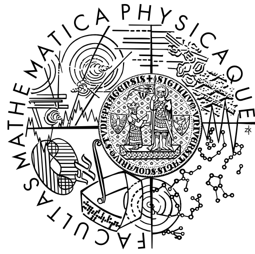


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

**MASTER THESIS**

Bc. Štěpán Hojdar

# Using neural networks to generate realistic skies

Department of Software and Computer Science Education

Supervisor of the master thesis: doc. Ing. Jaroslav Křivánek, Ph.D.
Study programme: Computer Science
Study branch: Computer Graphics and Game Development

Prague 2019

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ........ date ............ signature of the author

I would like to thank doc. Ing. Jaroslav Křivánek, Ph.D. for continuous support and counsel, Tobias Rittig, M.Sc. for advice regarding photography equipment and the rest of the *Computer Graphics Group* at Charles University for the help they provided.

Title: Using neural networks to generate realistic skies

Author: Bc. Štěpán Hojdar

Department: Department of Software and Computer Science Education

Supervisor: doc. Ing. Jaroslav Křivánek, Ph.D., Department of Software and Computer Science Education

Abstract: Environment maps are widely used in several computer graphics fields, such as realistic architectural rendering or computer games as sources of the light in the scene. Obtaining these maps is not easy, since they have to have both a high-dynamic range as well as a high resolution. As a result, they are expensive to make and the supply is limited.

Deep neural networks are a widely unexplored research area and have been successfully used for generating complex and realistic images like human portraits. Neural networks perform well at predicting data from complex models, which are easily observable, such as photos of the real world.

This thesis explores the idea of generating physically plausible environment maps by utilizing deep neural networks known as generative adversarial networks. Since a skydome dataset is not publicly available, we develop a scalable capture process with both low-end and high-end hardware. We implement a pipeline to process the captured data before feeding it to a network and extend an already existing network architecture to generate HDR environment maps. We then run a series of experiments to determine the quality of the results and uncover the directions of possible further research.

Keywords: deep learning, generative adversarial networks, image-based lighting

# Contents

**Introduction** **3**
Motivation . . . 3
Goals . . . 4
Thesis outline . . . 5

**1 Theoretical background** **7**
1.1 Deep neural networks terminology . . . 7
1.2 Convolutional networks . . . 7
1.3 Generative models . . . 10

**2 Related work** **17**
2.1 Stability of GANs . . . 17
2.2 GAN architectures . . . 18
2.3 High dynamic range networks . . . 19
2.4 Simulation based approaches . . . 20

**3 Dataset** **21**
3.1 Dataset acquisition . . . 21
3.2 Dataset processing . . . 28

**4 Neural network** **37**
4.1 Network architecture . . . 37
4.2 Network modifications . . . 41
4.3 Implementation details . . . 47

**5 Results** **51**
5.1 Dataset acquisition . . . 51
5.2 Network experiments . . . 59

**6 Future work** **99**
6.1 Spherical convolutions . . . 99
6.2 Skyline separation . . . 100
6.3 Latent space mapping . . . 101
6.4 Providing more input to the network . . . 102
6.5 Addition instead of generation . . . 102
6.6 Increasing the resolution . . . 103

**Conclusion** **105**

**Bibliography** **107**

**List of Figures** **113**

**List of Tables** **115**

**List of Abbreviations** **117**

**A Attachments** **119**
A.1 Digital attachments . . . 119
A.2 Data from the internet . . . 121
A.3 Dataset shooting manual . . . 122

# Introduction

In this chapter we explain the motivation behind our work, as well as the main goals we hope to achieve and the reasoning behind them. We also provide a general outline of the thesis to help the reader navigate the text.

## Motivation

The term computer graphics encompasses a wide field of different disciplines. One of the more prominent ones is realistic rendering of scenes created by a 3D artist, known to the general public as *Computer-generated imagery* or *CGI*. In order to make these scenes look realistic, the artist has to utilize assets captured in the real world – mainly textures and material properties. In order to achieve realistic lighting of the scene, the artists can use environment maps to get the radiance values of the sky. An environment map is a high dynamic range, 360° photograph of the entire surroundings of the camera, as shown in Figure 1.

Figure 1: An example of a low dynamic range environment map. Since the map covers all 180 polar degrees, we can see the camera tripod in the bottom of the image.

As we already mentioned, these maps have to have a high dynamic range of luminance values, since some elements of the sky (the Sun or sunlit clouds) are brighter than the rest of the scene by orders of magnitude, referred to as HDR images. Artists also require large resolution for these images – the current standard is 15000 by 7500 pixels, which is roughly 110 megapixels in every image. These demands make capturing the maps require a significant amount of manual work, as well as special and expensive equipment.

Having a fixed set of images does not give the artist a lot of creative freedom either, since these environment maps cannot be adjusted in any way, except for horizontal (azimuthal) rotation. If the artist likes a certain environment, but the scene requires more clouds than the image contains, there is no way to increase the cloud coverage in the image and the artist has to discard it.

Another common task in architectural visualization is rendering a given building several times, simulating the look of the building over the course of the whole day. This requires a sequence of skydome images, shot in the same place over the course of several hours. Similar problem arises in movie renderings, where the rendered assets (like buildings or characters) have to be lit correctly, corresponding to the time of the day in the movie. This, again, requires a sequence of environment maps, since both the clouds and the Sun in the sky have to move in the movie.

## Goals

The above requirements could be solved by generating synthetic environment maps, which the artists could use in their projects. This could both give the artists greater control over the generated results, as well as allow them to generate sequences of images with realistic cloud movement.

The goals of this thesis are based on the requirements we mentioned so far. In order to be usable in *CGI*, the generated environment maps have to fulfill the following criteria:

- **Realistic looking results** – the generated results need to look physically plausible. This includes both the cloud formations, as well as the lighting of the sky.
- **High dynamic range (HDR)** – as we already mentioned, the goal is to use these environment maps as sources of light in the scene. High dynamic range is very important, since the difference of radiance emitted from different surfaces is greater than the range of RGB images (e.g. the Sun is a thousand times brighter than a streetlamp).
- **High image resolution** – while not as important as the others, image resolution is still a constraint we have to keep in mind. If the environment map has a low resolution, the sky and any reflections (such as water surfaces) will look blurry and reduce the realism of the scene.

While this list is not exhaustive, it gives a good high-level idea of what is required of the results to be used by the general public. Once these requirements are met, other goals (like giving the artist the tools to influence the generation) can be taken into consideration.

There have recently been significant improvements in utilizing neural networks to generate images from complex data models (like generating human portraits). Neural networks have performed well in this discipline but their capabilities of generating high dynamic range data are largely unexplored.

In this thesis we focus on generating skydome images by utilizing neural networks. Training neural networks requires a large dataset and because there is no publicly available HDR dataset of skydome data, we develop a complete method to acquire this dataset and use it to expand a dataset gathered from various sources online. We utilize an already existing neural network, run a few experiments to see if we can match the image quality produced by the authors in our smaller dataset. We implement additional layers with the goal of improving the

image quality of our results. We also convert the state of the art network for low dynamic range images to generate high dynamic range images and perform experiments on this network. These experiments aim both to improve the visual quality of the generated images as well as to evaluate the generated dynamic range. The goal of this thesis is to determine if the three goals of generating HDR environment maps can be fulfilled by a neural network.

In conclusion, our contributions are as follows:

- We develop a complete pipeline to capture and process skydome data as a dataset for our neural network.
- We utilize this pipeline to expand a dataset we collected from various sources online.
- We convert a state of the art neural network to generate high dynamic range (HDR) images.
- We implement two neural network layers in an attempt to improve the generated image quality. We run experiments to evaluate our new layers, as well as the precision of the neural network.
- We provide several ideas for future improvement of our results, which we believe solve some of the issues we encountered during this work.

# Thesis outline

In the following chapters, we first provide a theoretical background necessary to understand both the related work and our own (Chapter 1). In Chapter 2 we give an overview of the existing research in the image generation research area. Chapter 3 explains all of the research we did to design the best dataset acquisition method and provides reasoning for both the hardware and software choices we made. Chapter 4 describes in more detail the neural network we used for our research, as well as introduces and explains new layers we implemented and the reasons behind them. In Chapter 5 we present all our results, both in dataset acquisition and neural network training. Afterwards we highlight several approaches for future improvement of this research in Chapter 6. Finally, in the chapter Conclusion we summarize our work and the goals we achieved.

# 1. Theoretical background

This chapter is intended to define the terms we will use in the thesis, as well as to review the fundamental concepts of neural networks and their applications in image fabrication.

## 1.1 Deep neural networks terminology

*Deep neural networks* are machine learning models, which have been gaining popularity over the last few years. By adjusting its parameters $\theta$ a neural network is trying to best approximate a function $f$. The function can perform any arbitrary task, for example, a classifier function $f(x) = y$ will, given an example $x$, classify the example into a category $y$. An image generating function can take a vector of features $x$ and generate a completely new image $y$.

When talking about neural networks in this work, we usually mean *feedforward networks*. This means that the data flows from the network from the input, through its layers, directly to the output – without any feedback loops, which would utilize the output of the network.

In order to have better control over the network's architecture and therefore a better control over the final function $f$, we think about the networks as composed of different *layers*. These layers are smaller component functions which the input sequentially flows through. If we denote the first layer $f_1$ and the second layer as $f_2$, the output of this network will have the form of $f_2(f_1(x))$.

If we stack multiple layers together, we call the resulting network *deep*. The first layer of the network is usually called the *input layer*, the functional layers in the middle are called *hidden layers*, since we do not directly observe the outputs of these layers, and the last layer is called the *output layer*. There is no standard way of distinguishing a "shallow" network from a "deep" network, so the exact wording is not crucial. Deep neural networks can have four layers or one hundred layers. *Network architecture* is the particular way a single network is composed into different layers – a network composed of two layers $f_2(f_1(x))$ and a network composed of a different number of different layers $g_3(g_2(g_1(x)))$ have different architectures, though their purpose might be the same.

By neural network *training* we mean adjusting the parameters $\theta$ in such a way, that the network's approximation of the function $f$ improves. This is done by minimizing the *loss function* which is a function approximating the inaccuracy of the prediction of $f$.

We do not go into any more detail in this section, instead we direct the reader to a great resource of Goodfellow et al. [2016], which covers this topic extensively.

## 1.2 Convolutional networks

*Convolutional neural networks*, or *CNNs* for short, are a widely adopted type of neural networks, first proposed by LeCun et al. [1989]. They are mainly used in the image processing field for all different kinds of image based tasks, be it object classification, object detection or content generation. A *convolutional network* is

simply a network which utilizes mathematical convolution in at least one of its layers [Goodfellow et al., 2016]. Computing the convolution over the whole image means computing $S(i,j)$ for every $[i,j]$ – a pixel of the image:

$$S(i,j) = \sum_{m=0}^{K_1} \sum_{n=0}^{K_2} I(i-m, j-n)K(m,n) \tag{1.1}$$

where $K_1$ and $K_2$ are the dimensions of the *convolutional filter (kernel)*. This convolutional filter gets applied over the input image. While it is possible to apply the convolutional filter to every pixel of the image, as the Equation 1.1 suggests, it is often changed to move "faster" across the image skipping some pixels in the process. This speed of the convolutional filter is called the *stride*.

While we describe a convolution as a 2D operation, there are several 3D convolutions happening in a single convolutional layer. If we have an input feature volume with dimensions $(H, W, C)$ and specify the kernel size as $(3, 3)$ and the number of output channels $C_1 = 16$, a single convolutional filter used in the layer will have the dimensions $(3, 3, C)$ and there will be $C_1 = 16$ of them. Each of these sixteen filters will produce one feature channel in the output volume. Each of the feature channels receives the same information from the input feature volume, but uses different weights of the kernel to compute the convolution. This single convolutional layer is also illustrated in Figure 1.1 with $C = 3$.

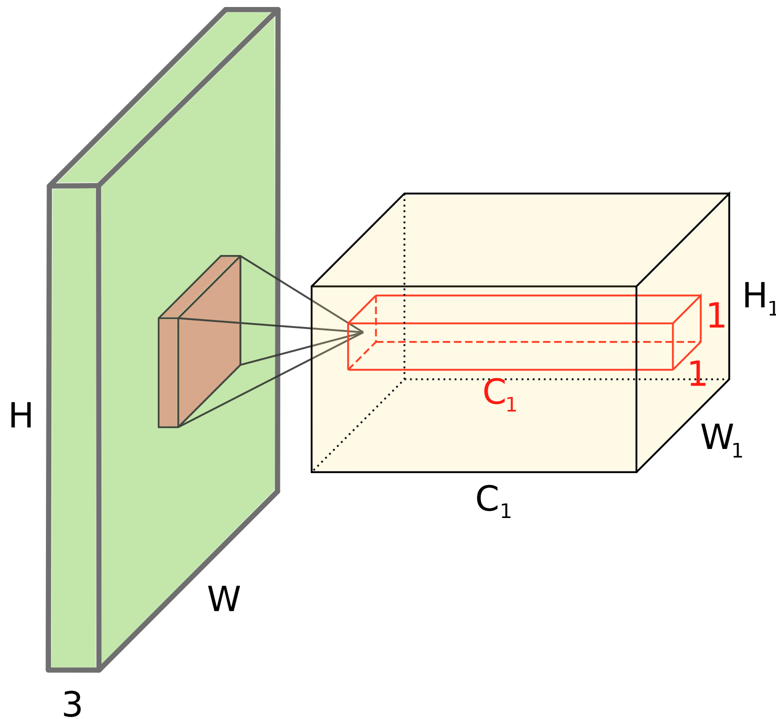



Figure 1.1: An illustration of a single convolutional layer filtering. The original image with dimensions $(H, W, 3)$ gets transformed into convolutional features with dimensions $(H_1, W_1, C_1)$. The red area illustrates one single $(H, W, 3)$ kernel computing one channel of the output feature volume.

A convolutional layer usually consists of the convolution itself, as well as a *pooling layer*. The pooling layer somehow locally combines the outputs of the convolution. There are several statistics, which can perform this function such

as averaging a neighborhood, computing different norms of the neighborhood, or simply just taking the maximum of the neighborhood. The last example is called *max pooling* and is one of the better known variants. Pooling helps to make the calculated representation invariant to small translations, which is a very useful property [Goodfellow et al., 2016].

Goodfellow et al. [2016] (chapter 9, figures 9.18 and 9.19) also provides us with a deeper understanding of convolutional layers. Strong evidence suggests that lower level convolutional layers (the first layers to see the image) tend to learn response functions similar to response functions in the primary visual cortex in the human brain. The original authors LeCun et al. [1989] also mention this in their text. After a convolution is applied to an image, we call the output of the layer the image's *features*, largely because we expect that the convolutional layer extracted some important information about the image by using its filters.

**Transposed convolution** Because this thesis focuses on generative models, we need to also mention the *transposed convolutional layer*, which is often used to fabricate delicate image details, instead of condensing image details into more general features, like the standard convolutional layer. Transposed convolution provides the main functionality in image synthesis networks, taking a lower resolution image (or image representation, in features), and "dreaming" new details in the higher resolution. This is often utilized for both super-resolution tasks (enlarging an already existing image into a higher resolution and keeping the same level of detail) as well as synthesizing a completely new image. We will discuss image synthesis at length in the next section.

The article of Dumoulin and Visin [2016] provides a detailed explanation for both types of convolution. We will briefly explain how transposed convolution works and refer the reader to said article for a more in-depth analysis.

A simple way of calculating a convolution is transforming the kernel into a sparse matrix which, when multiplied with a vector containing the flattened input image, produces the result of the convolution. The shape of the matrix is easily expressed as $A \times B$, where $A$ is the number of pixels after the convolution, and $B$ is the number of pixels before the convolution. A $4 \times 16$ matrix would take a $4 \times 4$ image and reduce it to a $2 \times 2$ using a convolution (with a filter size $3 \times 3$). Since the pattern for constructing the sparse matrix is somewhat complicated, we again refer the reader to Dumoulin and Visin [2016] for details.

Once we realise that convolutions can be expressed as the already mentioned $A \times B$ matrix, the natural question which arises is: "What happens if we transpose the matrix to be $B \times A$". The answer to this question is precisely a *transposed convolution*, also sometimes referred to as a *deconvolution*. Since the term *deconvolution* can also refer to the inverse of convolution (which is not equal to the transposed convolution), we will avoid it and use the term *transposed convolution*.

A transposed convolution with a $16 \times 4$ matrix can be multiplied by a vector of length 4, producing a vector of length 16. This means it is possible to use this matrix to upscale images from the resolution $2 \times 2$ to the resolution of $4 \times 4$. This approach is very often exploited in super-resolution networks and generative models alike.

## 1.3 Generative models

*Generative models* are neural networks built and trained for the task of generating new images from a given distribution. The network is given a training dataset of images, and its goal is to learn to generate images similar to the training images. It is an unsupervised machine learning method, as the data is not labeled in any way.

There are several measures of the quality of the generative network's output. We want the images to be as *similar* to the images in the dataset as possible – meaning the distribution of the model should be similar to the data distribution of the dataset. However, a network focused purely on the quality of reproduction has no room for modification – it will learn to replicate the images of the dataset. This is counteracted by the second measure we seek to maximize – the *generalisation* of the network. A network *generalises* well if it is able to produce new images, which are not contained in the dataset. A high quality generative network will, therefore, be able to produce images that look to have come from the dataset, but are completely new. These two metrics of quality are obviously opposite to each other, and the goal of the network is to strike a good balance.

There are several outcomes of training a generative model, besides the correct one. We say the network *diverged* if the network's parameters do not converge during training. This can have multiple outcomes in terms of the quality of the resulting images, but most often the images look like random noise. There is another condition of generative models called mode collapse. This refers to the network completely ignoring a few examples from the dataset – for example, on a mnist dataset [LeCun et al.] which consists of handwritten digits 0 to 9, the network might only generate ones and threes, completely omitting the other digits.

Generative models use a set $\mathcal{X}$ of realisations of a random variable $X$. This set $\mathcal{X}$ is called a dataset. This random variable can be, for example, the skydome images or human portraits. When we shoot a picture, we generate one additional sample $x \in X$. The goal of generative models is to be able to sample P(X) and thus generate new images.

Both generative models we mention in the following paragraphs utilize an unobserved latent space $Z$. This assumption gives the formula for image generation $P(x) = P(z) * P(x|z)$ for $x \in X, z \in Z$. Because the function $P(x|z)$ is complex, we utilize neural networks to learn this function.

With these terms defined and explained, we now take a brief look at the two most prominent generative model architectures – variational autoencoders and generative adversarial networks.

### 1.3.1 Variational Autoencoders

Traditional *autoencoders* are neural networks with a *encoder-decoder* architecture. The autoencoder architecture takes a single input datapoint $x \in \mathcal{X}$, encodes it using the *encoder* part of the network into a latent space variable $z \in Z$. The *decoder's* goal is to then reproduce the $x$ given the latent vector $z$, which is exactly $P(x|z)$.

However, these traditional autoencoders cannot be used to generate new im-

ages because there is no incentive for the network to fill the latent space with real data, since each training datapoint corresponds only to a single latent vector. This causes the network to not generalise very well and generate non-realistic images.

*Variational autoencoders* or *VAEs*, introduced by Kingma and Welling [2013], on the other hand, strive to fill the latent space $Z$ with real examples. This is achieved by modifying the encoder to generate not only a single latent vector, but a distribution of possible latent vectors in the latent space $Z$. The network is incentivized to fill the latent space with these distributions around each training example, which allows the user to draw a random $z \in Z$ and have the network produce a viable result.

Taking a deeper look at the *VAE* architecture, the encoder is usually denoted as $Q_\phi(z|x)$. Its goal is to take a datapoint $x \in \mathcal{X}$ and produce a distribution of latent vectors $z \in Z$. This distribution is a subspace of $Z$ and represents the data contained in $x$. In practice, the encoder generates the parameters for a gaussian distribution (the mean and the variance) and we assume the distribution of vectors $z$ corresponding to $x$ is gaussian. Since this function is usually intractable, we again approximate it using a neural network – the encoder with parameters $\phi$.

The second part of the network is the decoder, usually denoted as $P_\theta(x|z)$ and is the logical opposite to the encoder. It takes a latent vector $z \in Z$ and generates a new datapoint $x \in X$ corresponding to the $z$. As we have already mentioned, this $P(x|z)$ is approximated by the decoder network with parameters $\theta$.

Training of variational autoencoders is straightforward and can be done *end-to-end* (the whole network – all of its parameters – at the same time). The network first reduces a datapoint $x$ down to the parameters of the gaussian distribution, producing a distribution $N(\mu, \sigma^2)$. A single latent vector $z$ is then sampled from this distribution. The vector $z$ is then used to generate a datapoint $x'$ similar to $x$ (we say that $x$ was first encoded into $z$ and then decoded back to $x'$) – this architecture is sometimes called the hourglass architecture, because of its visual resemblance, as you can see in Figure 1.2.

As we have already mentioned, the variational autoencoder is incentivized to fill the latent space $Z$ with the distributions of the real data. This is accomplished through one part of the loss function, called a latent loss. This part of the loss function computes the Kullback–Leibler divergence between the $Q_\phi(z|x)$ distributions and a given prior, usually a $N(0, 1)$ distribution. This penalizes the network for not filling the latent space with its training data. The second part of the loss is called the reconstruction loss, and on the other hand penalizes the network for not reproducing the images with enough clarity. These two parts of the loss function work against each other, since to minimize the latent loss, the network would just make all of the $Q$ distributions encompass the whole latent space (and generate the average image), while minimizing the reconstruction loss would result in a zero variance distribution, which would turn the network into a simple autoencoder (since each training example $x$ would have just one latent vector $z$ assigned). We do not explain the theory behind VAEs further and instead refer the reader to the original text of Kingma and Welling [2013], as well as a more complete resource for variational inference Kingma [2017].

Variational autoencoders are widely used, mainly in applications which ben-

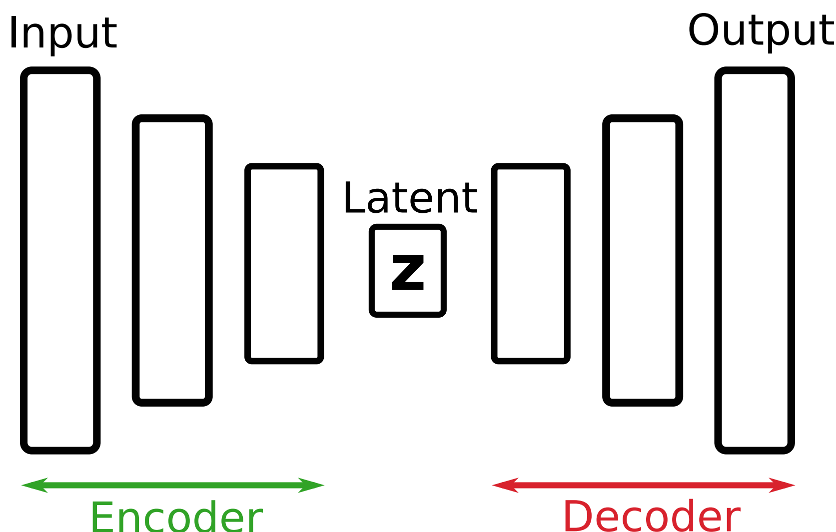



Figure 1.2: An illustration of the Variational autoencoder architecture. An encoder compresses the input image into a latent space vector $z$. A decoder can then create the same image from the compressed data.

efit from encoding a datapoint into a condensed latent space. Some examples include extracting features from images to achieve a semi-supervised learning as proposed by Kingma et al. [2014], or the synthesis of new molecules by traversing and *Stochastic Gradient Descent* optimization in the latent space $Z$, which is not possible in other molecular representations Gómez-Bombarelli et al. [2018]. Variational autoencoders have limited use for image generation, since the outputs of VAEs tend to be blurry which detracts from the realism of the generated images. The exact cause of the blurriness is not well known and is still a subject of ongoing research as is described in Kingma [2017].

### 1.3.2 Generative Adversarial Networks

*Generative adversarial networks* or *GANs* are generative neural networks taking a different approach than VAEs. They are based on game theory and contain two neural networks, which play a minimax game, competing against each other. Introduced by Goodfellow et al. [2014], GANs have been a hot topic of research since the first publication of the original paper. Since this chapter puts focus on the theoretical background, this section will explain the basic workings of a Generative adversarial network, while the following chapter will present the reader with the follow up research published around this topic.

As we already mentioned, a GAN is composed of two different networks – the networks are called the *generator* and the *discriminator*. Given the observed data $\mathcal{X}$, we define a prior probability distribution of the latent space as $P(z)$. This prior can be any fixed distribution and is determined by the author during the network design. The generator, denoted as $G(z, \theta_g)$, represents the mapping of the noise $P(z)$ into the actual data domain $X$. The second neural network, the discriminator, is a standard image processing deep convolutional network, which, given a single datapoint $x \in \mathcal{X}$, generates the probability of $x$ being from the real dataset $\mathcal{X}$ instead of being generated by the generator as an approximated sample from $X$.

The original formulation of the GAN architecture as a two-player minimax game with the value function V [Goodfellow et al., 2014] is presented in Equation 1.2:

$$\min_G \max_D V(G, D) = \mathbb{E}_{x \sim p_{data}(x)}[\log D(x)] + \mathbb{E}_{z \sim p_z(z)}[\log(1 - D(G(z)))] \quad (1.2)$$

This equation illustrates the underlying idea very well. In this original GAN formulation the function D returns 1 if the example $x$ is real and 0 if it is fake. Since the discriminator is trying to maximize the value of $V$ the optimal response is 1 for every real image and 0 for every fake image. The generator's goal is to fool the discriminator and minimize $V$, therefore the ideal situation for the generator is when the discriminator can only guess (and thus has a 50% chance to guess right). Unfortunately, this formulation of $V$ with the discriminator responses being only 0 and 1 has a few problems which make the training unstable in practice and significant research was put into eliminating these problems. However, before we further examine this research, we explain the architecture and the training process of a generative adversarial network in detail.

**Architecture** While the original proposal of GANs defined both the generator and discriminator networks as a multilayer perceptron, there has been a follow up research done by Radford et al. [2015] who use deep convolutional networks for both parts of the network, this architecture was named DC-GAN (deep convolutional GAN) by its authors. The DC-GAN architecture has become a de facto standard since its publication, which is why we present this particular form of the network.

We have already hinted at the architecture of the **discriminator network** – it is a deep convolutional network consisting of a sequence of convolutional layers of decreasing image resolution and increasing number of convolutional channels. Batch normalization (proposed by Ioffe and Szegedy [2015]) is applied to each layer during the training, and each batch normalized layer is activated by a leaky ReLu (rectified linear unit) activation function. While the authors of the DC-GAN architectures argued that fully connected layers should be omitted from the architecture altogether, we still see several state of the art networks (like [Karras et al., 2017] and [Karras et al., 2018]) utilize these layers. The dense layer can be added as the last step of the discriminator, evaluating the features extracted by the convolutional layer and predicting the final probability of the image being fake or real. This prediction can also be handled by a single convolution, as is the case in the DC-GAN architecture. Similar to the usage of dense layers, the authors of DC-GAN argue for elimination of the pooling layers. These layers are also still used in some state of the art networks, however, the DC-GAN network does not utilize them in any way.

The **generator network** is, in a sense, opposite to the discriminator network. Instead of reducing an image into a single value, it takes a latent vector z and fabricates a new image, using a sequence of transposed convolutional layers we have mentioned in Section 1.2. The sequence starts by projecting the high dimensional (usually approximately 100 dimensional) latent vector z into a $4 \times 4 \times 1024$ convolutional representation. This representation is then gradually enlarged in the width and height domain, while being flattened by reducing the number of

channels, until a $W \times H \times C$ image is produced. Each additional layer in the hierarchy reduces the resolution one step further until the final numbers $W, H, C$ are reached. In the DC-GAN paper, $W = 64, H = 64andC = 3$, which gives us a standard RGB image with a $64 \times 64$ pixel resolution.

An illustration of the whole GAN architecture is presented in Figure 1.3.

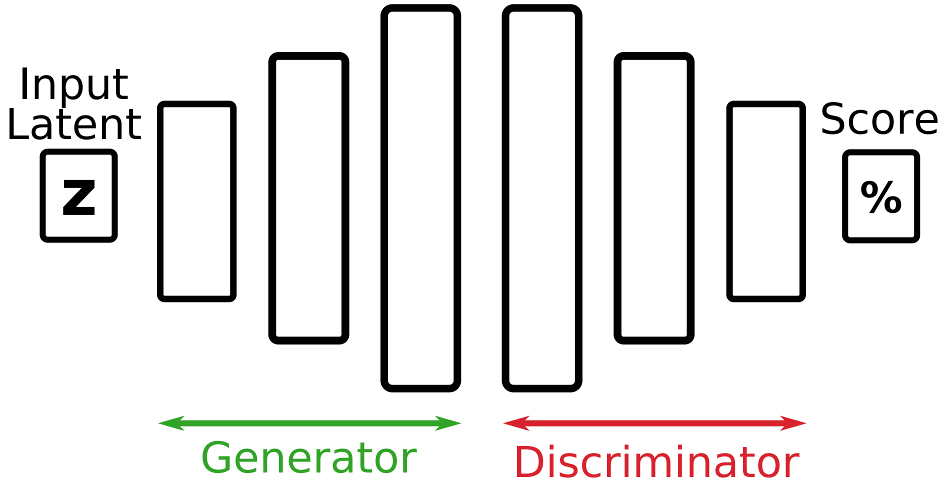



Figure 1.3: An illustration of the Generative adversarial network architecture. The generator takes a latent space vector $z$ and generates a fake image. The discriminator then takes the image and predicts the probability distribution of the image being fake or real.

**Training** Now that we have a general idea on what the network looks like, we explain the rather simple training process. The network is trained end-to-end, on a set of unsupervised data – usually a set of images that we want to be able to imitate. The number of these images has to be rather high, some of the smaller state of the art datasets having no less than 10000 images. Increasing the number of training data seems to greatly improve the network's generalisation capabilities, which is why we usually want to make the dataset as big as possible.

Because increasing the dataset size has this desirable property, *dataset augmentation* is popular. Augmenting a dataset means taking the unique images and slightly altering them in order to make the network less focused on the exact details and thus improve generalisation. A good example of image augmentation is vertically flipping the image. We explain the benefit of augmenting the dataset on a practical example – a network supposed to detect oranges (the fruit) in an image. If our training data has a lot of images with an orange in the right third of the picture, the network might start to only classify oranges that appear in the right third of the picture, because it has noticed the accidental correlation. This is, of course, an undesired effect. Augmenting the dataset in this manner helps the network learn that the position of the orange does not matter and that it should still detect oranges in all corners of the image. Other data augmentation methods include adding artificial noise into the image, cropping out different parts of the image (to help the network learn to detect partially obstructed oranges that are not perfectly round), stretching and resizing the image, and more.

The training of the network itself as presented by Goodfellow et al. [2014] is a rather simple algorithm, which you can see in Algorithm 1. While the details

of the algorithm (like the specific loss function) have been altered in the follow up research, it still gives a largely complete picture of the process.

**Data:** original data $\mathbf{x}$
**parameter:** discriminator training steps $k$
**parameter:** minibatch size $m$
**for** *number of training epochs* **do**
  **for** $k$ *steps* **do**
    sample $z = \{z_1, ..., z_m\}$ from the prior $p_g(z)$;
    sample $x = \{x_1, ..., x_m\}$ from the real data $\mathbf{x}$;
    update $\theta_d$ with $\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} [\log D(x_i) + \log(1 - D(G(z_i)))]$
  **end**
  sample $z = \{z_1, ..., z_m\}$ from the prior $p_g(z)$;
  update $\theta_g$ with $\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} [\log(1 - D(G(z_i)))]$
**end**

**Algorithm 1:** High level idea of GAN training. First the discriminator gets trained for a $k$ amount of steps, then the generator gets trained using the discriminator.

Updating the parameters of the networks can be done using *stochastic gradient descent* (or *SGD* for short) introduced in Robbins and Monro [1951], or more advanced methods like Adam [Kingma and Ba, 2014]. As you can see, the network alternates between training the discriminator and generator. This observation is important because it produces some of the undesired problems we mentioned at the start of the chapter.

The main problem of generative adversarial networks has always been the instability of the training. Since the original article was published, there has been significant research made to stabilize the training of the network and make it easier to train, since the earlier versions required manual tuning of the network's hyperparameters for the network to converge.

# 2. Related work

While the previous chapter contained some basic knowledge and understanding of neural networks, this chapter focuses on providing an overview of their recent developments, mainly in the field of generative adversarial networks. We divide this chapter into individual sections aiming to introduce the related work on several specific topics, namely the stability of GANs, state of the art GAN architectures and networks dealing with high dynamic range images. In the last section we also cover other approaches of generating skydomes with clouds without utilizing neural networks.

## 2.1 Stability of GANs

One of the main problems of generative adversarial networks is the stability of their training. The original versions of the GAN architecture required extensive hyperparameter tuning and experimentation before the network started producing some reasonable images and would not experience *mode collapse*. Because of this some researchers have even developed heuristical methods of finding stable GAN architectures. Lately, there has been excellent research both formulating why the networks are so difficult to train and providing a way to improve the stability of GANs, without requiring the user to manually search through a hyperparameter space to find a stable configuration. In this section, we present some of the more prominent results that relate to the network we used in this thesis.

In Arjovsky and Bottou [2017] the authors explore the sources of instability of training GANs. The paper shows that if we train the discriminator close to optimality before training the generator, the network minimizes a Jensen-Shannon divergence between the real data distribution and the generator's distribution. The authors prove that this is prone to either vanishing gradients or very noisy gradients, which in both cases significantly decreases the generator's ability to learn, since it will not receive any meaningful gradients from the discriminator and will not be able to improve the image quality anymore. This paper also proposes to use the *Earth Mover's (EM) distance* with added noise instead of the current loss function (which is the J-S divergence if the discriminator is trained until optimal) as the solution.

In a follow-up paper Arjovsky et al. [2017], the idea of the EM distance is utilized to define *Wasserstein GANs*, utilizing Kantorovich-Rubinstein duality [Villani, 2009] to formulate a new loss function for GANs. This new formulation, however, requires the network to learn the weights of the discriminator in such a way that it both achieves a supremum of the EM distance, as well as keeps the function *k-Lipschitz continuous*. The authors satisfy the Lipschitz constraint by clamping the range of the weights W of the discriminator to a compact space (for example $[-0.01, 0.01]$) and leave this topic for future work. In its theoretical section, this paper also nicely illustrates the fact that Jensen-Shannon divergence can provide a zero gradient in a relatively simple case.

Gulrajani et al. [2017] provides a better way of enforcing the Lipschitz constraint by introducing a so called *Gradient Penalty* to the discriminator loss func-

tion, at the cost of omitting batch normalization from the network. The authors call their loss function WGAN-GP and showcase the major improvements over the DC-GAN [Radford et al., 2015] network we mentioned in Chapter 1, as well as the previously mentioned Wasserstein GAN (WGAN) loss. WGAN-GP loss is widespread in its use and it is regarded as a major stepping stone in improving the training of GANs.

There have been several improvements since the WGAN-GP, from which we mention Miyato et al. [2018]. This paper proposes a novel way to enforce the Lipschitz constraint by keeping the spectral norm of each layer's weight matrix to a constant (usually 1) and proves that this is enough to satisfy the constraint over the whole network. Since the computation of the spectral norm using the Singular Value Decomposition method is computationally heavy, the paper uses a power iteration to approximate the spectral norm. The results of this paper showcase an improvement of stability over WGAN-GP on the CIFAR-10 [Torralba et al., 2008] and STL-10 [Coates et al., 2011] datasets and the algorithm's relatively simple implementation gives this work the potential to replace WGAN-GP as the state of the art GAN stabilization method. The neural network we use in this thesis utilizes a combination of several different loss functions, but the main loss used is the WGAN-GP function.

## 2.2 GAN architectures

In this section we present a handful of state of the art GAN architectures which had significant impact on the neural network research field. Since our goal is to produce images in high resolution, we only include a select few papers which are capable of generating high resolution.

The work of Karras et al. [2017] on generating fake celebrity portraits had a major impact on both the scientific community and the general public – several newspapers and magazines published an article about this research, like, for example, The New York Times in Metz and Collins [2018]. This paper is one of the first to present a network which is able to generate high resolution images (the original paper's target resolution was $1024 \times 1024$ pixels) with enough fidelity in details to make the images believable. The authors used a progressively growing network, which started on training on a very low resolution (down to $4 \times 4$ pixels) and gradually faded in new layers and produced bigger and bigger images. Since this is the architecture this thesis uses as a basis for our own experiments, we will describe this neural network in detail in Chapter 4.

Following up on the previous neural network, the authors released a second paper Karras et al. [2018]. The architecture of this network is very novel and is influenced by the style transfer task. Instead of starting the generation from a latent vector z, the network starts generating the new image from a learned constant vector. The output of the network is shaped by introducing a latent space $W$, drawing a sample $w \in W$, and performing an affine transformation to convert the latent features $w$ into a style vector $y_i$ which influences the output of each convolutional layer on every one of $i$ influence layers. The results of this paper are also on interactive display on the website *thispersondoesnotexist.com* [Wang], which allows the user to generate images from the network. Toying around with this website is very educational, since one can very quickly get a

general idea of the quality of images produced by state of the art GANs.

Another GAN architecture capable of generating high resolution images is the BigGAN network introduced in Brock et al. [2018]. This network serves a slightly different purpose than the previous ones – it is trained on labeled data and can be conditioned to generate images of only one class (like "dogs"). While the code of Karras et al. [2017] also supports this functionality, it does not seem to be the core idea of the paper. The authors of BigGAN state that the goal of their research is to explore the methods for increasing the output resolution of GANs, combining state of the art approaches from several different papers. They avoid using explicit multiscale methods like Karras et al. [2017] and can successfully produce images of $512 \times 512$ pixels of image resolution. They, however, admit their model still experiences mode collapse which in practice requires to stop the training before this happens.

A very recent work of Park et al. [2019] combines the semantic control of BigGAN (achieved by conditioning the model on the label) and the stylistic control of StyleGAN (achieved by influencing the convolutional layers with a style vector). This network allows the user to input a semantically segmented image and a second image used as the source of style. The network is then able to generate an image which contains the objects specified in the semantically segmented source image while having similar style to the source style image. We leave out a detailed description of this network, since the goals of our work and this paper differ and instead refer the reader to the paper itself.

## 2.3 High dynamic range networks

Finally, we provide a brief overview of work that has been done on high dynamic range processing using neural networks. The work in this field has been rather sparse, compared to the rest of the neural network field. To our best knowledge there has not been any major publication attempting to directly generate completely new HDR images utilizing neural networks. The majority of work was centered around the process of converting between low and high dynamic range images (either by dreaming up a high dynamic range from an LDR image or tone mapping an HDR image).

The work of Patel et al. [2018] utilizes an image-to-image GAN, first introduced by Isola et al. [2016] to generate a tone mapping for an input HDR image. The generator network has the "hourglass" architecture (more akin to a VAE architecture) because it also has to read the original input image, in addition to the latent vector. We did not discuss this type of GANs in detail, since it is not related to our work. It does, however, provide a way to expand this work – if we had a dataset of sky images with cloudy skies, GPS position and time, we could train an image-to-image network that would take an analytically generated skydome model (such as Hosek and Wilkie [2012]) and let the network hallucinate the clouds on top. We will discuss this further in Chapter 6.

Several papers have also been presented on the inverse problem – generating a high dynamic range equivalent of an input low dynamic range image. The work of Eilertsen et al. [2017] utilizes a traditional encoder-decoder architecture, as does the work of Zhang and Lalonde [2017]. The latter focuses on reconstructing high dynamic range outdoor panoramas, which is a problem very close to our

own goal. On the other hand, Marnerides et al. [2018] solves the same problem of expanding an LDR image into an HDR image without the use of GANs or AEs, simply by utilizing three deep convolutional neural networks, each of them focusing on a different level of detail in the resulting image.

## 2.4 Simulation based approaches

The research topic of generating plausible looking skydomes is not a new research area. There are several different use cases for skydomes with clouds – applications which require the full 3D structure of the cloud volumes, such as flight simulators, and applications which view the skydome from a point close to ground level, which can only utilize two dimensional skydome images (such as environment maps). Architectural visualization is a good example of a product which does not require 3D cloud data.

Some approaches to generate 3D cloud data include describing the cloud formations within a formal grammar, particularly the L-systems as described by Kang et al. [2015]. The other popular approach is using different noise functions (like Perlin noise [Perlin, 1985]) like in the work of Schneider and Vos [2015].

The 2D approaches also utilize different noise functions but this time the noise is only two dimensional, usually mapped on a hemisphere which simulates the skydome around the camera [Roden and Parberry, 2005].

The results of most of these methods do not look realistic enough to be used in architectural visualization. The second significant drawback is the fact that even realistic looking cloud textures do not provide realistic luminance values from the sky – this has to be simulated by some light-transport algorithm where the cloud is represented as a participating medium. These simulations usually take a lot of time to converge, which does not make them practical to the average user.

# 3. Dataset

In this chapter, we present our research on dataset acquisition. Since there is no large dataset that would capture high dynamic range skies, we had to explore both the methods to acquire the raw data, as well as post-processing and conversion of the data into the best format for the neural network. This chapter is divided into two sections, the first section discusses the process of shooting the raw data, while the second one discusses the post-processing and conversions.

## 3.1 Dataset acquisition

While there does not exist a large enough dataset, there is a lot of related work and know-how in the field of high dynamic range imagery, even specifically targeted at *CGI*. This work provides us with a rough idea for the methods we should explore.

### 3.1.1 Related work

Since environment maps are used a lot in *CGI*, there are internet shops which sell high quality environment maps as their main product. There are, however, also websites with a more open approach and license. The two most prominent ones are Zaal and NoEmotion. Not only do the authors of these websites provide the environment maps free of charge, but also include a detailed description of how the maps were shot in Zaal [2019] and NoEmotion. We draw a lot of inspiration and know-how from these two sources, even though the goal of the websites (manually shooting a high quality complete environment map) and ours (shooting a large dataset as automatically as possible) differ somewhat. We note that our original goal was to acquire a full 360 degree dataset but we were forced to compromise and only take the upper hemisphere due to the reasons we explain in this chapter.

### 3.1.2 Hardware survey

We have several key pieces of hardware available, which enables us to explore a few different setups for shooting HDR skies, in order to determine which will suit our purpose the best. The two key pieces of gear are a Canon 5D Mark II camera and a Canon 8-15 mm fisheye lens. We also bought a 360° camera called Mi Sphere, which was highly reviewed for shooting still images (as opposed to shooting 360° video). We then tried several shooting scenarios to see which of them would be the easiest to perform on a larger scale – we want our dataset pipeline to be able to handle shooting thousands of images in the long run. This means that manual work should be minimized as much as possible since it is the major limiting factor. Here we present our findings on several different setups for shooting the panoramic images.

**Canon full-frame DSLR** Having a full-frame *Digital single-lens reflex* camera or a *DSLR* has several benefits – the captured images will be of high quality (with low amounts of noise), as well as high pixel resolution. Since the camera is full-frame, there is no crop factor on the focal length, which allows us to shoot at

the 8 mm focal length of the lens, which produces a full 180° field of view (FOV) fisheye image, as seen in Figure 3.1

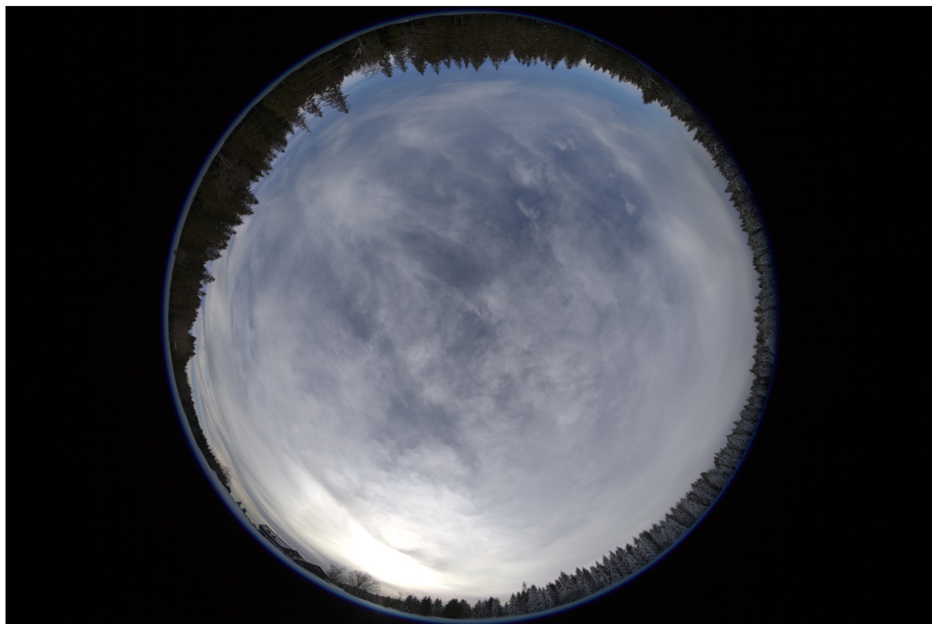


Figure 3.1: An image captured with a 8mm focal length. The field of view is roughly 180°. There is severe distance compression on the edges of the image.

You can see that objects on the edges of the image appear to be very far away – this effect is called *distortion* or *compression* of marginal objects. We will discuss this in detail in the following section. This is, however, a considerable disadvantage, since we do not capture the marginal clouds in as much detail as we might want, while the *zenith* gets captured in a higher resolution.

One of the main advantages of the 8 mm focal length is the fact that we capture the whole sky at once, which eliminates the need to rotate the camera, which in turn limits the manual work. The lack of camera movement also has one more advantage, that is not immediately clear to those unfamiliar with high dynamic range shooting. Taking a high dynamic range image is usually done by capturing the same scene at several different exposure levels, usually by adjusting the shutter speed, while keeping the aperture and ISO setting the same. This means that each picture the camera shoots is actually composed of several (usually around 7) images taken immediately after each other with varying shutter speed – these images are called *exposure brackets*. If the clouds above move fast (for example on a windy day), the cloud formation will be different in each image and the resulting HDR image will be blurred. We showcase this undesired effect in Figure 3.2.

We can clearly see that even one sequence of images, without moving the camera, can be problematic in high winds and result in loss of quality. If we want to shoot a full 360° panorama, this problem gets even more severe. Taking a full panorama is done by rotating the camera around and shooting a sequence for every rotation. To produce a 15000 × 7500 pixel panorama with a 15 mm focal length of the lens, you need at minimum six rotation steps and you also need to add a photo upwards and downwards (to include details of the clouds and the ground). Added up, this is 8 images, with 7 brackets per image – in total

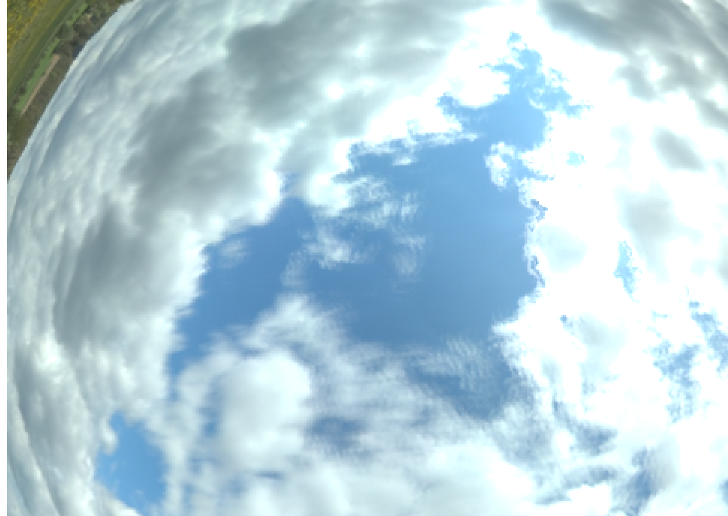


Figure 3.2: An outcrop of a high dynamic range image, which was captured in too high winds – the clouds were moving too fast and thus appear to be blurred. There is also another type of undesired artifact near the bright clouds.

roughly 56 images. These need to be shot in a time interval under approximately 30 seconds, for the result to not be blurred.

This obviously requires intensive manual labor, as well as a significant amount of practice to be able to perform correctly. The other disadvantage of the complete panoramic shoot is the need to stitch the result together into one large image, as well as merge the different exposure brackets together, finally producing a full 360° HDR panorama.

We also note that the 8 images we calculated with are the bare minimum. Professional environment maps used in *CGI* scenes are usually stitched together from a lot more images – we can see that the author of Zaal [2019] shoots 28 different images per panorama.

In conclusion there are two main approaches to shooting the sky with a DSLR camera:

- **upper hemisphere only** – shooting one sequence of roughly 7 exposures directly up, with a 8 mm focal length lens.
- **full panorama** – shooting a full 360° scene which is stitched from several different images, each image again taken at several exposure levels.

Both of these have their advantages and disadvantages, which we will summarize after we explore the 360° Mi Sphere camera we mentioned earlier.

**Mi Sphere** 360° **camera** One of the disadvantages of shooting a full 360° panorama is the extensive amount of manual work required. A 360° camera is a logical solution to this problem, because it can capture the whole surroundings at the same time, again reducing us only to the different exposure levels. Our Mi Sphere camera can be seen in Figure 3.3.

A 360° camera is usually composed of two fisheye lenses, one on each side of the camera. Because of this, the resulting image is what is usually called a dual fisheye image. An example of a dual fisheye image can be seen in Figure 3.4.

Figure 3.3: Our Mi Sphere camera. The same lens is duplicated on the other side of the camera. The camera is only about $10 \times 10 \times 2$ centimeters in size, making it very portable.

Because of this, stitching the picture into a full panorama is again required, although the amount of manual work is significantly lower than the amount required to stitch a panorama shot by a DSLR camera. This stitching can also be automated, since the relative position of the two images will always be the same.

The camera is very compact, which has its advantages and disadvantages. The advantage is the low weight and the portability which it implies. The weight is also important for a different reason – because the camera isn't heavy it can be mounted on lighter tripods or even flash stands. This reduces the amount of ground which is occluded by the tripod significantly, while also limiting the weight of the photographer's backpack.

However, the disadvantages are also very important. The biggest downside is the smaller sensor[1], which only has a 7.7 mm diagonal (compared to the full-frame DSLR which has around 43 mm). This means the image is noisier, less sharp and overall worse in quality. The second major disadvantage is the battery life – the battery cannot be exchanged and has to be charged through a USB

[1] According to the manufacturer, the sensor is *Sony IMX206 CMOS*.

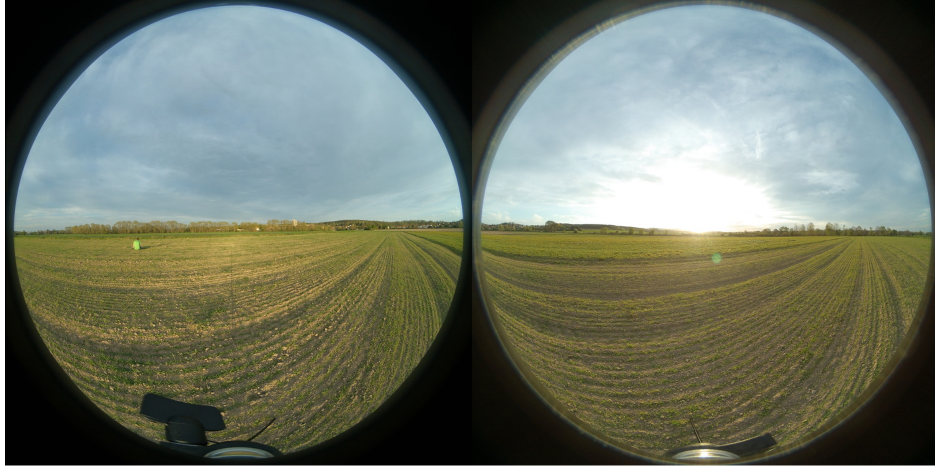


Figure 3.4: A *dual fisheye* image taken by the Mi Sphere camera.

port.

The camera comes with an application for Android devices which allows the user to set the ISO and shutter speed, which is critical for our task. The application, however, does not allow the user to set a bracketing of more than three consecutive pictures, which is not enough for our purposes. This means that the user has to set the exposure brackets manually, which increases the level of manual work.

**Conclusion** Now that we have reviewed all of our options, we list all significant requirements for the data acquisition method and compare the three mentioned methods based on these requirements.

1. **Reasonable quality** – while the quality of the images does not have to be excellent just for the purpose of training a network, if we shot a quality set of 360° environment maps usable by *CGI* artists as well as the network, the contribution would be significant.

2. **High degree of automation** – we aim to shoot tens of thousands of images. For this to be a realistic goal, the pipeline has to be as automated as possible, with the least amount of human interaction. This is important for both saving time as well as making the method less error-prone.

3. **Robust** – since the main goal of this thesis is to shoot clouds, the method has to be robust enough to not be affected by fast moving clouds noticeably. If the method produces significant artifacts with fast moving clouds, the dataset will not be able to generate such clouds.

The choice of the method came largely down to the second point. Shooting a production-ready full 360° environment map with a DSLR is simply too time consuming to be viable on a larger scale. Since the 360° camera application does not allow the user to neither set a custom bracket sequence nor repeat the shoot every $N$ minutes, it also cannot be used in its current form to shoot large datasets.

On the other hand, the upper hemisphere fisheye method fulfills all three of our criteria. While the quality of the marginal objects is not the greatest, it still

is enough for neural network training. The camera can be highly automated and the shooting times are fast enough to allow capturing fast moving clouds. This is why we chose this method as our primary method to acquire the data for our dataset.

We also note that developing a custom application for the Mi Sphere 360° camera would be largely beneficial. If we had an application which allows us to specify a custom bracketing sequence, as well as repeat the shoot every $N$ minutes, the 360° camera could also be used to capture a part of the dataset. It could, for example, be used on more remote places, where the weight of the DSLR would be a limiting factor to the photographer.

### 3.1.3 Dataset shooting

Now that we have presented our method and explained the reasons why we chose this particular way to shoot the data, we go over some practical considerations we have to keep in mind while shooting the dataset. We provide a complete list of all equipment used as well as a detailed step by step manual in Attachment A.3.

**Clipping the dynamic range** Shooting on a day when the Sun is clearly visible and unobstructed in the skies represents a problem even for high-end camera equipment. The Sun is brighter than the rest of the sky by orders of magnitude and capturing this high intensity is hard. It is usually done by placing a *Neutral Density filter (ND filter)*[2] in front of the camera and shooting with the lowest shutter speed and smallest aperture. The addition of the neutral density filter allows the user to capture additional exposure brackets, which increases the intensity values of the Sun in the final image.

However, this process is more manually demanding and as a result, a lot of HDR environment maps have a so called *clipped* Sun. This means the values of the Sun are arbitrarily clipped and are not realistic. For example, if the Sun is clipped in the image, its pixels will have the intensity around $60 - 100$. The intensity of an unclipped Sun on the other hand will be around 300000. This discrepancy is significant and affects the lighting conditions in the scene.

The usual lighting set up in a scene consists of a *Sun element* to provide the immense radiance value to light an outdoor scene, as well as an environment map to provide the much less intense details of the lighting. If the environment map is unclipped, however, the artist cannot use the Sun element, since the environment map already provides both components of the light. This limits the artist's creative freedom to adjust the Sun's position slightly to potentially remove some undesirable shadows. During rendering, sampling the unclipped environment map is also significantly harder than just sampling the Sun element, because the renderer knows where the Sun element is, as opposed to the Sun in the environment map, which the renderer has to discover through adaptive sampling.

[2]By a neutral density filter we mean a photographic accessory which reduces the intensity of the incoming light by several exposure steps. It is called neutral density because it reduces the intensity across all wavelengths. This filter is often used to reduce overexposure or increase the exposure time. In our case, the ND filter would allow us to capture more exposures of the Sun – giving us greater dynamic range.

Because of the rendering pipeline we mentioned, as well as the fact that installing an ND filter on a fisheye lens is almost impossible, we chose to only shoot with a clipped dynamic range, mainly to stay true to our goals from the previous chapter and keep the manual work to a minimum. We do, however, utilize the camera's dynamic range to the fullest and include a lowest shutter speed and smallest aperture image in our exposure settings – this way we clip the image the least amount we can given our camera hardware.

**Custom bracketing and intervalometer** *Custom bracketing* and an *intervalometer* are two of the most important features we need to be able to make the dataset acquisition as automated as possible. By custom bracketing we mean the ability to set the camera to take a sequence of images instead of one single image. This sequence will have varying shutter speed and thus varying exposures, which enables us to create an HDR photo.

The *intervalometer* is a simple piece of software which will automatically trigger the camera to shoot the sequence of bracketed exposure photos every $N$ minutes or seconds. This would allow us to just place the camera and let it shoot in precise intervals without any human interaction.

Despite the relative simplicity of these two features, many cameras on the current market seem to lack one or both of these features. While the majority of cameras have a compatible intervalometer in the form of a separate device connected via a cable, the ability to set a custom bracketing sequence cannot be usually added to the camera afterwards.

As we have already stated, a high dynamic range photo is usually produced by stacking several images taken at different exposure levels. The current standard for lower and mid-range cameras is shooting 3 exposure brackets with adjustable exposure (usually with an exposure step limited to up to 2 EV or 3 EV). This allows the user to, for example, shoot one photo at $-3$ EV, one photo at 0 EV and one photo at $+3$ EV. While this is enough for artists to capture beautiful HDR photos, it is not enough for our academic purposes. For the purposes of gathering an academic dataset with higher precision, we need approximately seven images per sequence. These seven images usually have a 2 EV step between each of them. If the darkest image is taken at 1/8000s, 22F and ISO 100, the second darkest image will be taken at 1/2000s, 22F and ISO 100, and so on.

Our solution to this issue is a free software add-on called Hudson [2009]. This firmware extension for a select few Canon cameras allows the user to customize the shooting sequence freely. This software is installed on the memory card and then allows us to set the number of images taken to 7, with a 2 EV exposure step in between. Such bracketing sequence covers a dynamic range of 12 EV, which is enough to capture the majority of outdoor scenes (including unobstructed Sun, as we have discussed in the few paragraphs about clipping). We note, however, that this custom firmware is not able to help us with clipping the Sun. The level of clipping is determined by the lowest exposure our camera can shoot (which is 1/8000s, 22F and ISO 100) and not by the following exposure brackets.

Magic Lantern also provides a quality intervalometer, which eliminates the need of a remote shutter control. The complete Magic Lantern settings we use are displayed in Figure 3.5. This combination makes the shoot fully automatic, requiring human interaction only to initiate and terminate the shoot.

Figure 3.5: Exposure bracketing and intervalometer settings of Magic Lantern software. The *HDR bracketing* setting signifies shooting 7 images with a 2 EV step in between each pair, starting from the darkest one and increasing the exposure. The *intervalometer* is set to shoot a sequence of pictures every two minutes.

## 3.2 Dataset processing

In this section we mention all data processing we have had to do while creating this dataset. This includes both processing the images we shot into a final dataset-ready image, as well as converting the dataset into the training data for a neural network by utilizing data augmentation.

### 3.2.1 Photo processing

Firstly, we go over post processing our captured data. There are several steps necessary to transform the sequence of RAW images into a high quality HDR image. We explain why each step is required and our chosen approach in the following text.

**Post-processing** Before we can merge the image sequence into one HDR image, we first need to process the RAW data to get rid of camera artifacts such as vignetting, try to remove the chromatic aberration, reduce the color noise slightly, as well as choose if we want to perform a white balance correction.

We adopted the profile for our specific camera and adjusted the chromatic aberration setting. We found the results of the preset satisfactory for our needs. We refer the reader to the source for any specific details about the preset's behavior. We also include our modified preset in the digital attachments, as seen in Attachment A.1.

White balance correction is an issue similar to the clipped dynamic range, which we discussed in the previous section. While shooting a sunset, the light from the Sun gets an orange tint (also called a *warm tone*, or *low temperature* derived from blackbody radiation). It is up to the user to choose if he wants to

correct the tone of the image or not. This is the only major decision the user needs to make when adopting the RawTherapee preset we mentioned above.

This effect is easily corrected by a post-processing program or even in camera. However, correcting this effect makes the resulting light to be a neutral white instead of the warm orange sunset. If the artist picks a sunset environment map, which was color corrected to emit a neutral white light, he will have a scene with a setting Sun but the objects in the scene will not be lit by the warmer orange light coming from a sunset.

For the purposes of getting the pipeline as simple as possible, we chose not to correct the white balance. This effect, contrary to the dynamic range clipping, is not hard to correct on the whole dataset at a later date, should the need arise.

By utilizing RawTherapee's batch processing, we can easily apply this modified preset to all images we took on a shooting session, and process them into a 16-bit compressed `.TIF` format. These photos are now ready to be transformed into the correct projection and merged into the resulting HDR image.

**Projection correction and HDR merging** After the images have been processed to remove the minor camera artifacts, we are left with several sequences of `.TIF` files. We now want to convert the projection, which is, at the moment, defined by our lens' optical system, into a standard projection. We need to perform this conversion because we want to have precise control over what the data looks like. We also need to be able to convert images from other existing environment map databases (like the ones from Zaal and NoEmotion) using the same projection. This way we can seamlessly merge images we shot with images from other sources, without creating new artifacts in the dataset, such as changing perspective (which would only confuse the networks). We will discuss the choice of one particular projection in the following Section 3.2.2.

For this projection correction, we chose a program called PtGui [B.V.]. It is a commercial graphics user interface over a panorama library called PanoTools [Dersch]. It specializes in stitching panoramas and is also capable of creating an HDR panorama. More importantly, it can be automated to a high degree, which makes it perfect for shooting a high-volume dataset.

We created a PtGui template for our purpose, which merges one sequence of images into an HDR image and creates a stereographic fisheye projection with precisely a 180° field of view. PtGui is able to detect the distinct sequences in a folder full of images, create a project for each of these sequences and apply our template to all generated projects. The projects then get queued and rendered one by one.

This pipeline requires minimal work from the user, which is critical for our purpose. It also handles both correcting the projection and merging the sequence into an HDR photo, which eliminates the need to use another software to perform the merge. We provide the complete step by step manual to our photo processing pipeline in the second part of Attachment A.3, which explains each step in detail. We also attach our PtGui template in the digital attachments of this thesis in Attachment A.1.

### 3.2.2 Neural network data format

In this second part of dataset processing we no longer talk about exclusively our newly shot images. By dataset we now mean every image of the upper hemisphere we could find and use for this thesis. This includes the data we shot with our camera, as well as other sources for similar data, which we already mentioned in Section 3.1.1. We list every source of our data in Attachment A.2.

**Choosing a projection for our data** The most important choice when processing the data lies in choosing the projection to use for the data. A perfect projection of a hemisphere onto a two dimensional plane (an image) does not exist and there are many different projections to choose from. This problem is well researched, since cartography has been trying to project the surface of the Earth onto paper as efficiently as possible for centuries. We list the most prominent projections and explain our choice.

- *Equirectangular projection*, also called a latitude-longitude (lat-long) projection is commonly used to distribute environment maps. An example of an image mapped in equirectangular projection can be seen in Figure 3.6.

  This projection projects the whole sphere (not just the upper hemisphere) onto a rectangular image with the image width being twice the image's height. The left-to-right axis corresponds to longitude, whereas the top-to-bottom axis corresponds to latitude. Both the forward and the backward mapping functions are simple to evaluate, and only require to evaluate one cosine.

  The major disadvantage is the considerable deformation which increases with growing distance from the equator. Since we are mainly interested in the upper hemisphere, this deformation is at its lowest on the horizon line, while it is significant in the upper sections of the hemisphere, where we want to achieve high fidelity of the generated clouds for both high quality scene lighting and reflections (e.g. reflections on bodies of water). On the other hand, the deformation is low for the horizon line which also suits our purposes, since the camera in a *CGI* scene will usually see part of the horizon.

  The equirectangular mapping also contains some embedded continuity constraints, some of which are non-obvious. The first constraint requires that for a specific projection, the edges of the projection correspond to each other and are seamless (that is, if you rotate the environment map horizontally by 180 degrees, you will not see a seam). The second, less obvious and more complicated constraint holds for the upper edge of the mapping – the upper edge defines the zenith of the mapping. This means the mapping has to correctly collapse from an edge into a point without any discontinuities. While these constraints obviously hold true for a correctly remapped hemisphere, a standard deep convolutional neural network has limited capabilities to recognize these constraints and might break them while generating a new image.

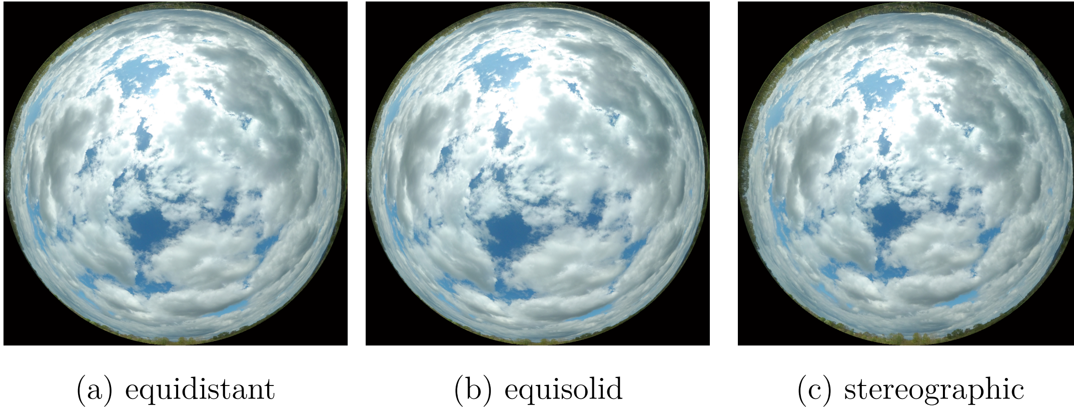
  We showcase the results of a network trained on equirectangular representation of the data in Chapter 5, where we observe this behavior.

Figure 3.6: An example of equirectangular projection. Notice the extreme distortion of the clouds in the top region of the image.

- *Fisheye projection*, which we already mentioned in relation to our camera equipment, only projects one hemisphere onto the plane. Fisheye projection is most commonly encountered by looking into a peephole in the front door – the lens commonly used for this purpose is a fisheye lens, in order to provide a wide field of view to the user. The image's center represents the camera's target (the zenith of the hemisphere), and by moving to one of the edges of the image we widen our field of view (looking closer to the equator). There exist several fisheye mapping functions, each mapping function having a set of slightly different properties as described by Bettonvil [2005], which we will now briefly reiterate, with a focus on the properties that are of importance for our work. We showcase the three most important to our work in Figure 3.7.

  - *Equidistant* fisheye projection maintains angular distances and is usually used for measuring angles (zenith and azimuth angles) because of this property. The compression of marginal objects is significant, which makes it impractical for our purposes.
  - *Equisolid* angle projection maintains a linear relationship between the projected surface on the image plane and the area of the hemisphere which got projected. This property is relevant to our cause, since it would preserve the relation between the cloud sizes. This projection unfortunately compresses the marginal areas the most.
  - *Stereographic* fisheye projection maintains the angles between curves in the image. This makes smaller objects (like clouds in the sky) look natural, because the angles do not get distorted (a circular cloud stays circular), though the area does. This projection compresses the marginal objects the least, and is therefore very suitable for our cause because of both of these properties.
  - *Orthographic* projection is a special projection which highly distorts the image as the viewing angle approaches 180 degrees. This projection

is not at all usable for our purpose.



(a) equidistant (b) equisolid (c) stereographic

Figure 3.7: A comparison of all three fisheye projections which are usable for our purpose. We can see that equisolid projection compresses the marginal objects the most, while stereographic projection compresses them the least. Notice the size of the trees when comparing the compression – the trees are clearly visible in image (c) but almost invisible in image (b).

As we can see, the *equirectangular* projection is not really the best suited for our cause – it highly distorts the parts of the hemisphere we focus on the most (the clouds high in the sky) and has some discontinuity problems, which are hard to solve in a neural network environment. We also note that the problems we mentioned for equirectangular projections hold true for all *cylindrical* projections, so choosing another cartographic projection will not help to solve these core issues.

On the other hand, a *stereographic* fisheye projection seems to eliminate the discontinuity problems, while providing very low distortion at the zenith part of the hemisphere and only compressing the marginal objects on the horizon line, which are of secondary importance to us. The considerable downside to all fisheye projections is the fact that the image is only valid inside the inscribed circle, leaving the corners of the image black. This property loses us some pixels of resolution, since if the network outputs a $1024 \times 1024$ image, we will only use $\pi * (512)^2$ pixels out of the $1024 * 1024$ available, which amounts to 78.5% of the pixels. As we will see in Chapter 4, we can add a layer to a deep convolutional network, which helps the network to disregard the outside of the unit circle, focusing the network on the actual image.

We also note here that we could avoid projecting the hemisphere altogether, if we used a spatial neural network, which we would train on the exact hemispherical data. We will discuss the idea in more detail in Chapter 6. We did not pursue this idea, since convolutional networks working over a custom spatial domain (like a hemisphere) are still not a well-researched topic.

**Preparing the data for the network** Having decided to use the stereographic projection, the next step is converting all images of our dataset into stereographic projection, with the same field of view and image size, as well as augmenting the data and exporting the correct format for our neural network to process. We will now describe each step of this pipeline.

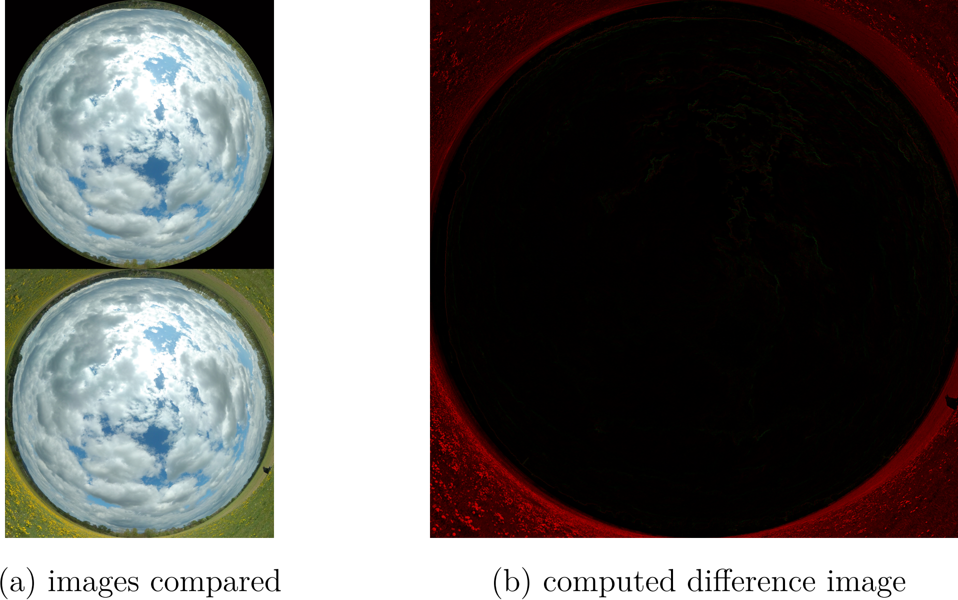
Since we have decided to use a fisheye projection, we need a program to convert all the equirectangular data into the stereographic projection. We implemented a C++ program which converts an equirectangular image into a fisheye view of the upper hemisphere. The program utilizes the OpenCV library [Bradski, 2000] to handle the image manipulation, as well as the OpenMP library [Dagum and Menon, 1998] to perform the conversion in parallel, which improves the performance when the target resolution is high. The program samples the equirectangular map using several samples per pixel, which are pseudo-randomly distributed throughout each pixel. It utilizes bilinear interpolation to extract the information provided by each sample.

We compared the results of our program with the results of the PtGui software we mentioned previously. We took an equirectangular panorama $P$ and transformed it into a 180 degree field of view, stereographic projection fisheye image $P_o$. We also used the PtGui software to transform $P$ into a fisheye image $P_p$ with the same parameters (stereographic projection, 180 degree FoV). The two images $P_p$ and $P_o$ can be seen in figure Figure 3.8 on the left. We computed a difference image between $P_o$ and $P_p$, which can be seen in Figure 3.8 on the right. We note that our program converts data only inside the physically viable unit circle of the fisheye lens and not outside, unlike the PtGui software. This is what causes the image to be missing the ground element in the corners, which is highlighted in red in the corners of Figure 3.8. The ground is beyond the 180 degree field of view and would not be visible through a real optical system with 180 degree field of view. We also see some minor discrepancies within the unit circle itself (usually present on clear edges like the horizon line or cloud edges), which we attribute to different sampling methods used by both programs. This conversion is precise enough for our purposes and will not limit the performance of the network in any way

The next step is performing some form of **data augmentation**, which should hopefully both enlarge the dataset and provide some variance to avoid overfitting by the network. While there are multiple ways of data augmentation for images, we unfortunately cannot use a majority of the most frequently used ones. For example, we cannot utilize zooming or cropping for our augmentation, since we want to only train the network on complete fisheye images. However, the fact that our images represent a full 360 azimuthal degree hemisphere allows us to rotate the view around the upward-facing axis, which provides us with an excellent method of data augmentation.

While rotating the images we avoid as much resampling as possible. The conversion program to generate fisheye images out of equirectangular maps has a setting to also rotate the camera by $N$ degrees. This way the rotated image is generated directly, avoiding the resampling which would occur if we first generated a non-rotated image and then rotated the image using an affine transform. For images which were already captured as a fisheye image and not generated from an equirectangular panorama, we use the affine transform method. We also do not want to rotate strictly by a given angle, since the network would learn the fact that the Sun is always in these discrete rotation steps. To counteract this, we introduce a small amount of random jittering of the rotation angle.

We also utilize the traditional horizontal flip of the image as an additional source of image augmentation, which is done directly inside the network during



(a) images compared

(b) computed difference image

Figure 3.8: Image (a) showcases both generated images, the top one being the result of our conversion program $P_o$ and the bottom one the result from PtGui software $P_p$. Image (b) is a difference image between the output of the PtGui software and our conversion program. Displayed is the $i - r$ metric for $i$ the compared image (image generated by us) and $r$ the reference image (the PtGui software output). Red color means the error metric was negative, green color indicates a positive difference. We note that the red error in the corners is due to the fact that our program does not convert an image outside of the viable fisheye unit circle.

the runtime.

After the data augmentation is complete, we also have to downsize the images into several resolutions. This is due to the network's architecture, which we will discuss at length in Chapter 4. While training, the network gradually increases its generated resolution, so we have to provide training data for each of these steps. These resolutions are powers of two, starting at 4 by 4 pixels, increasing to the target resolution (e.g. for a target resolution of 256 by 256, we would have 4, 8, 16, 32, 64, 128 and 256 pixels for both width and height of the image).

Similarly to the previous paragraph, we do not want to resample the image multiple times to avoid loss of quality. The C++ conversion program is able to produce an image of said target resolution, which again avoids one resampling step.

Once the images are in the correct projection, augmented by rotating and downsized to all target resolutions, the final step is to convert them all into a Tensorflow [Abadi et al., 2016] compatible format, since this is the framework our neural network uses.

Tensorflow has a special format called *TFRecord*, which the framework supports. Detailed guide and description can be found in the Tensorflow documentation[3]. It is a simple binary format without any compression, which the framework can iterate through and generate training examples from. We wrote

[3]https://www.tensorflow.org/guide/datasets

a simple script which takes an input folder of images and generates a `.tfrecord` file out of them. This way we are able to quickly generate the TFRecord files for all resolutions automatically by iterating through the directory structure

To verify the whole pipeline, we also created a script which reads a TFRecord file, decodes one sample image at random and also calculates some basic information about the images inside the file – the minimum value found in all pixels of the file, the maximum value and the file count. The minimum and maximum values of all pixels are a useful debugging tool for the rest of the pipeline, since the pixels should always be non-negative. Producing a negative value will raise an error later on in the training phase, so it is important to detect this early.

We have already mentioned the technical details of the equirectangular to fisheye conversion program, which is written in C++, to maximize performance. The rest of this pipeline is written in Python, since high performance is not as critical as the much quicker development cycle of Python. The Python programming language also makes the pipeline easy to use on both personal computers as well as servers. It also makes the pipeline easily understandable and extendable. We compiled the image processing functions into one single module called Image Utils, which is then imported into the scripts which handle both input projections (fisheye captured by us, as well as equirectangular from other sources).

To make this pipeline easy to use, we also provide a Docker [Merkel, 2014] image, which installs all system prerequisites to operate our pipeline, such as Python, OpenCV, Tensorflow, etc.

We also note that our pipeline is able to handle both low-dynamic range images (such as the `.jpg` and `.png` formats) as well as high-dynamic range images in both OpenEXR (`.exr` format) and RGBE (also known as Radiance HDR, `.hdr` extension) formats. This is handy because the majority of state of the art neural networks are not able to receive high-dynamic range images without modification of the network, and training on LDR images also provides viable feedback regarding the network structure.

We have now described and explained the whole dataset acquisition and processing pipeline, which starts by photographing the images and ends with a complete dataset transformed into a network-compatible format. For further technical details, we refer the reader to the source code of the thesis. The structure of the source code is described in Attachment A.1. A detailed read-me file is also included, which should help the reader to navigate the directory structure.

# 4. Neural network

In the previous chapter, we covered the whole process of creating a dataset for the purpose of training a neural network. In this chapter, we describe the neural network we chose as the basis of our solution, then describe the modifications we introduced and explain the reasoning behind them. We also offer a short technical review of the hardware setup we used to train this neural network, for the purposes of enabling the reader to re-create and use our work.

## 4.1 Network architecture

As a starting point of this thesis, we chose the neural network of Karras et al. [2017], with the goal of modifying the network to produce HDR environment maps. In this section we give the reasons for choosing this particular network architecture, as well as describe the network in detail, to hopefully make it easier to introduce our modifications later on.

### 4.1.1 Reasons for choosing the neural network

The first choice we had to make was if we should develop our own architecture, or use one which already exists. While developing a new architecture gives us more creative freedom to combine effective approaches from related work, it does come with a high development time, which has the potential to overshadow the actual reason we chose neural networks – as a means to generate HDR images with enough accuracy for the *CGI* purposes.

Using an already existing network, on the other hand, eliminates this development time, but comes with its own problems. Understanding a completely new (and usually complicated) neural network architecture is not fast either and since we did not develop the architecture, we do not have a solid idea about its behavior at the outset. It does, however, eliminate the need to search for the best hyperparameters (like the batch size, learning rate or number of convolutional channels for each layer, etc.) and convergence speed optimization (in optimizing the architecture to achieve the best performance by merging some layers or rearranging them) When using an already existing layer, we avoid this work since it has already been performed by the authors who usually highlight the settings they found to best perform in their report.

We chose to take the neural network of Karras et al. [2017] and modify it, rather than create our own neural network. We chose not to create our own network, because this architecture has performed very well in low dynamic range image generation and it therefore would be interesting to see if we could generate an HDR image using a successful LDR GAN network. This particular network had a few advantages over the other state of the art GAN architectures. The implementation was open-sourced by the authors under the Creative Commons CC BY-NC 4.0 license[1], which allowed us to use the original source code and train the network with our own datasets and directly compare both the quality

[1] https://creativecommons.org/licenses/by-nc/4.0/legalcode, accessed on 24. 06. 2019

and the training speed with the results of the authors. The second advantage of the network was the relatively small size of the training dataset mentioned in the paper of Karras et al. [2017] – the training dataset had ”only” 30000 images, unlike the more complicated follow-up work, where the dataset size increased to 70000 [Karras et al., 2018]. Since the dataset size was a major limiting factor in our research, we wanted to select a network, which had a higher chance of being able to train on smaller datasets.

### 4.1.2 Network architecture

As we will see, the underlying architecture of the network as proposed by Karras et al. [2017] is simple and straightforward. We note that unlike the architecture, the behavior of the network is not simple and straightforward and we had to perform a few experiments to understand how the training usually progresses and what is indicative of a successful training and what behavior indicates a failed training run. The network uses the following layers for both the generator and the discriminator:

- *a convolutional layer* is used in both the generator and discriminator to both compute features of the image (in the discriminator) and alter them (in the generator).
- *a transposed convolutional layer* is used in the generator to upscale the image and compute a convolution at the same time. This is the main source of enlarging the photo in the generator.
- *a nearest-neighbour upscaling layer* is used to increase the resolution by using a simple nearest-neighbour filtering. This operation is used while growing the network from one resolution to another. We will discuss this growing in greater detail in the following paragraph.
- *a average-pooling downscaling layer* is a simple layer to decrease the resolution using a moving average pooling filter. In a similar spirit to the upscaling layer, this layer is mainly utilized to grow the network's resolution.
- *a dense layer* is utilized at the end of the discriminator to compute the score of the input image.

**The generator** of the network is composed of blocks of layers which are repeated for each resolution step. One block is shown Table 4.1:

The first layer, `conv_up`, is a *transposed convolutional layer* which increases the resolution of the input twice (e.g. from $8 \times 8$ to $16 \times 16$ as illustrated in Table 4.1). The second layer, `conv`, is a convolutional layer, designed to introduce more detailed features in addition to the features already introduced during the upscaling by the previous layer. Both of these layers had a convolutional kernel of shape $(3, 3)$. A second convolutional layer, called `ToRGB`, is introduced to convert the 512 channels into 3 channels of RGB. This layer has a kernel of shape only $(1, 1)$, since format conversion is a local operation. Notice that this layer reduces the number of channels to three, down from 512 in this case. These three channels are precisely red, green and blue channels of the generated image. The following

| **Layer name** | **Output shape** (batch, channels, width, height) | **Activation function** (after the layer) |
|---|---|---|
| **block input** | $(?, 3, 8, 8)$ | — |
| `16x16Conv_up` | $(?, 512, 16, 16)$ | LReLU |
| `16x16Conv` | $(?, 512, 16, 16)$ | LReLU |
| `ToRGB` | $(?, 3, 16, 16)$ | — |
| `Upscale2D` | $(?, 3, 16, 16)$ | — |
| `Grow` | $(?, 3, 16, 16)$ | — |
| **block output** | $(?, 512, 16, 16)$ | — |

Table 4.1: Architecture of a single building block of the generator network.

two layers, which complete a single block, are there for the purposes of growing the network, and get ignored once the training is finished. Since one block increases the resolution twice, the number of blocks is thus directly proportional to the target resolution of the network.

We have already pointed out that the `ToRGB` layer reduces the number of channels down to 3. We want to clarify that once the network is trained, only the green rows of Table 4.1 are utilized. This means that the network does not compress the information in the convolutional channels down to three channels every block. The `ToRGB` layer is only applied directly before we want the network to produce an image. It is a part of the block because we want each block to be able to produce an image, as we will explain in the following paragraph about how the network grows from a smaller resolution to a bigger one.

Growing the network is a procedure integral to this network's architecture. In Table 4.1 we see the last two layers called `Upscale2D` and `Grow`. The `Upscale2D` is the simple nearest-neighbour upscaling layer we mentioned previously. This layer takes the image generated by the previous block $I_p$ and enlarges it twice (this is why we utilize the `ToRGB` layer in each block). The `Grow` layer then interpolates this upscaled lower resolution image $I_p$ and the new image $I_n$ produced by the `ToRGB` layer. This interpolation produces a new image $I = I_n + (I_p - I_n) * t$. When $t = 0$, only the new image gets output from this block into the next block. When $t = 1$, only the enlarged image from the previous block gets output from this block. Changing $t$ from $t = 1$ to $t = 0$ is called growing the network, because it enables the network to use one additional block, which produces higher resolution images. Note that Figure 4.1 illustrates this process of growing for $a$ going from 0 to 1, which is different from the actual network implementation which we explained above. This approach is very similar to a neural network technique called skip connections, except for the fact that in the architecture we use the output image is always full target resolution – the untrained layers do not get skipped but only enlarge the image without increasing the detail. This is because even though the output of the later blocks does not get used, the blocks still upscale the low resolution using the nearest-neighbor layer and thus the blocks are not skipped entirely. Figure 4.1, reproduced from the work of Karras et al. [2017], illustrates the growing process of adding a single block of resolution $32 \times 32$. The authors call this growing fading-in, since it happens in a linear fashion.

This fading-in of a new block is done only after the previous block has been

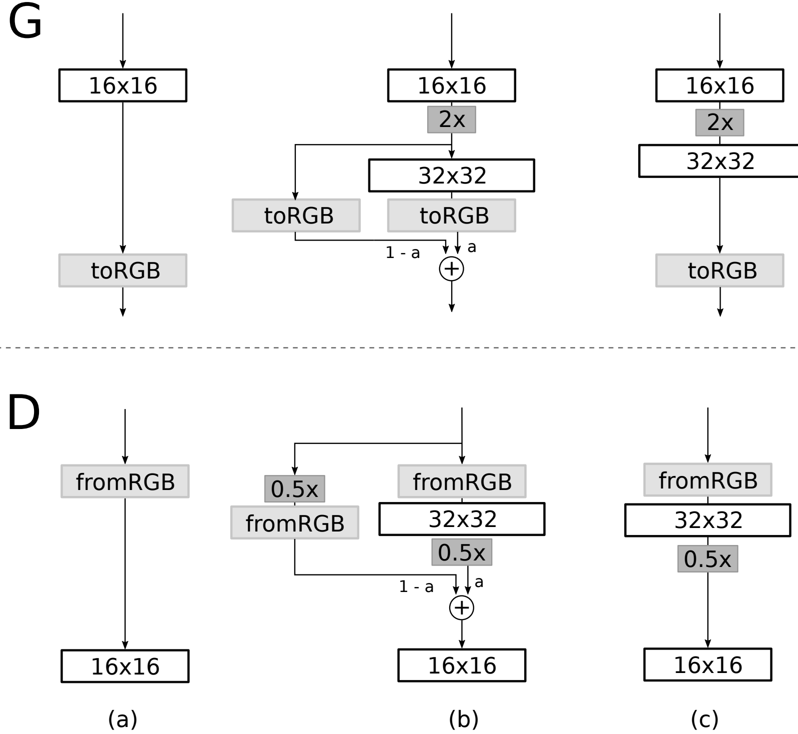



Figure 4.1: A diagram of the growing process, reproduced from [Karras et al., 2017]. Image (a) illustrates the network operating at a $16 \times 16$ resolution. Image (b) introduces the new $32 \times 32$ block and gradually fades it in by increasing $a$ from 0 to 1. Image (c) showcases the final $32 \times 32$ network. The layers *fromRGB* and *toRGB* refer to a $1 \times 1$ convolutional layer which extracts image features from RGB and projects features into RGB respectively. The $2\times$ and $0.5\times$ layers use nearest-neighbour upsampling and average pooling downsampling respectively to alter the image resolution.

trained for a while, which hopefully stabilizes the network, allowing it to converge before introducing a more complex architecture. The metric for correct alternation between the two phases (training the current blocks and introducing a new block) the authors use is the number of real images the discriminator has seen. This number which indicates the switch is $800k$ in the full version of the network, and $600k$ in a smaller, lower-capacity version of the same network (obtained by reducing the number of convolution channels per layer and the resolution).

**The discriminator** is very similar to the generator, and we will just briefly showcase the building blocks in Table 4.2 and point out some differences to keep this text concise.

In a similar fashion to the generator block, the discriminator block is composed of two convolutional layers, the second of which reduces the resolution of the processed image. The three layers highlighted in red in Table 4.2 are again used for growing the network. The `Downscale2D` layer reduces the resolution of the raw input image (hence only three channels on the output), passes it to the `FromRGB` layer, which decodes new features (which have the same shape as the features which came as the input to the block) and the `Grow` layer which linearly interpolates between these two feature sets of the downsampled image and the featureset calculated by the consecutive blocks.

| **Layer name** | **Output shape** (batch, channels, width, height) | **Activation function** (after the layer) |
|---|---|---|
| **block input** | $(?, 256, 64, 64)$ | — |
| `64x64/Conv0` | $(?, 256, 64, 64)$ | LReLU |
| `64x64/Conv1_down` | $(?, 512, 32, 32)$ | LReLU |
| `Downscale2D` | $(?, 3, 32, 32)$ | — |
| `FromRGB` | $(?, 512, 32, 32)$ | — |
| `Grow` | $(?, 512, 32, 32)$ | — |
| **block output** | $(?, 512, 32, 32)$ | — |

Table 4.2: Architecture of a single building block of the discriminator network.

One important note to understand in the discriminator is the inversion of directions – the first block of the discriminator which gets trained (the $4\times4$ block) is actually the last in the discriminator hierarchy, whereas the $4 \times 4$ block of the generator is the first block in the generator hierarchy. This immediately explains why the blocks pass a new conversion from RGB into convolutional channels if they are not getting trained (if the outputs of the green layers in Table 4.2 are not used) – the next layer expects to receive a tensor of convolutional filters, not an RGB image.

Apart from this dissimilarity, the discriminator is practically the same as the generator and as such we will not describe its architecture in as much detail. We instead refer the reader into the original text of Karras et al. [2017] for more information, or to the GitHub code repository of the network[2] for the detailed source code of the whole network.

## 4.2 Network modifications

While our ultimate goal is to explore the capabilities of generating high-dynamic range images by generative adversarial networks, there were a few modifications to the general network architecture which were simple to implement and had the potential to improve the output of the network. The first of these two modifications was adding a boolean fisheye masking layer to force the network to generate valid fisheye images (which have a circular shape inscribed in a square image). We also implemented an alternative way to enlarge images, instead of the traditional transposed convolution, since there has been some research pointing towards the transposed convolution producing unwanted image artifacts which decrease the overall image quality Odena et al. [2016].

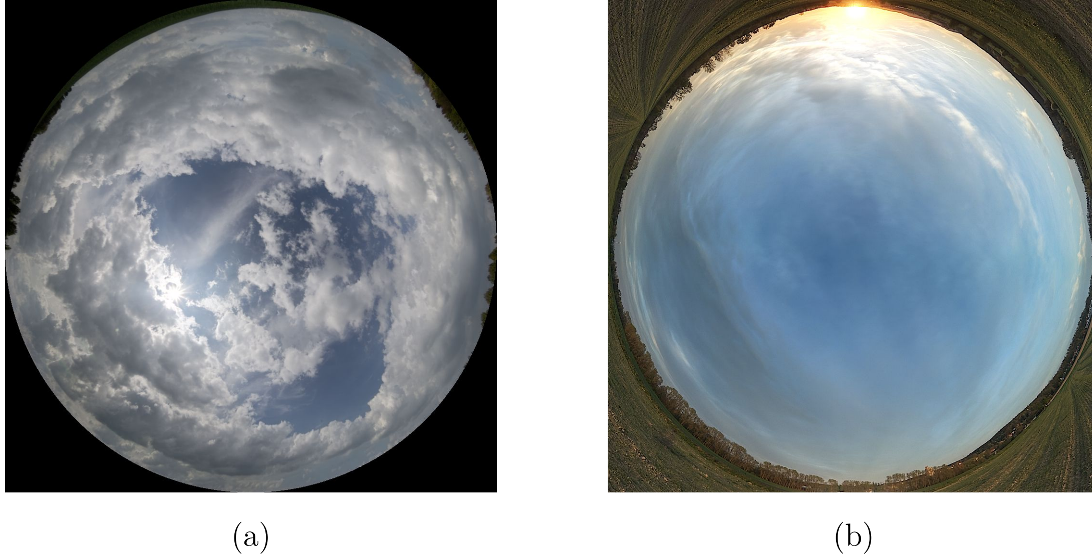
In this section we describe these additional layers we implemented, as well as the network's conversion to high-dynamic range images.

### 4.2.1 Fisheye masking

As we have already discussed in Chapter 3, we chose the *stereographic fisheye projection* with 180° of field of view for our image mapping. As you can see

[2]https://github.com/tkarras/progressive_growing_of_gans, accessed on 24. 06. 2019

in Figure 4.2(a), the skydome is compressed within the circle inscribed into the image frame and the corners of the image are filled with black color.



(a) (b)

Figure 4.2: Images captured by our dataset acquisition pipeline. Image (a) is the upper hemisphere projected using the stereographic fisheye projection with 180 degree field of view. The image is contained entirely within an inscribed circle of the image frame. Image (b) was captured by the Mi Sphere 360 camera and exported with PtGui. The corners outside of the fisheye circle are filled with the panorama's lower hemisphere.

If we started training the original network on these images, the network would have to learn to follow this precisely calculated circle. While this is not a complicated property of the dataset for a neural network to learn, as we will see in Chapter 5, we can force the network to generate this unit circle explicitly, which has several positive effects. We first discuss the simple solution to force the network to generate circular images and then explain the positive effects.

As we explained in the previous section discussing the network architecture, the generator network always outputs an image at the full target resolution, even if several upscaling layers have not been trained, because the resulting image passes through the upscaling layers of the following blocks. This helps us greatly, since as we know our target resolution (how big the resulting image will be), we can generate a boolean mask image, which has the shape of the unit circle for the given resolution. We illustrate a few generated masks in Figure 4.3.

We can perform pixel-wise multiplication of the output image from the generator with this mask and therefore set anything outside this unit circle to a fixed color. We also have to process the incoming images in the discriminator the same way. If we did not, the discriminator would very quickly notice if the fake images had a different value set in the corners than the real ones (for example, the real ones could have some leftover noise from the camera) and the training would diverge, since the generator would not be able to fix this issue. We experienced this divergence after not including the masking layer in the discriminator and the discriminator was able to learn this difference within the first 5 epochs of training and correctly classify every single fake image as a result. If we, however, multiply both the output of the generator, as well as the input of the discriminator by

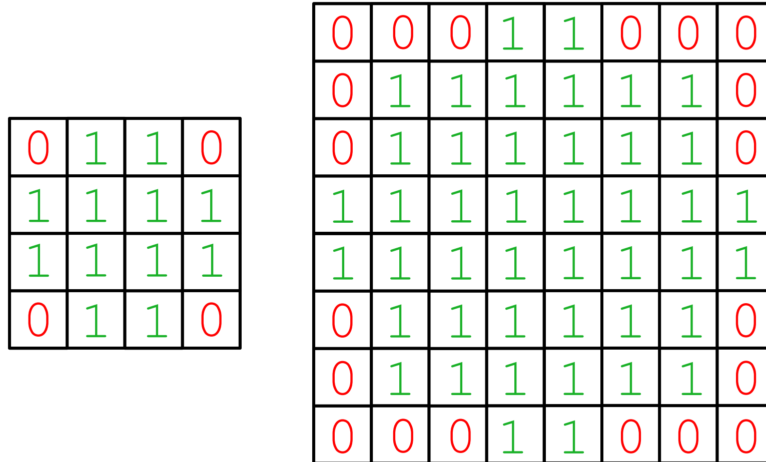


Figure 4.3: An illustration of a $4 \times 4$ and $8 \times 8$ circular mask. Pixels with a red zero will get masked, while pixels with the green one will go through the layer unchanged.

the same boolean mask, we effectively set the gradients of the pixels outside the unit circle to zero. This is because the pixels are exactly the same in all fake and real images and thus the discriminator cannot utilize these pixels to make the distinction between real and fake images. As a result, the discriminator will not ever create a gradient which would signal the generator to adjust these pixels. This effectively forces both networks to focus on the details inside the circle and disregard anything outside.

The first positive effect of this masking layer is the reduction of complexity. Our hypothesis was that since the generator network does not have to learn to generate a precise circle, it can utilize the computation time to actually generate the correct skydome. The discriminator also cannot learn to punish the generator for an imprecise circle, since for the image generated is inside exactly the same circle as the real data. We tested this hypothesis and report the results in the benchmark experiment.

The second positive effect is entirely dataset based and has nothing to do with the neural network. This masking allows us to ignore any artifacts of the camera's optical or sensor system in the image outside of the unit circle (like lens flares and sensor noise) and different outputs from the stitching software. If we shoot a skydome with the Sun visible in the sky, the optical system might create lens flares inside the whole image – even outside the inscribed fisheye circle. This would force the network to predict whether a lens flare will be present outside of the fisheye image circle and where, which is information we do not need and would increase the difficulty of our dataset in vain. The second reason was the fact that the stitching software does not respect the fisheye lens restriction of the unit circle and generates an image with corners filled with content from the second hemisphere (if we took a 360° panorama), as illustrated in Figure 4.2(b). Since we now focus on predicting the appearance of the upper hemisphere, this is an undesired effect, which we eliminate using the fisheye masking layer.

### 4.2.2 Replacing the transposed convolution

While training the network on low-dynamic range images, we noticed one of the most prevalent artifacts reducing the image quality was a checkerboard pattern artifact introducing very regular square-shaped cloud shapes, as can be seen in Figure 4.4.

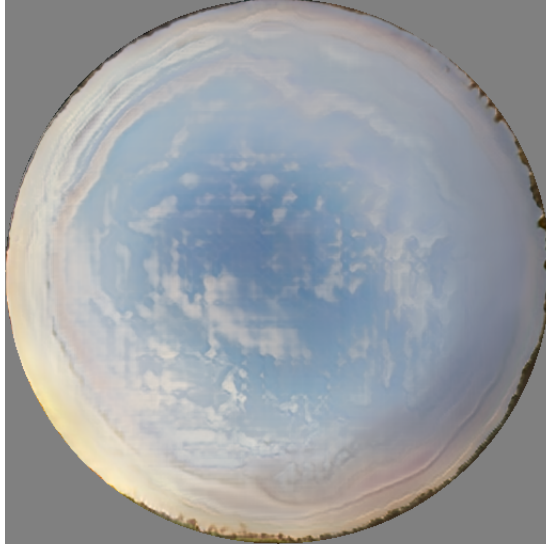


Figure 4.4: A low dynamic range image generated by one of our networks. The clouds have a distinct square-like shape, which we consider an artifact which reduces the image quality.

While searching for articles related to this problem, we found the work of Odena et al. [2016] who explore the cause of a very similar problem and conclude that the transposed convolution increasing the image size can degrade the image's quality. They also propose to exchange the transposed convolutional layer for a simple nearest-neighbour upscaling and a convolutional network on top, which is correctly padded to avoid overlapping.

We have already discussed the transposed convolutional layer in Section 1.2 but we briefly restate the mechanism here and illustrate it in Figure 4.5. During the computation of the transposed convolution, the input layer first gets padded around the edges, then a $(k, k)$ filter iterates over the input data and computes a new value for every pixel of the larger output layer.

During a resize-convolution, the input layer first gets upscaled to twice the resolution using a nearest-neighbour upscaling layer (which the network already utilizes as we have discussed in Section 4.1.2). After the data is upscaled a standard convolution is computed over the data to remove the upscaling artifacts and add new detail. An illustration of a resize-convolution can be seen in Figure 4.6.

This alternative approach should help to eliminate a so called checkerboard artifact which is created by the filter of the transposed convolution overlapping multiple times – which get stacked in the output layer and the checkered artifacts appear. For a more in-depth explanation we refer the reader to the original article which proposed this new layer by Odena et al. [2016].

We implemented this so called resize-convolution and will discuss the results of this modification in Chapter 5, but we do not think this modification made a

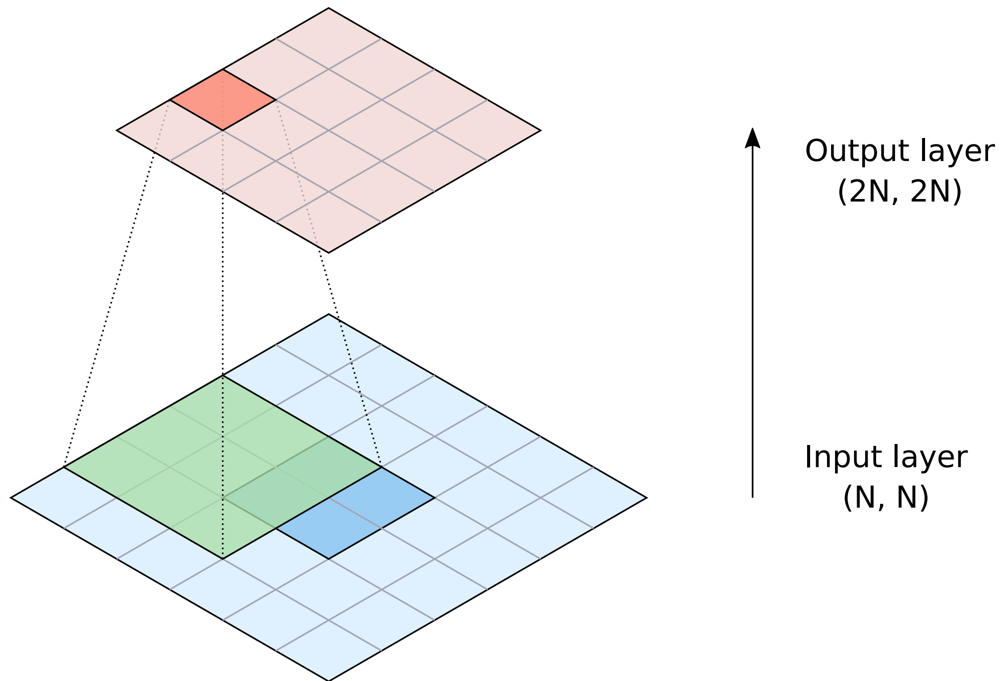



Figure 4.5: An illustration of a deconvolutional layer upscaling the input twice. The dark blue area is the input data, the light blue area is the padded input. The green filter then moves over the padded input producing the red output layer. The current output cell being computed right now is highlighted in bright red.

significant impact on the image quality. This, however, is hard to confirm, since image quality is a rather subjective measure.

### 4.2.3 Conversion to high dynamic range

As we have already mentioned, we spent a significant amount of time using the network for low dynamic range images at first. Reproducing the image quality the authors achieved on low dynamic range images was crucial, because without doing this first, we could not effectively produce a comparison of the performance on low and high dynamic ranges. Once we were satisfied with the performance of the low dynamic range networks, we converted the network to process and generate high dynamic range data. Since the authors provided the tensorboard[3] summary data for their training runs, we could directly compare the values of the summaries captured during training and see whether the behaviour we are observing is similar to the behaviour observed by the authors. Even though the training was executed on different hardware setups (we cover our hardware setup in Section 4.3), we found our training runs behaved in a very similar fashion to those of the authors of the network.

The conversion consisted of several modifications as we needed for the network to be able to both load a dataset in the high dynamic range format, as well as generate and output the high dynamic range generated images.

Since the network was designed for LDR images, the dataset loading operated in integer values. We did not want to hardcode the floating point arithmetic

[3]https://www.tensorflow.org/tensorboard, accessed on 24. 06. 2019

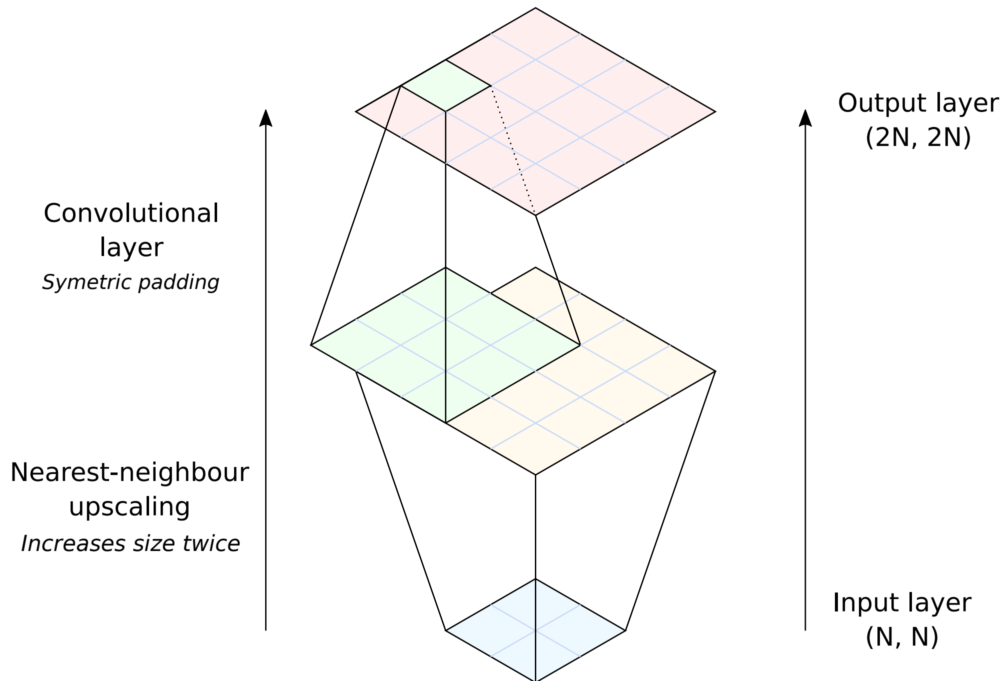



Figure 4.6: An illustration of a *resize-convolution layer* upscaling the image twice. The input first gets upscaled twice using a nearest neighbour filtering, the result is them symmetrically padded and put into a convolutional layer, which generates the additional detail and removes the upscaling artifacts.

instead, so we extended the network to accept both kinds of datasets (from which the user can choose by using the appropriate constructor parameter).

The second and bigger problem is the fact that HDR images tend to, per definition, have a high range of utilized values inside the image. This range, defined as the difference between the brightest and darkest spot in the image, is usually around $10^4$ for brighter images with the Sun directly visible (the Sun having values in the hundreds, while the shadows having values around 0.01), and around $10^3$ for the less bright images (for example overcast skies). As we can see, this range is much bigger than the $[0, 255]$ used for LDR RGB images. This high range is also not evenly represented in the image – the majority of the image is contained within a few exposure steps, but the Sun is extremely bright. This relationship can be complicated for the network to learn as is, so a mapping function converting the dataset into a different domain, to alter the big dynamic range, might be beneficial.

To have more control of the dynamic range, the network can operate in the logarithm domain [Eilertsen et al., 2017]. This is done by applying a natural logarithm to the value of each training image, after shifting the data by a constant $c$. This shift is necessary, because images often contain the value 0, which could not get mapped using the natural logarithm. Depending on the constant $c$, the shift either increases the dynamic range (for $c < 1$) or decreases the dynamic range (for $c > 1$). We did not hardcode this constant $c$ and instead made it adjustable through the network's configuration file. We will showcase the difference the different choices of $c$ make in Chapter 5.

Since the network learns to generate data mapped this way, once we generate a new image, we need to shift it back before we save it. The inverse function of

this mapping is very simple, and the whole process is described in Algorithm 2.

**Data:** hdr image $I$, image $G$ generated by the network
**parameter:** shifting parameter $c$
$I' = I + c$;
$I' = log_e(I')$;
*... train the network, generate G ...*
$G' = \exp(G)$;
$G' = G' - c$;
*... save the image G'*

**Algorithm 2:** The whole process of mapping an HDR image using a logarithm, with adjustable constant $c$.

In the original neural network, the RGB images got remapped from $[0, 255]$ to $[-1, 1]$ before training and remapped the images back before saving, much like in our case of Algorithm 2, except the mapping used was a simple linear transformation. Since the original network also operates on floating point numbers, we did not have to change the underlying architecture.

For the purpose of saving the generated image, we utilize the Python version of the OpenCV library [Bradski, 2000]. This allows us to choose an arbitrary format to save the generated data. For example, since the majority of the dataset we collected on the internet is saved as RGBE (`.HDR`) format, we chose this format for training on this dataset. Once we started utilizing our dataset, which is exported in OpenEXR format (`.EXR`), we changed the output format to `.EXR`. This is because when invoked, the generator returns the unmodified, unshifted data which it generated, which we then have to shift back according to Algorithm 2, and then can save as a format of our choosing. This allows the user to swap between formats as he sees fit.

## 4.3 Implementation details

In this section we give a detailed description of both our hardware and our software setup to allow the reader to better reproduce our work. We also mention the average training time on this hardware, to give the reader a general idea of how fast the network converges to the images we showcase in Chapter 5.

While the authors provide us with a list of requirements for the python environment, as well as system requirements, they do not use the Docker [Merkel, 2014] container software to make the network easier to use. In this section, we provide an overview of both our hardware and software setup to run the network. We utilize the Docker software and have created Docker images which are pre-configured to contain all prerequisites to run the network.

**Hardware we used to train the network** At first, we utilized one powerful computer to hold four GPU cards from Nvidia[4], namely the nVidia GTX 1080Ti. The computer configuration looked like this:

- *CPU* – 16-core AMD Ryzen Threadripper 1950X

[4] https://www.nvidia.com/en-us/, accessed on 24. 06. 2019

- *RAM* – 128 GB
- *GPU* – 4× nVidia GTX 1080Ti
- *Storage* – 2× 1 TB NVMe SSD
- *Operating system* – SMP Debian 4.9.144-3.1 (2019-02-19)

While this had the advantage of having four GPUs at our disposal at the same time, there were several issues with overheating and temperature based shutdowns, if the GPUs were used continuously for a longer time. The temperature on the GPUs would rise to about 91°$C$, which is the operational limit of these cards, as provided by Nvidia [Nvidia].

We also experienced a problem with loading the data onto the GPUs because of PCIe lanes. When two GPUs on the same lane were being utilized for the same task in parallel, the task would freeze and the computation would not start. If two GPUs which were not on the same PCIe lane were used, the task would perform fine.

These two flaws made us re-configure the setup and move two of the GPU cards to a different computer, leading to the following hardware setup in Table 4.3 and Table 4.4.

| **Computer** 1 | |
|---|---|
| *CPU* | 16-core AMD Ryzen Threadripper 1950X |
| *RAM* | 128 GB |
| *GPU* | 2x nVidia GTX 1080Ti |
| *Storage* | 6TB HDD (local), 2x 1 TB NVMe SSD (docker root) |
| *Operating system* | SMP Debian 4.9.144-3.1 (2019-02-19) |

Table 4.3: Hardware configuration of Computer 1. The storage is divided into local storage for users and docker storage for faster container performance.

| **Computer** 2 | |
|---|---|
| *CPU* | 24-core Xeon E5-2680 v3 |
| *RAM* | 256 GB |
| *GPU* | 2x nVidia GTX 1080Ti |
| *Storage* | WD 3TB HDD (local) |
| *Operating system* | SMP Debian 4.9.144-3.1 (2019-02-19) |

Table 4.4: Hardware configuration of Computer 2.

These two computers have a shared network storage, which allows the users to keep just one copy of the source code of the network.

Utilizing one of these computers, the average training time for a reasonably converged network was around $4-6$ days for resolutions from $128 \times 128$ to $512 \times 512$. We also note in this section that training a $1024 \times 1024$ with only 11 GB of GPU memory, while not impossible, would require significant tuning of the network's batch parameters. This is because the size of a tensor with shape $(batch, channels, 1024, 1024)$ is significant. For high resolutions, the authors used $batch = 3$ and $channels = 32$ for the final stage of the convolutional pyramid, which already is almost 400 MB just for the current training batch. We have ever only trained the network up to $512 \times 512$ for the purposes of faster evaluation and bigger batch sizes (which result in a faster training speed).

**Software setup** Now that we have a good idea of the hardware we used and its performance, we want to briefly mention the software setup of these machines. This is both to highlight the Docker approach we have chosen, as well as to enable the reader to fully reproduce our environment.

As we can see in the previous Tables 4.3 and 4.4, the operating system we used was a Unix-like system Debian. Additionally, we had to install the following software to be able to train the network on these computers.

- **Docker** – which we already mentioned, is a virtualization tool which allows the user to define a custom container for a project. This container is based on a certain operating system and allows the user to install all software requirements of his project, without interfering with other users' configuration.
- **Nvidia drivers** – since we utilize Nvidia GPU cards, we need to install the proprietary Nvidia drivers for these cards.
- **CUDA** – a parallel computing platform developed by Nvidia, which allows high performance computations GPUs. Neural network training is one such computation which can be sped up by running on a GPU. We used version 9.0 on all of our machines.
- **nvidia-docker** – an extension of the Docker platform developed by Nvidia, which allows the users to access Nvidia GPUs.. This piece of software is crucial for our computers, since without it, we would not be able to utilize the CUDA computations.

Once this computer environment is setup, we can now build and run our Docker images to train our network. Since the Docker image defines and automatically installs all the network's prerequisites, the user does not need to worry about installing Python, Tensorflow, NumPy, OpenCV or any other software on the computer itself.

# 5. Results

In this chapter we first present the results of our dataset acquisition, which we presented in Chapter 3. We then showcase some data generated by the network as we implemented different features discussed in Chapter 4. We also present several benchmarks of the network which we ran to get a better understanding of the network's capabilities.

## 5.1 Dataset acquisition

As we have already mentioned, developing a method to create a dataset is important since no skydome dataset exists. In Chapter 3 we described the process and explained our decisions regarding the method we devised. Here we briefly showcase the results of our dataset acquisition.

It is important to note that we have had several people from Charles University willing to set-up the camera and take the pictures. Here we accumulate all pictures taken by our research group.

As we have already described in Chapter 3, deciding on the exact method required experimentation with both hardware and general approach. This included shooting full 360° panoramas with the *DSLR* camera using the panoramic head, shooting full panoramas with the Mi Sphere 360° camera and finally shooting the upper-hemisphere skydome images with the 8 mm fisheye lens. In this section we present all the results, even though we have abandoned shooting the full 360° panoramic images due to the disproportionate amount of manual work, as we have discussed.

We also note that the majority of these images was shot in Prague, Czech Republic. We are aware that this fact is going to bias the dataset heavily, but at the moment we do not have a long-term solution to this problem available.

Having said this we now list the numbers of images captured, illustrate some examples and provide the positions which we shot the images at.

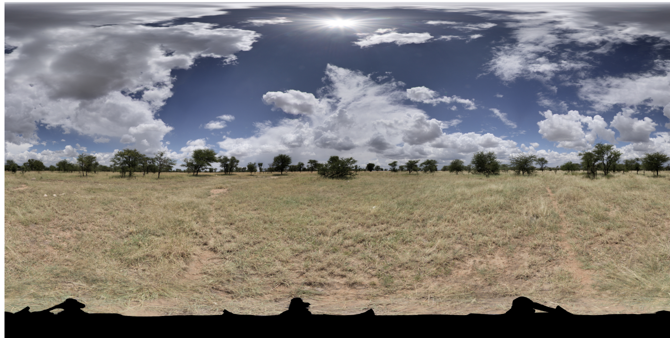
### 5.1.1 Full 360° panoramas

We first present the full 360° environment maps we shot as experiments to see if this way of gathering the dataset would be sustainable in large quantities. In in Chapter 3 we have determined that shooting these environment maps requires too much manual work. We concluded this after manually processing the following images.

Figure 5.1 illustrates a few examples of full $15000 \times 7500$ environment maps we shot in Tanzania, Africa. These maps were composed of anywhere from 60 to 70 images each. These photos take up roughly 1.8 *GB* of memory on the disk and have to get loaded into PtGui to stitch these panoramas. This requires a reasonably powerful personal computer in order to avoid the risk of running out of memory.

We have data for roughly 20 environment maps of varying qualities. The varying quality results from the fast movement of clouds in some conditions. This movement unfortunately makes the panoramas very difficult to properly

(a) Shot on 18.02.2019 at 13:30.



(b) Shot on 16.02.2019 at 11:40.

(c) Shot on 17.02.2019 at 09:45.

(d) Shot on 16.02.2019 at 10:00.

Figure 5.1: A few 360° panoramas shot with our Canon 5D DSLR. These images were taken in Tanzania, Africa. The images are stitched from around 60 to 70 shots using the PtGui software [B.V.]. For illustration, we masked the tripod in the first three images and left the tripod in the shot in the last image.

stitch, because fast moving clouds result in blurry images and visible seams. We did not include any such ”incorrect” panoramas in Figure 5.1.

We also shot around 20 full 360° panoramas with $15000 \times 7500$ pixel resolution in Prague, within the city. The sky is more obstructed than we would want for our purpose and since this is a full resolution environment map, all the manual work was still present. We include one image to illustrate the shooting place in Figure 5.2.

Figure 5.2: One full 360° panorama shot in Prague. The image was taken on 31.03.2019 at 16:45.

As we have mentioned in our hardware survey in Chapter 3, we also experimented with a 360° camera, namely the Mi Sphere. The shooting process with this camera is a little easier since no manual work with the hardware is involved. However, the camera lacks both the *exposure bracketing* and the *invervalometer* feature we discussed earlier and thus manual work is required to control the camera's software. We have managed to get 63 shots total in sunny days, overcast days and sunsets. We showcase a few images in Figure 5.3.

These images were captured as experiments to see whether this acquisition method is viable and how much work it entails. Because of this, we have neither vast quantities nor exact procedures to unify the processing of the photos.

(a) Shot on 22.04.2019 at 19:51.

(b) Shot on 25.04.2019 at 19:45.

(c) Shot on 25.04.2019 at 20:06.

(d) Shot on 26.04.2019 at 19:25.

Figure 5.3: A few 360° panoramas shot using the Mi Sphere 360° camera. All were shot in the vicinity of České Budějovice city during one week. The images are stitched from two 180° fisheye shots using the PtGui software [B.V.].

### 5.1.2 Upper-hemisphere skydome images

After determining that 360° shots require too much manual work for our small research group to deal with, we decided to simply shoot the upper hemisphere of the environment. We have described our reasoning for this decision in Chapter 3. In the following pages we present the numbers of images we shot, as well as showcase examples from different shoots.

We present all the data acquired in Table 5.1. Note that this list is true at the time of writing this thesis and will grow significantly in the future. We only include the most important data in Table 5.1, we also attach the whole spreadsheet in Attachment A.1.

| Date | Interval | Shots taken | Location |
|---|---|---|---|
| 17.5.2019 | 0:02:00 | 12 | Staré Hodějovice |
| 18.5.2019 | 0:02:00 | 40 | Staré Hodějovice |
| 24.5.2019 | 0:02:00 | 62 | Prague, Šárka |
| 4.6.2019 | 0:02:00 | 7 | Prague, Žižkov |
| 4.6.2019 | 0:02:00 | 16 | Prague, Žižkov |
| 4.6.2019 | 0:02:00 | 52 | Prague, Šárka |
| 5.6.2019 | 0:02:00 | 61 | Prague, Žižkov |
| 5.6.2019 | 0:02:00 | 61 | Prague, Šárka |
| 6.6.2019 | 0:02:00 | 14 | Prague, Žižkov park |
| 6.6.2019 | 0:02:00 | 48 | Prague, Parukářka park |
| 7.6.2019 | 0:02:00 | 248 | Germany, Hammerschmiede |
| 8.6.2019 | 0:02:00 | 8 | Germany, AdventureSteinbruch |
| 10.6.2019 | 0:02:00 | 61 | Germany, Hammerschmiede |
| 12.6.2019 | 0:02:00 | 28 | Prague, Stromovka, CGBBQ |
| 16.6.2019 | 0:01:00 | 128 | Prague, Vinohrady rooftop |
| 20.6.2019 | 0:00:30 | 428 | Prague, Vinohrady, rooftop |
| 20.6.2019 | 0:01:00 | 177 | Prague, Vinohrady, rooftop |
| 20.6.2019 | 0:01:00 | 512 | Prague, Vinohrady, rooftop |
| 21.6.2019 | 0:00:30 | 314 | Prague, Vinohrady, rooftop |
| 22.6.2019 | 0:00:30 | 44 | Statek Výštice |
| 23.6.2019 | 0:00:30 | 50 | Statek Výštice |
| 23.6.2019 | 0:00:30 | 262 | Prague, Vinohrady, rooftop |
| 24.6.2019 | 0:00:30 | 400 | Prague, Vinohrady, rooftop |
| 25.6.2019 | 0:00:30 | 242 | Prague, Vinohrady, rooftop |
| 26.6.2019 | 0:00:30 | 308 | Prague, Vinohrady, rooftop |
| 26.6.2019 | 0:00:30 | 498 | Prague, Vinohrady, rooftop |
| 26.6.2019 | 0:00:30 | 378 | Prague, Vinohrady, rooftop |

Table 5.1: The upper-hemisphere skydome images we have taken to create our dataset.

As we can see, we have been gradually decreasing the interval at which we take the photos. The is largely because we noticed that even with medium winds, the image changes drastically during 30 seconds, as the clouds change form fast. The total number of images we shot as of writing this thesis is 3935 and as we can see from the Table 5.1, the majority was captured during the month of June (except

for 104 photos from May). This nicely showcases the capability of our pipeline and the fact that it is viable to shoot around 5000 photos per month if we get enough human resources to keep the camera operating. This means that during the course of about half a year we should be able to collect a dataset approaching the sizes of other smaller neural network datasets, such as the portrait dataset of Karras et al. [2017] and Karras et al. [2018].

We follow the table with example images from the different shoot locations in Figure 5.5. We also include Figure 5.4 which illustrates one single approximately 4 hour shoot at different time points. The images in both figures are processed in RawTherapee, tone mapped from HDR and converted into the stereographic projection.

While we have shot a lot of photos, we can neither show them here nor attach them as a digital attachment, since they take up a lot of space. We do, however, include some small resolution ($1024 \times 1024$) LDR time-lapses in the Attachment A.1 to hopefully give the reader a general idea of the quality and variance in our images. We want to make the dataset available to the general public but we have to figure out a solid means of distribution first.

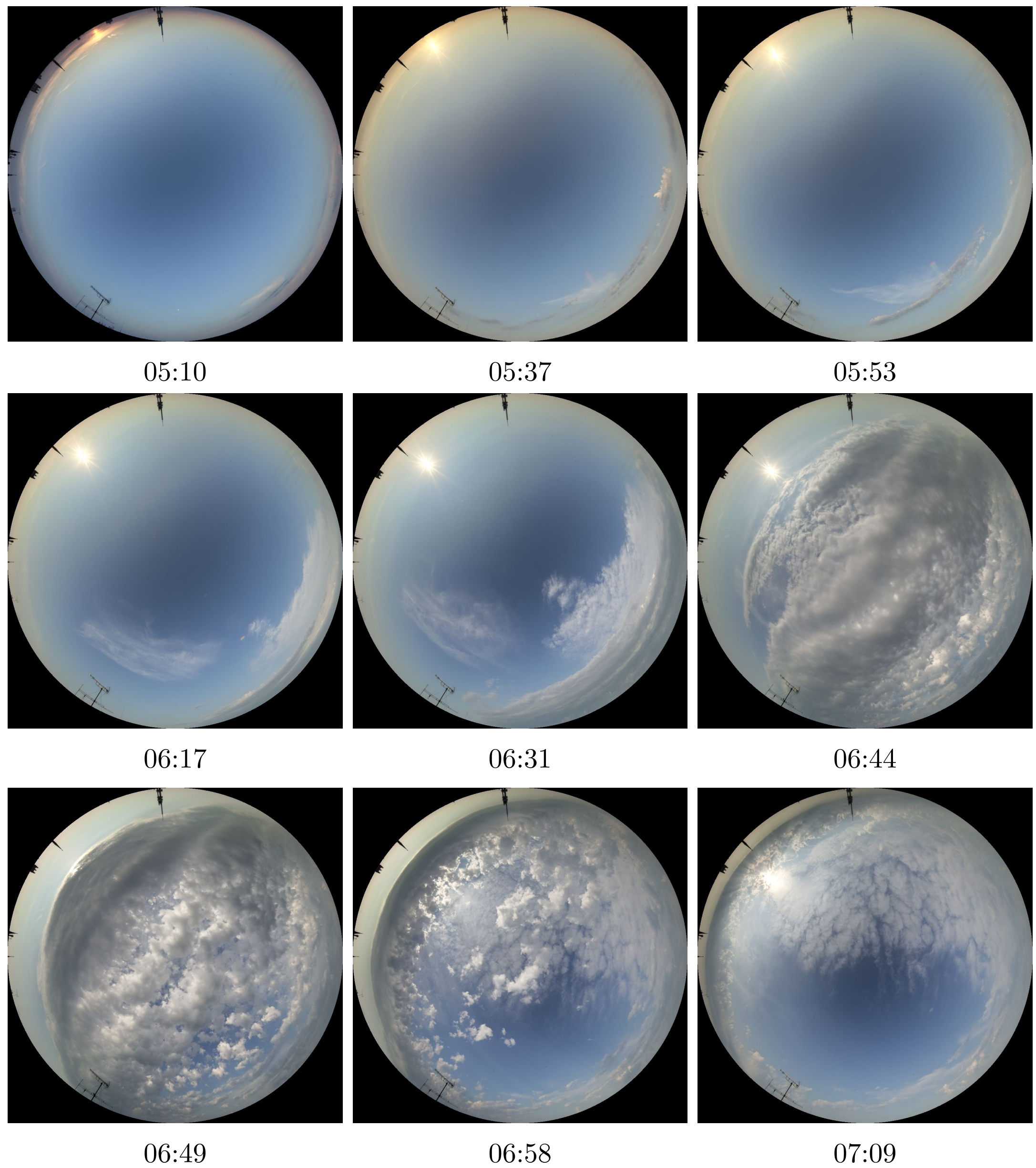



Figure 5.4: A small subset of an approximately 4 hour time-lapse taken in Prague, starting from 4:50 and ending at 8:30. Each image is marked with the time it was taken, to further illustrate how drastically the sky changes in a few minutes.

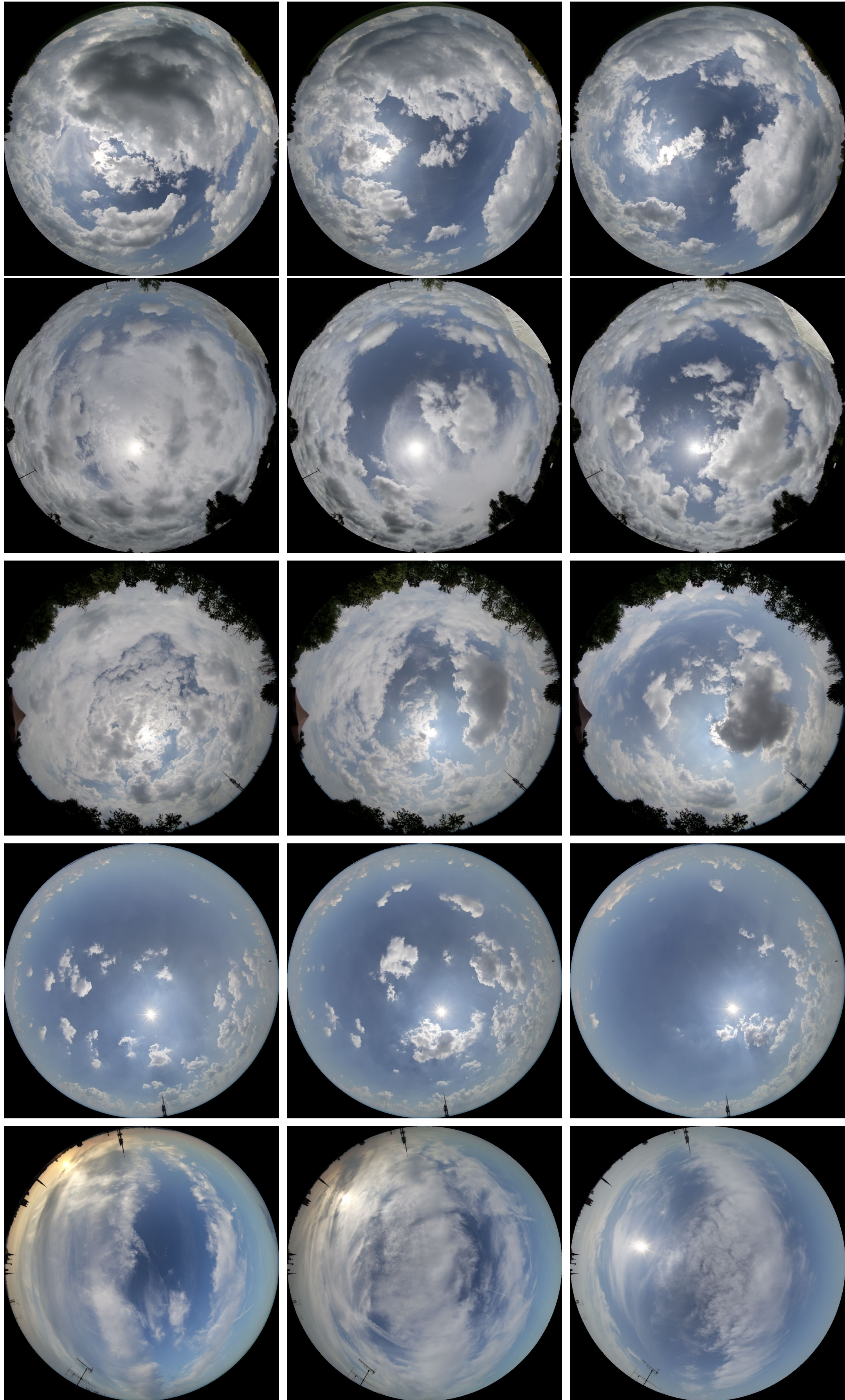


Figure 5.5: Samples from different dataset shoots. Each row represents one shoot, sampled at three different times.

## 5.2 Network experiments

In this section we describe all of our experiments we performed on the network during our research. We also conducted some "benchmarks" of the network, testing its ability to overfit on HDR data, as well as experimented with omitting the new layers we described in Chapter 4.

### 5.2.1 Low dynamic range

As we have already mentioned in the previous chapters, we started with a low dynamic range network format. This is because we wanted to make sure we can reproduce the performance of the network in our hardware setup, as well as on our rather limited dataset. For this reason, the first few experiments we describe in the following paragraphs are performed on low dynamic range networks only. We will notify the reader when we switch to HDR imagery.

We also want to point out the fact that results showcased here highlight our research process. This is why the first networks do not utilize the fisheye projection, as well as the additional layers we described in Chapter 4.

We will now cover the contents of our dataset as well as our exact data augmentation parameters which should, along with Section 3.2 of Chapter 3 give the reader all information about our dataset augmentation.

Our dataset consisted of roughly 650 images gathered from the internet. We describe all the sources in Attachment A.2. All of these images were at least 2000 by 1000 pixels in resolution, providing us with enough detail for the purpose of the neural network generation.

While we have already explained our dataset augmentation and its reasoning in Section 3.2, we reiterate here and provide specific constants we chose. Since the data we gathered from the internet is HDR, we tone map the images using the Reinhard operator [Reinhard and Devlin, 2005] in OpenCV [Bradski, 2000] with intensity set to 1 and light adaptation set to 0 to make the operator global. Once we obtain LDR images by tone mapping the HDR data, the best data augmentation method we have available is the rotation around the vertical axis (rotating the image azimuthally). We chose to rotate with $\phi = 36°$ as the base rotation, which gets randomly offset at the start by an angle $\phi_o \in [0, 36°]$ before the rotation to keep the network from learning the discrete Sun positions. We then resize the tone mapped and rotated images into all resolutions the network goes through while training. For example, for a $64 \times 64$ target resolution, we need $4 \times 4$, $8 \times 8$, $16 \times 16$, $32 \times 32$ and $64 \times 64$. As the last augmentation step the network itself uses vertical mirroring of the images during the runtime.

Finally, the authors of the network provided some hyperparameter presets for training the network on different numbers of GPU cards. Because these presets are targeted at the full $1024 \times 1024$ resolution, we eventually created our own for lower resolutions, but at the start, we used these presets to train our networks. We will mention the preset used in each experiment.

**Low resolution experiment** In the beginning we sought to find out if we would be able to use the network provided by Karras et al. [2017] in our research. We created an experiment to see if the network would produce reasonable results

at low resolutions in our hardware and software setup, which would help us determine whether we can use the network for further research. We showcase a part of the training set in Figure 5.6. As you can see, remapping a high-resolution environment map into a $32 \times 32$ image reduces the quality significantly, as well as distorts the image even more. This is of little concern, as our primary goal was to figure out if we can get the network to generate similar data, using the progressive architecture of Karras et al. [2017].

Figure 5.6: A sample from the real dataset from the NoEmotion team. Resized to $32 \times 32$ resolution and tone mapped.

We downloaded all day and evening HDR environment maps from the NoEmotion team. Utilizing our pipeline we converted the images into low dynamic range and generated all resolutions the network would need to train a target resolution of $32 \times 32$ pixels. We ran the network on the `v2-1gpu` preset for 6 hours, until the network started training the full resolution. We made a mistake at this point and shut down the training, even though the network only trained for 2 epochs on the target resolution (once all blocks have been faded in) which we later found out was too early. For comparison, the authors of the network let the network train on the target resolution for several days.

Regardless, after 6 hours of training we obtained results shown in Figure 5.7.

These results suggest that the network is capable of reproducing the quality seen in our dataset, albeit the fake images are a bit more blurry than the dataset. It is possible that this blurriness was caused by us shutting the network down too early.

Nevertheless, based on this experiment we can say that the network is capable of training and producing images of reasonable quality on our setup. This result was a signal for us to continue using this network for our further research.

Figure 5.7: A matrix of images generated by our trained network. The network was trained for 6 hours on 1 GPU card.

**Scaling into a bigger resolution** As the generated images in the last experiment are of satisfying quality, we wanted to find out how this quality would change when scaled into a bigger resolution.

Since the previous dataset, containing only images from the NoEmotion team, was small in size containing only about 100 unique images, we increased the dataset to the full 650 images we found on the internet (all the sources can be seen in Attachment A.2). We also removed the lower hemisphere (containing the ground) from the images, leaving us with a 360° (longitude) by 90° (latitude) equirectangular mapping. This also further distorts the images when we remap them into a square image, which the original network requires. We then again converted the images into LDR, rotated them to augment the data, and generated all resolutions required to train the network to $256 \times 256$ pixel resolution. We chose this resolution because it is a significant improvement over the resolutions which GAN architectures were able to generate just a few years ago. A sample of the dataset can be seen in Figure 5.8. We utilized 2 GPU cards and the `v2-2gpu` preset from the network's authors. The network was trained for 3 days and 17 hours, this time avoiding the mistake of shutting down the training too soon (the network this time saw $9.4M$ real images and trained on the target resolution for 55 epochs).

A few generated images can be seen in Figure 5.9. We will now discuss the properties of the generated images to evaluate the results of this experiment.

As we can see from the generated images, the scenes do look plausible, albeit low resolution and blurry. One thing we immediately noticed was the *checkerboard artifacts* the network seems to create in larger areas of clouds, showcased closely in Figure 5.10. This later prompted us to look into the work of Odena et al. [2016], described in Section 4.2.2 to improve the quality.

The big problem with equirectangular projection, however, is the continuity constraints on the image. In Figure 5.11 we showcase two generated equirectangular images from the network, which we manually converted into a fisheye projection, to see if the network learned the continuity constraints. As we can see, neither of the two main constraints – the zenith collapse (the image's top border has to collapse into one singular point – the zenith) nor the border continuity (the leftmost and rightmost edge of the image have to be seamless) got recognized by the neural network. While we could implement a custom convolutional layer which could "wrap" around the leftmost and rightmost edges in a cylinder topology to potentially solve the second issue of the seam seen in Figure 5.11 in image (c), the zenith collapse is a harder problem to solve. This particular problem prompted us to investigate other projections, as we have discussed in Section 3.2 of Chapter 3 and to choose the *stereographic fisheye projection* for our further experiments.

To conclude this experiment we note that we were glad to see the image quality not deteriorating too much with a significantly higher resolution. We also identified several potential weak points of the current architecture – the continuity constraints and the checkerboard artifacts.

Figure 5.8: A sample from the real data from the bigger dataset containing 650 unique images. Resized to $256 \times 256$ resolution and tone mapped.

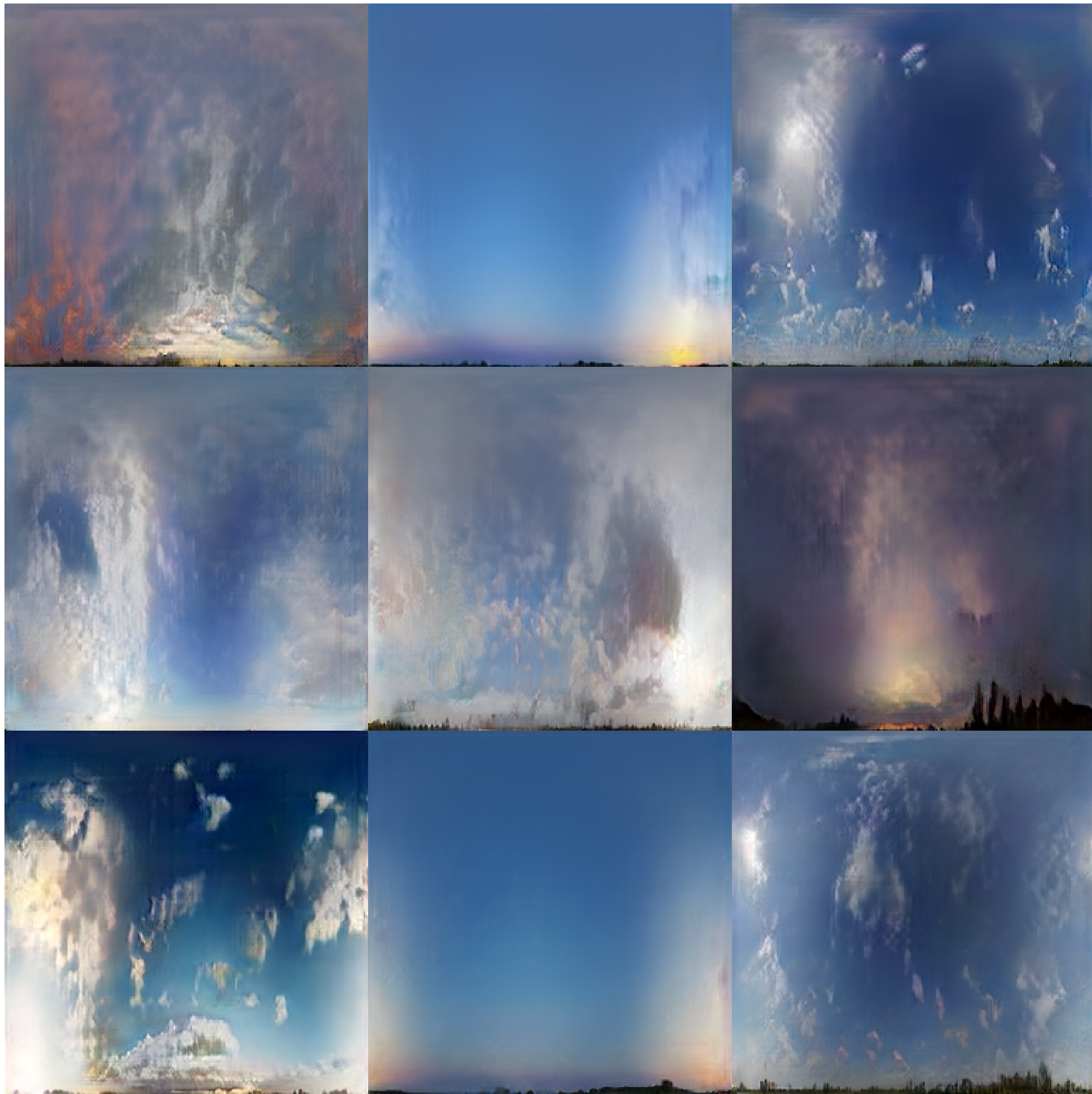


Figure 5.9: A matrix of images generated by our trained network. The network was trained for 3 days and 17 hours on 2 GPU cards.

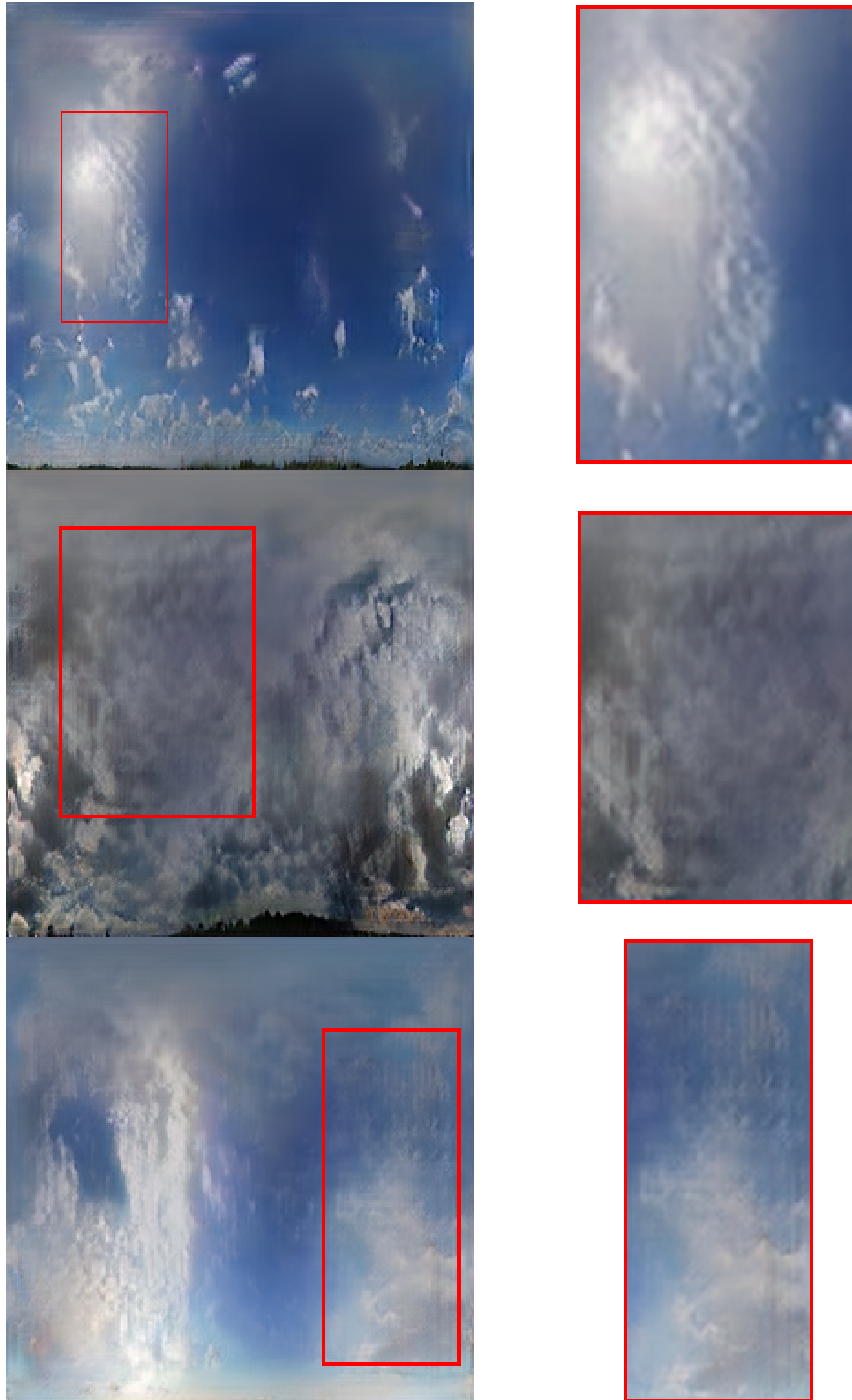


Figure 5.10: Examples of the so called *checkerboard artifacts*. The network creates the clouds with these distinct patterns inside, where brighter and darker spots alternate in rows and columns.

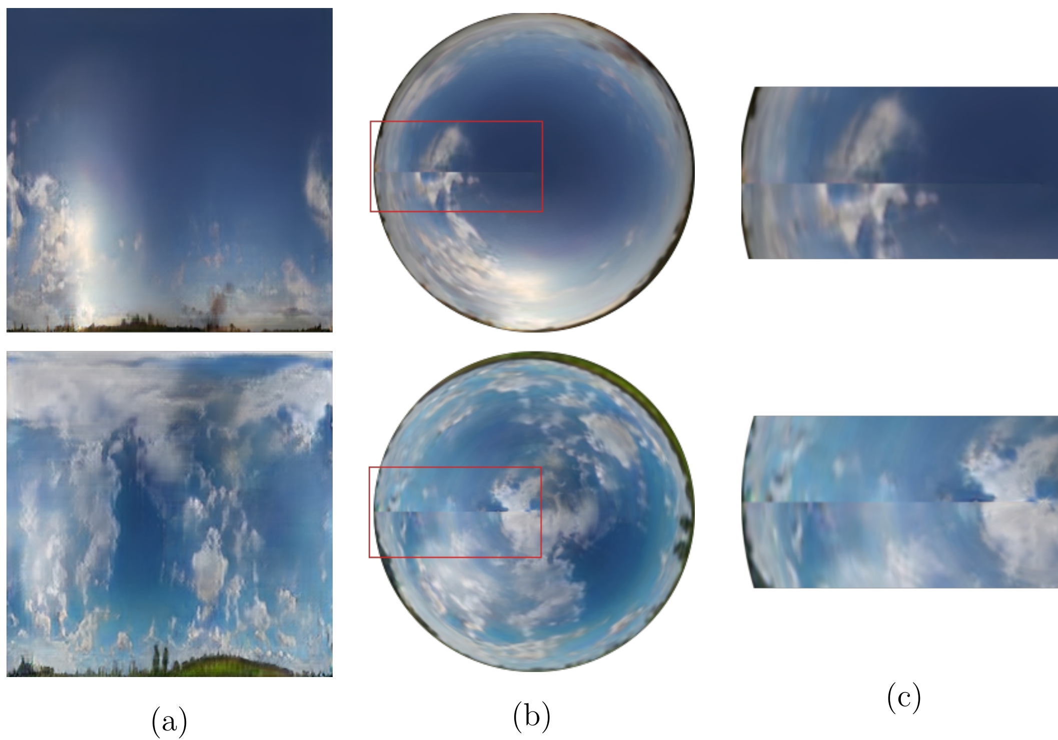



Figure 5.11: An analysis of continuity constraints of the equirectangular projection. The leftmost images (a) showcase the image as it was generated by the neural network. The images (b) have been manually converted to a fisheye projection. Images (c) are outcrops from images (b) highlighting the fact that the network did not learn either of the two constraints (zenith collapse and edge-continuity).

**Fisheye projection** As we have discussed in Section 3.2, we decided to use the *stereographic fisheye projection*. Our next experiment was designed to see if the network can generate this projection without any additional image quality decreases. At this point we also introduced the fisheye masking we explained in Section 4.2.1. The last goal of this experiment was to see if we gain any image clarity by letting the network train on even bigger images and thus we used a target resolution of $512 \times 512$. We note that although the network now generates stereographic fisheye projection, we are still interested in the equirectangular mapping as well. This is because the areas around the horizon, which are best visible on an image projected with equirectangular projection, are important since the camera in the *CGI* scene will most likely see at least part of the horizon. For this reason, we include both the fisheye as well as the equirectangular projections when we present generated results.

We used the same dataset containing around 650 images as in the last experiment, only this time we remapped the equirectangular panoramas into a stereographical fisheye projection. The tone mapped and augmented fisheye images were then resized to the target resolution of $512 \times 512$. We show a sample of the training dataset in Figure 5.12. We used the `v2-2gpu` preset and trained the network for 6 days and 1 hour, to let it converge as much as possible. Note that this experiment does not yet utilize the work of Odena et al. [2016].

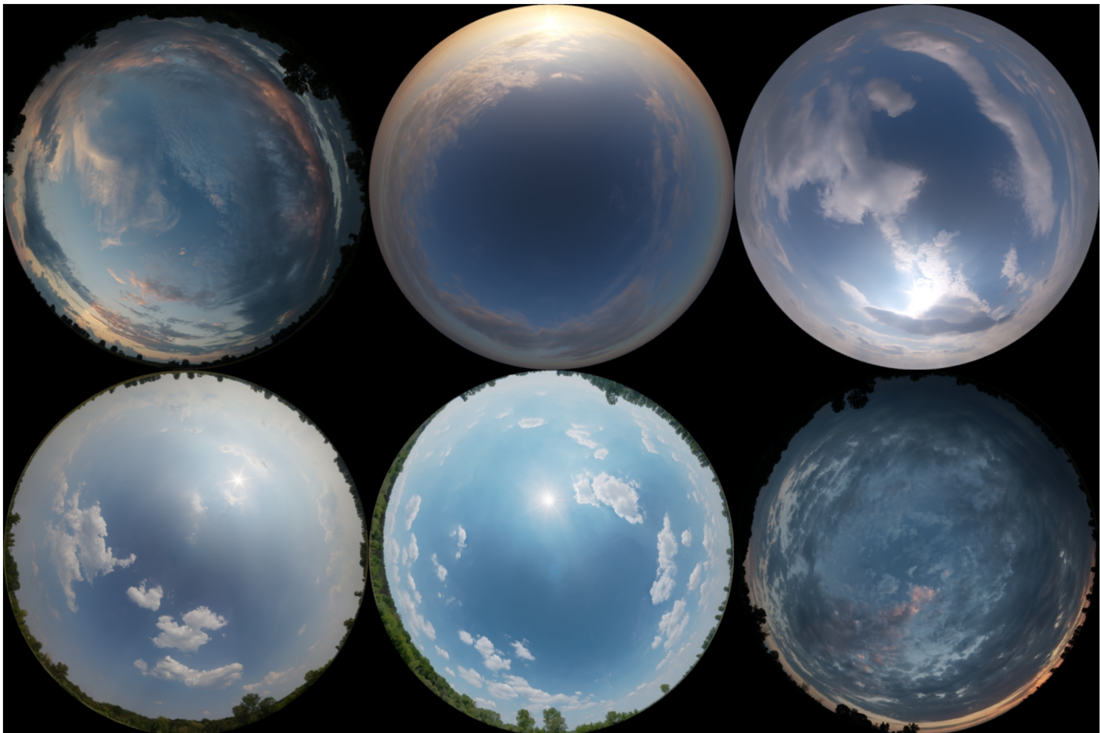


Figure 5.12: A sample from the real data from the fisheye projected dataset. Stereographic fisheye projection was used in these images. Images were resized to $512 \times 512$ resolution and tone mapped.

Figure 5.13 illustrates several images as generated by the trained network in the stereographic projection. We also include Figure 5.14 in which we converted the fisheye projection back into equirectangular projection for better understanding. We will now further discuss the quality, as well as the lighting conditions

and the realism.

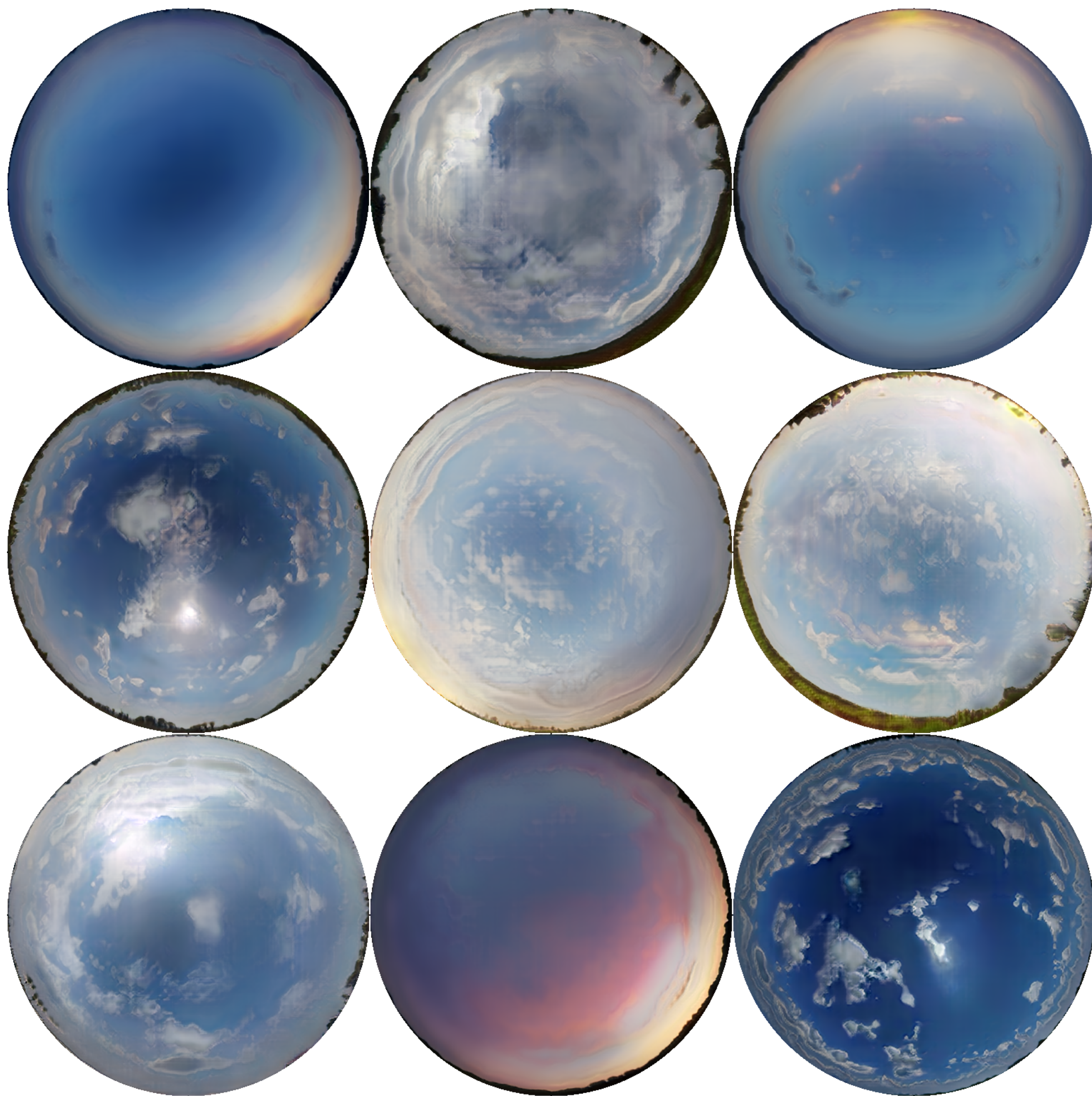


Figure 5.13: A sample from the data generated by the LDR fisheye network.

As we can see on the rightmost image in Figure 5.15, the checkerboarding artifacts which started to appear during the last training run still prevail. This fact was what made us decide to implement the new upscaling method of Odena et al. [2016] to try to reduce this behaviour. We can also see some unnatural behavior on the edges of the clouds, where the network creates some artifacts which look similar to biological cells or oil stains. Figure 5.16 showcases this type of artifacts. These artifacts appear consistently throughout the rest of this thesis and while we did research to figure out how to remove this behaviour, we did not manage to find out the exact reason of this happening. We would probably require a significantly bigger dataset to provide further insight into this type of artifact.

Other than these two artifacts, the images overall look plausible. We generated 250 random images and manually labeled them into them into four categories – plausible looking, unrealistic because of wrong lighting, unrealistic because of wrong sky composition, and plain wrong results. Figure 5.17 highlights the three failure cases. Out of the 250 generated images, we subjectively classified 23 as

having incorrect cloud formations (left column of Figure 5.17), 10 as having incorrect lighting (middle column of Figure 5.17) and 18 as being a complete failure case (right column of Figure 5.17). This amounts to 51 images out of 250. We think that increasing the dataset would help significantly improve this ratio and the overall quality of the images.

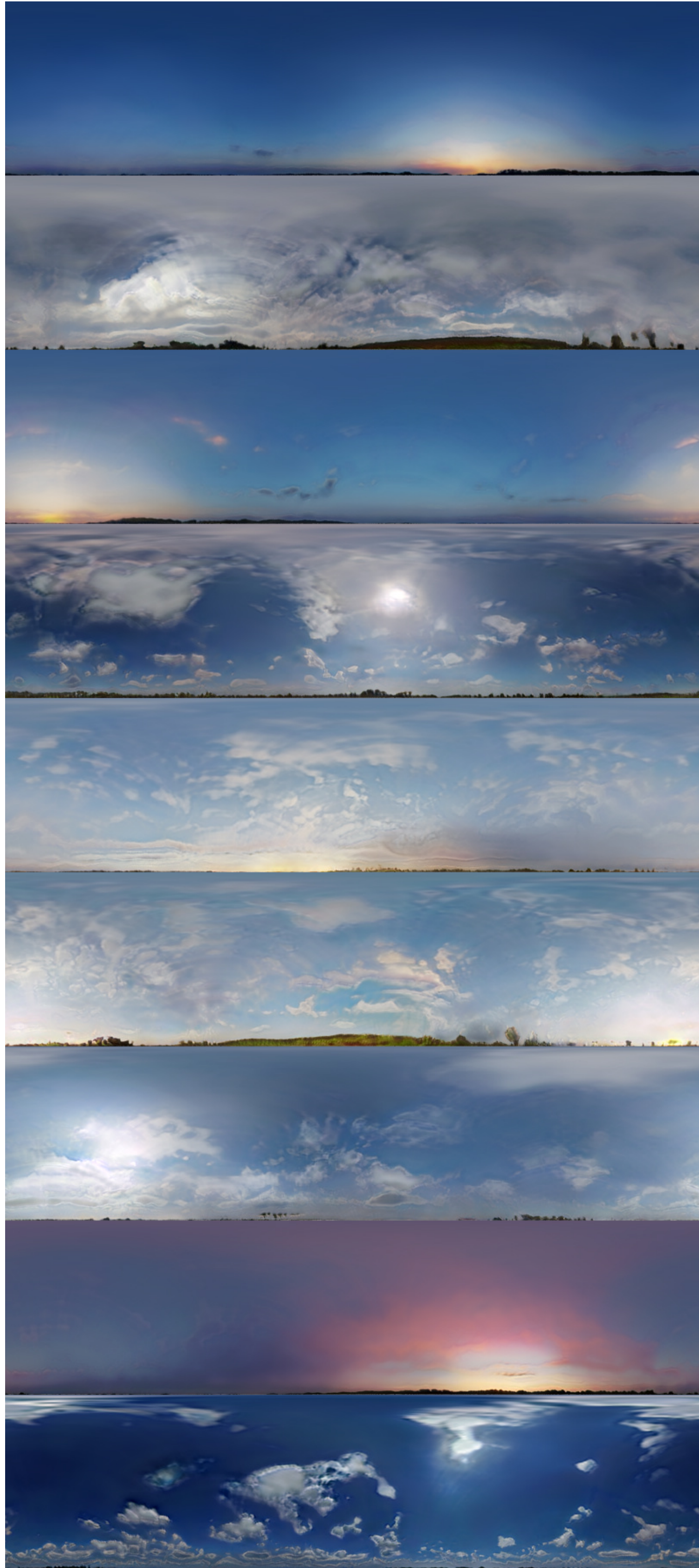


Figure 5.14: A sample from the data generated by the fisheye network remapped into equirectangular projection. This figure contains the same images as Figure 5.13.

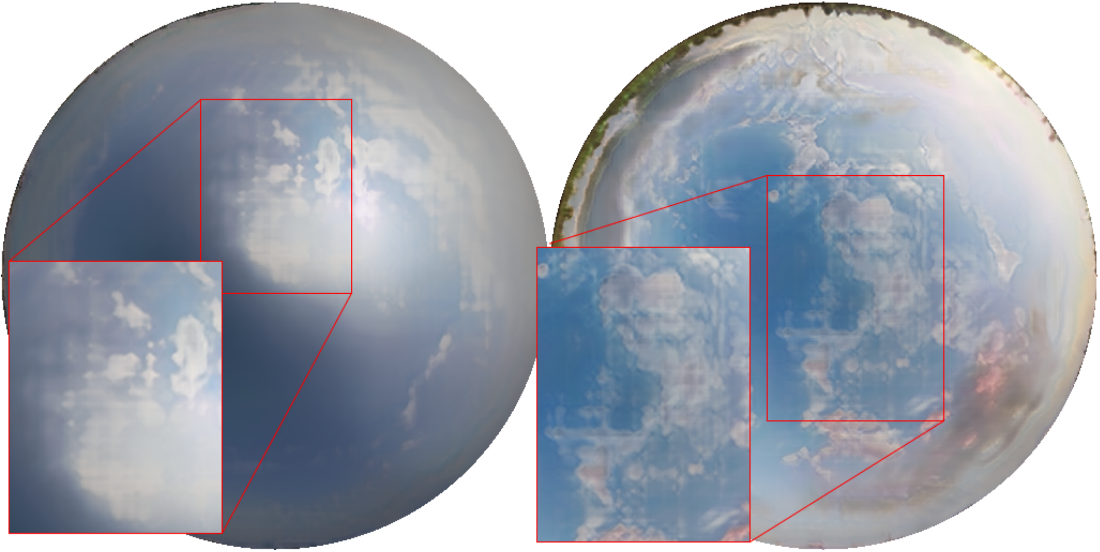


Figure 5.15: An illustration of the *checkerboard artifact* in the fisheye network. These artifacts were visible in many generated images and were not rare.

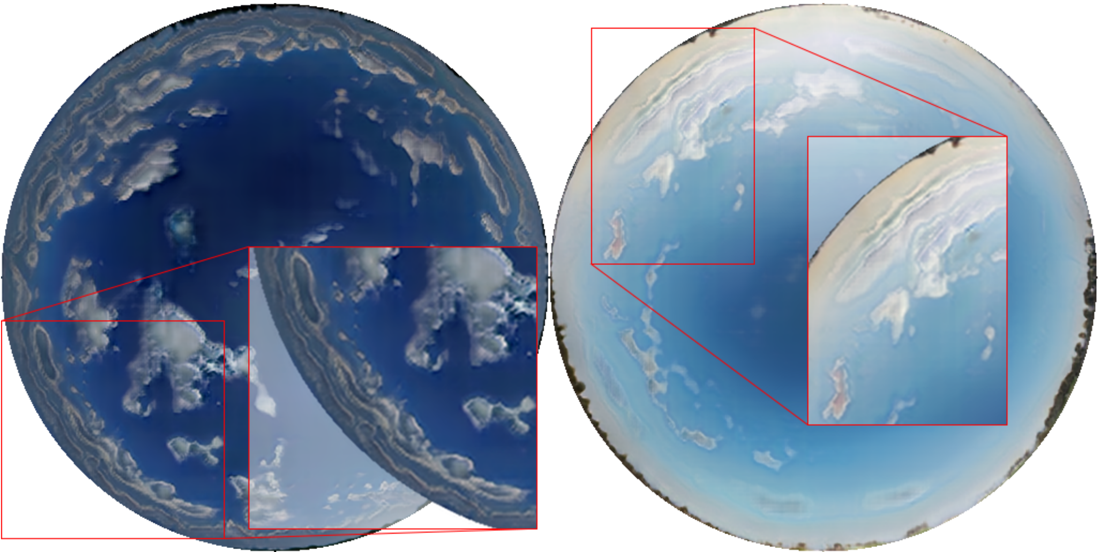


Figure 5.16: An illustration of the *oil stain-like artifact* in the fisheye network. These artifacts appear in almost every image, usually in the marginal region.

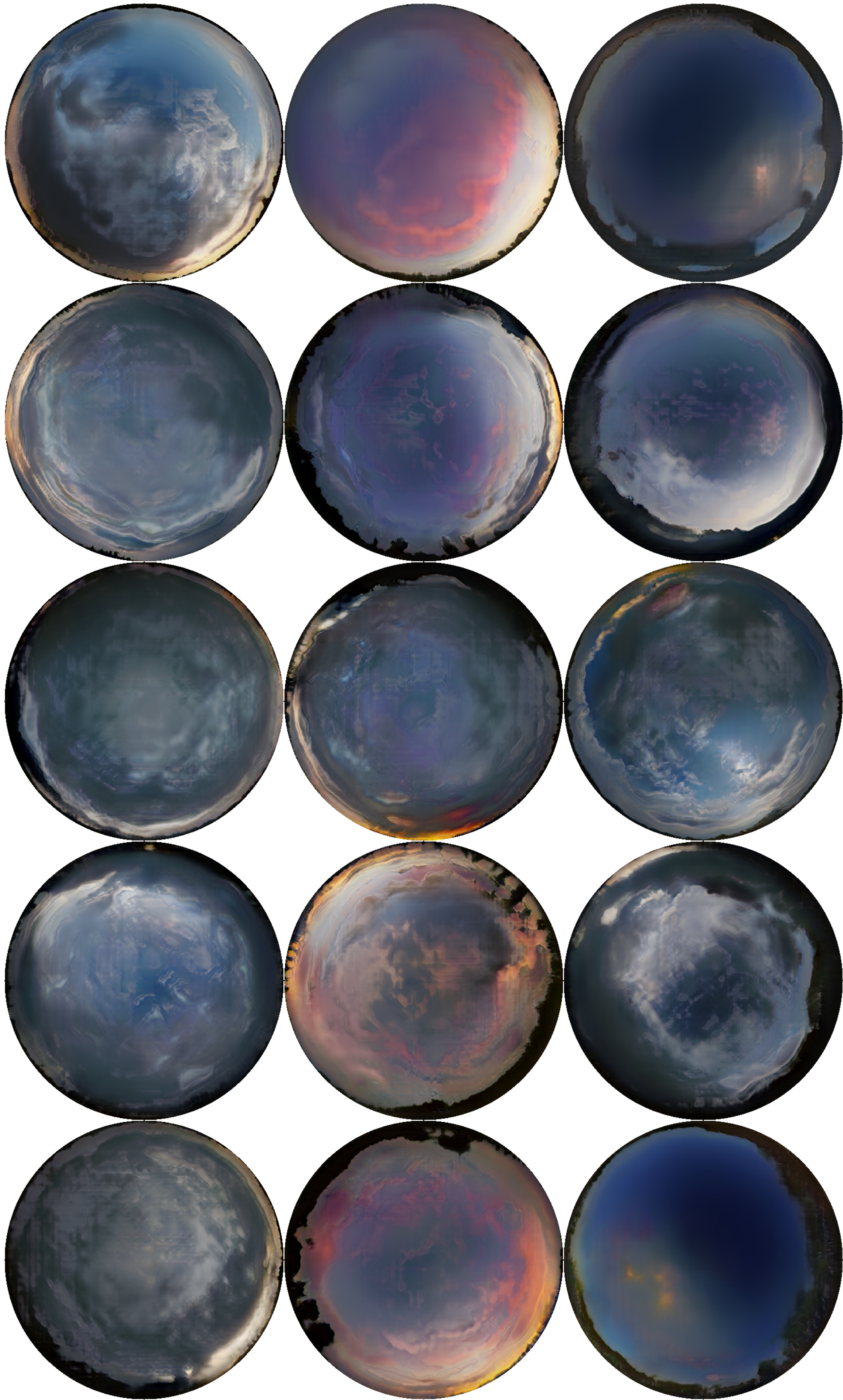


Figure 5.17: An sample of failure cases of the fisheye network. We subjectively counted 51 failures in 250 generated images. The left column are images we classified as having the wrong cloud formations, the middle column are images with incorrect lighting and the right column are images being a complete failure case.

**Interpolation experiment** We also designed a follow-up experiment utilizing the network we trained for the fisheye projection experiment. We wanted to test if the network is capable of producing a physically plausible transition between two skydome images. Exploring this area is important, since if we were able to generate a realistic looking transition, we could animate the environment maps for the purpose of CGI movies, or provide a sequence of environment maps for the purpose of rendering an architectural scene during different times of the day. We think, however, that generating realistically plausible skydome transitions would require modifying the network to include some information about time. This information is readily available from the EXIF data contained in the images.

By transition between two images we mean taking the latent representation (the latent vector $l \in Z$ which led to generating the image) of two images $l_s, l_e \in Z$ and generating $N$ latent vectors which are a linear interpolation of $l_s$ and $l_e$ as $l_i = t \cdot l_s + (1 - t) \cdot I_e$ where $t \in [0, 1]$. We then generate a new image for each such $t$. This gives us $N$ images which are part of an "animation" transforming the image represented by $l_s$ to the image represented by $l_e$. For this experiment, we used the network we trained for the last experiment, loaded the snapshot and instructed it to generate 24 images (plus the two original ones).

The result of this experiment can be seen in Figure 5.18, note that the number of images was reduced from 24 to 10 to better illustrate the transition. As we have already mentioned, we also include the equirectangular remapping of these images in Figure 5.19.

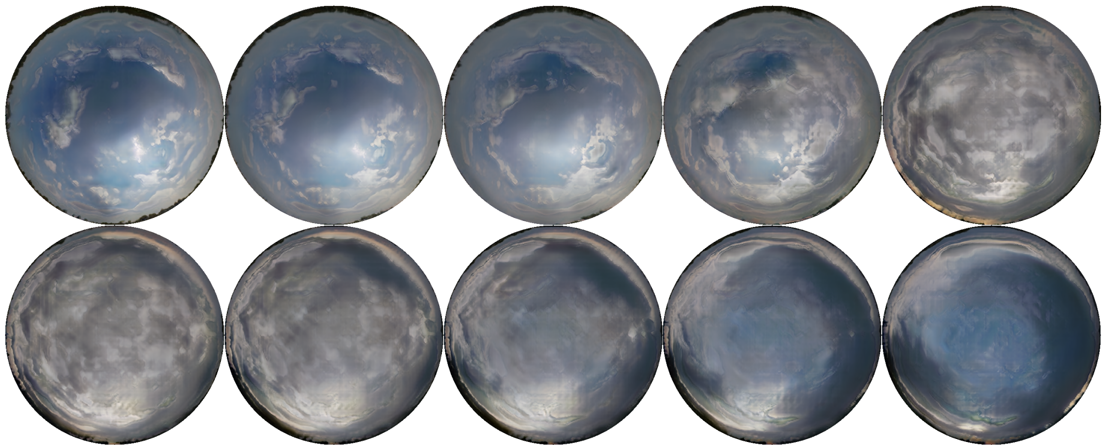


Figure 5.18: Performing linear interpolation between the latent vector of the first image (leftmost, first row) and the latent vector of the last image (rightmost, second row).

While not all of the images in the transition look physically realistic, most of them look plausible. This result may suggest that the network could be able to generate a plausible looking transition given a bigger dataset and longer training time. If we were able to gather enough timelapse shots with full upper hemisphere skydome, we might be able to train a network generating custom timelapses of the sky. We have already explained that this would be immensely helpful in the architecture visualization branch of *CGI*.

Figure 5.19: A sequence interpolating between the top and the bottom image generated by the fisheye network. The generated outcome was remapped into equirectangular projection. This figure contains the same images as Figure 5.18.

**New layers benchmark** In Chapter 4 we have described two new layers we have implemented to see if they will improve the quality of the images, increase the speed of the network or have any other beneficial effects. These two layers were the *fisheye masking layer* and the *resize-convolution layer*. In this experiment we have completed three identical training runs in which we trained a network without these improvements, a network with only the fisheye masking layer and a network with both the fisheye masking layer and the resize-convolution layer. The question we seek to answer is whether any of these new layers had any impact on the network training process or the results.

We set up the training scheme in the following manner for all three training runs. We used the 650 image dataset, with a target resolution of $256 \times 256$, tone mapped into low dynamic range. We utilized a new preset modified by us to only scale up to 256 resolution and not further, which we called `v3-2gpu`. We also created a variant of the preset for just 1 GPU card, which we will utilize for later training, called `v3-1gpu`. The reader can find the preset in the digital attachments in Attachment A.1. We stopped each network after it has seen $9040k$ real images, which was after a roughly 4 days of training time (the networks differed by a few hours). This training time consisted of approximately 2 days of the network growing and 2 days spent on training the full size resolution.

The results, as produced by the network, are shown in Figure 5.21. We also remapped the results back into equirectangular projection as can be seen in Figure 5.20. We will now discuss the differences.

Figure 5.20: Comparison of three the versions of the network we ran this test for, remapped to equirectangular projection. This figure contains the same images as Figure 5.21.

We think that between the three networks there are not any significant changes in image quality. We see the same artifacts we have already discussed (both the checkerboarding and oil stain-like artifacts) in every version of the network. This leads us to believe that either this is a hard problem for the network to solve or our dataset is not big enough for the network to learn all the small nuances of clouds and their structure.

Comparing the three networks, we conclude that our modifications have not improved the output quality, though the fisheye masking layer does have the additional benefit of giving the user control over the shape of the generated fisheye, as we have discussed in Section 4.2.1. Improving the quality of the generated images is important to the goal of our thesis, as such we will discuss our other ideas of improving the image quality in Chapter 6.

Figure 5.21: Comparison of the three versions of the network we ran this test for. Leftmost column is the control run – no fisheye masking, no resize-convolution. Middle column is a network including the fisheye masking layer. Rightmost column is the network including both fisheye masking as well as a resize-convolution layer.

### 5.2.2 High dynamic range

Once we conducted these experiments we felt we had a solid understanding of the network as well as managed to produce images of reasonable quality. This made us shift our efforts back to our original goal of generating high dynamic range images, and as a result converting the neural network of Karras et al. [2017] to generate HDR images. We have already covered the technical details of the conversion in Section 4.2.3 and here we present a few experiments we performed to evaluate the image quality of the HDR network.

Before we start describing our experiments we want to describe our dataset in greater detail. In Attachment A.2 we list all the data we downloaded from the internet. We mention that we had some proprietary data, which we cannot disclose. This data counted roughly 40 images which were only low dynamic range. This means that while our LDR dataset counted roughly 650 images, only 610 of those were HDR. We also want to mention that the dataset of Zaal is captured with the *unclipped dynamic range* of the Sun, whilst the rest of the dataset has clipped dynamic range. We have discussed the clipping phenomenon in the few paragraphs about clipping.

We now present several experiments we performed on the high dynamic version of the network, including a failed training run which resulted in a *mode collapse*.

**Training without the logarithmic remapping** In Section 4.2.3 we have discussed our logarithmic remapping of the dataset for the purpose of the network training. The question we wanted to answer with this experiment was whether this remapping is needed at all and how much it improves the quality when introduced.

To test this we utilized the full dataset as we have described above, consisting of around 610 images. We ran the network on 2 GPU cards, with the custom `v3-2gpu` preset we have mentioned in the last section. We targeted the resolution of $128 \times 128$ and trained on this setup for 2 days. We chose this smaller resolution because we consider HDR image generation a harder task than LDR and wanted to reduce the difficulty somewhat.

In Figure 5.22 we present the results of the network after 2 days of training. Since the network produces a high dynamic range image we cannot directly represent the high dynamic range in this document. Because of this, we only include the images sampled at one exposure level, namely the unadjusted exposure 0 EV.

This result may suggest that the network is not capable of learning a larger dataset without adjusting the dynamic range. The collapse did not happen "suddenly" (meaning over the course of the last few training epochs) but the network was experiencing atypical artifacts and oscillations throughout the whole training process. By oscillations we mean the phenomenon where the network eliminates some unwanted artifact in one epoch and re-introduces it back in the few following epochs. We include the whole training run in the digital Attachments A.1 for the curious reader.

With this training run failing to produce any tangible results we decided to implement the logarithmic shift described in Section 4.2.3, which some architectures utilize [Eilertsen et al., 2017].

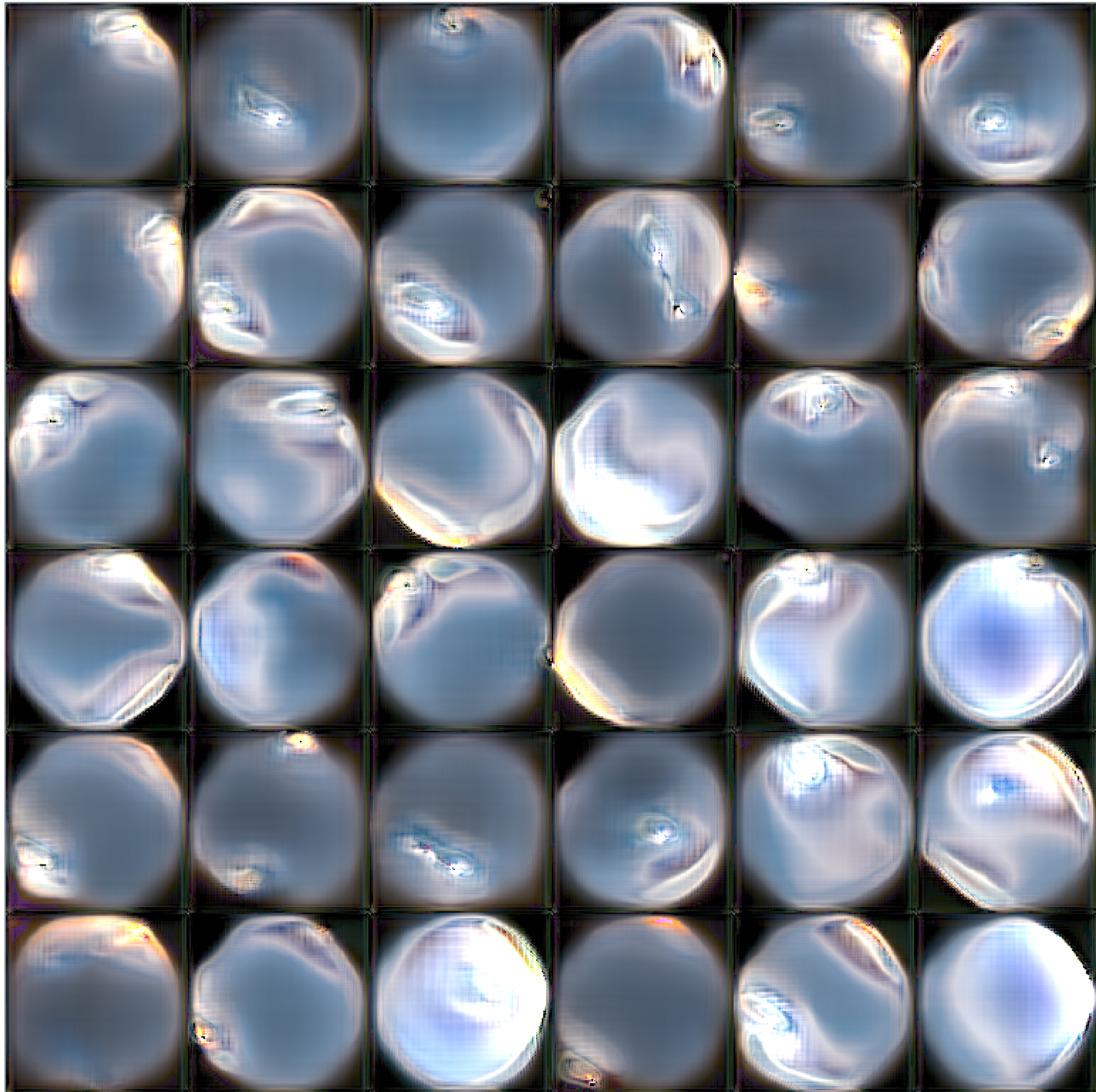


Figure 5.22: The results of a failed training run of a high dynamic range network.

**Training with logarithmic remapping** As a result of our last experiment we implemented the logarithmic shift described in Section 4.2.3 and wanted to find out if this improvement will help the network stabilize and learn to generate images in a similar quality we have achieved on the low dynamic range in Section 5.2.1.

We again use our full dataset of 610 images, train the altered network on 2 GPU cards with the `v3-2gpu` preset. We again chose the target resolution of $128 \times 128$ to keep the experiments consistent. The logarithmic adjustment constant is set to $c = 0.0001$, which replicates the $1/255$ setting of Eilertsen et al. [2017]. Unlike the previous experiment which we only let train for 2 days because it very clearly diverged, we trained this neural network for 5 days and 3 hours, in which the (discriminator) network saw $11827k$ real images. We illustrate the training data in Figure 5.23, along with RGB histograms.

We present a bigger sample of the generated data, without the RGB histograms in Figure 5.24, along with a few generated images with RGB histograms in Figure 5.25. The histograms also contain information about the maximum value in the image (across all channels) and the minimum *non-zero* value across all channels. The true minimum value is clearly 0, since the pixels outside of the fisheye unit circle are set to zero.

Unlike the previous experiment, after converting the images into the logarithm domain the network was able to train in a stable manner and produce images which, to our eye, look of similar image quality as the our LDR results from the previous section. From the histograms we also see that the network is able to learn the high dynamic range (note the minimum and maximum values as we mentioned in the previous paragraph).

These results suggest that the high dynamic range network is able to produce similarly looking images as the low dynamic range version of the network. There are several questions this experiment opens up, namely what should we set the constant $c$ to for best performance and whether the network is able to accurately reproduce the luminance data. We will attempt to answer these questions in the next experiments.

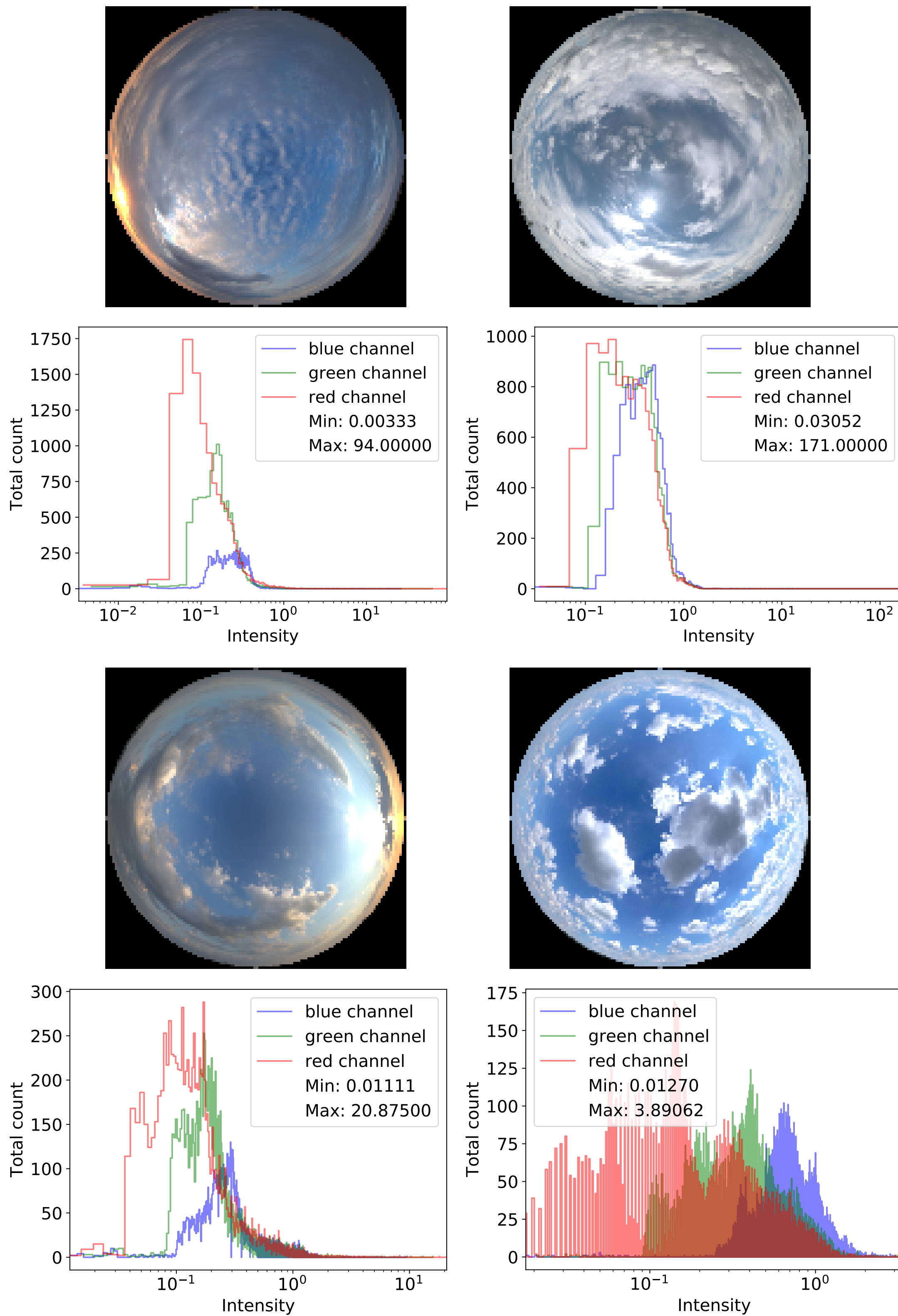



Figure 5.23: A sample of the real data input into the HDR network. Each image also has a RGB histogram included, which illustrates the red, green and blue channels as well as the smallest non-zero and the maximum channel value.

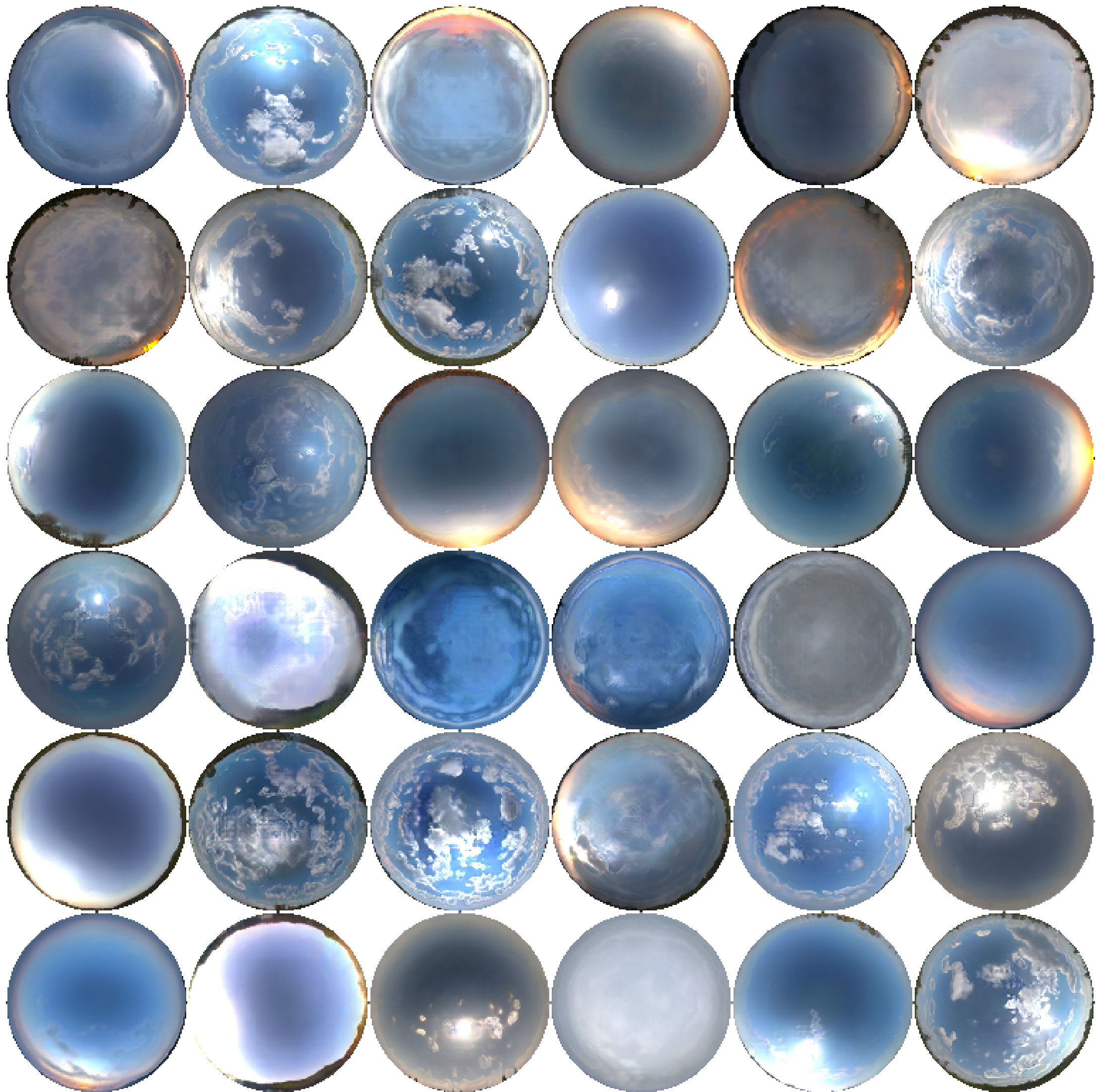


Figure 5.24: A bigger sample of data generated by the network. Since the data is HDR and we cannot display it in this work, we only include one snapshot at 0 EV and refer the reader to the digital attachments to preview the HDR image personally.

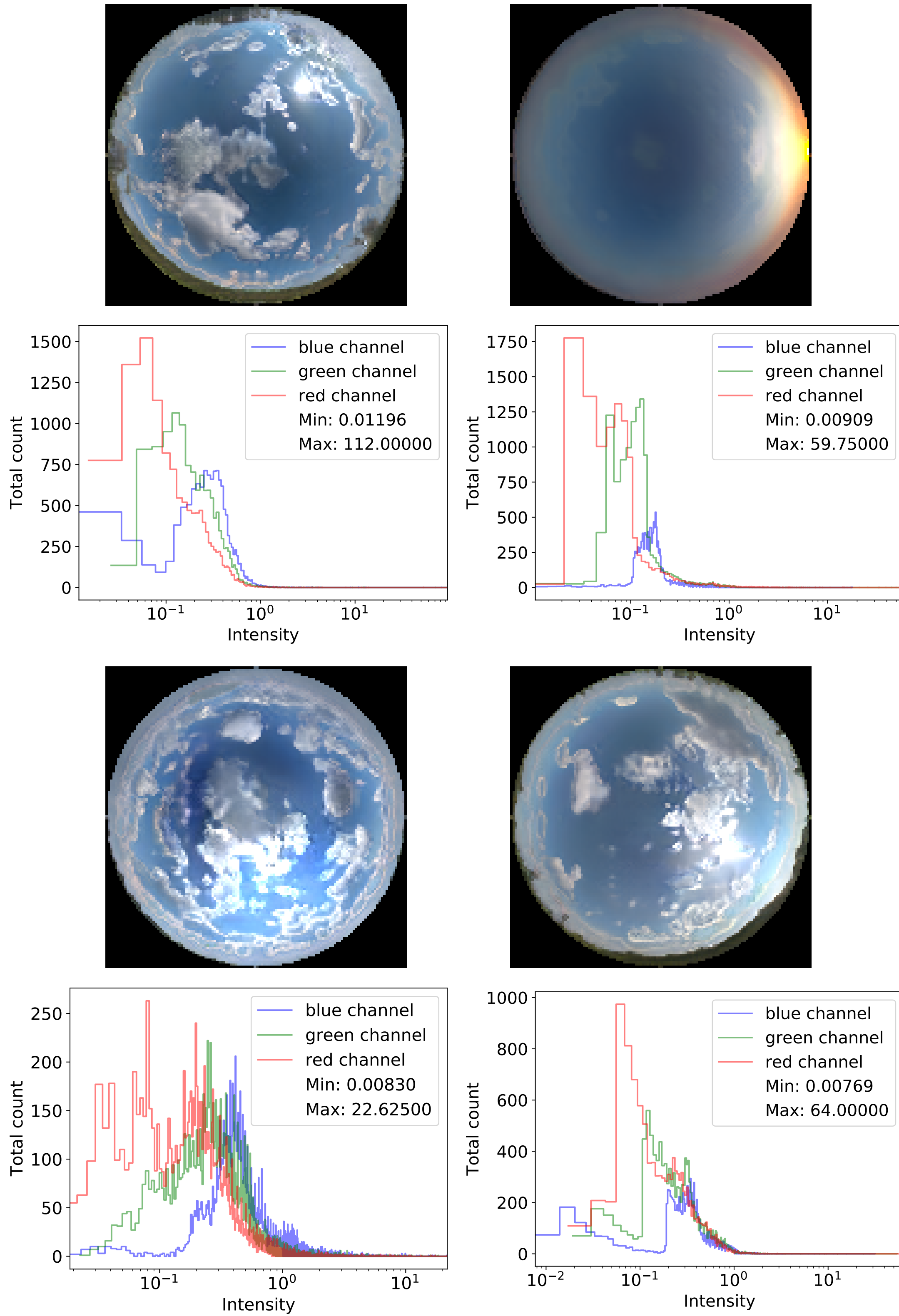



Figure 5.25: A sample of the data generated by the network with the accompanying RGB histograms. More generated images can be see in Figure 5.24.

**Overfitting benchmark** Having modified the existing network, we wanted to ensure that the architecture of the network is capable of reproducing the dynamic range and high image quality in the dataset images and is not failing systematically because of a flaw in its design. We performed two simple experiments which should provide some evidence for this.

We decided to train the network on a single image as a dataset, and compare the image reproduced and the training image to see if there are some errors which would point to more than a noisy reproduction. We used the `v3-1gpu` preset and trained the network on 1 GPU card for 4 days. The training image can be seen in Figure 5.26, along with a histogram for the image. We chose the resolution of $256 \times 256$, since the last network performed well on a large dataset with a smaller resolution. We also used the logarithmic adjustment with $c = 0.0001$, since the image quality of the last network was comparable to our low dynamic range results. We also left the mirroring augmentation of the dataset, which the network performs during its runtime, enabled, to provide a slight bit of variance.

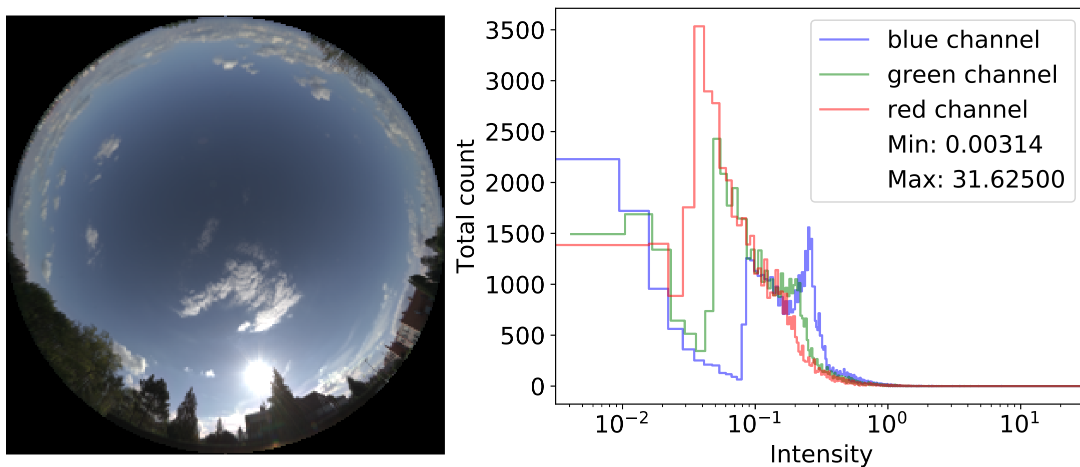



Figure 5.26: The HDR over-fitting experiment dataset image, with the RGB histogram.

Figure 5.27 showcases one of the generated images, as well as its histogram. We do not include more generated images in the text, since all of them are very similar, save for some very insignificant random noise. We include other generated images in Attachment A.1.

As we can see, the figures 5.26 and 5.27 are very similar and pretty much identical to the human eye. Thanks to the maximum channel value reading in the histograms we see that the difference in the highest value in the difference image is 1.125.

We think that this experiment proves that, save for some insignificant noise, the network is able to reproduce an image with high accuracy, which answers the question this experiment has been designed for.

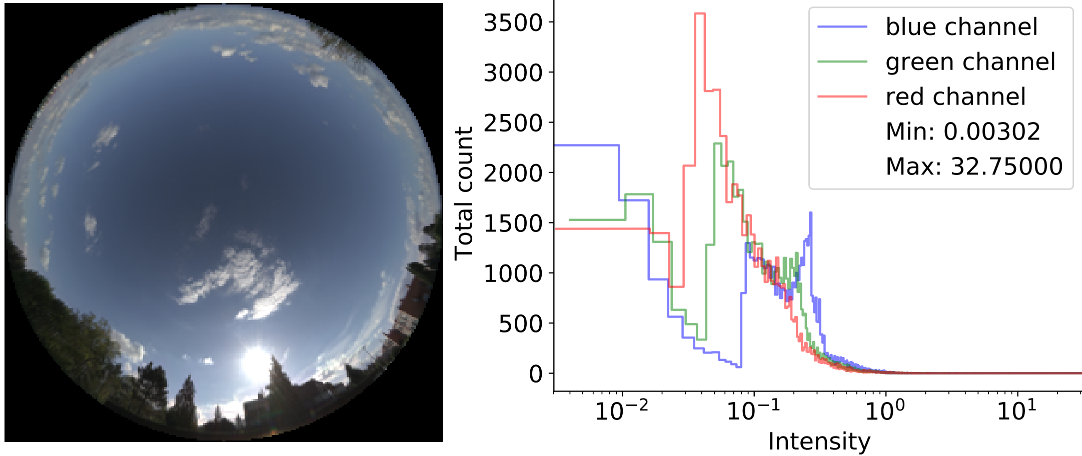



Figure 5.27: A sample produced by the HDR over-fitting experiment, with the RGB histogram. Every other image produced was the same, except for random noise.

**Logarithmic shift benchmark** We have already mentioned that we were not sure about the exact setting of the logarithmic parameter $c$, which we defined in Section 4.2.3. We already know that $c = 0.0001$ generates reasonable output from our previous experiments, which is curious, because this setting compresses the dynamic range high in the spectrum (the brightness of the Sun and the sunlit clouds), while *increasing* the dynamic range of the darker parts of the image (the shadows, darker clouds). The fact that this is what helps the network learn is, in our opinion, counter-intuitive since neural networks usually benefit from having their data as normalized as possible [Shanker et al., 1996], [Sola and Sevilla, 1997], [Jayalakshmi and A, 2011]. The standard method of normalization is recalculating the data in such a way that the average of the feature over the whole dataset is roughly zero [LeCun et al., 1998]. This includes, for example, the original network of Karras et al. [2017], which remaps the RGB images from $[0, 255]$ to $[-1, 1]$.

On the other hand, the setting $c = 1$ would shift the whole data into positive range and compress the higher dynamic ranges significantly, resulting in a more normalized data with a smaller variance, which should help the network train. However, the fact that the setting shifts the whole data into the positive range breaks the "rule of thumb" about having a zero average. Since both of these settings seem to have a positive as well as a negative side, we designed an experiment to directly compare the image quality between these two shifting constants.

To briefly recap our **hypothesized** pros and cons of each setting — $c = 0.0001$ compresses the higher end (a pro), increases the range of the lower end (a con) and shifts the data into the negative range (a pro) by computing the natural logarithm of numbers lower than 1. On the other hand, the setting $c = 1$ shifts the data entirely into the positive range (a con) but also lowers the range of the high end of the data significantly (a pro).

For the training of this experiment, we finally acquired results from the dataset acquisition pipeline we described in Chapter 3 and utilized some of the images shot, which we showcased in Section 5.1. This allowed us to add around 561

photos, effectively doubling our current dataset, which resulted in a dataset of 12311 images once augmented. We trained on the target resolution of $256 \times 256$, on 1 GPU card with the preset `v3-1gpu`. We present some hand-picked (to show the variance of the dataset) samples from the real dataset in Figure 5.28.

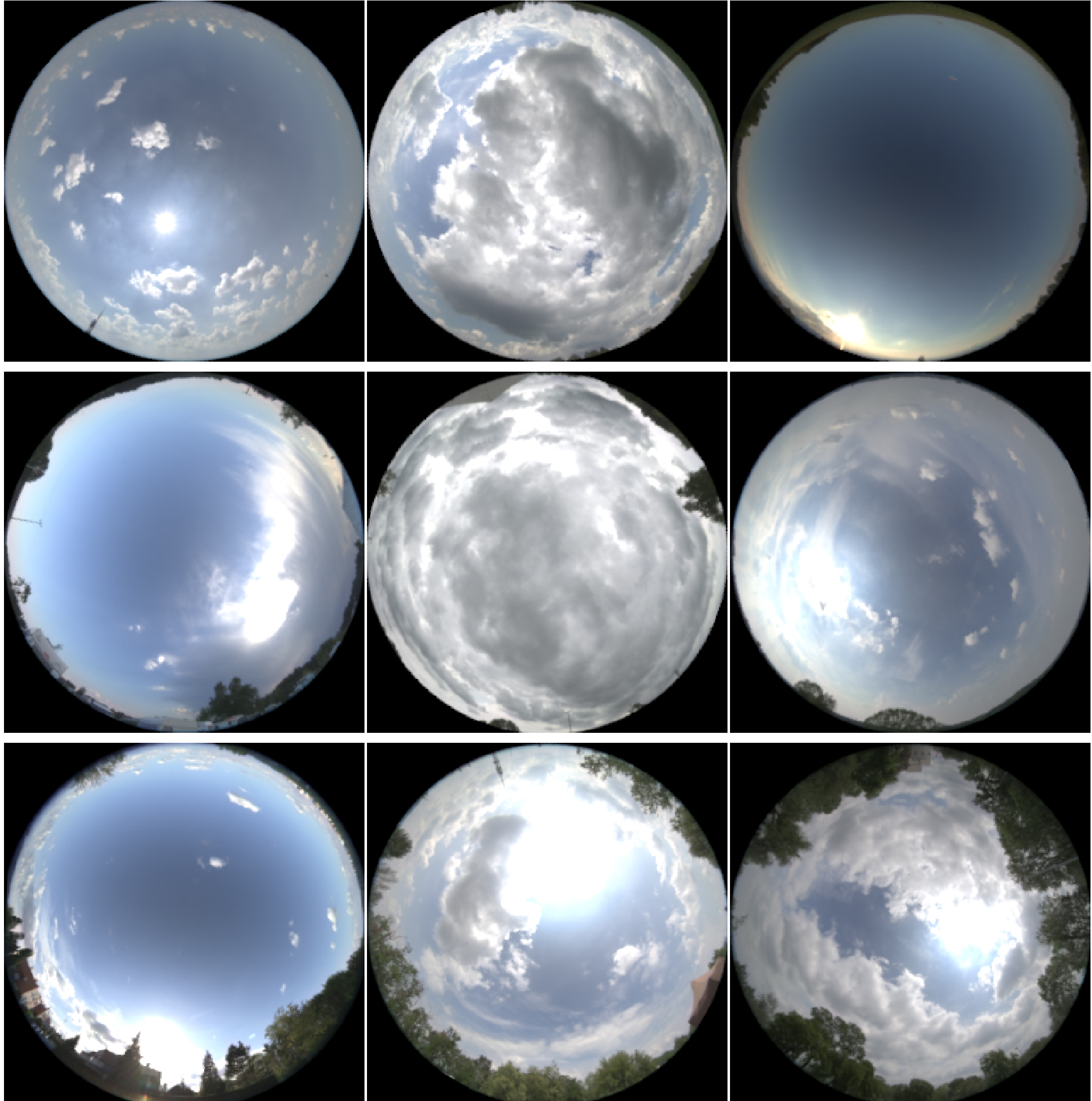
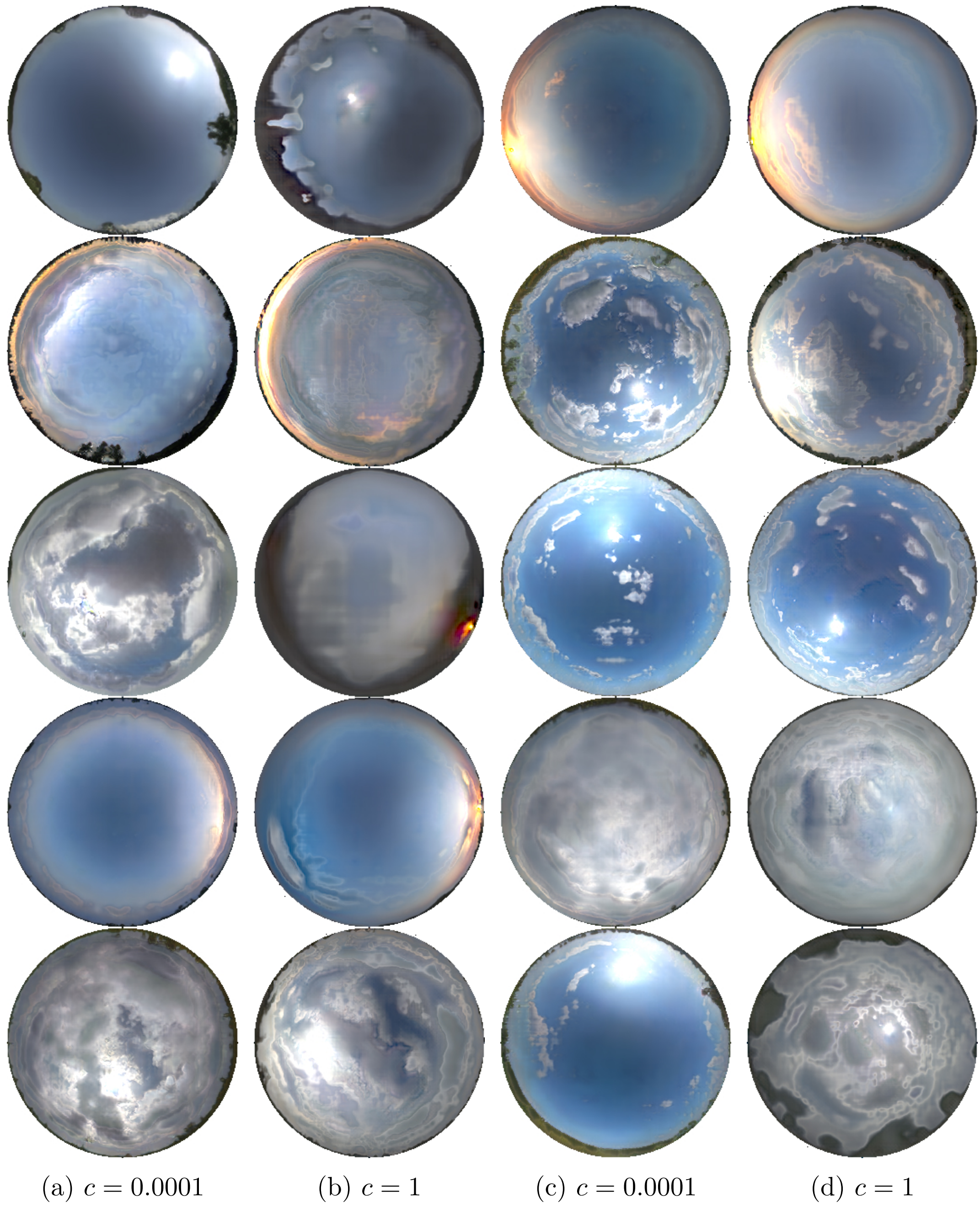


Figure 5.28: A sample from our new bigger dataset. This sample only includes new photos, which we shot ourselves.

We attempt to illustrate the differences on several pairs of photos which were manually picked based on their visual similarity in Figure 5.29. We also showcase that both types of artifacts we mentioned in the low dynamic range experiments still prevail in both of these settings, as can be seen in Figure 5.30 and Figure 5.31. Lastly, in Figure 5.32 we provide graphs of the scores given by the discriminator to the real and fake images during training. The *real score* is affected by the loss function and kept to be close to 0. Both scores indicate how sure the discriminator is about classifying the image. The higher the real score is, the more sure the discriminator is about classifying the real images. The lower the fake score is, the more sure the discriminator is about classifying fake images. While these graphs can often be misleading, we think that in this case they are quite interesting.

We have generated 300 images from each of the networks, we include 50 of these images for each network in Attachment A.1 to allow the reader to get a more complete picture of why we came to the following conclusions. We will now discuss these results one by one. Figures 5.30 and 5.31 illustrate that the network still suffers from both of the artifacts we have already discussed at length. We also notice the fact that the network trained with $c = 1$ generates a lot more unrealistic images, as can be seen in Figure 5.29 – distorting the ground very severely. The network trained with $c = 0.0001$ does not generate these images at all. The image quality of images which look realistic is comparable but we think that the network trained with $c = 0.0001$ does produce better quality overall. The fake scores seen in Figure 5.32 also suggest that the setting of $c = 1$ makes it very hard for the discriminator to classify the fake images, while the discriminator is perfectly able to classify the fake images with $c = 0.0001$. We think that this is the primary reason for the discrepancy in the image quality between the two networks.

In conclusion, this experiment indicates that setting $c = 0.0001$ seems to produce significantly more plausible looking images without severe ground artifacts. The image quality is marginally better as well. In this thesis, we use the setting $c = 0.0001$ for all future experiments as a result.



(a) $c = 0.0001$ (b) $c = 1$ (c) $c = 0.0001$ (d) $c = 1$

Figure 5.29: A sample of images produced by the HDR logarithm shifting experiment. The columns are alternating between $c = 0.0001$ and $c = 1$, see the caption.

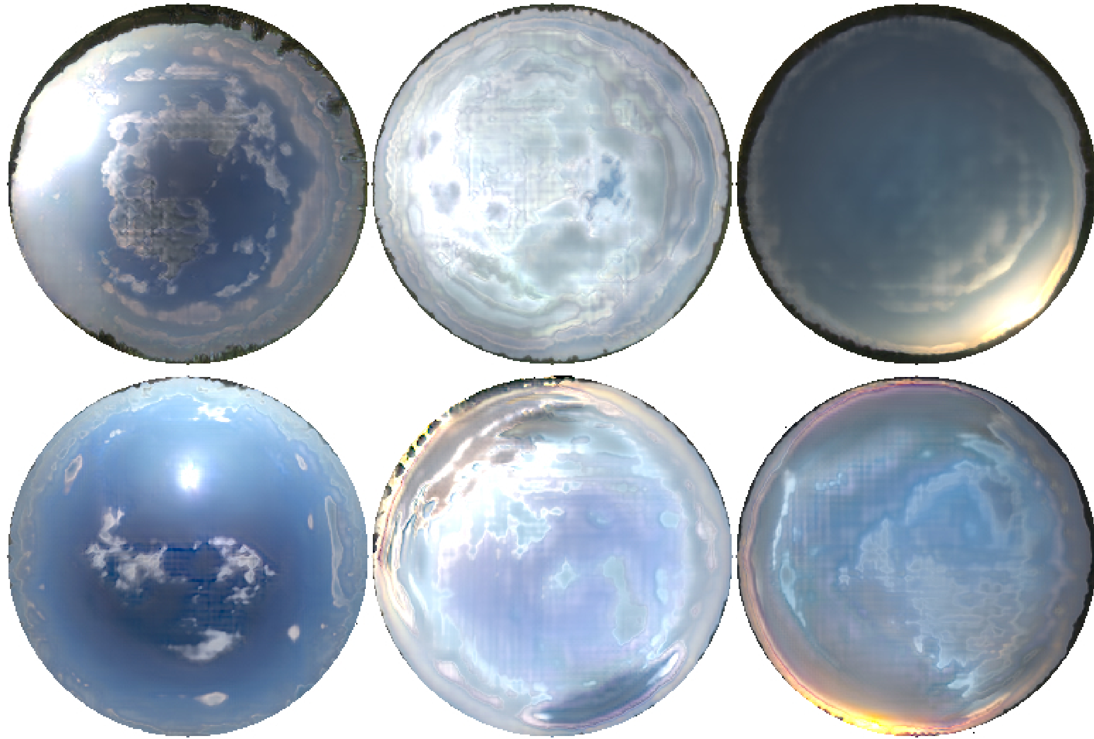


Figure 5.30: An example of the prevailing checkerboard artifact. The first row are images from the $c = 0.0001$ network. The second row contains images from the $c = 1$ network.

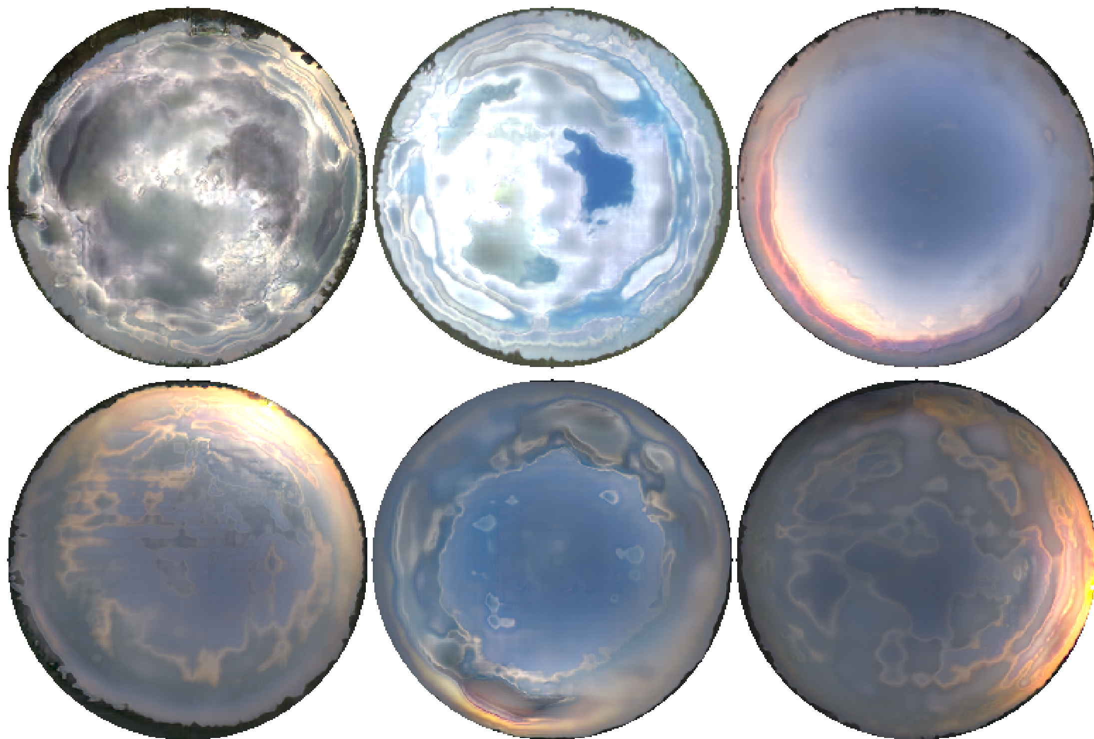


Figure 5.31: An example of the prevailing oil-like artifact. The first row are images from the $c = 0.0001$ network. The second row contains images from the $c = 1$ network.

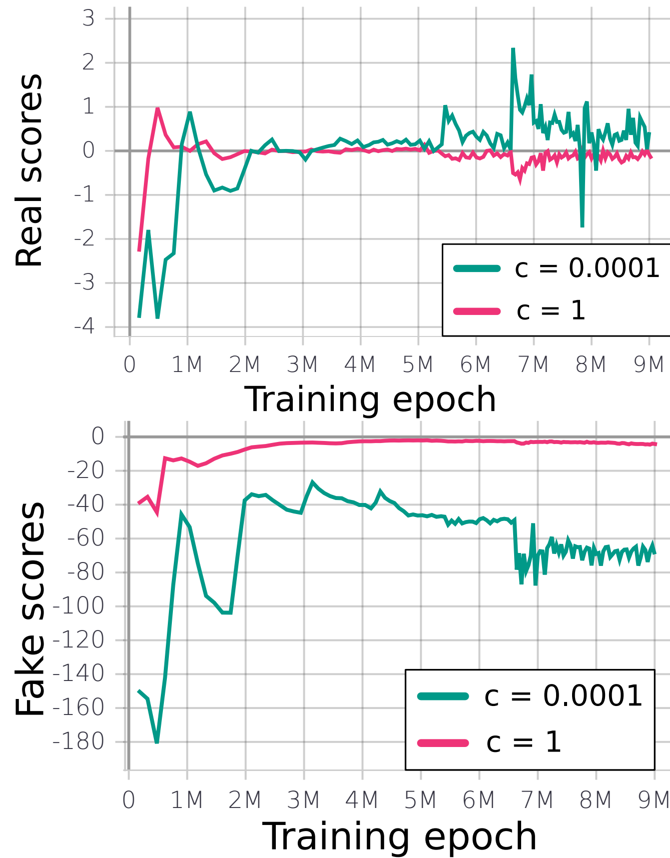



Figure 5.32: Real and fake scores for the networks in the logarithm benchmark experiment. The *real score* is affected by the loss function and kept to be close to 0. Both scores indicate how sure the discriminator is about classifying the image. The higher the real score is, the more sure the discriminator is about classifying the real images. The lower the fake score is, the more sure the discriminator is about classifying fake images.

**Training on a single timelapse** In the previous experiments we saw how the size of the dataset affects the quality of the images and the generalization abilities of the network. If the network is only given a small dataset to learn, it overfits really well and the quality of the images is high, without any visible artifacts. If, on the other hand, the network is tasked with learning a big and diverse dataset, it is able to generalize but it is not able to produce high quality results and produces artifacts in the images (mainly the *checkerboarding* and *oil-stain* artifacts). In this experiment we wanted to get a better idea of where this line dividing these two behaviors lies by training the network on a single timelapse shot during 2 hours and 35 minutes from a single location, taking one picture every 30 seconds. We wanted to see if the network could learn to fill in the blanks in between the 30 second pauses and learn to interpolate the sky correctly, while maintaining the high quality we saw from the overfitting tests.

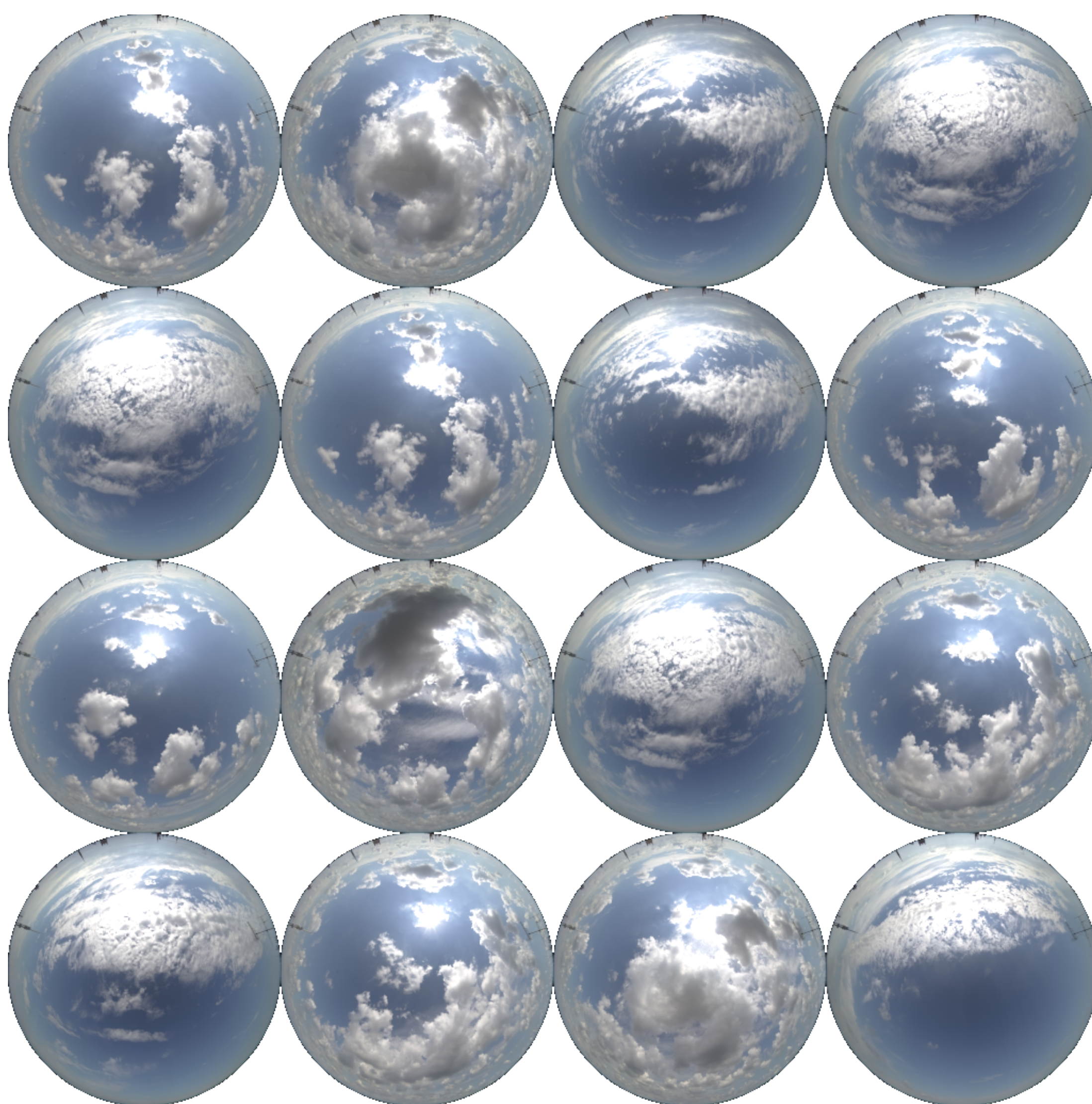


Figure 5.33: A sample of real data our network trained on during the single timelapse experiment.

We trained the network on 311 images, with a target resolution of $256 \times 256$. A few images from the training dataset can be seen in Figure 5.33. We used the logarithm shift, with $c = 0.0001$ and used a `v4-1gpu` preset, which was very

similar to the already mentioned `v3-1gpu` setting, different only in increased batch sizes (which increases the training speed marginally). As the name suggests, we trained this network on a single GPU card, this time training for an extensive 9 days during which the network saw $12M$ real images. We also note that we *did not augment* this data in any way. After the network finished training, we also created a series of images interpolating between two latent vectors, to see if the network is able to provide a plausible interpolation of the timelapse.

We present a sample of the images generated by the trained network in Figure 5.34. Figures 5.35 and 5.36 showcase the interpolation and its shortcomings, which we will discuss in the following paragraphs.

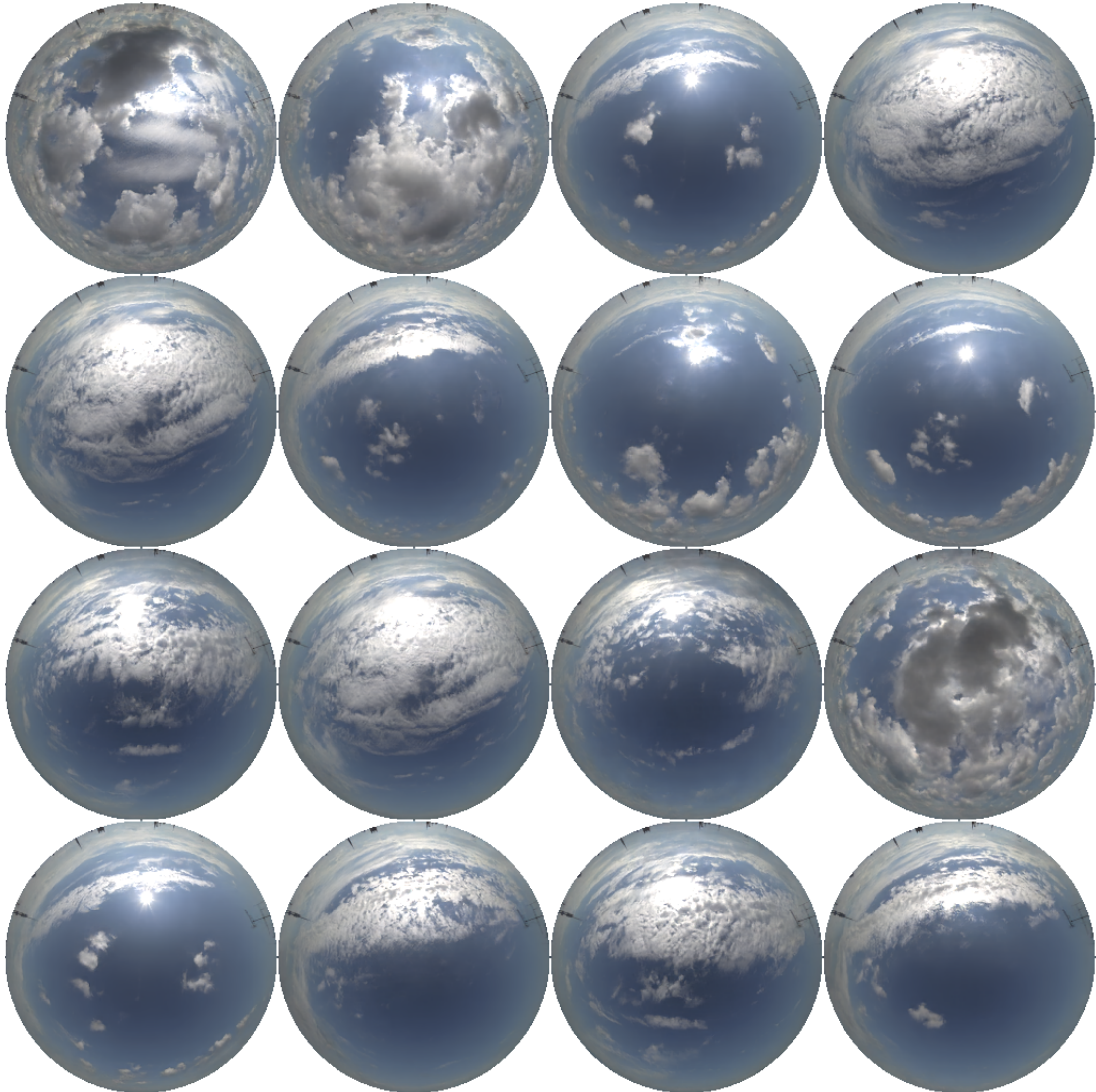


Figure 5.34: A sample of data our network generated at the end of training during the single timelapse experiment.

Looking at the generated data we see that the quality is very close to the real data and there are no visible artifacts of any kind in the images. This leads us to believe that the network is still overfitting on the dataset. This hypothesis is further supported by the interpolation result as seen in Figure 5.36 where the network fails to correctly move the clouds and Sun along the sky and at some point (from image 3 to image 4 in the figure) just blends the two images together. On

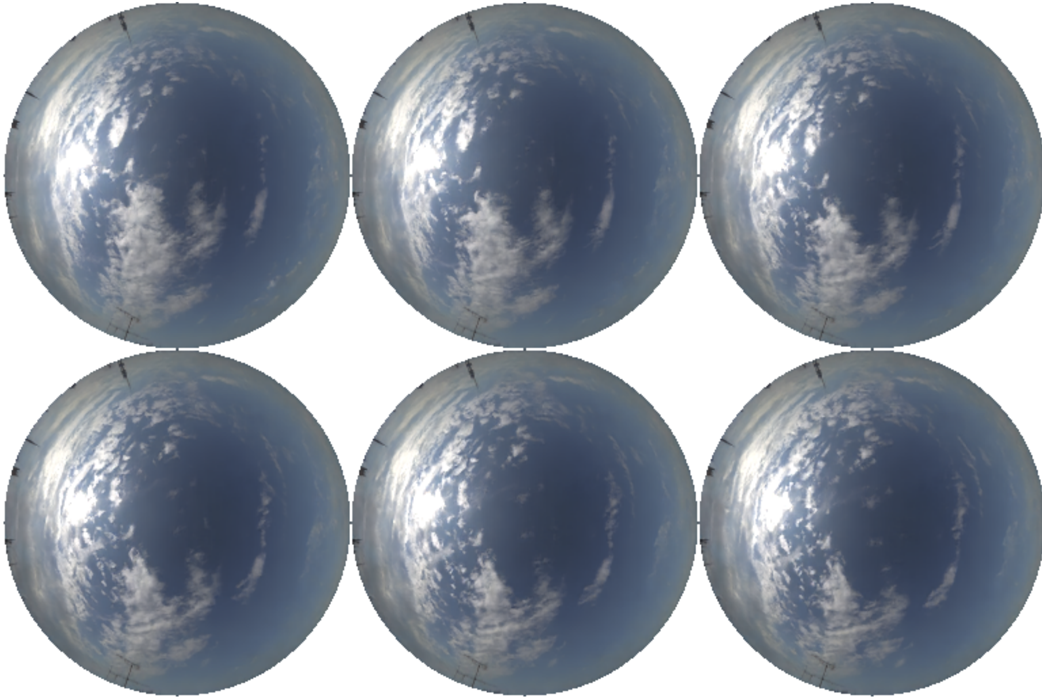


Figure 5.35: This figure illustrates a correct interpolation behavior. The images are not blended together and instead the network generates the correct cloud movement.

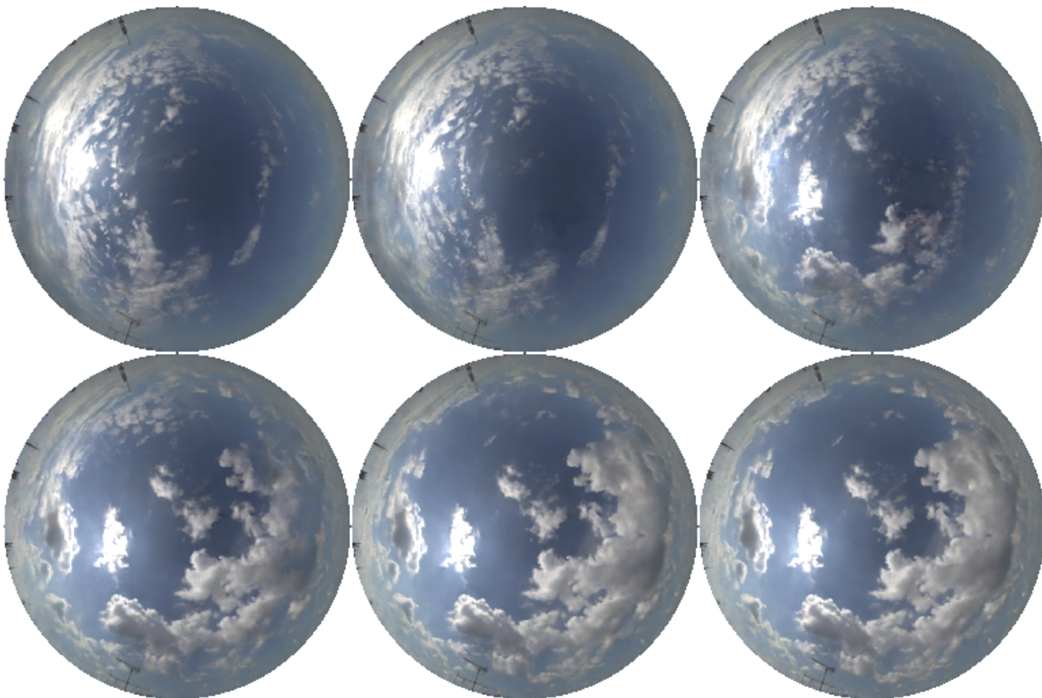


Figure 5.36: This figure illustrates an incorrect interpolation behavior. The third image on the first row is generated by blending the second and fourth image together and not producing the correct cloud movement. Notice the doubled Sun in the third image.

the other hand, in Figure 5.35 the progression of the clouds in the sky is correctly interpolated. These different interpolations may suggest that the network has a

hard time moving the Sun in the sky, whereas moving the clouds is easier. These two different interpolation outcomes also suggest that the network was not able to correctly notice the fact that the training dataset consists of consecutive images, since it interpolates with different precision between the photos. This fact could probably be remedied by inputting additional data into the network, such as the time of the day. We will discuss this extension further in Chapter 6.

To conclude this experiment, we found out that the network still overfits on a single timelapse of 311 images and will probably require a significantly bigger dataset to get the desired generalization behaviour. On the other hand, even with 311 different images the network is able to produce images of great quality, which is a good sign for further research in this area and prompted us to perform one final experiment, which follows.

**Training with only our acquired data** Since we started gathering the dataset, we always trained the network with both the data downloaded from the internet as well as the data we shot ourselves mixed together to form the largest dataset we could. However, the data downloaded from the internet is of a vastly different kind than the data we shot – while we shoot long timelapses, usually over 2 hours in duration, the data from the internet consists mainly of one single image shot at each location. This means that our shooting locations are present hundreds of times when compared to a single photo for each internet location. In this experiment, we tried training the network only on the data we shot, because we thought that this large discrepancy within the dataset might confuse the network and reduce the quality of the cloud coverage we get from the network.

Similar to the last experiment, we trained the network on a `v4-1gpu` preset for 9 days and 5 hours, to complete the training by showing the discriminator $12M$ real images. We also used the logarithmic shifting set to $c = 0.0001$ and trained on a target resolution of $256 \times 256$. We gathered all our photos, subsampled some longer shoots in order to make the dataset balanced in terms of shooting locations and augmented the data using the standard 36° rotation around the upward facing axis. This provided us with 19431 images after the augmentation, which we used to train the network. A sample from the images can be seen in Figure 5.37.

We present a sample from the generated images in Figure 5.38. We also showcase some comparison images, which were hand-picked to be similar to the images in the dataset and illustrate both the images and their RGB histograms in Figures 5.39 and 5.40.

As we can see the quality of the generated clouds seems to be of a higher quality than the previous big-dataset experiments, even though the quality-reducing artifacts are still present in a smaller capacity. We include the images showcasing the prevalent artifacts in Attachment A.1. The network also seems to be generating the correct shape of the histogram, but increasing the dynamic range more than the real images from the dataset – the generated images being anywhere from 50% to 100% higher in dynamic range.

We also performed a similar evaluation to the one we mentioned in the Fisheye projection experiment and generated 300 images from the network, which we then subjectively classified as either *plausible* or implausible because of either *wrong lighting*, *wrong cloud formations* or *a complete failure case*. This time we only

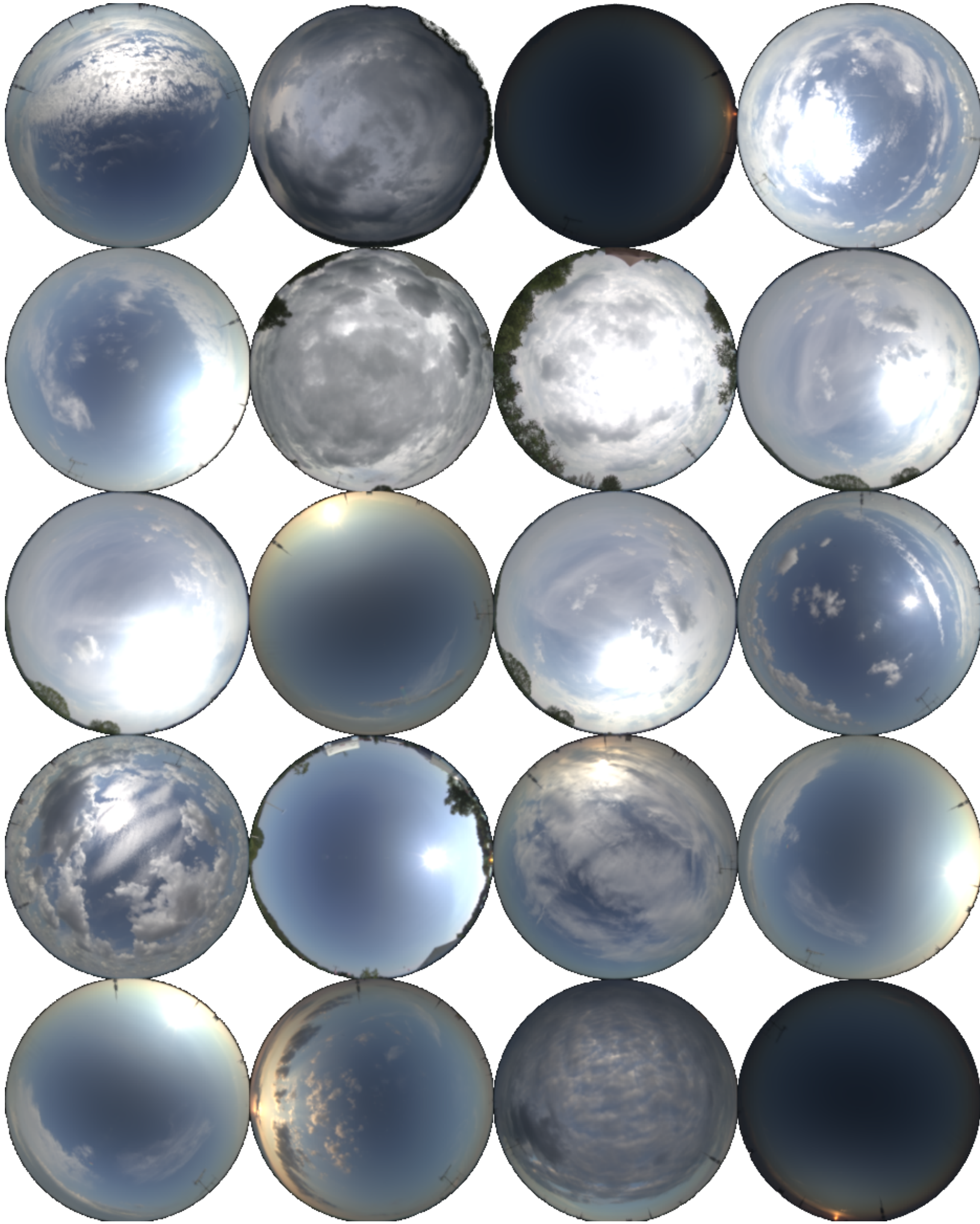


Figure 5.37: A sample of real data for the experiment utilizing only the data we shot during this thesis.

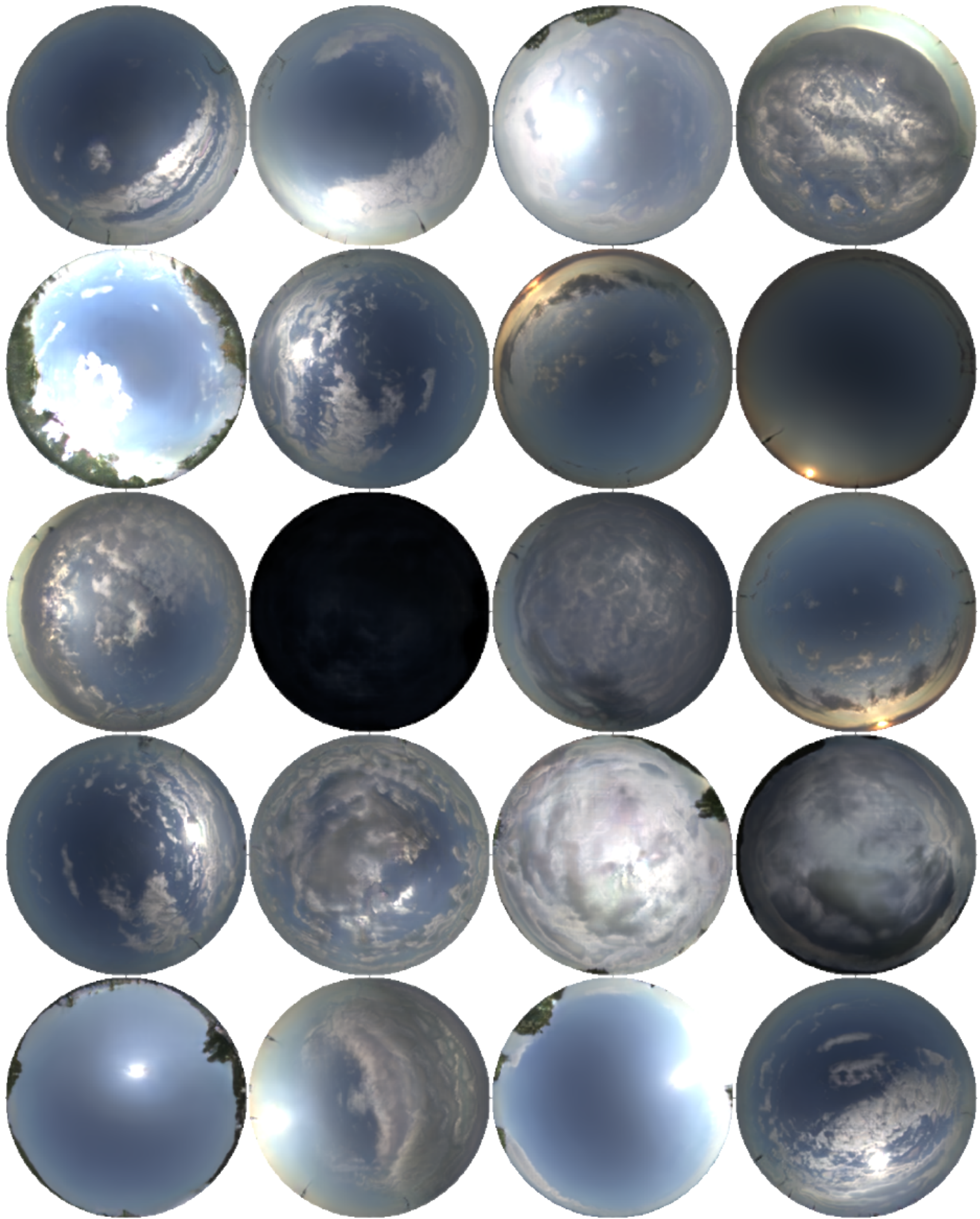


Figure 5.38: A sample of data generated by the network trained on only the data we shot during this thesis.

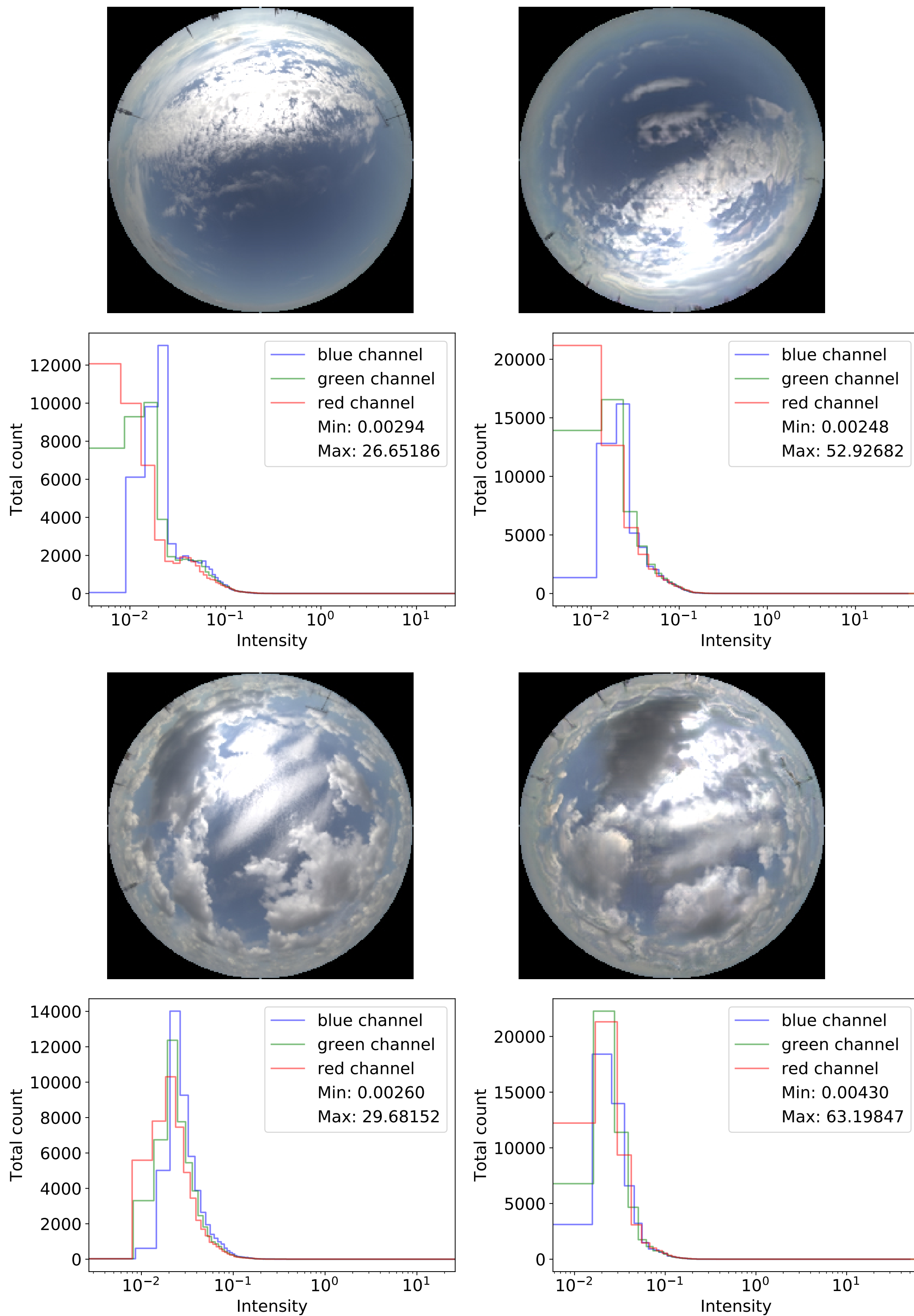



Figure 5.39: This figure shows a comparison of a real image (on the left) and a similar handpicked generated image (on the right) and their RGB histograms.

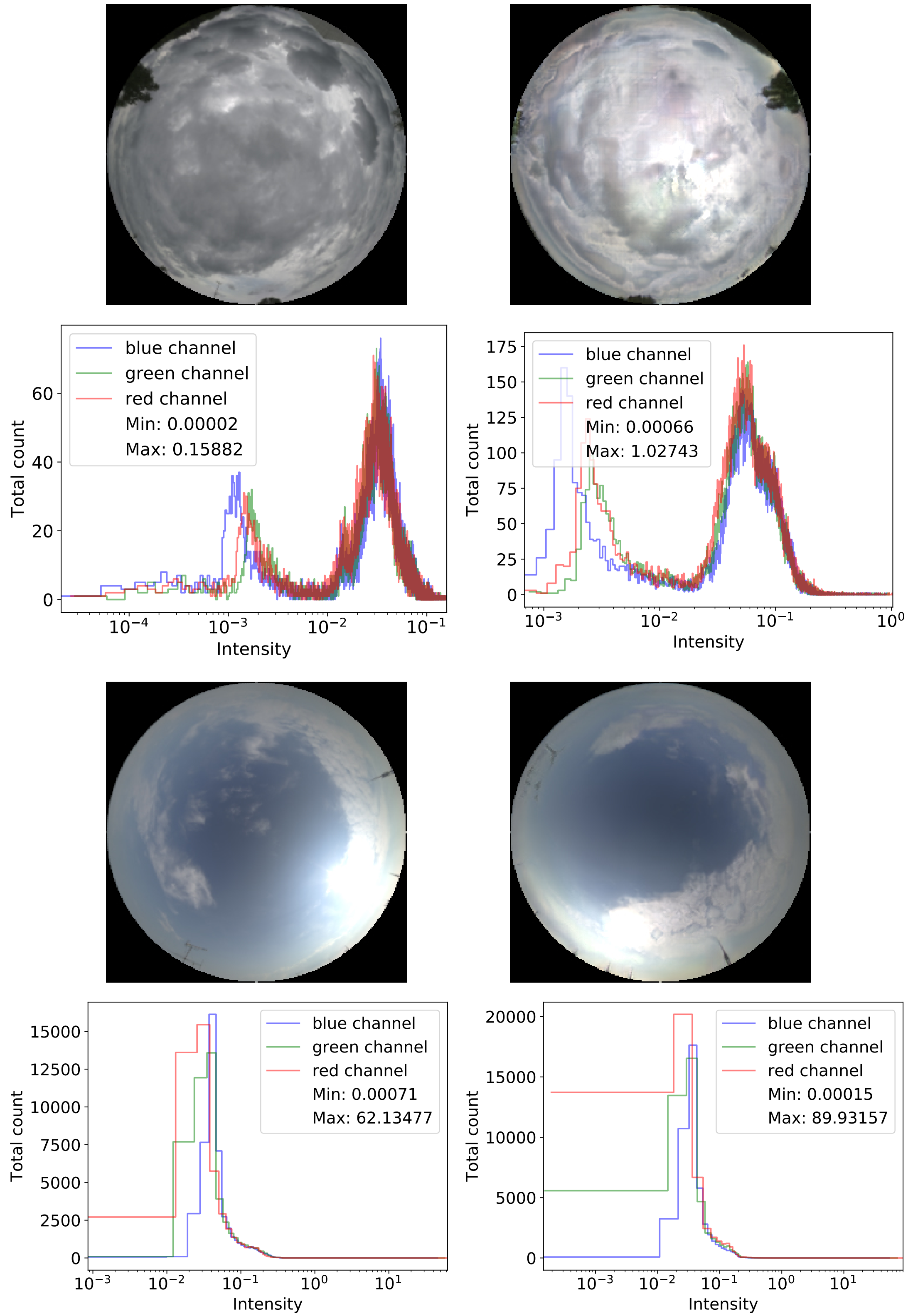



Figure 5.40: This figure shows a comparison of a real image (on the left) and a similar handpicked generated image (on the right) and their RGB histograms.

classified around 15 images as *wrong cloud formations* and we did not see any cases of *wrong lighting* or any *complete failure* cases. This might suggest that the network was getting confused by the single images in the dataset and now focuses a lot more on generating correct cloud formations.

In conclusion, this last experiment suggests that as we gather more data, the plausibility of generated data could increase without significantly decreasing the image quality. Once we shoot more of our data, we should see if removing the data from the internet improved the quality because the total image count got lowered or because the single images were confusing the network.

# 6. Future work

In this chapter we provide several ways to continue our work, explain the reasoning behind them, and why they would be useful. We put a lot of focus on this chapter since the goal of our work was to understand if and how well generative adversarial networks can be utilized to generate high dynamic range images. The future work that we propose should provide the reader with a good understanding of what we perceive has to be done before our research can be of use to the general public (for example to *CGI* artists).

## 6.1 Spherical convolutions

One of the problems which we noticed during our dataset projection research in Section 3.2 was the fact that we are unnecessarily stacking several projections on top of each other, which further increases the data distortion. When the photographs of the upper hemisphere of the skydome get taken, we are already projecting a locally flat object (the sky can be thought of as flat in the vicinity of the photographer) onto a hemisphere. We then additionally project the hemisphere onto a plane using, for example, a fisheye or equirectangular projection. Since each projection introduces additional error into the image, removing one of these projections might be beneficial.

The other problem with stacking the projections in this way is the fact that once the hemisphere gets projected onto a plane, the network does not receive the information about some parts of the image being distorted in a different way. This is true for both the equirectangular projection, where the distortion is minimal at the equator and increases with increasing latitude, as well as for the fisheye projections, where the distortion increases with increasing angle with the optical axis.

Using a traditional convolutional layer in this setting is not correct, since the weights in the convolutional filter are shared over the whole image and do not take into account the changing distortion inside the image.

A more valid convolutional layer for the fisheye projected image could include several different filters, arranged in concentric circles inside the image. This way, the areas of the image which are processed by a single filter would be subject to similar distortion. We attempt to illustrate this setup for the fisheye projection in Figure 6.1. Likewise for an equirectangular projection, the filters would be arranged in different latitude bands. For example, every 10 degrees of latitude would be computed by a different filter. This way the filters could implement the increasing distortion.

Another viable approach which is starting to get explored by the neural network research community is spherical convolutions, which compute the convolution directly over hemispherical data instead of mapping the hemisphere onto a plane. The applications of this approach are far reaching and include robotics [Khasanova and Frossard, 2017] or astronomical cosmology [Perraudin et al., 2018]. There are several approaches emerging in the last years, such as simply letting the network learn the convolutional filters as proposed by Su and Grauman [2017], utilizing Fast-Fourier Transform as proposed by Cohen et al. [2018]

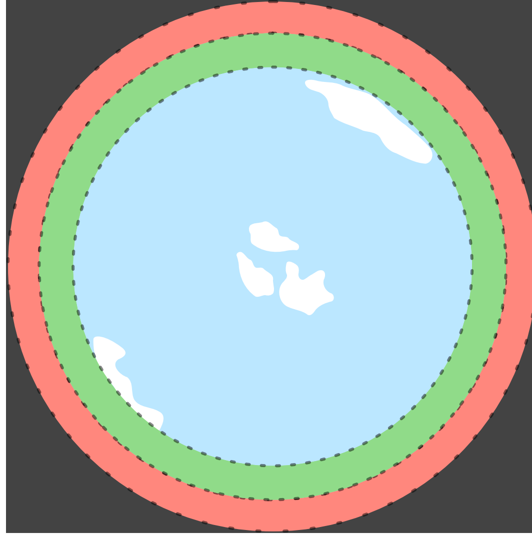


Figure 6.1: An illustration of concentric convolutional filters over an image using a fisheye projection. The colored bands (green and red) illustrate two different filters used at two different distortion levels. This way, the weights of the filters are shared only between regions with similar distortion.

or transforming the spherical data into a graph as can be seen in Khasanova and Frossard [2017] and Perraudin et al. [2018].

We note, however, that even a spherical convolution will not solve the inherent perspective change occurring in the skies. That is, when we look up, we are looking at the clouds above our heads from below. On the other hand, near the horizon, we see clouds from their side. This inherent perspective change will not be solved by using spherical convolutions but rather in using different convolutional filters for different parts of the sky, to capture this perspective shift.

However implemented, we think that taking the changing distortion of the perspective projection into account could significantly improve the quality of both the generalization as well as the generated images.

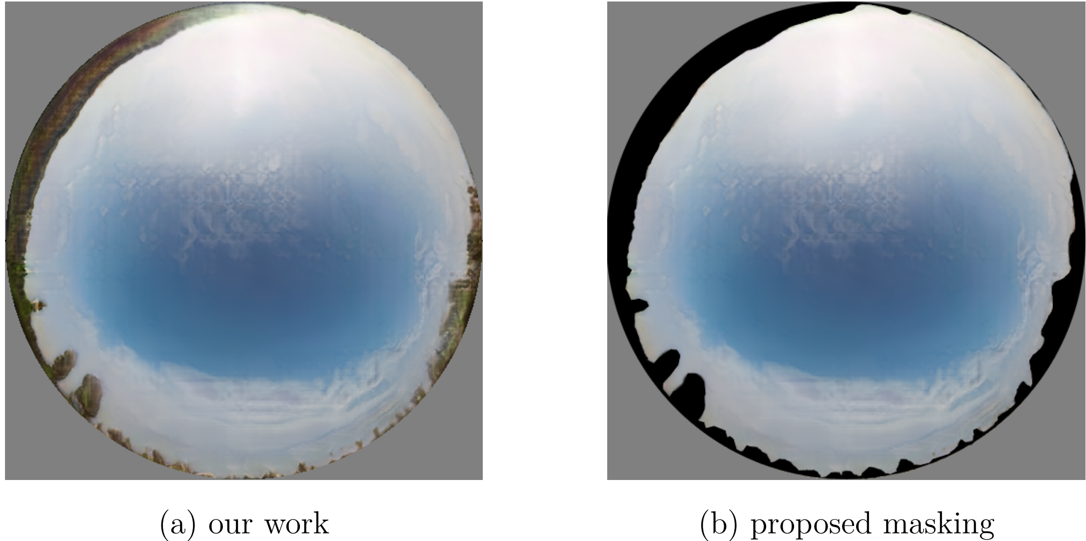
## 6.2 Skyline separation

Our dataset acquisition pipeline, which we described in Chapter 3, captures the whole upper hemisphere. While we try to select locations which have enough open ground not to capture a lot of nearby trees or mountains, this is hard to avoid completely.

Because of this the network also learns to replicate some form of skyline in the generated image, as can be seen in the left image (a) of Figure 6.2. These forests, trees, mountains or buildings the network is trying to reproduce are of very low quality since the network did not have enough data to learn to generate these marginal objects in high detail.

We propose, as an extension to our work, to include a skyline separation algorithm which would automatically mask any parts of the image which are below the horizon line for the current image generate a pixel-wise mask for each dataset image. The mask can be seen in Figure 6.2 on the right image (b). We could then pass this mask into the network and use a very similar trick as we

used in Section 4.2.1. We would multiply the input image in the discriminator with this skyline separation mask before passing it to the discriminator network. This would effectively mean that the discriminator would never see the non-sky parts and thus not force the generator to produce them. Our hypothesis is that a network modified in this way would always generate a full 180° skydome with zero obstructions.



(a) our work

(b) proposed masking

Figure 6.2: An illustration of our proposed skyline separation. On the left in image (a) is our current work. Because the network did not have a lot of data to learn how trees and mountains look like, the quality is very poor. On the right in image (b) we masked the horizon by hand to showcase the core goal of our idea.

## 6.3 Latent space mapping

As we have already introduced in Chapter 1, generative adversarial networks map the training data into a so called *latent space*. While generating a new image, a random vector in the space is sampled and the network generates the new image according to this vector. Unlike variational autoencoders, GANs do not allow us to compute the inverse function of the vector-to-image mapping. That is, for a given image, we cannot find a corresponding latent vector using the GAN network.

However, navigating the latent space is a very useful tool to have as it enables us to create tools for the artists to generate a skydome to their own liking. This could be done by, for example, providing a computer program where the artist can choose several parameters (percentage of cloud cover, time of the day, turbidity of the skydome, type of clouds, etc.) and generate only skydomes which observe these parameters. This would be implemented by taking the artist's parameters, translating them into the latent space as constraints and then generating latent vectors within these constraints. It would also enable us to order the network to generate a time sequence of the changing sky (by fixing every parameter except for time).

Mapping the latent space is a hard problem to solve and one possible way to

solve it might be to use a variational autoencoder, trained on the real skydome dataset. Once trained, this VAE would interpret the images generated by the network and we would receive a mapping function between the latent vectors $g \in G$ from the GAN (used to generate the images) and latent vectors $v \in V$ generated by the VAE, by feeding the images generated for the latent vector $g$ into the VAE and receiving the corresponding latent vectors $v$ from the VAE.

This way we might be able to perform some analysis on the VAE's latent space $V$ and then directly translate this knowledge to the GAN's latent space $G$.

## 6.4 Providing more input to the network

While we think of GAN networks as a network which process only images, because we usually do not have any additional information which could help the network, this is not true in our case. There is a lot of useful data we can either simply read from the *EXIF* data of the image or compute with little additional overhead. This includes the time and date when the image was taken, the Sun elevation, the photo orientation (azimuthal, i.e. whether the up-axis is north or east), *GPS* coordinates, etc.

These additional inputs might help to shape the latent space into a more navigable space, because the network might try to correlate this additional information with the information contained within the image. This is desirable because, for example, the Sun position in the image and time is highly correlated (because of the Sun elevation being a function of world position and current time).

As a result of this modification, we might be able to condition the generated images based on some of this data, for example getting the network to only generate images taken at a given time or with a given Sun elevation.

This point of future work is strongly related to the latent space navigation we discussed in the previous section, attempting to perform a similar task. But while the previous method tries to accomplish the goal by training a new neural network, this approach extends the current network instead.

## 6.5 Addition instead of generation

A big room for improvement of this work might lie in abandoning the idea to generate the whole skydome from scratch and instead utilize an image-to-image generative network such as the work of Isola et al. [2016]. This idea is based on the fact that there exists an analytical model of the skydome radiance by Hosek and Wilkie [2012].

Utilizing this model, we could train a network which receives the generated skydome from the analytical model and adds correct clouds on top of it. To do this, we would need to label our dataset with parameters for the Hosek-Wilkie model, which would generate the analytical skydome very similar to the sky in our captured picture. The model, apart from the Sun elevation parameter, has two other parameters which are the turbidity of the sky and the horizon blur (as implemented in Corona Renderer [ChaosCzech, 2019]). We would need to guess these parameters for each of our dataset pictures. This could probably be done by utilizing a simple regression model. Computing the Sun elevation requires the

exact time and position where the photo was taken. Creating this dataset should not be that hard, since we already capture both of these readings in our pipeline.

Once we obtain this dataset, we would then generate the clear skydome for each of the dataset pictures, and train an image-to-image network to take the clear skydome picture and dream new clouds on top of it, and train it against the real photo we shot.

This approach would have a tremendous advantage to the users of the common commercial renderers, since they are already used to using this skydome model and controlling the look of the sky by altering a Sun object within their scene. This advancement would allow them to alter the look of the sky by altering the Sun object (changing the Sun elevation to set the correct ”time” of the scene) and afterwards press a button and generate additional cloud layer on top of this skydome. Generating the environment map this way would also mean that the Sun elevation stays unchanged between the different randomly generated examples of the sky, which might be a desirable property.

## 6.6 Increasing the resolution

As we have already seen from the previous chapters, our neural network is not able to produce images with resolutions close to $15000 \times 7500$ pixels. Since GAN architectures are not advanced enough to generate high resolutions immediately, we want to utilize *super-resolution* neural networks to provide us with the extra level of detail we need for our applications.

This could effectively mean that we will utilize our HDR GAN network for the purpose of generating the rough semantic idea of the sky, and then apply a super-resolution neural network such as SRCNN [Dong et al., 2014] to provide the additional detail.

Since many state of the art super-resolution networks also utilize GAN architectures, there also exists a possibility to connect our HDR network with the super-resolution network and practically create a new joint architecture, which could be trained end-to-end.

# Conclusion

In this thesis we explored the concept of generating realistic high dynamic range environment maps using neural networks. Since very little research was published about this topic, we wanted to provide some basic research which would lay the ground work for further research. To accomplish this we had to perform several different tasks, which we will now summarize.

To be able to produce realistic looking images we first needed to gather a dataset. Because no such dataset is available, we developed a complete dataset capturing pipeline and utilized it to capture thousands of HDR images. The complete results can be seen in Section 5.1. We also discussed at length the advantages of different projections and found the *fisheye projection* to be the best compromise for our purpose.

In parallel to gathering our dataset we performed experiments on a state of the art neural network, to evaluate the image quality of the architecture before and after converting it to produce high dynamic range images. We provided in-depth discussion about our experiments and their results in Section 5.2.

As a result of these experiments we were able to identify several areas of improvement, as well as provide a proposed solution to each of these problems. We explain these ideas in the chapter Future work. We will continue expanding our dataset as well as improving the network in order to achieve industry quality results in the future.

# Bibliography

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 Mane, 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 Viegas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. *arXiv:1603.04467 [cs]*, March 2016. URL `http://arxiv.org/abs/1603.04467`. arXiv: 1603.04467.

Martin Arjovsky and Léon Bottou. Towards Principled Methods for Training Generative Adversarial Networks. *arXiv:1701.04862 [cs, stat]*, January 2017. URL `http://arxiv.org/abs/1701.04862`. arXiv: 1701.04862.

Martin Arjovsky, Soumith Chintala, and Léon Bottou. Wasserstein GAN. *arXiv:1701.07875 [cs, stat]*, January 2017. URL `http://arxiv.org/abs/1701.07875`. arXiv: 1701.07875.

F. Bettonvil. Fisheye lenses. *WGN, Journal of the International Meteor Organization*, 33:9–14, February 2005.

G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

Andrew Brock, Jeff Donahue, and Karen Simonyan. Large Scale GAN Training for High Fidelity Natural Image Synthesis. *arXiv:1809.11096 [cs, stat]*, September 2018. URL `http://arxiv.org/abs/1809.11096`. arXiv: 1809.11096.

New House Internet Services B.V. PTGui. URL `https://www.ptgui.com/`. Accessed on 23. 06. 2019.

ChaosCzech. Corona Renderer, 2019. URL `https://corona-renderer.com/`. Accessed on 24. 06. 2019.

Adam Coates, Andrew Ng, and Honglak Lee. An Analysis of Single-Layer Networks in Unsupervised Feature Learning. In Geoffrey Gordon, David Dunson, and Miroslav Dudík, editors, *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, volume 15 of *Proceedings of Machine Learning Research*, pages 215–223, Fort Lauderdale, FL, USA, April 2011. PMLR. URL `http://proceedings.mlr.press/v15/coates11a.html`.

Taco S. Cohen, Mario Geiger, Jonas Koehler, and Max Welling. Spherical CNNs. *arXiv:1801.10130 [cs, stat]*, January 2018. URL `http://arxiv.org/abs/1801.10130`. arXiv: 1801.10130.

L. Dagum and R. Menon. OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1): 46–55, March 1998. ISSN 10709924. doi: 10.1109/99.660313. URL `http://ieeexplore.ieee.org/document/660313/`.

Helmut Dersch. Panorama Tools. URL `http://panotools.sourceforge.net/`. Accessed on 23. 06. 2019.

Chao Dong, Chen Change Loy, Kaiming He, and Xiaoou Tang. Image Super-Resolution Using Deep Convolutional Networks. *arXiv:1501.00092 [cs]*, December 2014. URL `http://arxiv.org/abs/1501.00092`. arXiv: 1501.00092.

Vincent Dumoulin and Francesco Visin. A guide to convolution arithmetic for deep learning. *arXiv:1603.07285 [cs, stat]*, March 2016. URL `http://arxiv.org/abs/1603.07285`. arXiv: 1603.07285.

Gabriel Eilertsen, Joel Kronander, Gyorgy Denes, Rafa\l Mantiuk, and Jonas Unger. HDR image reconstruction from a single exposure using deep CNNs. *ACM Transactions on Graphics (TOG)*, 36(6), 2017.

Rafael Gómez-Bombarelli, Jennifer N. Wei, David Duvenaud, José Miguel Hernández-Lobato, Benjamín Sánchez-Lengeling, Dennis Sheberla, Jorge Aguilera-Iparraguirre, Timothy D. Hirzel, Ryan P. Adams, and Alán Aspuru-Guzik. Automatic chemical design using a data-driven continuous representation of molecules. *ACS Central Science*, 4(2):268–276, February 2018. ISSN 2374-7943, 2374-7951. doi: 10.1021/acscentsci.7b00572. URL `http://arxiv.org/abs/1610.02415`. arXiv: 1610.02415.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative Adversarial Networks. *arXiv:1406.2661 [cs, stat]*, June 2014. URL `http://arxiv.org/abs/1406.2661`. arXiv: 1406.2661.

Ishaan Gulrajani, Faruk Ahmed, Martin Arjovsky, Vincent Dumoulin, and Aaron Courville. Improved Training of Wasserstein GANs. *arXiv:1704.00028 [cs, stat]*, March 2017. URL `http://arxiv.org/abs/1704.00028`. arXiv: 1704.00028.

Gábor Horváth. RawTherapee, 2019. URL `https://rawtherapee.com`. Accessed on 23. 06. 2019.

Lukas Hosek and Alexander Wilkie. An Analytic Model for Full Spectral Sky-Dome Radiance. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2012)*, 31(4), July 2012.

Trammell Hudson. Magic Lantern, 2009. URL `https://magiclantern.fm/`. Accessed on 24. 06. 2019.

Sergey Ioffe and Christian Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *arXiv:1502.03167 [cs]*, February 2015. URL `http://arxiv.org/abs/1502.03167`. arXiv: 1502.03167.

Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, and Alexei A. Efros. Image-to-Image Translation with Conditional Adversarial Networks. *arXiv:1611.07004*

*[cs]*, November 2016. URL http://arxiv.org/abs/1611.07004. arXiv: 1611.07004.

T Jayalakshmi and Santhakumaran A. Statistical normalization and back propagation for classification. *International Journal Computer Theory Engineering (IJCTE)*, 3:89–93, 2011.

SeokYoon Kang, Kyoung Choon Park, and Ki-Il Kim. Real-Time Cloud Modeling and Rendering Approach Based on L-system for Flight Simulation. *International Journal of Multimedia and Ubiquitous Engineering*, 10(6):395–406, June 2015. ISSN 19750080. doi: 10.14257/ijmue.2015.10.6.38. URL http://www.sersc.org/journals/IJMUE/vol10_no6_2015/38.pdf.

Tero Karras, Timo Aila, Samuli Laine, and Jaakko Lehtinen. Progressive Growing of GANs for Improved Quality, Stability, and Variation. *arXiv:1710.10196 [cs, stat]*, October 2017. URL http://arxiv.org/abs/1710.10196. arXiv: 1710.10196.

Tero Karras, Samuli Laine, and Timo Aila. A Style-Based Generator Architecture for Generative Adversarial Networks. *arXiv:1812.04948 [cs, stat]*, December 2018. URL http://arxiv.org/abs/1812.04948. arXiv: 1812.04948.

Renata Khasanova and Pascal Frossard. Graph-Based Classification of Omnidirectional Images. *arXiv:1707.08301 [cs]*, July 2017. URL http://arxiv.org/abs/1707.08301. arXiv: 1707.08301.

Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. *arXiv:1412.6980 [cs]*, December 2014. URL http://arxiv.org/abs/1412.6980. arXiv: 1412.6980.

Diederik P. Kingma and Max Welling. Auto-Encoding Variational Bayes. *arXiv:1312.6114 [cs, stat]*, December 2013. URL http://arxiv.org/abs/1312.6114. arXiv: 1312.6114.

Diederik P. Kingma, Danilo J. Rezende, Shakir Mohamed, and Max Welling. Semi-Supervised Learning with Deep Generative Models. *arXiv:1406.5298 [cs, stat]*, June 2014. URL http://arxiv.org/abs/1406.5298. arXiv: 1406.5298.

D.P Kingma. *Variational inference & deep learning: A new synthesis*. 2017. ISBN 978-94-6299-745-5. OCLC: 8086899107.

Jean-François Lalonde, Louis-Philippe Asselin, Julien Becirovski, Yannick Hold-Geoffroy, Mathieu Garon, Marc-André Gardner, and Jinsong Zhang. The Laval HDR Sky Database, 2016. URL http://sky.hdrdb.com. Accessed on 29.5.2019.

Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation Applied to Handwritten Zip Code Recognition. *Neural Computation*, 1(4):541–551, December 1989. ISSN 0899-7667, 1530-888X. doi: 10.1162/neco.1989.1.4.541. URL http://www.mitpressjournals.org/doi/10.1162/neco.1989.1.4.541.

Yann LeCun, Corinna Cortes, and Christopher J.C. Burges. THE MNIST database of handwritten digits. URL http://yann.lecun.com/exdb/mnist/. Accessed on 20. 06. 2019.

Yann LeCun, Leon Bottou, Genevieve B. Orr, and Klaus Robert Müller. Efficient BackProp. In Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, Genevieve B. Orr, and Klaus-Robert Müller, editors, *Neural Networks: Tricks of the Trade*, volume 1524, pages 9–50. Springer Berlin Heidelberg, Berlin, Heidelberg, 1998. ISBN 978-3-540-65311-0 978-3-540-49430-0. doi: 10.1007/3-540-49430-8_2. URL http://link.springer.com/10.1007/3-540-49430-8_2.

Demetris Marnerides, Thomas Bashford-Rogers, Jonathan Hatchett, and Kurt Debattista. ExpandNet: A Deep Convolutional Neural Network for High Dynamic Range Expansion from Low Dynamic Range Content. *arXiv:1803.02266 [cs]*, March 2018. URL http://arxiv.org/abs/1803.02266. arXiv: 1803.02266.

Dirk Merkel. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux J.*, 2014(239), March 2014. ISSN 1075-3583. URL http://dl.acm.org/citation.cfm?id=2600239.2600241.

Cade Metz and Keith Collins. How an A.I. 'Cat-and-Mouse Game' Generates Believable Fake Photos, February 2018. URL https://www.nytimes.com/interactive/2018/01/02/technology/ai-generated-photos.html. Accessed on 26. 5. 2019.

Takeru Miyato, Toshiki Kataoka, Masanori Koyama, and Yuichi Yoshida. Spectral Normalization for Generative Adversarial Networks. *arXiv:1802.05957 [cs, stat]*, February 2018. URL http://arxiv.org/abs/1802.05957. arXiv: 1802.05957.

NoEmotion. NoEmotion HDRs. URL http://noemotionhdrs.net/. Accessed on 29.5.2019.

Nvidia. GeForce GTX 1080 Ti. URL https://www.nvidia.com/en-us/geforce/products/10series/geforce-gtx-1080-ti/. Accessed on 24. 06. 2019.

Augustus Odena, Vincent Dumoulin, and Chris Olah. Deconvolution and Checkerboard Artifacts. *Distill*, 1(10):10.23915/distill.00003, October 2016. doi: 10.23915/distill.00003. URL http://distill.pub/2016/deconv-checkerboard.

Taesung Park, Ming-Yu Liu, Ting-Chun Wang, and Jun-Yan Zhu. Semantic Image Synthesis with Spatially-Adaptive Normalization. *arXiv:1903.07291 [cs]*, March 2019. URL http://arxiv.org/abs/1903.07291. arXiv: 1903.07291.

Vaibhav Amit Patel, Purvik Shah, and Shanmuganathan Raman. A Generative Adversarial Network for Tone Mapping HDR Images. In Renu Rameshan, Chetan Arora, and Sumantra Dutta Roy, editors, *Computer Vision, Pattern Recognition, Image Processing, and Graphics*, volume 841, pages 220–231. Springer Singapore, Singapore, 2018. ISBN 9789811300196 9789811300202. doi:

10.1007/978-981-13-0020-2_20. URL http://link.springer.com/10.1007/978-981-13-0020-2_20.

Ken Perlin. An image synthesizer. *ACM SIGGRAPH Computer Graphics*, 19 (3):287–296, July 1985. ISSN 00978930. doi: 10.1145/325165.325247. URL http://portal.acm.org/citation.cfm?doid=325165.325247.

Nathanaël Perraudin, Michaël Defferrard, Tomasz Kacprzak, and Raphael Sgier. DeepSphere: Efficient spherical Convolutional Neural Network with HEALPix sampling for cosmological applications. *arXiv:1810.12186 [astro-ph]*, October 2018. URL http://arxiv.org/abs/1810.12186. arXiv: 1810.12186.

Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks. *arXiv:1511.06434 [cs]*, November 2015. URL http://arxiv.org/abs/1511.06434. arXiv: 1511.06434.

E. Reinhard and K. Devlin. Dynamic Range Reduction Inspired by Photoreceptor Physiology. *IEEE Transactions on Visualization and Computer Graphics*, 11 (01):13–24, January 2005. ISSN 1077-2626. doi: 10.1109/TVCG.2005.9. URL http://ieeexplore.ieee.org/document/1359728/.

Herbert Robbins and Sutton Monro. A Stochastic Approximation Method. *The Annals of Mathematical Statistics*, 22(3):400–407, September 1951. ISSN 0003-4851. doi: 10.1214/aoms/1177729586. URL http://projecteuclid.org/euclid.aoms/1177729586.

Timothy Roden and Ian Parberry. Clouds and stars: efficient real-time procedural sky rendering using 3d hardware. In *Proceedings of the 2005 ACM SIGCHI International Conference on Advances in computer entertainment technology - ACE '05*, pages 434–437, Valencia, Spain, 2005. ACM Press. ISBN 978-1-59593-110-8. doi: 10.1145/1178477.1178574. URL http://portal.acm.org/citation.cfm?doid=1178477.1178574.

Andrew Schneider and Nathan Vos. The Real-Time Volumetric Cloudscapes of Horizon Zero Dawn, August 2015. URL http://advances.realtimerendering.com/s2015/The%20Real-time%20Volumetric%20Cloudscapes%20of%20Horizon%20-%20Zero%20Dawn%20-%20ARTR.pdf. Accessed on 19. 06. 2019.

M. Shanker, M.Y. Hu, and M.S. Hung. Effect of data standardization on neural network training. *Omega*, 24(4):385–397, August 1996. ISSN 03050483. doi: 10.1016/0305-0483(96)00010-2. URL https://linkinghub.elsevier.com/retrieve/pii/0305048396000102.

J. Sola and J. Sevilla. Importance of input data normalization for the application of neural networks to complex industrial problems. *IEEE Transactions on Nuclear Science*, 44(3):1464–1468, June 1997. ISSN 00189499. doi: 10.1109/23.589532. URL http://ieeexplore.ieee.org/document/589532/.

Yu-Chuan Su and Kristen Grauman. Learning Spherical Convolution for Fast Features from 360{\deg} Imagery. *arXiv:1708.00919 [cs]*, August 2017. URL `http://arxiv.org/abs/1708.00919`. arXiv: 1708.00919.

A. Torralba, R. Fergus, and W.T. Freeman. 80 Million Tiny Images: A Large Data Set for Nonparametric Object and Scene Recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 30(11):1958–1970, November 2008. ISSN 0162-8828. doi: 10.1109/TPAMI.2008.128. URL `http://ieeexplore.ieee.org/document/4531741/`.

Cédric Villani. *Optimal Transport*, volume 338 of *Grundlehren der mathematischen Wissenschaften*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. ISBN 978-3-540-71049-3 978-3-540-71050-9. doi: 10.1007/978-3-540-71050-9. URL `http://link.springer.com/10.1007/978-3-540-71050-9`.

Phil Wang. This Person Does Not Exist. URL `https://thispersondoesnotexist.com/`. Accessed on 26. 5. 2019.

Greg Zaal. HDRi Haven. URL `https://hdrihaven.com/`. Accessed on 20. 06. 2019.

Greg Zaal. How to Create High Quality HDR Environments, February 2019. URL `http://blog.hdrihaven.com/how-to-create-high-quality-hdri/`. Accessed on 29.5.2019.

Jinsong Zhang and Jean-François Lalonde. Learning High Dynamic Range from Outdoor Panoramas. *arXiv:1703.10200 [cs]*, March 2017. URL `http://arxiv.org/abs/1703.10200`. arXiv: 1703.10200.

# List of Figures


1 An example of a low dynamic range environment map . . . . . . . 3

1.1 An illustration of a single convolutional layer filtering. . . . . . . . 8
1.2 An illustration of the Variational autoencoder architecture. . . . . 12
1.3 An illustration of the Generative adversarial network architecture. 14

3.1 An image captured with a 8mm focal length. . . . . . . . . . . . . 22
3.2 High dynamic image captured in too high winds. . . . . . . . . . . 23
3.3 Our Mi Sphere camera. . . . . . . . . . . . . . . . . . . . . . . . . 24
3.4 A *dual fisheye* image taken by the Mi Sphere camera. . . . . . . . 25
3.5 Exposure bracketing and intervalometer settings of Magic Lantern 28
3.6 An example of equirectangular projection . . . . . . . . . . . . . . 31
3.7 Fisheye projections comparison. . . . . . . . . . . . . . . . . . . . 32
3.8 Difference image between the output of PtGui and our program . 34

4.1 A diagram of the growing process. . . . . . . . . . . . . . . . . . . 40
4.2 Images captured by our dataset acquisition pipeline. . . . . . . . . 42
4.3 An illustration of a $4 \times 4$ and $8 \times 8$ circular mask. . . . . . . . . . 43
4.4 A low dynamic range image generated by one of our networks. . . 44
4.5 An illustration of a deconvolutional layer. . . . . . . . . . . . . . . 45
4.6 An illustration of a *resize-convolution layer*. . . . . . . . . . . . 46

5.1 A few 360° panoramas shot with a DSLR. . . . . . . . . . . . . . 52
5.2 Full 360° panorama shot in Prague. . . . . . . . . . . . . . . . . . 53
5.3 A few Mi Sphere 360° panoramas. . . . . . . . . . . . . . . . . . . 54
5.4 A timelapse captured in Prague. . . . . . . . . . . . . . . . . . . . 57
5.5 Samples from different dataset shoots. . . . . . . . . . . . . . . . 58
5.6 A sample from the real dataset from the NoEmotion team. . . . . 60
5.7 A matrix of images generated by our trained network. . . . . . . . 61
5.8 A sample from the bigger dataset containing 650 unique images. . 63
5.9 A matrix of images generated by our trained network. . . . . . . . 64
5.10 Examples of the so called *checkerboard artifacts*. . . . . . . . . . . 65
5.11 An analysis of continuity constraints in equirectangular projection. 66
5.12 A sample from the real data from the fisheye projected dataset. . 67
5.13 A sample from the data generated by the LDR fisheye network. . 68
5.14 Generated images remapped into equirectangular projection. . . . 70
5.15 An illustration of the *checkerboard artifact* in the fisheye network. 71
5.16 An illustration of the *oil stain-like artifact* in the fisheye network. 71
5.17 An sample of failure cases of the fisheye network. . . . . . . . . . 72
5.18 Linear interpolation on latent vectors. . . . . . . . . . . . . . . . . 73
5.19 Interpolated images remapped into equirectangular projection. . . 74
5.20 Equirectangular version of the benchmarking results. . . . . . . . 75
5.21 Comparison of the three versions of the network. . . . . . . . . . . 76
5.22 A *mode collapse* of an HDR network. . . . . . . . . . . . . . . . . 78
5.23 The real data input into the HDR network. . . . . . . . . . . . . . 80
5.24 A big sample of fake images generated by the HDR network. . . . 81
5.25 Fake data generated by the HDR network. . . . . . . . . . . . . . 82

5.26 HDR over-fitting experiment dataset image. . . . . . . . . . . . . 83
5.27 A sample produced by the HDR over-fitting experiment. . . . . . 84
5.28 A sample of the bigger dataset. . . . . . . . . . . . . . . . . . . . . . 85
5.29 A sample produced by the HDR over-fitting experiment. . . . . . 87
5.30 An example of the prevailing checkerboard artifact. . . . . . . . . 88
5.31 An example of the prevailing oil-like artifact. . . . . . . . . . . . . 88
5.32 Real and fake scores for logarithm benchmark training. . . . . . . 89
5.33 A sample of real data in our single timelapse experiment. . . . . . 90
5.34 A sample of images generated in our single timelapse experiment. 91
5.35 An illustration of a correct timelapse behaviour. . . . . . . . . . . 92
5.36 An illustration of a wrong timelapse behaviour. . . . . . . . . . . 92
5.37 A sample of real data for the experiment with our data. . . . . . . 94
5.38 A sample of data generated during the experiment with our data. 95
5.39 Histograms for similar real and fakes photos. . . . . . . . . . . . . 96
5.40 Histograms for additional similar real and fakes photos. . . . . . . 97

6.1 An illustration of concentric convolutional filters. . . . . . . . . . 100
6.2 Masking the horizon with skyline separation. . . . . . . . . . . . . 101

A.1 Goal of the dataset shoot. . . . . . . . . . . . . . . . . . . . . . . . 122
A.2 Description of the camera controls. . . . . . . . . . . . . . . . . . . 123
A.3 Description of the camera and lens controls. . . . . . . . . . . . . 124
A.4 MagicLantern software booted up correctly. . . . . . . . . . . . . 125
A.5 Exposure bracketing and intervalometer settings of Magic Lantern 126
A.6 MagicLantern intervalometer in practice. . . . . . . . . . . . . . . 127
A.7 Correctly setup camera on the tripod. . . . . . . . . . . . . . . . . 128
A.8 Applying the profile in RawTherapee. . . . . . . . . . . . . . . . . 129
A.9 Detecting panoramas in PtGui. . . . . . . . . . . . . . . . . . . . . 130
A.10 Detected panoramas in PtGui. . . . . . . . . . . . . . . . . . . . . 131

# List of Tables

4.1 Architecture of a single building block of the generator network. . 39
4.2 Architecture of a single building block of the discriminator network. 41
4.3 Hardware configuration of Computer 1. . . . . . . . . . . . . . . . . 48
4.4 Hardware configuration of Computer 2. . . . . . . . . . . . . . . . . 48

5.1 The upper-hemisphere skydome images we have taken to create our dataset. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 55

# List of Abbreviations

| | |
|---|---|
| **HDR** | High dynamic range |
| **LDR** | Low dynamic range |
| **GAN** | Generative adversarial network |
| **VAE** | Variational autoencoder |
| **CGI** | Computer-generated imagery |
| **CNN** | Convolutional neural network |
| **SGD** | Stochastic gradient descent |
| **EM distance** | Earth mover's distance |
| **WGAN** | Wasserstein GAN |
| **WGAN-GP** | Wasserstein GAN with gradient penalty |
| **FOV** | Field of view |
| **DSLR** | Digital single-lens reflex camera |
| **ND filter** | Neutral density filter |
| **EV** | Exposure value |
| **RGB** | Red, green, blue (image format) |
| **CPU** | Central processing unit |
| **GPU** | Graphics processing unit |
| **RAM** | Random access memory |
| **PCIe** | Peripheral Component Interconnect Express |
| **EXIF** | Exchangeable image file format |
| **GPS** | Global Positioning System |

# A. Attachments

## A.1 Digital attachments

In this section, we describe all the digital attachments we include.

- Dataset – we include several files important to the capturing of the dataset, as we have discussed in Chapter 3.
  - *Pre-shoot checklist* – since the shooting manual (Attachment A.3) is long and impractical to read on-site, we have developed a simple one sheet checklist, which should help both the person capturing the time-lapse to correctly setup the camera, as well as the person converting the data into a dataset-compatible format.
  - *Profiles* – we include both the PtGui and RawTherapee profiles we utilize to process our photos after they are captured. Please note that these profiles are subject to change.do
  - *Photos shot* – a document with dates, GPS coordinates, times and numbers of images taken for each of our shoots.
  - *A LDR timelapse example shoot* – we include complete timelapse which was shot using the methods described in this thesis. To save space, we only include the LDR version of the shoot.
- Neural networks – we include all our training runs, examples of the dataset, the code used to run them and the results (both the generated images and the Tensorboard log files). Here we describe the structure of the included experiments.
  - *Dataset examples* – we include 3 different photos to illustrate how the dataset of this experiment looks for most of the training runs. These photos are consistent across the experiments (if the photos were utilized), to make the comparison better.
  - *Run setup* – contains all the code and docker file required to run the network. This code is left in the state it was run in, so for reproducing the experiment, one only needs to run the appropriate bash script.
  - *Run result* – the results of the network. This includes the log file, the config file, the Tensorboard summaries and the fake images generated after each epoch.
  - *Extra data* – some experiments generated extra data (such as extra images, interpolations, etc.). We include these extra images as well.
- Dataset processing pipeline – we include all of our scripts to resize, augment and convert all images before feeding them into the network, which we have discussed in Chapter 3. These scripts are:
  - *Augmentation* scripts for both equirectangular and fisheye projections, which augment the images by rotating the image around the upward facing axis.

- *Conversion from/to `.TFRecords` format* – we include scripts for both converting a folder of images into a `.TFRecords` file and a script to convert the `.TFRecrods` file back to images, as well as calculate the average dynamic range.
- *Equirectangular to fisheye* C++ program, which we have already mentioned in Chapter 3. The program can convert an equirectangular map into a fisheye projection, with *stereographic*, *equisolid* and *equidistant* projections to choose from.
- *Docker environment* prepared to run all of the mentioned scripts on any machine.

We also include readme files which describe the contents in more detail.

## A.2 Data from the internet

In this attachment, we describe the data we collected on the internet, from freely available sources. We do not include this data as part of our digital attachment, since it is downloadable from the internet.

1. *NoEmotion HDRs* – http://noemotionhdrs.net/ – between the day and night HDR images, this website provides 106 high resolution environment maps. 34 of those are shot during the day, 72 are shot during the evenings. The Sun is clipped in these photos, but they still provide a good dynamic range of around $10^3$.

2. *HDRi Haven* – https://hdrihaven.com/ – while this research was going on, this website provided 74 usable environment maps with a visible skydome. Since this website is really active, there are probably additional photos which were added after research. This website provides photos with **unclipped** Sun, which makes it rather unique.

3. *HDRI Skies* – https://hdri-skies.com/ – this website provides around 429 environment maps. Although the full resolution photos are paid, one can download the lower resolution ($2048 \times 1024$ pixels) images for free.

4. The Laval HDR Sky Database [Lalonde et al., 2016] – this database provides a big amount of data with very low variance. In order to avoid overfitting, we need to subsample these first, to only include a portion of the images. More importantly, this database also includes **metadata** like the Sun coordinates, date and time. The data is also classified as clear or cloudy, making it easier to pick images we would like (with clouds present). We only found this database later in the research and as such it is not included in many of the training runs.

## A.3 Dataset shooting manual

This attachment is a field manual to read before shooting the dataset. It is used to get the user familiar with the camera, with the goals and the means to achieve those goals. Our goal is shooting images similar to what can be seen in Figure A.1.

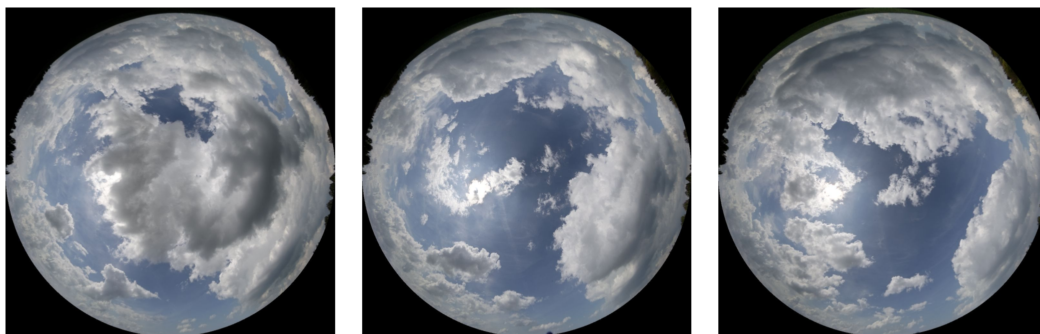


Figure A.1: Three photos taken in the same dataset shoot. Shooting these pictures is the goal of this manual.

### Required gear

In this part of our manual, we briefly go over the gear we utilize to shoot the dataset. Several of the pieces are not strictly necessary, but make the process a little easier.

- Canon 5D Mark II – a DSLR full-frame camera, any full-frame camera which is able to set a custom bracketing sequence and shoot the sequence every $N$ minutes is enough.
- Canon EF 8-15 mm f/4.0 L USM fisheye lens – a 8 mm focal length lens is required to capture the whole upper hemisphere of the sky. This is also the reason why APS-C or any other smaller sensors are discouraged for our shoot, since achieving an equivalent of 8 mm is almost impossible.
- Memory card – Canon 5D Mark II uses the CF memory card format
- Manfrotto 055 tripod[1] with a ball head[2] – a solid and heavy tripod is very important, since it lowers the amount of camera shake due to wind and therefore produces better photos.
- Panoramic head NOVOFLEX VR-SYSTEM PRO[3] – an optional piece of gear, used mainly in full panoramic shoots. However, this tripod head combined with a ball head on the tripod allows us to level only the head without changing the height of the tripod legs, which makes the setup easier.

[1]https://www.manfrotto.co.uk/collections/supports/055-series
[2]https://www.manfrotto.us/xpro-ball-head-in-magnesium-with-200pl-plate
[3]https://www.novoflex.de/en/715/the-professional-allrounder-vr-system-pro-ii.html

- Lens-cleaning brush[4] – dust on the lens is a problem which can be very easily fixed before the shoot. Having a brush also helps with accidental smudges on the lens while handling the camera.
- Compass, GPS – used to provide the EXIF data of position and heading. This might be required in the future, should we attempt to augment the training with some geographical data.

We also include a brief camera controls description in Figure A.2 and Figure A.3.

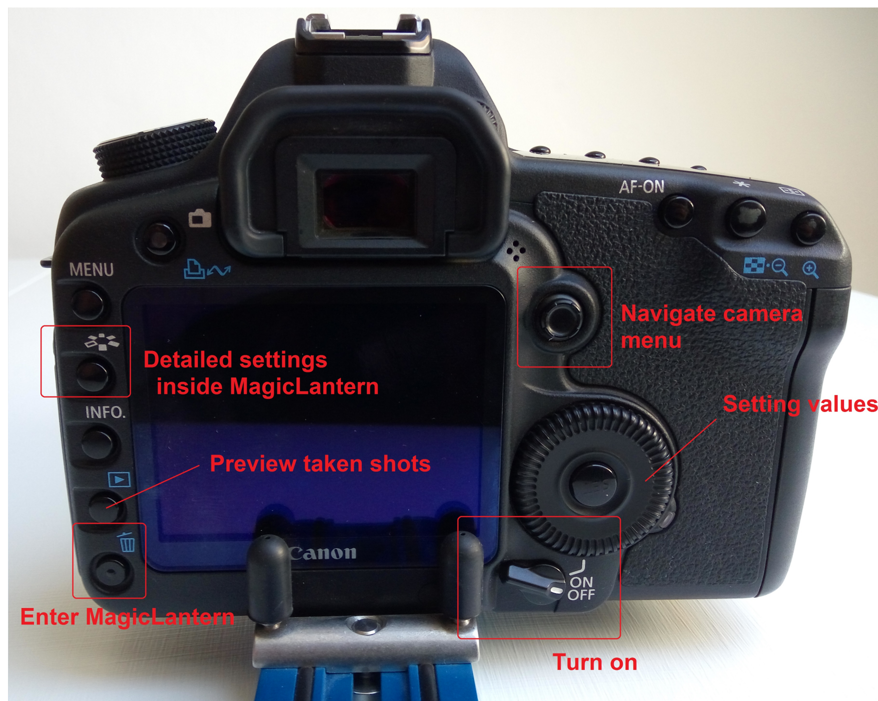



Figure A.2: Description of the camera controls on the back of the camera.

## Dataset shooting

First, we present a to-do list before heading out for the shooting location.

1. Clear the CF card from old data. The photos get stored in the DCIM folder, so you can delete the contents of this folder. You need to preserve the ML/ folder, as well as `5D2-212.fir` and `autoexec.bin` files in the root. Safest rule is to just delete `.CR2` data from within `/DCIM/100EOS5D/`.
2. Make sure MagicLantern (ML) boots up correctly.
   (a) Insert the card.
   (b) Turn on the camera.
   (c) If the main window shows the battery life in percent (see Figure A.4), ML is booted correctly.

[4]https://www.fotoskoda.cz/lenspen-stetec/

Figure A.3: Description of the camera controls on the top of the camera and the controls of the lens.

3. Charge the camera's battery
4. Fix the Novoflex blue rail to the camera, if it has been undone

If all things are performed correctly, we will see the screen showcased in Figure A.4 on the camera's back screen. Notice the small text like battery percentage, time, remaining gigabytes and "HDR 7x2EV". If MagicLantern did not boot correctly, this text will be missing.

If you have never used the camera, make sure to take time to familiarize yourself with the controls. The following two images should help you with using the camera.

Press the "Enter MagicLantern" button (as described on the first picture of the camera with labels), use the button in the middle of the circular rotary switch "setting values" to enable/disable a setting, press the "Detailed settings inside MagicLantern" to enter e.g. HDR Bracketing or Intervalometer settings menu and use left/right "Navigate camera menu" joystick to change the value. Then press the "Detailed settings inside MagicLantern" to close the detailed settings and "Enter MagicLantern" button to go to the main screen.

Find a place which does not have any obstructions (like trees or buildings) in the immediate surroundings of approximately 20-50 meters. Anything further

Figure A.4: An illustration of the MagicLantern software booted up correctly.

away shouldn't be too visible on the fisheye. The bigger the gap, the better the spot.

Place the tripod and extend it so the camera position will roughly match the height of your eyes. You will need to make this a little lower, since you will have to level the panoramic head (alternatively, you will need something to stand on). **Leveling is more important** than eye-height.

Before you mount the camera to the tripod, **clean the lens** with the provided lenspen brush.

**Level** the panoramic head by adjusting the Manfrotto ball head. Note that you do not need to level the tripod itself (manipulate with its legs), only the upper panoramic head. **Rotate** the head so that the flash mount is pointing north if the camera is facing upwards. Also take note of the **GPS coordinates** for the location of your shoot.

Adjust camera settings to:

1. Set the lens to **Manual Focus** – MF.
2. Set the lens to 8 mm. Make sure it is really all the way to the left.
3. Set the lens focus. Since you do not have any objects in the immediate vicinity, **infinity** should be good.
4. Set the ISO to 100, unless shooting night or very low light.
5. Find out the starting exposure setting:
   (a) For shots where the **Sun is visible**, this will be 1/8000 exposure time (lowest), Aperture 22.
   (b) **If the Sun is not visible**, you will not need such a dark exposure. Start the exposures at 1/2000, shoot one sequence and inspect it. If no highlight is clipped, use this. If there are clipped highlights (such as a sun behind the clouds), revert to 1/8000.

(c) The preview in the camera should help you determine if there are clipped highlights. Look at the histogram, there should not be any values tightly packed on the max range of the histogram.

6. Setup MagicLantern similar to Figure A.5. Navigating the MagicLantern menu is described in the following paragraph.

Figure A.5: Exposure bracketing and intervalometer settings of Magic Lantern software. The *HDR bracketing* setting signifies shooting 7 images with a 2 EV step in between each pair, starting from the darkest one and increasing the exposure. The *intervalometer* is set to shoot a sequence of pictures every two minutes.

Press the "Enter MagicLantern" button (as described on the first picture of the camera with labels), use the button in the middle of the circular rotary switch "setting values" to enable/disable a setting, press the "Detailed settings inside MagicLantern" to enter e.g. HDR Bracketing or Intervalometer settings menu and use left/right "Navigate camera menu" joystick to change the value. Then press the "Detailed settings inside MagicLantern" to close the detailed settings and "Enter MagicLantern" button to go to the main screen.

**HDR Bracketing** should be set up similar to what we see in Figure A.5. You can alter the number of exposures ($7 \times 2$ $EV$), if your scene has high/low dynamic range. Overcast days will not require 7 exposures. As a rule of thumb, $7 \times 2$ $EV$ with 1/8000 F22 start should be good on sunny days (row 1 and 3) and on overcast days use $6 \times 2$ $EV$ 1/2000 instead. You should **keep to** 2 $EV$ **steps**, unless you need to shoot really fast (very fast moving skies – in **high winds**) and a high dynamic range (exposed sun as row 1 or 3) at the same time. If that is the case, $5 \times 3$ $EV$ is equivalent to $7 \times 2$ $EV$ in exposure value.

Intervalometer must be ON, with the interval you want to shoot. Shooting every 2 minutes is a good start, yielding 30 HDR images per hour. If you are shooting a fast-changing scene like fast moving clouds, sunsets or sunrises, use a

smaller interval – 1 minute or 30 seconds. In many cases, clouds are moving fast enough to use 30 second steps.

With the exposures we already mentioned (1/8000 F22 ISO 100 and 1/2000 F22 ISO 100), the camera can easily take the sequence every 30 seconds. If you need longer exposures for low light, consider increasing the aperture (F22 → F16 or more) rather than the time.

Once you exit out of MagicLantern, you should see this screen (as can be seen in Figure A.6). Notice the new box, covering our format setting. This informs you that the next image will be shot in 101 seconds, and you have so far taken one shot. You can cancel the shoot by pressing the preview button.

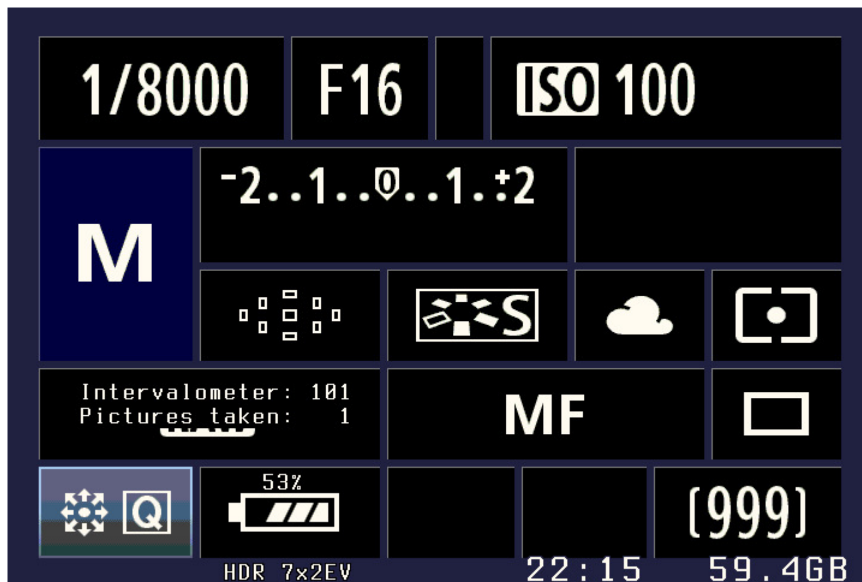



Figure A.6: An illustration of the intervalometer in action, once the shoot starts. This window informs us that the next image will be shot in 101 seconds, and we have so far taken one shot.

Mount the camera on the tripod, make sure to lock it tight in the upright position. Also make sure the end of the panoramic head is not visible in the shot. Do not forget to remove the lens covers: both the protective cover and the ”sun screen”. Also make sure to double check the leveling just before you start to shoot. The final assembled camera can be seen in Figure A.7.

We also include a one page checklist which covers every single part of the shoot we mentioned here. It is available in Attachment A.1.

## Processing the photos

The first thing to do is to download the photos from the CF card and back them up. After that, you can start with the post-processing steps.

Delete any photos taken by mistake. Also delete the photos you took when you were guessing the correct exposure (if you took any). If you leave the photos now, not only will they get processed in vain, they will also confuse PtGUI and require more manual work later.

At this point, you should have $N$ exposures per shot, and no other extra photos. To make sure, check if the number of files in your folder can be divided

Figure A.7: The camera correctly setup on the tripod.

by $N$. The $N$ is the HDR bracketing setting from MagicLantern, i.e. the 7 in $7 \times 2\ EV$.

Process the photos with RawTherapee [Horváth, 2019]. This is a very simple process, but takes a lot of time to do, so use the best machine you have for this. RawTherapee is free and open-source, so you can download it on as many machines as you want.

The process is as follows:

1. Open RawTherapee, and navigate to your taken photos
2. Select all the photos you want to process.
3. Apply our custom correction profile using right-click context menu. This is a custom profile that you can find in the digital attachments and place into `/c:/Users/<USER>/AppData/Local/RawTherapee/profiles/`. This step is illustrated in Figure A.8.
4. After a while, a check-mark will appear on the photos you selected.
5. Select them again, and choose *Put to Queue*.
6. Now, on the right side, go into the Queue manager, select `TIFF-16bit`, compressed and not-float, specify the target folder, which we call `/path/converted` and start the queue.

After you have converted the `.CR2` data into `.TIF` by using RawTherapee, you are now ready to stitch them with PtGUI [B.V.]. Unlike RawTherapee, PtGUI requires a license to use.

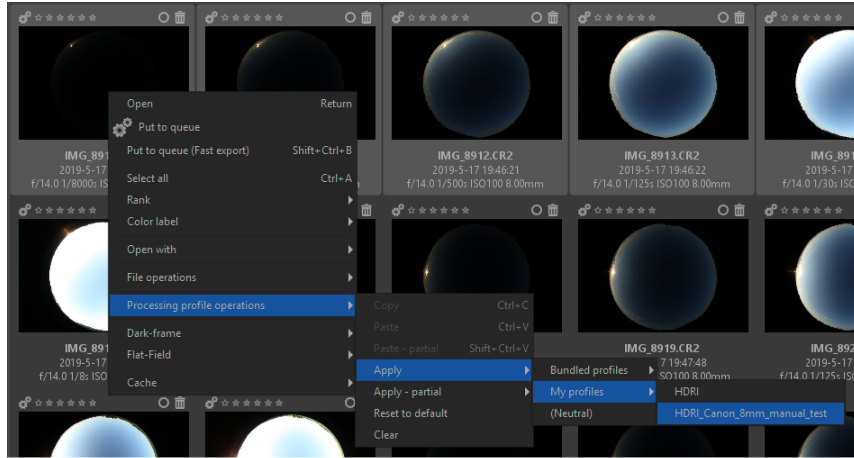



Figure A.8: A screenshot from the processing pipeline – applying our custom preset in RawTherapee.

Having your converted `.TIFs` in `/path/converted`, the next step is to convert the projection using PtGui.

First, you need to create a template for your shoot. Take the first sequence (first $N$ shots), and load them into PtGui. Apply the template from Attachment A.1 and edit the Metadata tab of PtGUI – input the **GPS** coordinates and the heading of the camera to 0 (because you aligned the camera's flash mount to north). This template is created for $N = 7$, if you have 6 photos (e.g. if you started on 1/2000), use this template and double check the result.

If you have more photos (8 or more) you only need to link the last image together with the rest in the Source Images tab of PtGui. You can find a more detailed description of PtGui's settings at the end of this manual.

Save the template as your own – named `beautiful_location_8mm_N`. Once you save the template, you can close the project without saving. Now you can apply this template to all sequences at the same time:

1. Open PtGUI
2. From Tools select *Batch Builder*. You can also use Alt + Shift + B.
3. Click *Detect Panoramas*, and set up the detection similar to what you see in Figure A.9.
4. Click Detect panoramas. You should now see all of your taken series in a list below each other.
5. Select the new template you just created – `beautiful_location_8mm_N`. Make sure that batch builder detected the panoramas correctly - $N$ shots per row, as illustrated if Figure A.10.
6. Click *Generate projects*, and after PtGUI is done generating, confirm again to stitch them all using batch stitcher.

You should now have a folder nearby with all shots correctly merged into both `.JPG` and `.EXR` formats. All photos should be $1024 \times 1024$ in size, ready to be processed by the network pipeline and used to train.

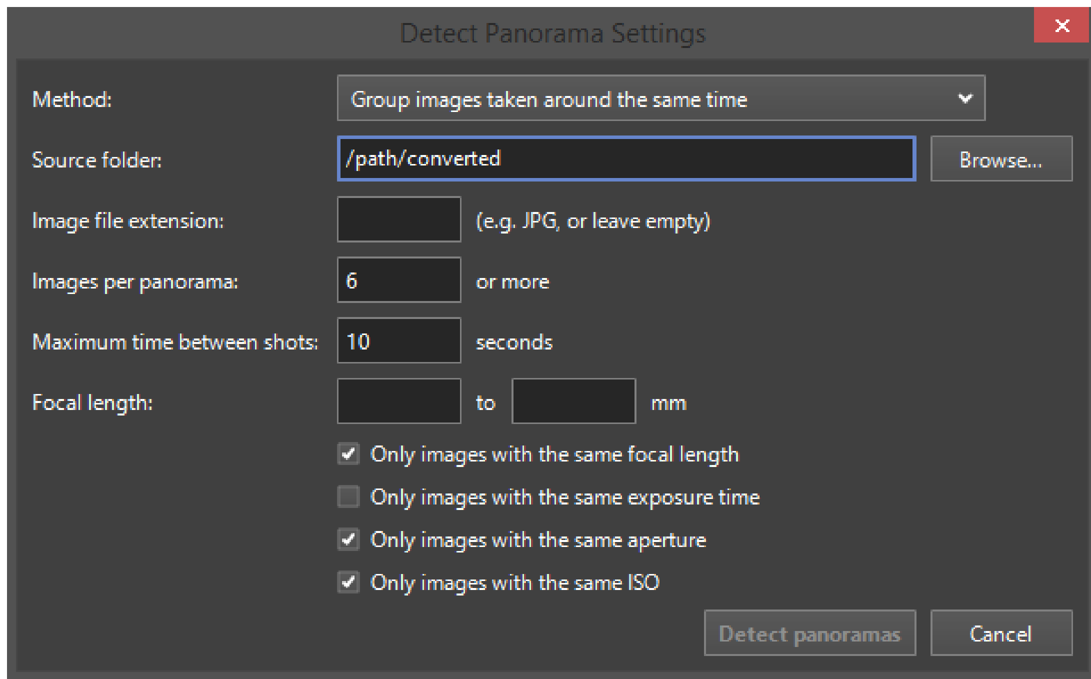



Figure A.9: Setting up the panorama detection in PtGui software. This setting should detect all panoramas we have taken.

Clean up – you might want to delete all the intermediate data, such as the `.TIF` files, as they take a lot of space, RawTherapee `.pp3` profiles, etc. Keep the `.RAW` data for further backup.

**PtGui settings** The following short list describes each sub-menu in PtGui in regards to our template. It lists everything you need to know to better navigate our template.

1. *Source images* – link a number of exposures (referred to as $N$ in this section).
2. *Lens settings* – should be loaded by default from EXIF.
3. *Crop* – the default crop is too restrictive, so if you are making a new one, extend this one a little. Make sure all the quality data (bottoms of trees, houses, etc) is included, but the refractions of the lens is not.
4. *Mask* – do not touch, no need to mask anything.
5. *Image parameters*, Control Points, Optimizer – do not touch.
6. *Exposure/HDR* – Select Group bracketed exposures, and then True HDR.
7. *Project settings* – deselect "Do Align Images" in Batch Stitcher, also make sure everything is checked in Template Behavior.
8. *Preview* – do not touch, no need to.
9. *Metadata* – GPS and heading is important here.

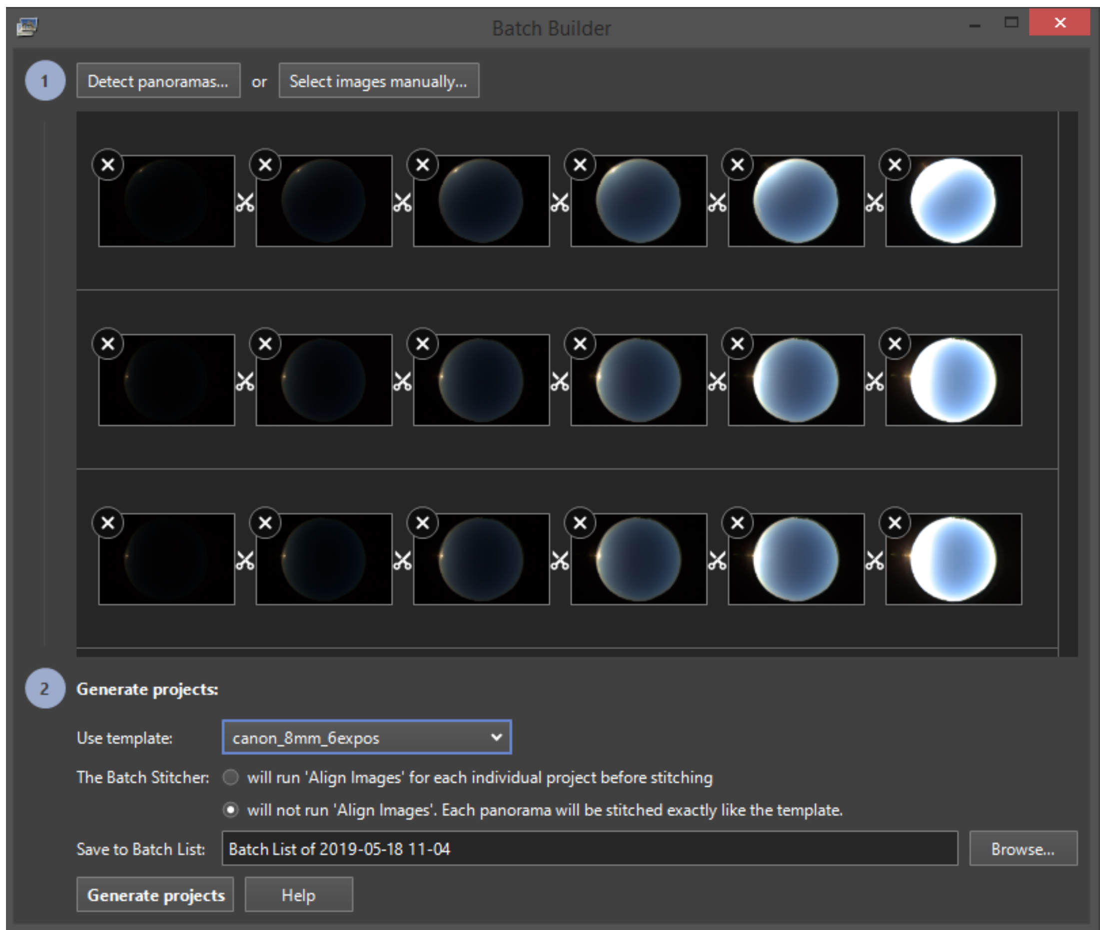



Figure A.10: Detected panoramas in PtGui. We then set the correct template below the preview window and click *Generate projects*.

10. *Create panorama* – Width and height (usually 1024), `.JPG` and `.EXR`, check both tonemapped panorama and HDR panorama.

11. *Panorama Editor* – brought up by pressing *Ctrl + E*, we select *stereographic down* in the projections list, and then use the *numerical transform* setting to change the view to *stereographic up*, by rotating it by 180°.