xin Tong. Modeling surface appearance from a single photograph using self-augmented convolutional neural networks. *ACM Transactions on Graphics*, 36(4):1–11, Jul 2017.

[Lew94] Robert R Lewis. Making shaders more physically plausible. In *Computer Graphics Forum*, volume 13, pages 109–120. Wiley Online Library, 1994.

[LFY+17] Yijun Li, Chen Fang, Jimei Yang, Zhaowen Wang, Xin Lu, and Ming-Hsuan Yang. Diversified texture synthesis with feed-forward networks. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 266–274, 2017.

[LLC+10] Ares Lagae, Sylvain Lefebvre, Rob Cook, Tony DeRose, George Drettakis, David S Ebert, John P Lewis, Ken Perlin, and Matthias Zwicker. A survey of procedural noise functions. In *Computer Graphics Forum*, volume 29, pages 2579–2600. Wiley Online Library, 2010.

[LSC18] Zhengqin Li, Kalyan Sunkavalli, and Manmohan Chandraker. Materials for masses: Svbrdf acquisition with a single mobile phone image, 2018.

[Lun] LunaPic. LunaPic. https://www7.lunapic.com/editor/.

[LW93] Eric P Lafortune and Yves D Willems. Bi-directional path tracing. 1993.

[LW16] Chuan Li and Michael Wand. Precomputed real-time texture synthesis with markovian generative adversarial networks, 2016.

[Mit97] Tom M Mitchell. *Machine learning*. McGraw-Hill computer science series. McGraw-Hill, New York ; London, international edition edition, 1997.

[MMZ+18] Abhimitra Meka, Maxim Maximov, Michael Zollhoefer, Avishek Chatterjee, Hans-Peter Seidel, Christian Richardt, and Christian Theobalt. Lime: Live intrinsic material estimation. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 6315–6324, 2018.

[MPBM03] Wojciech Matusik, Hanspeter Pfister, Matt Brand, and Leonard McMillan. A data-driven reflectance model. *ACM Transactions on Graphics*, 22(3):759–769, July 2003.

[ODO16] Augustus Odena, Vincent Dumoulin, and Chris Olah. Deconvolution and checkerboard artifacts. *Distill*, 2016.

[Pdo] pdoc. https://pdoc3.github.io/pdoc/.

[Per85] Ken Perlin. An image synthesizer. *SIGGRAPH Comput. Graph.*, 19(3):287–296, July 1985.

[Per02] Ken Perlin. Improving noise. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 681–682, 2002.

[PGM+19] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[PH89] Ken Perlin and Eric M Hoffert. Hypertexture. In *Proceedings of the 16th annual conference on Computer graphics and interactive techniques*, pages 253–262, 1989.

[Rei12] Reinalter, Stefan. Real-time radiosity, May 2012. https://blog.molecular-matters.com/2012/05/04/real-time-radiosity/.

[RFB15] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation, 2015.

[RMC15] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks, 2015.

[Ros58] Frank Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, pages 65–386, 1958.

[Rus98] Szymon M Rusinkiewicz. A new change of variables for efficient brdf representation. In *Eurographics Workshop on Rendering Techniques*, pages 11–22. Springer, 1998.

[RWS+11] Peiran Ren, Jiaping Wang, John Snyder, Xin Tong, and Baining Guo. Pocket reflectometry. *ACM Trans. Graph.*, 30(4), July 2011.

[Sch94] Christophe Schlick. An inexpensive brdf model for physically-based rendering. In *Computer graphics forum*, volume 13, pages 233–246. Wiley Online Library, 1994.

[SDSY16] Jian Shi, Yue Dong, Hao Su, and Stella X. Yu. Learning non-lambertian object intrinsics across shapenet categories, 2016.

[SHK+14] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[Smi67] Bruce Smith. Geometrical shadowing of a random rough surface. *IEEE Transactions on Antennas and Propagation*, 15(5):668–671, 1967.

[SZ14] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2014.

[TB15] Lucas Theis and Matthias Bethge. Generative image modeling using spatial lstms. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'15, page 1927–1935, Cambridge, MA, USA, 2015. MIT Press.

[Tib96] Robert Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society: Series B (Methodological)*, 58(1):267–288, 1996.

[TSH12] Yichuan Tang, Ruslan Salakhutdinov, and Geoffrey Hinton. Deep lambertian networks, 2012.

[TZL+02] Xin Tong, Jingdan Zhang, Ligang Liu, Xi Wang, Baining Guo, and Heung-Yeung Shum. Synthesis of bidirectional texture functions on arbitrary surfaces. *ACM Trans. Graph.*, 21(3):665–672, July 2002.

[ULVL16] Dmitry Ulyanov, Vadim Lebedev, Andrea Vedaldi, and Victor Lempitsky. Texture networks: Feed-forward synthesis of textures and stylized images, 2016.

[UVL16] Dmitry Ulyanov, Andrea Vedaldi, and Victor Lempitsky. Instance normalization: The missing ingredient for fast stylization, 2016.

[VCGLM19] Raquel Vidaurre, Dan Casas, Elena Garces, and Jorge Lopez-Moreno. Brdf estimation of complex materials with nested learning. In *2019 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pages 1347–1356, 2019.

[Wil79] Ralph A. Willoughby. Solutions of ill-posed problems (a. n. tikhonov and v. y. arsenin). *SIAM Rev.*, 21(2):266–267, April 1979.

[WL00] Li-Yi Wei and Marc Levoy. Fast texture synthesis using tree-structured vector quantization. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '00, page 479–488, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[WMLT07] Bruce Walter, Stephen R. Marschner, Hongsong Li, and Kenneth E. Torrance. Microfacet models for refraction through rough surfaces. In *Proceedings of the 18th Eurographics Conference on Rendering Techniques*, EGSR'07, page 195–206, Goslar, DEU, 2007. Eurographics Association.

[Wor96] Steven Worley. A cellular texture basis function. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '96, page 291–294, New York, NY, USA, 1996. Association for Computing Machinery.

[YBS+19] Ning Yu, Connelly Barnes, Eli Shechtman, Sohrab Amirghodsi, and Michal Lukac. Texture mixer: A network for controllable synthesis and interpolation of texture, 2019.

[ZZB+18] Yang Zhou, Zhen Zhu, Xiang Bai, Dani Lischinski, Daniel Cohen-Or, and Hui Huang. Non-stationary texture synthesis by adversarial expansion. *ACM Trans. Graph.*, 37(4), July 2018.