FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Jan Špaček

Generation of realistic skydome images

Department of Software and Computer Science Education

Advisor:	Ing. David Futschik
Supervisor:	doc. Alexander Wilkie, Dr.
Study programme:	Computer Science
Study branch:	Artificial Intelligence

Prague 2020

In: Ostrava, date: 2020-05-20 ....Jan Špaček........

Author's signature

Dedicated to the memory of Jaroslav Křivánek.

Title:	**Generation of realistic skydome images**
Author:	Jan Špaček
Department:	Department of Software and Computer Science Education
Advisor:	Ing. David Futschik Department of Computer Graphics and Interaction Faculty of Electrical Engineering Czech Technical University in Prague
Supervisor:	doc. Alexander Wilkie, Dr. Department of Software and Computer Science Education Faculty of Mathematics and Physics Charles University in Prague

Abstract: We generate realistic images of the sky with clouds using generative adversarial networks and propose a novel architecture for generating images in very high resolutions.

Keywords: Deep learning, machine learning, generative adversarial network, deep convolutional network, deep generative model, skydome, fish-eye.

Contents

Acknowledgements

This thesis is dedicated to Jaroslav Křivánek, who was the supervisor of the project before he tragically lost his life at the end of 2019.

I am very grateful to my advisor, David Futschik, who provided much-needed deep learning experience and insight, consulted the progress of the work with me for several months, and reviewed the thesis thoroughly. His help was essential for the research presented in this thesis.

I would like to thank Štěpán Hojdar, for starting this project and collecting the dataset; Chaos Czech, for providing support of this research; Alexander Wilkie, for supporting the project and my thesis; the KSVI and KSI departments, for supplying the GPUs and the thousands of kilowatt-hours of burned electricity; and the university library, for providing me with all the great books.

Na závěr děkuji svým rodičům za podporu nejen při psaní této práce, a Páji za všechen čas nestrávený nad klávesnicí a monitorem.

1. Introduction

Artists that create visualizations of architecture (see example in figure 1.1) need realistic sky as a background image (environment map). They require images that have high dynamic range (HDR), resolution at least 16K, and capture the whole skydome.

In practice, 3D artists use photographs of real skies as environment maps. Such images must be stitched from a large number of photos taken with a high-end digital camera at varying angles (because each photo only captures a small section of the sky) and multiple exposures (to obtain an HDR image from several low dynamic range images using the bracketing technique [1]). There are websites [2] that provide collections of such images that can be downloaded (see example in figure 1.2), but the number of images found on these sites is necessarily limited.

There are good computer models that can generate clear sky [3, 4], but there is no practical method that could generate a sky with clouds. Various heuristic procedural methods are used to generate clouds in computer games and similar applications, but these methods cannot produce photorealistic results.

With recent advances in volumetric rendering [5], it might soon become possible to render photorealistic clouds. However, synthesizing volumetric density data for clouds is a hard problem. There are some heuristic approaches (such as [6]) for procedural generation of volumetric clouds, but the results are of low quality. Accurate physical simulations of volumetrics such as smoke and fire produce very realistic results and are already used by 3D artists, but these methods cannot be used to simulate clouds forming, evolving and dissolving over a wide area. Moreover, providing artistic control over a full atmospheric simulation is challenging, because the atmosphere is a very complex system, so modifications of physical variables such as temperature, pressure or humidity have non-intuitive and unpredictable effects due to the chaotic nature of the atmosphere.

In this project, we explore a different approach and try to train a generative model from examples of real images of the sky, applying recent breakthroughs in deep generative models [7, 8, 9]. Using these techniques, novel content can be created quickly and no physical simulation of the atmosphere is necessary, because the model works directly on the level of HDR images. Essentially, we want to find out whether we can replace physically-based simulation and rendering derived from first principles (inductive approach) with a machine learning model trained to fit real data (deductive approach).

This thesis expands on the work by Štěpán Hojdar [10]. Our contributions are as follows:

- We significantly improve the quality of the generated images at medium resolutions (up to 1K).
- We propose a novel way of generating high-resolution images and use it to generate images at resolutions up to 4K. This approach could scale up to the target resolution of 16K-32K and is not limited to the generation of sky images, so it could be used to generate other kinds of high-resolution content.

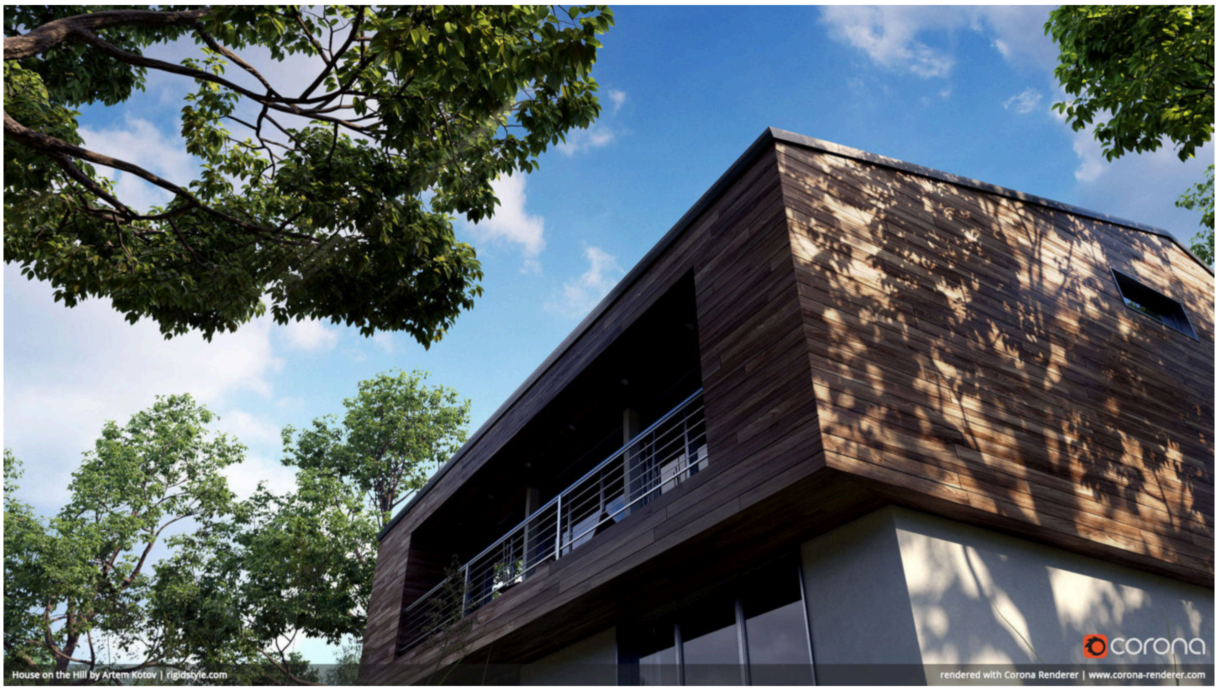

House on the Hill by Artem Kotov | rigidstyle.com
corona
rendered with Corona Renderer | www.corona-renderer.com

Figure 1.1: Architectural visualization “House on the Hill” by Artem Kotov (`http://rigidstyle.com`).

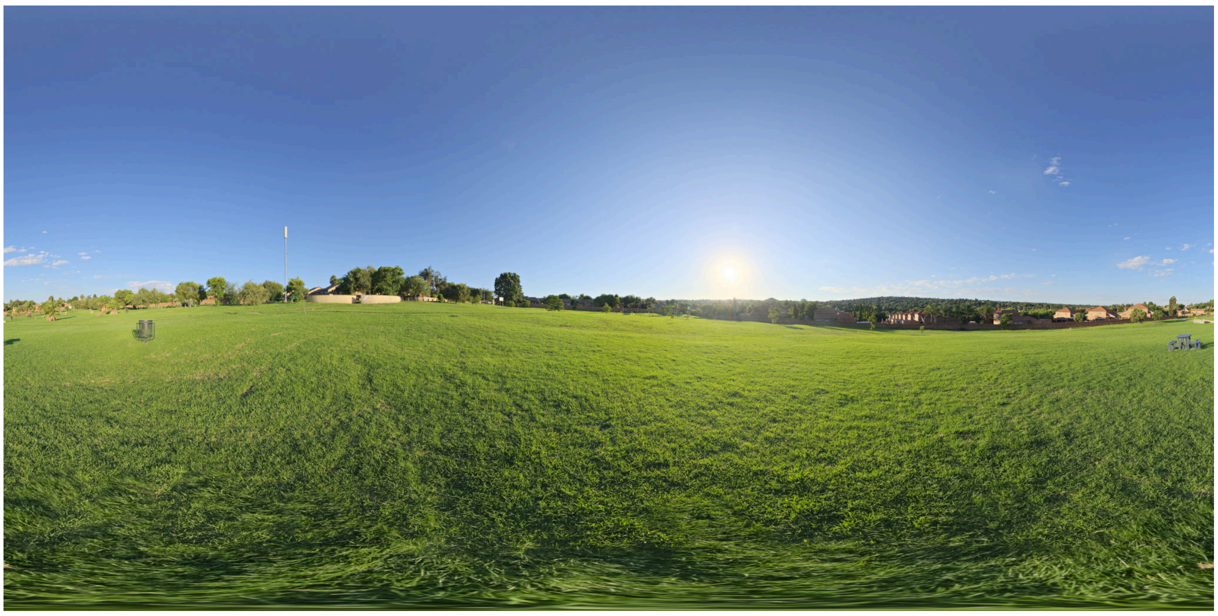


Figure 1.2: A sample HDR environment image from HDRI Haven [2] (Rooitou Park). This is a full 360° image in equirectangular projection, which also includes ground. Our aim is to reproduce only the sky.

2. Related work

This chapter will introduce the reader to a selected number of topics from deep learning that are relevant to this work. We assume prior background in computer science and basics of machine learning.

2.1 Deep learning

In this section we describe deep learning and modern neural networks. A reference that covers almost all of the material in this section is the excellent textbook by Goodfellow, Bengio and Courville [11]. We do not attempt to trace the development of basic ideas in deep learning to the original publications and refer the reader to [11] for extended biography and more detailed discussion of these topics.

2.1.1 Feedforward networks

Feedforward networks are general function approximators of the form

$$\boldsymbol{y}_1, \dots, \boldsymbol{y}_M = f(\boldsymbol{x}_1, \dots, \boldsymbol{x}_n)$$

where $\boldsymbol{x}_i$ are the inputs and $\boldsymbol{y}_j$ are the outputs. The network f is defined as a sequence of operations that compute the outputs $\boldsymbol{y}_j$ from the inputs $\boldsymbol{x}_i$. Internally, the network f may compute some intermediate values before producing the outputs $\boldsymbol{y}_j$. The structure of the operations that form the network is fixed, but we can define some intermediate values as unknown **weights $\boldsymbol{w}$** (or **parameters**), which must be learned from data.

The inputs, outputs, and intermediate values are K-dimensional arrays of real numbers or integers[1]. Traditionally, they are called **tensors** because they generalize scalars, vectors and matrices to higher dimensions, but no algebraic structure is associated with them.

Feedforward networks are used as models in machine learning. As an example, a feedforward network for classification takes the sample features as inputs $\boldsymbol{x}$ and produces a vector of class probabilities as the output $\boldsymbol{y}$.

A network can be represented as bipartite directed acyclic graph, where the two partitions are the tensors and the operations. Input tensors and parameters have no incoming edges, other tensors have exactly one incoming edge from the operation that computes their value. Each operation has one incoming edge for every input tensor and one outgoing edge for every output tensor.

Note that in this very general view, feedforward networks are not very different from SSA form [12] (used in compilers for imperative programming languages) and similar notations that represent computation as a graph.

2.1.2 Training

To use a feedforward network, we must determine values of the unknown parameters. This process is called **training**.

[1]Though integers can be used only in a limited way, because they are not differentiable; see below for description of network training.

2.1.2.1 Loss functions

We want to find "good" values of the parameters. To specify what "good" means, we need a **loss function** $\mathcal{L}(\boldsymbol{w})$ which maps network parameters $\boldsymbol{w}$ to a single scalar. Low values of the loss function correspond to better parameters, so our aim will be find parameters $\boldsymbol{w}$ that minimize the loss $\mathcal{L}(\boldsymbol{w})$.

The loss function can be arbitrary, the only restriction is that it should be differentiable with respect to the network parameters almost everywhere. In machine learning, where our goal is to fit some model to data, the loss measures the quality of the fit, which depends on the training dataset $\mathcal{D}$. In this case, the loss function is typically defined as an expectation over the dataset:

$$\mathcal{L}(\boldsymbol{w}) = \mathcal{L}_{\mathcal{D}}(\boldsymbol{w}) = \mathbb{E}_{\boldsymbol{x} \sim P_{\mathcal{D}}}[\mathcal{L}_{\boldsymbol{x}}(\boldsymbol{w})]$$

where $P_{\mathcal{D}}(\boldsymbol{x})$ is the distribution of samples $\boldsymbol{x}$ in the dataset $\mathcal{D}$ and $\mathcal{L}_{\boldsymbol{x}}(\boldsymbol{w})$ is the loss for a single sample $\boldsymbol{x}$.

We typically cannot evaluate $\mathcal{L}(\boldsymbol{w})$ directly, because the dataset is too large or even infinite. However, we can estimate the loss by sampling a list $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n \sim P_{\mathcal{D}}$ and computing the estimator $\widehat{\mathcal{L}}$:

$$\widehat{\mathcal{L}} = \operatorname*{mean}_{i} \mathcal{L}_{\boldsymbol{x}_i}(\boldsymbol{w})$$

The list $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$ is called a **minibatch** (or simply **batch**) of size N. In practice, the feedforward network can process whole minibatch at once: we stack the N tensors for each sample into a single tensor with an extra dimension of size N.

Some examples of common loss functions in machine learning are:

- For classification, where the network estimates $P(c \mid \boldsymbol{x})$ (probability of class c, given the input features $\boldsymbol{x}$), we typically use cross-entropy between the predicted distribution (produced by the network) and the ground truth distribution (from training data).
- For regression, where the network estimates $\mathbb{E}[\boldsymbol{y} \mid \boldsymbol{x}]$ (expected value of variable $\boldsymbol{y}$ given the input features $\boldsymbol{x}$), the standard choice is to use mean squared error: square of the difference between the predicted value and the true value.

In many cases, the loss is not directly the performance metric that we want to optimize, typically because the metric is not differentiable (classification accuracy), is intractable (the total reward of an agent in reinforcement learning), or because it is hard to describe formally ("the image looks good").

2.1.2.2 Optimization algorithms

Once we have the network and the loss function, training is reduced to a minimization problem:

$$\operatorname*{arg\,min}_{\boldsymbol{w}} \mathcal{L}(\boldsymbol{w})$$

This problem is well studied in the optimization literature [13, 14]. If the loss is not convex, which is typical, it is not feasible to find a global minimum, but we can find a local minimum that works well in practice. All optimization methods used in deep learning are iterative methods that update the parameters $\boldsymbol{w}$ in small steps, gradually decreasing the loss. To determine the direction and length of these steps, the methods use the gradient $\boldsymbol{g} = \nabla_{\boldsymbol{w}} \mathcal{L} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{w}}$.

If we estimate the loss $\mathcal{L}$ from a minibatch $\boldsymbol{x}_1, \dots, \boldsymbol{x}_n$, we can also estimate the gradient $\boldsymbol{g}$ using estimator $\widehat{\boldsymbol{g}}$:

$$\widehat{\boldsymbol{g}} = \operatorname*{mean}_i \nabla_{\boldsymbol{w}} \mathcal{L}_{\boldsymbol{x}_i}(\boldsymbol{w}) = \nabla_{\boldsymbol{w}} \widehat{\mathcal{L}}$$

This means that we simply treat the minibatch estimator $\widehat{\mathcal{L}}$ as the loss function. The optimization algorithms should converge to a local minimum even if these stochastic estimates are used instead of the true gradient.

We will briefly describe the most important optimization algorithms. The algorithms used in deep learning are quite straightforward, because we work with models with large number of parameters that cannot be efficiently and reliably trained using more sophisticated methods

Stochastic gradient descent (SGD): this is the simplest algorithm, where the step is simply the gradient $\widehat{\boldsymbol{g}}$ scaled with learning rate, which is given as a hyperparameter:

```
$\eta$ : learning rate
for $t = 0, 1, \dots$ do
    $\widehat{\boldsymbol{g}}_t \leftarrow \nabla_{\boldsymbol{w}} \widehat{\mathcal{L}_t}$
    $\boldsymbol{w}_t \leftarrow \boldsymbol{w}_{t-1} - \eta \cdot \widehat{\boldsymbol{g}}_t$
```

Stochastic gradient descent with momentum: to stabilize gradient descent, it is often beneficial to introduce momentum: instead of taking the step in direction of current gradient, we use an exponential moving average $\boldsymbol{v}$ of gradients from previous steps.

```
$\eta$ : learning rate
$\alpha$ : momentum
for $t = 0, 1, \dots$ do
    $\widehat{\boldsymbol{g}}_t \leftarrow \nabla_{\boldsymbol{w}} \widehat{\mathcal{L}_t}$
    $\boldsymbol{v}_t \leftarrow \alpha \cdot \boldsymbol{v}_{t-1} - \eta \cdot \widehat{\boldsymbol{g}}_t$
    $\boldsymbol{w}_t \leftarrow \boldsymbol{w}_{t-1} + \boldsymbol{v}_t$
```

RMSProp: to dynamically adapt the learning rate of different parameters, the RMSProp algorithm [15] accumulates exponential moving average $\boldsymbol{r}$ of squared gradients, which is then used to scale the update:

```
$\eta$ : learning rate
$\rho$ : decay rate
for $t = 0, 1, \dots$ do
    $\widehat{\boldsymbol{g}}_t \leftarrow \nabla_{\boldsymbol{w}} \widehat{\mathcal{L}_t}$
    $\boldsymbol{r}_t \leftarrow \rho \cdot \boldsymbol{r}_{t-1} + (1 - \rho) \cdot \widehat{\boldsymbol{g}}_t^2$ (elementwise)
    $\boldsymbol{w}_t \leftarrow \boldsymbol{w}_{t-1} - \eta \cdot \widehat{\boldsymbol{g}}_t \,/\, \sqrt{\boldsymbol{r}_t}$ (elementwise)
```

Adam: the Adam algorithm [16] ("adaptive moments") estimates the first and second moment of the gradient, so it in effect combines RMSProp with momentum:

```
$\eta$ : learning rate
$\beta_1, \beta_2$ : decay rates of the first and second moment
**for** $t = 0, 1, \ldots$ **do**
    $\widehat{\boldsymbol{g}}_t \leftarrow \nabla_w \widehat{\mathcal{L}_t}$
    $\boldsymbol{v}_t \leftarrow \beta_2 \cdot \boldsymbol{v}_{t-1} + (1 - \beta_2) \cdot \widehat{\boldsymbol{g}}_t^2$ (elementwise)
    $\boldsymbol{m}_t \leftarrow \beta_1 \cdot \boldsymbol{m}_{t-1} + (1 - \beta_1) \cdot \widehat{\boldsymbol{g}}_t$
    $\boldsymbol{w}_t \leftarrow \boldsymbol{w}_{t-1} - \eta \cdot \boldsymbol{m}_t \,/\, \sqrt{\boldsymbol{v}_t}$ (elementwise)
```

2.1.2.3 Backpropagation

The last missing piece of the puzzle is computing the gradients. Fortunately, derivatives in feedforward networks can be efficiently and exactly calculated using the method of **backpropagation**. To understand backpropagation, it is useful to return to the notion of the network as a graph described in subsection 2.1.1. We can describe the loss function $\mathcal{L}(\boldsymbol{w})$ using exactly the same kind of operations as the network, so we end up with a graph that has the network parameters $\boldsymbol{w}_i$ as inputs and the scalar loss $\mathcal{L}$ as its sole output. Our goal is to compute the derivatives $\partial\mathcal{L} \,/\, \partial\boldsymbol{w}_i$.

We will describe backpropagation as an inductive process which starts at the output and ends at the inputs. For this reason, this computation is often referred to as the **backward pass**, in contrast to the evaluation of the graph, which is the **forward pass**.

To start the induction, we note that the gradient of the output $\mathcal{L}$ is one, $\partial\mathcal{L} \,/\, \partial\mathcal{L} = 1$. Now, assume that for some operation F with inputs $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$ and outputs $\boldsymbol{y}_1, \ldots, \boldsymbol{y}_m$, we have already computed the gradients $\partial\mathcal{L} \,/\, \partial\boldsymbol{y}_j$ for all outputs $\boldsymbol{y}_j$. From the chain rule of calculus, we obtain:

$$\frac{\partial\mathcal{L}}{\partial\boldsymbol{x}_i} = \sum_j \frac{\partial\mathcal{L}}{\partial\boldsymbol{y}_j} \cdot \frac{\partial\boldsymbol{y}_j}{\partial\boldsymbol{x}_i} \tag{2.1}$$

where $\partial\boldsymbol{y}_j \,/\, \partial\boldsymbol{x}_i$ are the partial derivatives of function F. If the variable $\boldsymbol{x}_i$ is an input of multiple operations (or if it appears as an input of the same operation multiple times), the gradients from each operation are summed together. Note that when the variables are K-dimensional tensors, their gradients are also K-dimensional tensors with the same shape.

Therefore, for every function F that appears as an operation in the graph, we need to be able to compute 2.1. The backpropagation algorithm then correctly handles any combination of operations. The algorithm is completely general and can be used to evaluate gradients for any graph, not only for training feedforward networks.

2.1.3 Deep neural networks

In the previous subsections, we have described feedforward networks as arbitrary computational graphs to emphasize that the framework is quite general. However, the networks used in deep learning share many common traits and are, for historical reasons, called **neural networks**.

Neural networks are organized into **layers**, which are typically sequentially ordered. The input of the network flows into the first layer, and the output of the last layer is used as the output of the network. The values that flow inside the network are called **activations** or **features**.

2.1.3.1 Fully-connected layer

The most basic type of layer is the **fully-connected** layer (also **dense** layer or **linear** layer). The input to this layer is a vector $\boldsymbol{x}$ of size N, and the output is a vector $\boldsymbol{y}$ of size M. The layer has two parameters, a weight matrix $\boldsymbol{A}$ of shape $M \times N$ and bias $\boldsymbol{b}$ of size M, and performs affine transformation of $\boldsymbol{x}$, followed by the element-wise application of activation function f:

$$\boldsymbol{y} = f(\boldsymbol{A}\boldsymbol{x} + \boldsymbol{b})$$

The purpose of the activation function f is to make the layer non-linear. Many activation functions were proposed, but the most important ones are:

- **ReLU** (rectified linear unit; also **positive part** function in mathematics): $\mathrm{relu}(x) = x^+ = \max(x, 0)$ (see plot in figure 2.1a). This function keeps positive values unchanged, but clips all negative values to zero. The derivative is trivial ($\mathrm{relu}'(x) = [x > 0]$), so the layer is easy to train, because it is "almost linear". ReLU is typically the default choice of an activation function in deep learning.

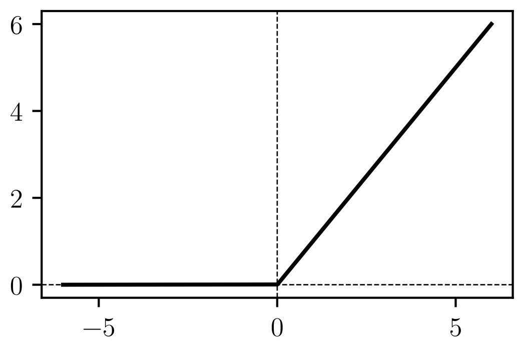
- **Leaky ReLU**: $\mathrm{lrelu}(x) = \max(x, \alpha x)$ (see plot in figure 2.1b), where α is a hyperparameter between 0 and 1. Like ReLU, this function keeps positive values unchanged, but the negative values are multiplied by α (so the negative value "leaks through" the function). The motivation behind Leaky ReLU is that some learning happens even for negative activations.

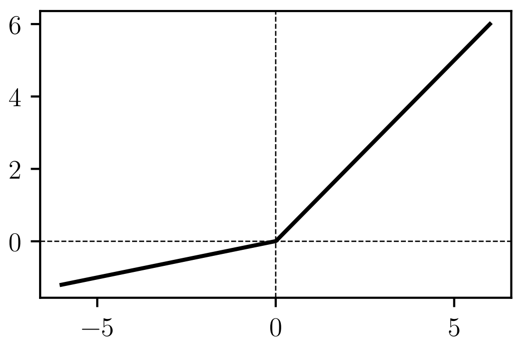
- **Sigmoid** (more precisely **logistic sigmoid**): $\mathrm{sigmoid}(x) = \exp(x) \mathbin{/} (1 + \exp(x))$ (see plot in figure 2.1c). The sigmoid function is useful to produce values interpreted as probabilities, because it maps any real number into $(0, 1)$. Note that for inputs with large absolute value, the derivative diminishes to almost zero and we say that the function **saturates**. Saturation is a problem for gradient-based learning, because the gradient flowing through the function vanishes and learning cannot proceed. For this reason, sigmoid is usually used only at the output of a network, to convert arbitrary activations into probabilities.

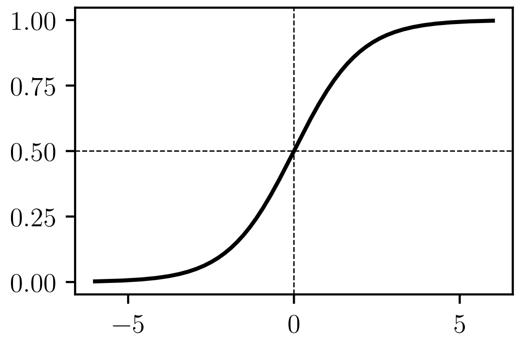
- **Softplus**: $\mathrm{softplus}(x) = \log(\exp(x) + 1)$ (see plot in figure 2.1d). The softplus function can be thought of as a "softened" variant of the positive part function x^+: for positive x, $\mathrm{softplus}(x) \approx x$, and for negative x, $\mathrm{softplus}(x) \approx 0$. It is not typically used as an activation function on its own, but it arises when we take logarithm of a sigmoid activation: $\log(\mathrm{sigmoid}(x)) = -\mathrm{softplus}(-x)$.

2.1.3.2 Batch normalization

Another type of layer is **batch normalization**, introduced by Ioffe and Szegedy in [17]. The authors observed that as the layers close to input are trained, the distributions of activations flowing into layers deeper in the network often significantly vary during training, so the deeper layers have to adapt to the changing inputs and thus learn slowly. Batch normalization is a way of normalizing the activations that helps to reduce this phenomenon, makes the effect of parameter updates more localized, accelerates learning, and makes it easier to train deeper networks.


(a) The relu function.


(b) The lrelu function.


(c) The sigmoid function.

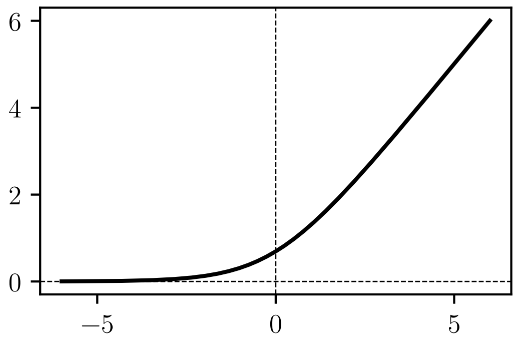

(d) The softplus function.

Figure 2.1: Plots of the most important activation functions.

In fully-connected layers, batch normalization is usually inserted after the matrix multiplication and before the activation function. For each feature i, it computes the mean and variance over all N samples in the minibatch and normalizes the input activations $\boldsymbol{x}_{\text{in}}(n, i)$ to normalized activations $\boldsymbol{y}(n, i)$ with zero mean and unit variance:

$$\begin{aligned}\boldsymbol{y}(n, i) &= \frac{\boldsymbol{x}_{\text{in}}(n, i) - \mu(i)}{\sigma(i)} \\ \mu(i) &= \operatorname*{mean}_{n} \boldsymbol{x}_{\text{in}}(n, i) \\ \sigma(i)^2 &= \operatorname*{mean}_{n} \left(\boldsymbol{x}_{\text{in}}(n, i) - \mu(i)\right)^2\end{aligned}$$

It is important that these operations are part of the computation graph of the network, so they participate in the backpropagation algorithm. In particular, this means that the gradient descent algorithm will not update the parameters in a way that simply shifts the bias or scales the variance, because such step would not change the output of the layer.

However, forcing the output of the layer to be normalized may reduce the expressive power of the network in the presence of nonlinear activation functions. For this reason, the normalized activations $\boldsymbol{y}(n, i)$ are typically scaled and shifted again using learned scale factors $\boldsymbol{a}(i)$ and biases $\boldsymbol{b}(i)$:

$$\boldsymbol{x}_{\text{out}}(n, i) = \boldsymbol{y}(n, i) \cdot \boldsymbol{a}(i) + \boldsymbol{b}(i)$$

At test time, normalization over minibatch would be problematic, so the activations are normalized using an exponentially smoothed average of $\mu(i)$ and $\sigma(i)$ computed from minibatches during training.

2.1.3.3 Regularization

The dichotomy between training performance and testing performance is a core issue in machine learning. We train our models to fit the training data, but our goal is to find models that generalize to unseen samples and perform well on the testing data, which is not available for training. **Regularization** is any method that aims to find more general models that do not necessarily perform better on the training data.

Most regularization methods define some measure of model complexity and add it to the loss function. In this way, the training explicitly searches for a solution that balances the two requirements: it must fit the data well, but it also should not be too complex and thus likely to overfit. The relative weight given to the regularization term in the loss function can be controlled and the optimal weight is typically determined through experimentation.

L^2 **regularization** (or L^2 **penalty**) is a basic method with regularization term $\mathcal{L}_{L^2}$ that penalizes the sum of squares of parameters:

$$\mathcal{L}_{L^2} = \frac{1}{2}\sum_i \|\boldsymbol{w}_i\|_2^2$$

This regularization is quite common in other areas of machine learning and statistics, where it is sometimes referred to as **ridge regression**. The basic intuition is that large parameter values are more likely to cause overfitting, so L^2 regularization pushes them towards zero.

Weight decay is a related regularization method that simply scales the weights with a constant in range $(0, 1)$ after each training iteration, which directly moves all weights towards zero. In fact, it can be shown that for gradient descent, weight decay is equivalent to L^2 regularization. However, this equivalence does not hold for adaptive optimization algorithms such as Adam [18].

The observation that models of lower complexity tend to generalize better corresponds to the principle of Occam's razor, which states that the simplest explanation is most likely to be true. Equivalently, in the Bayesian view, we can think of regularization as a prior probability distribution over the space of possible models which assigns higher probability to simple models. Learning then searches for the model that maximizes the posterior probability of the model, given the training data; the posterior probability incorporates the prior, so simple models are preferred. Indeed, the connection between some regularization methods and Bayesian priors can in some cases be formulated exactly.

Some aspects of training may also improve generalization as a side-effect, and can thus be viewed as indirect regularization methods. This includes stochastic gradient descent with large learning rate (which helps the network to avoid local minima of the loss function) and batch normalization (which helps generalization by making the layers more independent).

2.1.4 Convolutional networks

Convolutional networks (also **convnets** or convolutional neural networks, **CNNs**) are feedforward networks that are especially suited to process images. Images are represented as tensors of shape $(C, H, W)^2$, where C is the number of **channels**, W is the width and H is the height. Intermediate activations in the network, where the channels do not correspond to channels of the input image (such as RGB), are usually called **feature maps** and their channels are **features**.

2.1.4.1 Convolution

The **two-dimensional convolution** operation is the workhorse of convolutional networks. In its most basic form, it maps a tensor $\boldsymbol{x}_{\text{in}} : (C_{\text{in}}, H, W)$ to tensor $\boldsymbol{x}_{\text{out}} : (C_{\text{out}}, H - K_{\text{h}} + 1, W - K_{\text{w}} + 1)$ as follows:

$$\boldsymbol{x}_{\text{out}}(c_{\text{out}}, y, x) = \sum_{j,i,c_{\text{in}}} \boldsymbol{x}_{\text{in}}(c_{\text{in}}, y + j, x + i) \cdot \boldsymbol{k}(c_{\text{out}}, c_{\text{in}}, j, i) \tag{2.2}$$

where $\boldsymbol{k} : (C_{\text{out}}, C_{\text{in}}, K_{\text{h}}, K_{\text{w}})$ is a parameter that specifies the **kernel** (or **weights**). Each pixel (y, x) of the output image $\boldsymbol{y}$ depends only on a patch of $K_{\text{h}} \times K_{\text{w}}$ pixels of the input image, weighted by the elements of the kernel. The same kernel is used over the whole image, so the output values do not depend on the exact (y, x) coordinates. This property is called **translation invariance**, because if the input image is translated (shifted), the output image is translated by the same amount but is otherwise unchanged.

Note that the formula 2.2 in fact describes a cross correlation and not a convolution (because the kernel $\boldsymbol{k}$ is not flipped), but this distinction is not important for our purposes.

In convolutional networks, we typically use square kernels ($K = K_{\text{h}} = K_{\text{w}}$) with sizes between 2×2 and 7×7, most typical size being 3×3. A special case is the 1×1 convolution, which effectively reduces to pixel-wise matrix multiplication:

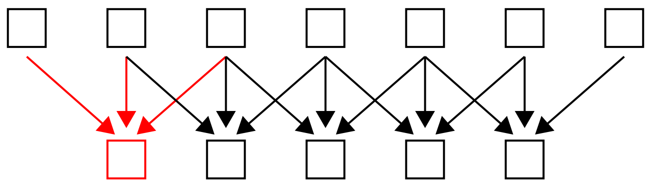
$$\boldsymbol{x}_{\text{out}}(c_{\text{out}}, y, x) = \sum_{c_{\text{in}}} \boldsymbol{x}_{\text{in}}(c_{\text{in}}, y, x) \cdot \boldsymbol{k}(c_{\text{out}}, c_{\text{in}})$$

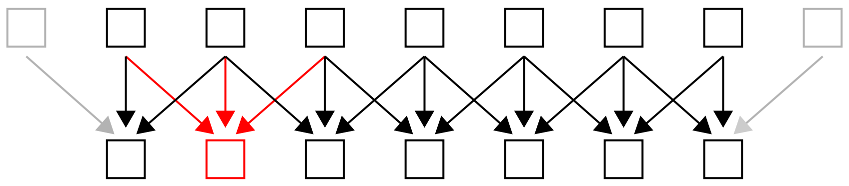
Convolutions can be defined for any number of dimensions. In particular, one-dimensional convolution maps a tensor $\boldsymbol{x} : (C_{\text{in}}, T)$ to tensor $\boldsymbol{y} : (C_{\text{out}}, T - K)$ using kernel $\boldsymbol{k} : (C_{\text{out}}, C_{\text{in}}, K)$. One-dimensional convolutions are not used in networks that process images, but they are easier to illustrate (in figure 2.2a).

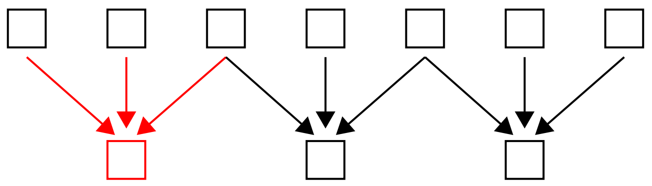
Padding (illustrated in figure 2.2b): For kernels of shape $K_{\text{h}} \times K_{\text{w}}$ larger than 1×1, the output from convolution is smaller than the input; height is reduced by $K_{\text{h}} - 1$ and width by $K_{\text{w}} - 1$. This is often undesirable, so we make the input image artificially bigger by padding it with extra pixels at the sides of the image: $(K - 1) \,/\, 2$ pixels are necessary to ensure that the output image has the same size as the input image. The extra pixels are usually filled with zeros (**zero padding**).

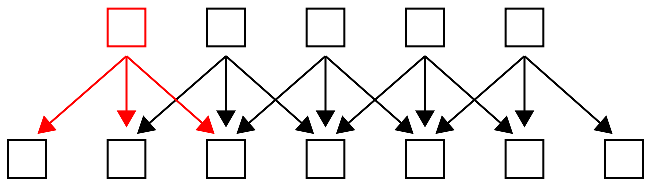
Stride (illustrated in figure 2.2c): The convolution operation may also specify a stride $S_{\text{h}}, S_{\text{w}}$, which replaces the term $\boldsymbol{x}(c_{\text{in}}, y{+}j, x{+}i)$ in formula 2.2 with $\boldsymbol{x}(c_{\text{in}}, S_{\text{h}}h{+}j, S_{\text{w}}x{+}i)$.

[2] The shape (H, W, C), which is more common in other areas of computer image processing, is also sometimes used.


(a) Convolution with kernel size 3.


(b) Convolution with kernel size 3 and zero padding 1.


(c) Convolution with kernel size 3 and stride 2.


(d) Transposed convolution with kernel size 3.

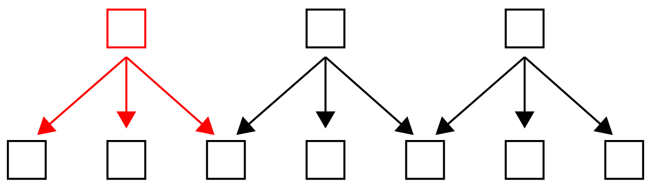

(e) Transposed convolution with kernel size 3 and stride 2.

Figure 2.2: Examples of various types of one-dimensional convolutions with kernel size 3.

In effect, we apply the kernel $\boldsymbol{k}$ only at spatial positions where y is a multiple of S_h and x is a multiple of S_w. Strided convolutions are used to downsample images: for example, a 4×4 convolution with stride 2 and padding 1 will halve the resolution of the image. Note that by setting stride $S_\mathrm{h} = S_\mathrm{w} = 1$, we obtain the basic convolution.

Transpose (illustrated in figures 2.2d and 2.2e): Convolution is a linear operator [19], therefore it has a transpose, referred to as the **transposed convolution** (some authors incorrectly use the term **deconvolution** to refer to the transpose). The transpose occurs naturally when computing the gradient, but it is sometimes useful on its own. Intuitively, the transpose turns the convolution on its head: in standard convolution, the flow of values into every output pixel is weighted by the kernel; while in transposed convolution, the flow of values out of every input pixel is weighted by the kernel.

For convolutions with stride 1, the transposed convolution is in fact almost the same as standard convolution, only the kernel is reshaped and the pixels at the border are handled differently. For higher strides, however, the transposed convolution upscales the image, so that a transposed 4×4 convolution with stride 2 and padding 1 will double the resolution of the image.

2.1.4.2 Convolution layer

The **convolution layer** in convolutional networks is a direct analogy of the fully-connected layer in neural networks, with matrix multiplication replaced by convolution. The layer has the weights $\boldsymbol{w}$ (the convolution kernel) and bias $\boldsymbol{b}$ as parameters, and it maps input feature map $\boldsymbol{x}_\mathrm{in}$ into output feature map $\boldsymbol{x}_\mathrm{out}$ as follows:

$$\boldsymbol{x}_\mathrm{out} = f(\mathrm{conv}(\boldsymbol{x}_\mathrm{in}, \boldsymbol{w}) + \boldsymbol{b})$$

where f is an activation function. Of course, the convolution operation may have stride, padding, or it may be transposed. Multiple convolution layers are often stacked into a convolution block, which contains multiple convolutions in a sequence, separated by activation functions.

2.1.4.3 Pooling layers

To produce a vector or a scalar from an image, it is necessary to downscale the feature maps several times. We can either apply a strided convolution or we can use a **pooling layer**, which combines values from multiple pixels in the pooling region (typically 2×2) into a single pixel. **Max pooling** takes the maximum value of each feature in the region of each pixel, while **average pooling** takes the mean value.

2.1.4.4 Normalization

Batch normalization can be easily applied to convolutional networks, with the statistics for each feature are calculated over all spatial positions.

Instance normalization [20] normalizes each feature over all spatial locations, separately for every sample in the minibatch. The most important advantage over batch normalization is that it can be applied when the batch size is small, and there is no difference between training and testing.

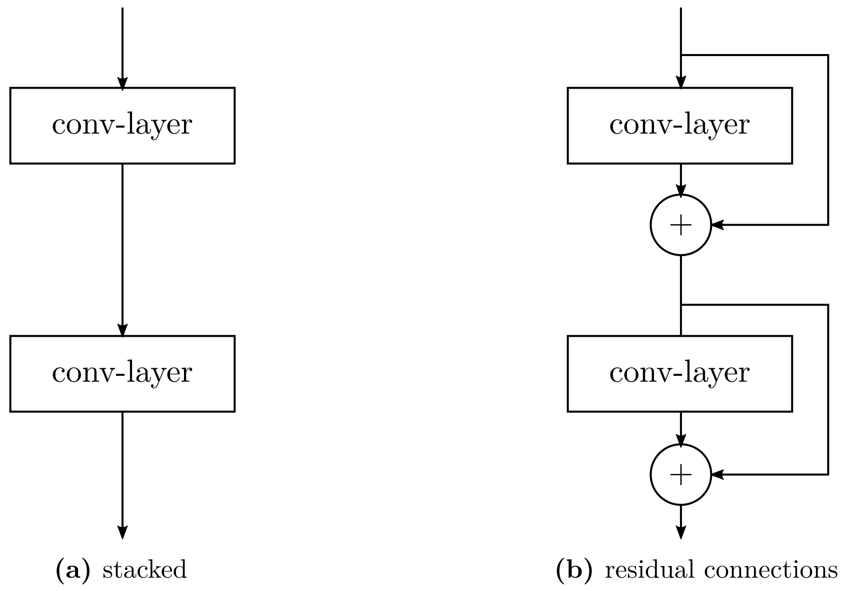


Figure 2.3: Comparison of sequentially stacked convolutions and convolutions with residual connections.

2.1.4.5 Residual connections

In a standard convolutional network with convolution layers $\mathcal{C}_1, \ldots, \mathcal{C}_N$, the layers are stacked in sequential order as follows (illustrated in figure 2.3a):

$$\boldsymbol{x}_n = \mathcal{C}_n(\boldsymbol{x}_{n-1})$$

so we have $\boldsymbol{x}_N = \mathcal{C}_N(\mathcal{C}_{N-1}(\ldots\mathcal{C}_1(\mathcal{C}_0(\boldsymbol{x}_0))\ldots))$. When residual connections [21] are used, we directly add the input of the convolution to its output (illustrated in figure 2.3b):

$$\boldsymbol{x}_n = \mathcal{C}_n(\boldsymbol{x}_{n-1}) + \boldsymbol{x}_{n-1}$$

This makes the network easier to train, because the gradient can directly propagate from $\boldsymbol{x}_n$ to $\boldsymbol{x}_{n-1}$.

2.1.4.6 Motivation

To understand how convolutional networks can learn something useful, consider the problem of deciding whether an image contains a cat (example in figure 2.4). The input to the network is the image as a tensor with RGB intensities of each pixel and the output is a probability that the image depicts a cat. Each convolution layer will then process the image into features at progressively higher level, until the last layer produces the estimated probability.

This first few layers of the network may learn the basic structure of an image, such as edges and some representation of texture. The following layers will combine these features to recognize higher-level phenomena, such as corners at positions where two edges meet ("pointy" feature) and more specific textures, such as fur ("furry" feature). Deeper layers may start to detect specific cues, such as the cat's ears ("ear" feature as a combination of "pointy" and "furry"), eyes ("eye" feature that recognizes "round" and "yellow"), and

(a) A cat[a].

[a]`https://commons.wikimedia.org/wiki/File:Felis_catus-cat_on_snow.jpg`

(b) A Coulommiers cheese[a] (not a cat).

[a]`https://commons.wikimedia.org/wiki/File:Coulommiers_lait_cru.jpg`

Figure 2.4: Comparison of a cat and a piece of cheese.

the M-shaped stripes on the forehead that are typical of tabby cats. The final layers will combine these features into an overall measure of catness.

In fact, features in any deep neural network, including convolutional networks, are notoriously hard to interpret, so the description in the previous paragraph is very idealized. For example, recent work [22] shows that contrary to the previous belief, convolutional networks recognize objects mostly by their texture and not by their shape.

2.1.4.7 Architectures

We will now describe some network architectures of convolutional networks that were created for image classification and then adapted for other purposes. Progress in this area is connected to the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) [23], a competition focused on classification accuracy on the ImageNet dataset [24] using 1000 classes.

AlexNet: The network from [25] is an early example of a deep convolutional network. It significantly outperformed other classification methods in the ILSVRC and became one of the most cited works in the field of deep learning. The network exhibits many properties of modern networks, such as ReLU non-linearity and GPU training, but with 5 convolution layers, it is quite shallow by today's standards. The convolutions are followed by three fully-connected layers, which transform the feature maps into class probabilities.

VGG: The VGG network [26] simplified the architecture by using only 3×3 convolutions and 2×2 max-pooling, and reached the depth of up to 16 convolutions, followed by 3 fully-connected layers (which account for most of the parameters of the network).

ResNet: Residual connections were introduced in [21, 27] to create the ResNet architecture. Two or three convolution layers form a block and a residual connection is routed around the block. This improves the flow of gradient in the network and allowed the authors to train extremely deep networks with up to 152 layers. The convolutions end with global average pooling and a single fully-connected layer that produces the class probabilities.

The increased depth allows the network to achieve higher accuracy than VGG, even though it uses fewer parameters.

2.2 GAN

Generative adversarial networks (**GAN**s) were introduced by Goodfellow et al. in [28] as a general framework for training a generative model of a probability distribution $P(\boldsymbol{x}) = P_{\text{real}}(\boldsymbol{x})$ for which we do not have an explicit model, but a dataset of samples $\boldsymbol{x}$ from $P_{\text{real}}(\boldsymbol{x})$ is available. We will describe GANs in detail because we use this approach in our work to model the sky.

A GAN consists of two feedforward networks, the generator G and the discriminator D. The generator is a function $\boldsymbol{x}_{\text{fake}} = G(\boldsymbol{z})$, where $\boldsymbol{z}$ is a **latent vector** that is sampled from the latent distribution $\boldsymbol{z} \sim P_{\text{z}}(\boldsymbol{z})$. The latent distribution is fixed by the designer of the network, typical choices are standard Gaussian or uniform distributions. Our aim is to make the distribution $P_{\text{fake}}(\boldsymbol{x})$ generated by $\boldsymbol{x}_{\text{fake}} = G(\boldsymbol{z}), \boldsymbol{z} \sim P_{\text{z}}(\boldsymbol{z})$ equivalent to the true data distribution $P_{\text{real}}(\boldsymbol{x})$, so that the generator G can be used to synthesize samples from $P_{\text{real}}(\boldsymbol{x})$ using $\boldsymbol{z}$ as a source of randomness.

The purpose of the discriminator network D is to act as a loss function for the training of G. The discriminator is trained as a classifier to distinguish between the real samples $\boldsymbol{x}_{\text{real}}$ from $P_{\text{real}}(\boldsymbol{x})$ (sampled from the dataset) and the fake samples $\boldsymbol{x}_{\text{fake}}$ synthesized by the generator. The output $D(\boldsymbol{x})$ of the discriminator is a **score** that measures how real or fake the input $\boldsymbol{x}$ looks to the discriminator. Therefore, we obtain a computable measure of “realness” of any given sample $\boldsymbol{x}$, which can be used for training the generator G to produce fake samples that cannot be told apart from the real samples by the discriminator.

The discriminator D is trained as a standard classifier network. We draw a sample $\boldsymbol{x}_{\text{real}}$ from the dataset and another sample $\boldsymbol{x}_{\text{fake}}$ from the generator (using $\boldsymbol{x}_{\text{fake}} = G(\boldsymbol{z}), \boldsymbol{z} \sim P_{\text{z}}(\boldsymbol{z})$), and train D to assign high scores to $\boldsymbol{x}_{\text{real}}$ and low scores to $\boldsymbol{x}_{\text{fake}}$. Note that during training of D, the generator G serves only as a means to generate the fake sample, we do not backpropagate through G and do not update its weights.

$$\begin{aligned}\text{train } D : \text{maximize } & D(\boldsymbol{x}_{\text{real}}), \text{minimize } D(\boldsymbol{x}_{\text{fake}}) \\ & \text{where } \boldsymbol{x}_{\text{real}} \sim P_{\text{real}}(\boldsymbol{x}),\ \boldsymbol{x}_{\text{fake}} = G(\boldsymbol{z}),\ \boldsymbol{z} \sim P_{\text{z}}(\boldsymbol{z})\end{aligned}$$

To train the generator, we generate a random latent sample $\boldsymbol{z}$ and evaluate $\boldsymbol{x}_{\text{fake}} = G(\boldsymbol{z})$. Then we compute the discriminator score $D(\boldsymbol{x}_{\text{fake}}) = D(G(\boldsymbol{z}))$ for the fake sample and train G to maximize this score. Note that we backpropagate through D and G, but we only update the parameters of G, the parameters of D are not modified.

$$\begin{aligned}\text{train } G : \text{maximize } & D(G(\boldsymbol{z})) \\ & \text{where } \boldsymbol{z} \sim P_{\text{z}}(\boldsymbol{z})\end{aligned}$$

In each iteration of the training process, we train both D and G, trapping them in an endless adversarial game: as D gets better at separating the real and fake samples, G gets better at synthesizing samples that seem more real, until the networks reach an equilibrium: G learns to produce perfect samples and D is no longer able to separate the real from the fake.

In reality, there is no guarantee that the two networks reach this equilibrium. A common failure is **mode collapse**, a situation where the generator G learns to ignore the latent sample $\boldsymbol{z}$ and produces a single sample $\boldsymbol{x}^*$ that (approximately) maximizes $D(\boldsymbol{x}^*)$; the networks then engage in a game of mouse and cat, where D attempts to "catch" $\boldsymbol{x}^*$ and assign a low score to it, while G "escapes" by quickly moving $\boldsymbol{x}^*$ to a region of higher score.

Another situation where the training diverges is when D becomes so strong that it confidently scores all outputs of G as uniformly fake and does not provide a useful gradient: in that case G cannot learn and the generated samples do not improve. On the other hand, when D is too weak, it may not be able to distinguish the real from the fake, so the generator G will not improve.

2.2.1 Loss

The stability of training of GANs is highly influenced by the exact formulation of the loss function used to train D and G. Various loss functions and regularization methods have been proposed, we are going to describe the most important contributions.

2.2.1.1 Logistic loss

Logistic loss (also saturating logistic loss, saturating loss, standard loss, SGAN) is the original loss function proposed in [28]. We define a value function $V(D, G)$:

$$\begin{aligned} V(D, G) &= \log(\mathrm{sigmoid}(D(\boldsymbol{x}_{\mathrm{real}}))) + \log(1 - \mathrm{sigmoid}(D(G(\boldsymbol{z}))) \\ &\quad \text{where } \boldsymbol{x}_{\mathrm{real}} \sim P_{\mathrm{real}}(\boldsymbol{x}),\ \boldsymbol{z} \sim P_{\mathrm{z}}(\boldsymbol{z}) \end{aligned}$$

We interpret $\mathrm{sigmoid}(D(\boldsymbol{x}))$ as the probability that $\boldsymbol{x}$ is real according to the discriminator D. When $D(\boldsymbol{x}) > 0$, we have $\mathrm{sigmoid}(D(\boldsymbol{x})) > \frac{1}{2}$ and $\boldsymbol{x}$ would be classified as real; when $D(\boldsymbol{x}) < 0$, we have $\mathrm{sigmoid}(D(\boldsymbol{x})) < \frac{1}{2}$ and $\boldsymbol{x}$ would be classified as fake.

Note that in line with common practice in statistics and machine learning, we define the value function in terms of log-probabilities, which makes the resulting optimization problem better behaved and more numerically stable.

The discriminator is trained to maximize $V(D, G)$: it simultaneously maximizes

$$\mathrm{sigmoid}(D(\boldsymbol{x}_{\mathrm{real}})) = P(\boldsymbol{x}_{\mathrm{real}} \text{ is real} \mid D)$$

and

$$1 - \mathrm{sigmoid}(\boldsymbol{x}_{\mathrm{fake}}) = P(\boldsymbol{x}_{\mathrm{fake}} \text{ is fake} \mid D)$$

so that it learns to correctly classify the samples. Negating the value function to obtain loss function and rewriting log-sigmoids as softpluses, we obtain:

$$\begin{aligned} \mathcal{L}(D) &= -\log(\mathrm{sigmoid}(D(\boldsymbol{x}_{\mathrm{real}}))) + \log(1 - \mathrm{sigmoid}(D(G(\boldsymbol{z})))) \\ &= \mathrm{softplus}(-D(\boldsymbol{x}_{\mathrm{real}})) + \mathrm{softplus}(D(\boldsymbol{x}_{\mathrm{fake}})) \\ &\quad \text{where } \boldsymbol{x}_{\mathrm{real}} \sim P_{\mathrm{real}}(\boldsymbol{x}),\ \boldsymbol{x}_{\mathrm{fake}} = G(\boldsymbol{z}),\ \boldsymbol{z} \sim P_{\mathrm{z}}(\boldsymbol{z}) \end{aligned}$$

On the other hand, the generator is trained to minimize $V(D, G)$: it minimizes

$$1 - \mathrm{sigmoid}(D(G(\boldsymbol{z})) = P(G(\boldsymbol{z}) \text{ is fake} \mid D)$$

so that it learns to generate $\boldsymbol{x}_{\text{fake}} = G(\boldsymbol{z})$ that are not classified as fake by the discriminator. Note that the first term of $V(D, G)$ does not depend on G, so we can drop it from the loss function of G to obtain:

$$\begin{aligned}\mathcal{L}(G) &= \log(1 - \text{sigmoid}(D(G(\boldsymbol{z})))) = \\ &= -\text{softplus}(D(G(\boldsymbol{z}))) \\ &\text{where } \boldsymbol{z} \sim P_{\text{z}}(\boldsymbol{z})\end{aligned}$$

2.2.1.2 Non-saturating logistic loss

Non-saturating logistic loss (also non-saturating loss, modified standard loss) solves the saturation problem of the previous loss. The problem is clear from the plot of the softplus function (in figure 2.1d). The slope of the function is close to 1 for positive values, but it diminishes almost to zero for negative values. Therefore, when the discriminator learns to assign negative values to $D(\boldsymbol{x}_{\text{fake}})$, the term softplus$(D(G(\boldsymbol{z}))$ in $\mathcal{L}(G)$ is pushed to the region of small slope of the softplus function, the gradients of G with respect to $\mathcal{L}(G)$ vanish and G stops learning.

This problem was already discussed in the original paper [28] and the proposed fix is simply to replace the generator loss with

$$\begin{aligned}\mathcal{L}(G) &= \text{softplus}(-D(G(\boldsymbol{z}))) \\ &\text{where } \boldsymbol{z} \sim P_{\text{z}}(\boldsymbol{z})\end{aligned}$$

Note that $\mathcal{L}(D)$ saturates when D becomes very good ($D(\boldsymbol{x}_{\text{real}})$) is high and $D(\boldsymbol{x}_{\text{fake}})$ is low), while the modified $\mathcal{L}(G)$ saturates when G becomes very good ($D(\boldsymbol{x}_{\text{fake}})$ is high). This effect helps to stabilize the training, because it is now harder for one network to overwhelm the other network.

2.2.1.3 LSGAN

Least-squares loss (**LSGAN**), introduced by Mao et al. in [29, 30], discards the probabilistic interpretations of the score from the discriminator and simply trains the discriminator to assign score $+1$ to real images and -1 to fake images:

$$\begin{aligned}\mathcal{L}(D) &= (D(\boldsymbol{x}_{\text{real}}) - 1)^2 + (D(\boldsymbol{x}_{\text{fake}}) + 1)^2 \\ &\text{where } \boldsymbol{x}_{\text{real}} \sim P_{\text{real}}(\boldsymbol{x}),\ \boldsymbol{x}_{\text{fake}} \sim G(\boldsymbol{z}),\ \boldsymbol{z} \sim P_{\text{z}}(\boldsymbol{z})\end{aligned}$$

A natural way to train G would be to try to make $D(G(\boldsymbol{z}))$ close to $+1$, but the authors of this loss reported in [30] that they obtained better results by training G to make $D(G(\boldsymbol{z}))$ close to 0:

$$\mathcal{L}(G) = D(G(\boldsymbol{z}))^2$$

LSGAN does not suffer from saturation and is reported to produce better behaved gradients, which make the training more stable. In our initial informal experiments, we found LSGAN to be more stable than both logistic losses (both saturating and non-saturating). However, it seems that the popularity of LSGAN waned in the last few years, being replaced with other loss functions.

2.2.1.4 WGAN

Wasserstein GAN (**WGAN**) was derived by Arjovsky et al. in [31] using a mathematical analysis of probability distances. Standard GAN can be shown to minimize the Jensen-Shannon divergence between $P_{\text{real}}(\boldsymbol{x})$ and $P_{\text{fake}}(\boldsymbol{x})$, but the authors of [31] argue that better convergence is obtained by minimizing the Wasserstein-1 distance (also called Earth-Mover or EM distance) and show that this distance may be approximated (up to a multiplicative constant) with:

$$\begin{aligned} W(P_{\text{real}}(\boldsymbol{x}), P_{\text{fake}}(\boldsymbol{x})) = & \max_{\boldsymbol{w}} \mathbb{E}[f_{\boldsymbol{w}}(\boldsymbol{x}_{\text{real}})] - \mathbb{E}[f_{\boldsymbol{w}}(\boldsymbol{x}_{\text{fake}})] \\ & \text{where } \boldsymbol{w} \in \mathcal{W},\ \boldsymbol{x}_{\text{real}} \sim P_{\text{real}}(\boldsymbol{x}),\ \boldsymbol{x}_{\text{fake}} \sim P_{\text{fake}}(\boldsymbol{x}) \end{aligned}$$

where $\{f_{\boldsymbol{w}}, \boldsymbol{w} \in \mathcal{W}\}$ is a family of K-Lipschitz continuous functions. Therefore, $f_{\boldsymbol{w}}$ plays the role of the discriminator D, which is trained to maximize $\mathbb{E}[D(\boldsymbol{x}_{\text{real}})] - \mathbb{E}[D(\boldsymbol{x}_{\text{fake}})]$ in order to estimate the Wasserstein distance. Negating W to transform maximization to minimization, we obtain the loss function:

$$\begin{aligned} \mathcal{L}(D) = & D(\boldsymbol{x}_{\text{fake}}) - D(\boldsymbol{x}_{\text{real}}) \\ & \text{where } \boldsymbol{x}_{\text{real}} \sim P_{\text{real}}(\boldsymbol{x}),\ \boldsymbol{x}_{\text{fake}} = G(\boldsymbol{z}),\ \boldsymbol{z} \sim P_{\text{z}}(\boldsymbol{z}) \end{aligned}$$

The generator G, on the other hand, is trained to minimize the Wasserstein distance to make $P_{\text{fake}}(\boldsymbol{x})$ close to $P_{\text{real}}(\boldsymbol{x})$. The first term of $W(P_{\text{real}}(\boldsymbol{x}), P_{\text{fake}}(\boldsymbol{x}))$ can be dropped because it does not depend on G, so the generator is trained to minimize:

$$\begin{aligned} \mathcal{L}(G) = & -D(G(\boldsymbol{z})) \\ & \text{where } \boldsymbol{z} \sim P_{\text{z}}(\boldsymbol{z}) \end{aligned}$$

We can see that the loss functions that we obtained are almost trivial: D learns to assign low scores to fakes and high scores to reals, and G learns to generate samples that produce high scores. However, to make the Wasserstein approximation valid, we must ensure that D remains approximately K-Lipschitz continuous. The original paper [31] uses weight clipping: the weights of the network D are clipped to a small range (such as $[-0.01, 0.01]$), which can be shown to enforce the Lipschitz constraint.

However, as the authors of [31] themselves stated, “weight clipping is a clearly terrible way to enforce a Lipschitz constraint”. Indeed, the method was quickly improved by Gulrajani et al. in **WGAN-GP** [32], which uses **gradient penalty** to enforce the Lipschitz constraint. The gradient penalty places a soft constraint on the norm of the discriminator with respect to its input:

$$\begin{aligned} \mathcal{L}_{\text{gp}} = & \|\nabla_{\widehat{\boldsymbol{x}}} D(\widehat{\boldsymbol{x}})\|_2^2 \\ & \text{where } \widehat{\boldsymbol{x}} \sim \text{uniform}(\boldsymbol{x}_{\text{real}}, \boldsymbol{x}_{\text{fake}}),\ \boldsymbol{x}_{\text{real}} \sim P_{\text{real}}(\boldsymbol{x}),\ \boldsymbol{x}_{\text{fake}} \sim P_{\text{fake}}(\boldsymbol{x}) \end{aligned}$$

where we evaluate the penalty at points $\widehat{\boldsymbol{x}}$ randomly sampled on line segments between pairs of real and fake samples. We therefore obtain the WGAN-GP discriminator loss:

$$\begin{aligned} \mathcal{L}(D) = & D(\boldsymbol{x}_{\text{fake}}) - D(\boldsymbol{x}_{\text{real}}) + \lambda \cdot \|\nabla_{\widehat{\boldsymbol{x}}} D(\widehat{\boldsymbol{x}})\|_2^2 \\ & \text{where } \widehat{\boldsymbol{x}} \sim \text{uniform}(\boldsymbol{x}_{\text{real}}, \boldsymbol{x}_{\text{fake}}),\ \boldsymbol{x}_{\text{real}} \sim P_{\text{real}}(\boldsymbol{x}),\ \boldsymbol{x}_{\text{fake}} \sim P_{\text{fake}}(\boldsymbol{x}) \end{aligned}$$

where λ is the weight given to the gradient penalty.

We can see that the scores $D(\boldsymbol{x})$ from the discriminator have no interpretation in terms of probabilities and their scale is determined solely by the gradient penalty, which ensures that the discriminator cannot reap unlimited rewards by making the absolute value of the scores arbitrarily large. Also note that there are no issues with saturation of $\mathcal{L}(D)$ and $\mathcal{L}(G)$, because in both cases the gradient from the loss flows directly to the network.

Note that $\mathcal{L}(D)$ does not change when a constant is added to the output of $D(\boldsymbol{x})$, so the outputs may "drift" very far away from zero, causing numerical problems. Therefore, in practice we add "epsilon penalty" to the loss in the form of $\epsilon \cdot D(\boldsymbol{x}_{\text{real}})^2$, where ϵ is a very small constant, to limit the drift.

2.2.1.5 R_1 and R_2 regularization

The convergence of GAN training was explored by Mescheder et al. in [33]. They show that several GAN loss functions, including WGAN and WGAN-GP, are not always locally convergent, demonstrate their divergence on simple examples, and discuss several regularization strategies. They propose R_1 and R_2 **regularization**, which place a penalty on the norm of the gradient of $D(\boldsymbol{x})$ with respect to $\boldsymbol{x}$:

$$\begin{aligned}
\mathcal{L}_{R_1} &= \|\nabla_{\boldsymbol{x}_{\text{real}}} D(\boldsymbol{x}_{\text{real}})\|_2^2 \\
&\quad \text{where } \boldsymbol{x}_{\text{real}} \sim P_{\text{real}}(\boldsymbol{x}) \\
\mathcal{L}_{R_2} &= \|\nabla_{\boldsymbol{x}_{\text{fake}}} D(\boldsymbol{x}_{\text{fake}})\|_2^2 \\
&\quad \text{where } \boldsymbol{x}_{\text{fake}} \sim P_{\text{fake}}(\boldsymbol{x})
\end{aligned}$$

The R_1 regularization penalizes the discriminator gradients on the real distribution, while the R_2 regularization penalizes the gradients on the current fake distribution.

2.2.2 DCGAN

GANs can be readily adapted to images. The original GAN paper [28] presented experiments that generated images using fully-connected and convolutional networks, but the architecture was improved by Radford et al. in **DCGAN** (Deep Convolutional GAN) [34], which utilized several improvements from the state-of-art classification networks, such as eliminating fully-connected layers, batch normalization, ReLU activation function, and greater depth.

The discriminator D (illustrated in figure 2.5a) is quite similar to the convolutional nets that achieved great performance in image classification. The input RGB image is passed through multiple convolution layers which gradually decrease the resolution of the features using strided convolutions. Leaky ReLU is used as the activation function, and batch normalization is employed to normalize the features after each convolution block except the first one. The output of the last convolution layer with resolution 4×4 is flattened from three dimensions to one and processed by a single fully-connected layer, which outputs the score of the input image.

The generator G (illustrated in figure 2.5b) is almost a mirror image of the discriminator D. The latent vector $\boldsymbol{z}$ is processed by a single fully-connected layer and reshaped to a 4×4 feature map, which is then processed by the convolutional layers. The convolutions are transposed convolutions with stride, so they perform upsampling. ReLU is used as the activation function, and all convolutions except the last one are followed by batch normalization.

The DCGAN paper also demonstrated that the generator learned to capture some semantic aspects of the images in the latent vectors. For example, on a network trained to generate human faces, we can find the vector that corresponds to the concepts of smiling woman, neutral woman, and neutral man (see figure 2.6). By computing the vector smiling woman − neutral woman + neutral man, we obtain an image that plausibly represents the concept smiling man This is similar to the semantic structure found in word embeddings generated by the word2vec algorithm [35] in the field of natural language processing.

2.2.3 ProGAN

An important milestone in the evolution of GANs was the progressive GAN (**ProGAN**) architecture by Karras et al. [7] from Nvidia Research. This work combined several improvements of the networks and the training method to generate high-quality images in large resolutions up to 1024 pixels (see figure 2.7 for some examples).

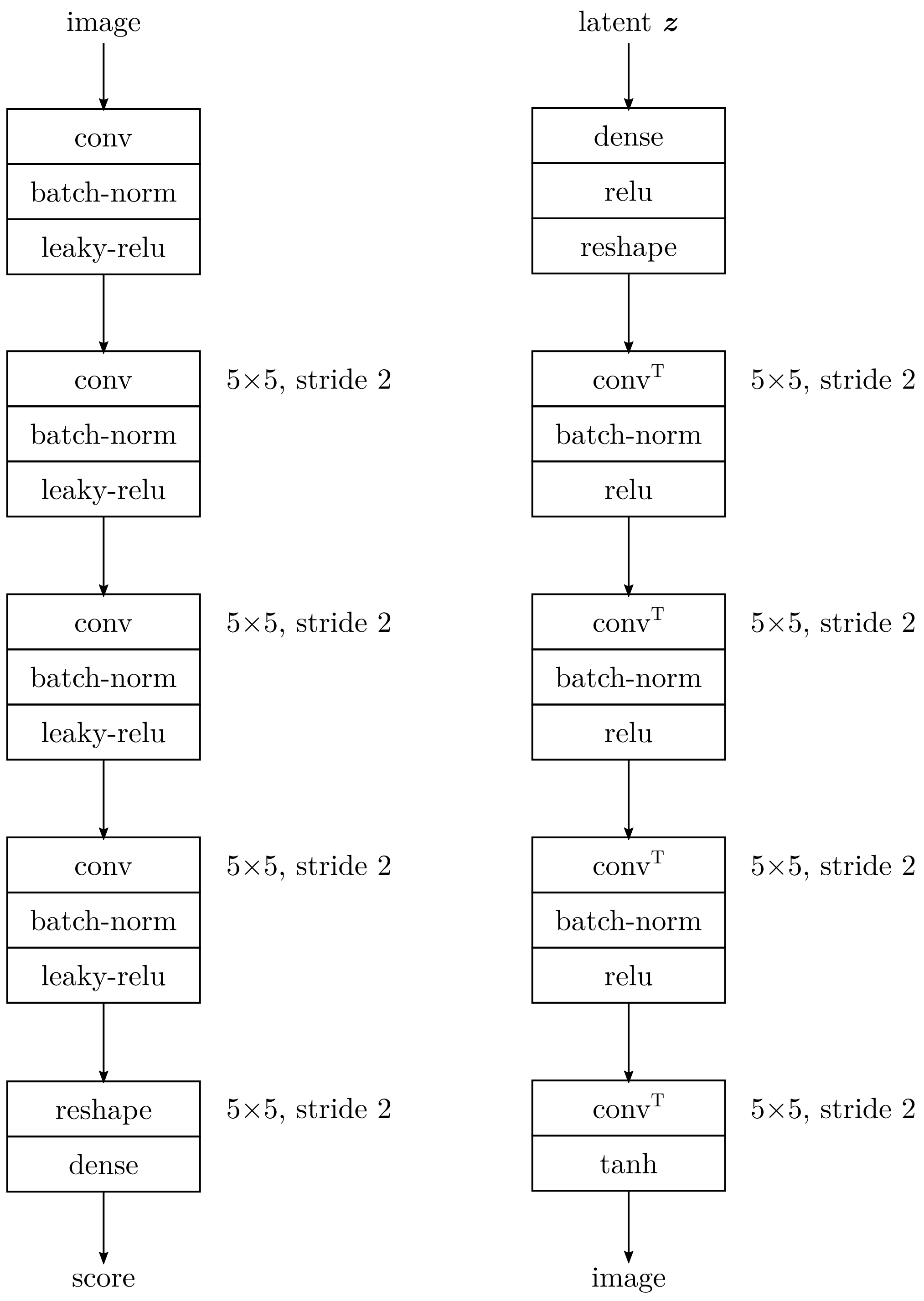
Progressive growing: The main contribution of this work is the method of progressive growing (illustrated in figure 2.8). At the beginning, the networks are trained with images at resolution 4×4, so both the generator G and the discriminator D are quite small. During training, the resolution of the networks is gradually increased by adding new convolution blocks to both networks, until both networks are grown to their full size. The output of blocks at higher resolution is gradually blended with the output of blocks at lower resolution in both networks, so that there are no discontinuous changes that could disrupt training (as seen in figure 2.8a). For each resolution, we have a block that projects feature maps into RGB intensities in G (G-to-rgb in figure 2.8a) and a block that projects RGB intensities into feature maps in D (D-to-rgb in figure 2.8a).

This method is an example of curriculum learning, a general principle of starting with easy tasks (low resolution) and gradually moving to harder tasks (high resolution). This makes the training more stable and also improves the quality of the generated images. Another benefit is that the training iterations at lower resolution are quite fast, so we can train the networks for more iterations in the same amount of time.

Minibatch standard deviation: to increase the amount of variation in the generated images, the discriminator is augmented with a minibatch standard deviation layer. This layer computes standard deviation of every feature at every spatial position over all images in the minibatch. The deviations are then all averaged together to produce a single scalar value, which is inserted back into the discriminator as another feature, replicated over all spatial positions and over all images in the minibatch.

This layer allows the discriminator to perceive the amount of variation among the samples in the minibatch, so it encourages the generator to increase variation and avoid mode collapse.

Equalized learning rate: Linear layers in deep feedforward networks (both fully-connected and convolutional) are usually initialized from the normal distribution scaled so that the layer approximately preserves the variation of the input signal [36]. Let $\boldsymbol{x}(j)$ be



(a) Each block in the discriminator contains a 5×5 convolution with stride 2, which downscales the image. The convolutions are followed with batch normalization and the Leaky ReLU activation function. In the last block, the feature map is flattened into a vector and processed with a fully-connected layer to yield the score.

(b) The generator is a mirror image of the discriminator. Each block contains a 5×5 transposed convolution with stride 2, which upscales the image. ReLU is used as the activation function. In the first block, the 100-dimensional latent $\boldsymbol{z}$ is processed with a fully-connected layer and reshaped into feature map.

Figure 2.5: Architecture of DCGAN [34].

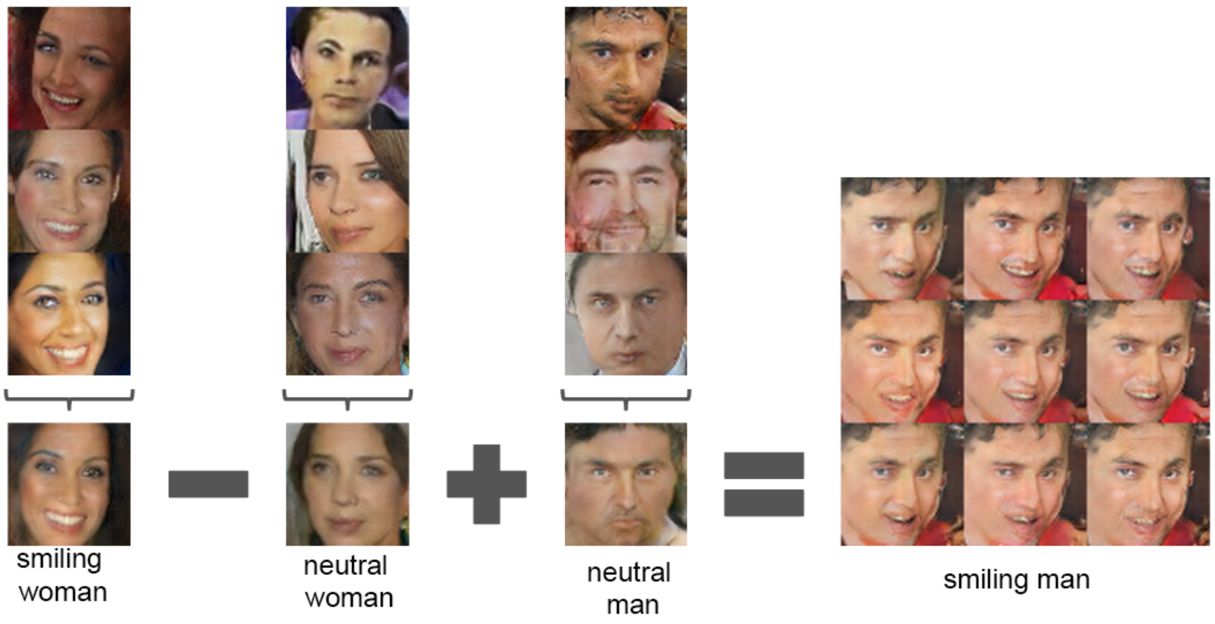


Figure 2.6: Latent vectors for smiling woman, neutral woman, and neutral man were obtained by averaging latents for multiple images that displayed these concepts. Using arithmetic on these vectors, the latent vector for smiling man was computed. (Reproduced from figure 7 of [34].)

the input to a fully-connected layer of size M, let $\boldsymbol{y}(i)$ be the output of size N, and let $\boldsymbol{A}(i,j)$ be the weight matrix of size $N \times M$, so we have:

$$\boldsymbol{y}(i) = \sum_{0 \le j < N} \boldsymbol{A}(i,j) \cdot \boldsymbol{x}(j)$$

Assume that $\operatorname{var}(\boldsymbol{x}(j)) = \sigma^2$, $\operatorname{var}(\boldsymbol{A}(i,j)) = a^2$, $\mathbb{E}[\boldsymbol{x}(j)] = \mathbb{E}[\boldsymbol{A}(i,j)] = 0$, and that elements of $\boldsymbol{x}$ and $\boldsymbol{A}$ are uncorrelated. We then have:

$$\begin{aligned}\operatorname{var}(\boldsymbol{y}(i)) &= \sum_{0 \le j < N} \operatorname{var}(\boldsymbol{A}(i,j)) \cdot \operatorname{var}(\boldsymbol{x}(j)) = \\ &= \sum_{0 \le j < N} a^2 \cdot \sigma^2 = \\ &= N a^2 \sigma^2\end{aligned}$$

Therefore, we can ensure that $\operatorname{var}(\boldsymbol{y}(i)) = \operatorname{var}(\boldsymbol{x}(j))$ by setting $a = \frac{1}{\sqrt{N}}$, which means that we initialize the weight matrix $\boldsymbol{A}(i,j)$ from $\mathcal{N}(0, \frac{1}{N})$. For a two-dimensional convolution with N input channels and kernel $K \times K$, a similar derivation shows that we should initialize the weights from $\mathcal{N}\left(0, \frac{1}{K^2 N}\right)$. This method is called the Kaiming He initialization after the first author of [36].

This initialization method ensures that the scale of network activations is approximately the same in the whole network; otherwise, the activations could either grow or diminish exponentially as they flow through the network, so some layers of the network would learn much faster than other layers, because the magnitude of stochastic gradient descent update is directly proportional to the magnitude of the gradient.

However, the authors of the ProGAN paper note that this reasoning does not apply to Adam and other optimizers that use the gradient to determine only the direction of the

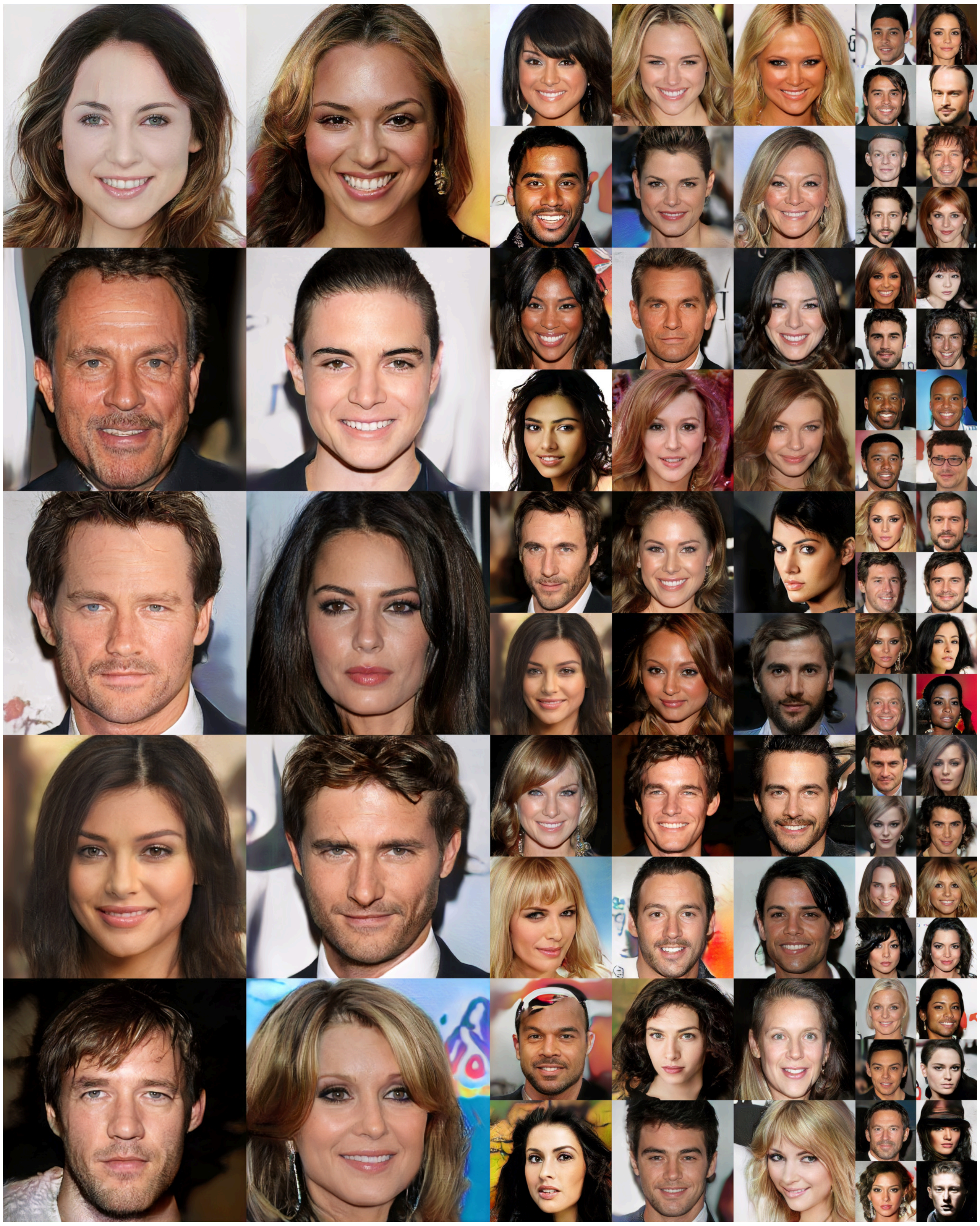
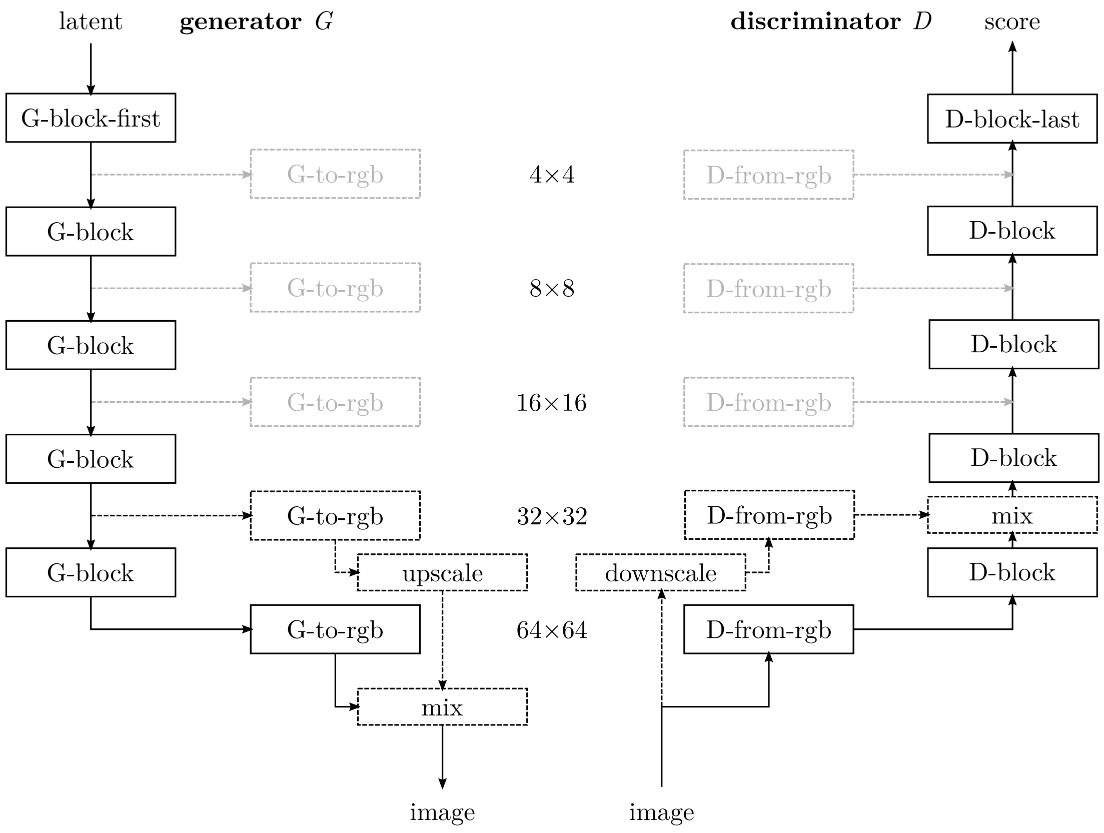
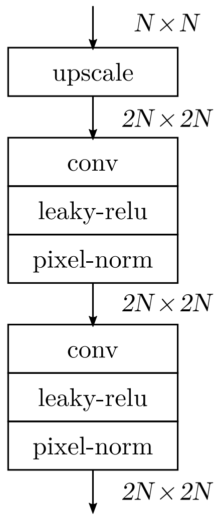


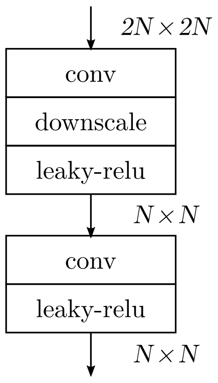
Figure 2.7: Uncurated images of synthetic celebrities generated by the ProGAN network. (Reproduced from figure 11 of [7].)



(a) Progressive growing. The dashed paths indicate the low-resolution image that is currently mixed with the high-resolution image to smooth the training. The gray paths have been used earlier in the training, but are not relevant anymore.



(b) Convolution block in G (G-block).



(c) Convolution block in D (D-block).

Figure 2.8: Architecture of ProGAN [7].

update, but the magnitude of the update is determined by the learning rate. Therefore, if the scale of the parameters in the network is not the same (as happens with the He initialization), some parameters may learn too slowly while other parameters learn too fast.

For this reason, the weights in ProGAN are initialized from $\mathcal{N}(0,1)$ and multiplied by $\frac{1}{K\sqrt{N}}$ at runtime, before they are used in the convolutions. This ensures that the scale of the activations does not spiral out of control, but all parameters have the same scale and are thus learned at the same rate.

Pixelwise feature normalization in G: neither G nor D use batch normalization, but G uses pixelwise normalization of the feature maps: for each spatial position, the vector of features is normalized to have unit length:

$$\begin{aligned}\boldsymbol{y}(c,y,x) &= \frac{\boldsymbol{x}(c,y,x)-\mu(y,x)}{\sigma(y,x)}\\ \mu(y,x) &= \operatorname*{mean}_c \boldsymbol{x}(c,y,x)\\ \sigma(y,x)^2 &= \operatorname*{mean}_c (\boldsymbol{x}(c,y,x)-\mu(y,x))^2\end{aligned}$$

Note that $\mu(y,x)$ and $\sigma(y,x)$ are computed variables, not parameters of the network. As in batch normalization, this computation is a part of the network and participates in backpropagation.

The convolutional blocks in G and D (illustrated in figures 2.8b and 2.8c) are composed from two 3×3 convolutions followed by Leaky ReLU activations. In the generator, pixelwise normalization is applied between the convolution and the activation function. The blocks that project between feature maps and RGB intensities (G-to-rgb in G and D-from-rgb in D) are 1×1 convolutions.

The networks are trained using the Adam optimizer with the WGAN-GP loss. The images are not generated directly from G, but from a smoothed copy G_s that is updated with the weights of G using exponential moving average with decay 0.999.

2.2.4 StyleGAN

StyleGAN by Karras et al. [8][3] improves ProGAN to generate images of higher quality (see figure 2.9 for some examples). All these improvements are focused on the generator, architecture of the discriminator was almost unchanged.

Mapping network: The latent vector $\boldsymbol{z}$ sampled from normal distribution is not used directly, but a multi-layer mapping network F transforms it into intermediate latent vector $\boldsymbol{w}$. This allows the generator to learn a custom latent representation, which better captures the semantics of the generated images. The mapping network F is composed from 8 fully-connected layers with Leaky ReLU activations.

AdaIN (**style modulation**): Adaptive instance normalization (AdaIN) layers modulate the feature maps in G using the latent $\boldsymbol{w}$ and replace pixelwise feature normalization from

[3]The same team that developed ProGAN.

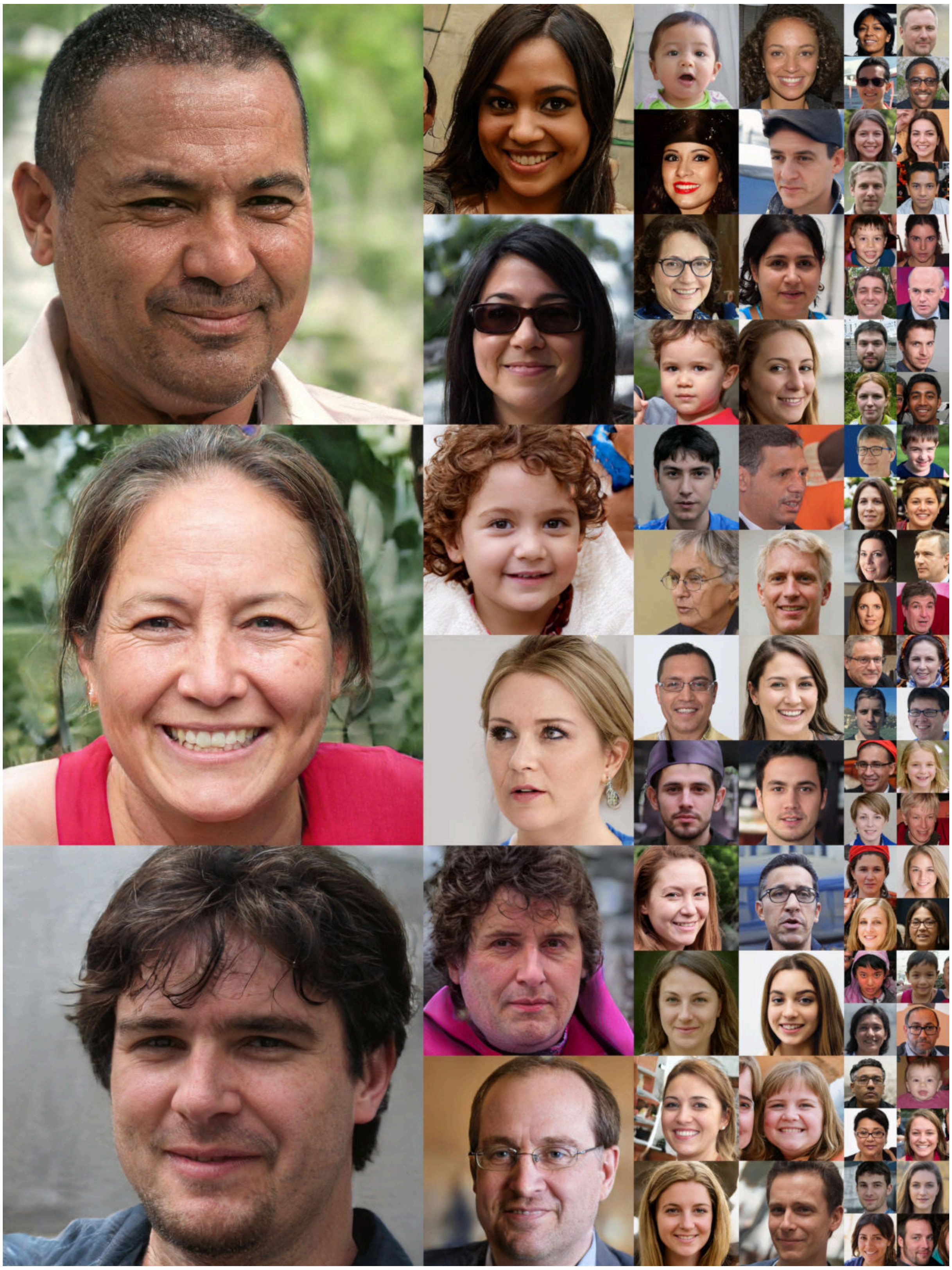


Figure 2.9: Uncurated set of samples generated by the StyleGAN network. (Reproduced from figure 2 of [8].)

ProGAN. Each feature map $\boldsymbol{x}$ is first normalized to unit variation and zero bias (instance normalization) and then scaled and biased with style vectors $\boldsymbol{s}_{\text{scale}}$ and $\boldsymbol{s}_{\text{bias}}$, which are computed from a learned affine transform with weights $\boldsymbol{A}_{\text{scale}}$, $\boldsymbol{A}_{\text{bias}}$ and biases $\boldsymbol{b}_{\text{scale}}$, $\boldsymbol{b}_{\text{bias}}$:

$$\begin{aligned}
\boldsymbol{x}_{\text{out}}(c,y,x) &= \frac{\boldsymbol{x}_{\text{in}}(c,y,x) - \mu(c)}{\sigma(c)} \cdot \boldsymbol{s}_{\text{scale}}(c) + \boldsymbol{s}_{\text{bias}}(c) \\
\mu(c) &= \operatorname*{mean}_{y,x}\ \boldsymbol{x}_{\text{in}}(c,y,x) \\
\sigma(c)^2 &= \operatorname*{mean}_{y,x}\ (\boldsymbol{x}_{\text{in}}(c,y,x) - \mu(c))^2 \\
\boldsymbol{s}_{\text{scale}} &= \boldsymbol{A}_{\text{scale}} \cdot \boldsymbol{w} + \boldsymbol{b}_{\text{scale}} \\
\boldsymbol{s}_{\text{bias}} &= \boldsymbol{A}_{\text{bias}} \cdot \boldsymbol{w} + \boldsymbol{b}_{\text{bias}}
\end{aligned}$$

The authors observed that when the generator can use the latent vector $\boldsymbol{w}$ to modulate the features with the AdaIN operation, it is no longer necessary to inject the latent into the first convolutional block: instead, a learned constant 4×4 feature map is used at the beginning of the network.

Noise inputs: To make it easier for the network to generate stochastic variation (such as positions of individual hairs in the images of human faces), white noise image sampled from $\mathcal{N}(0,1)$ is added to the feature maps after each convolution. The noise has only a single channel, which is broadcasted to all feature maps and multiplied using learned per-feature scaling factors $\boldsymbol{s}$ (initialized to zeros).

$$\begin{aligned}
\boldsymbol{x}_{\text{out}}(c,y,x) &= \boldsymbol{x}_{\text{in}}(c,y,x) + \boldsymbol{s}(c) \cdot \boldsymbol{n}(y,x) \\
\boldsymbol{n}(y,x) &\sim \mathcal{N}(0,1)
\end{aligned}$$

Bilinear up/down: Bilinear interpolation is used in the generator to upscale the feature maps. It is implemented as nearest-neighbor interpolation followed by a simple feature-wise blur. In the discriminator, the feature maps are blurred before being downscaled with average pooling. This is the only improvement that affects the discriminator network.

Otherwise, the generator G remains the same as in ProGAN. Notably, progressive growing and equalized learning rate are still used, and the network has two 3×3 convolutions in each convolutional block. Instead of WGAN-GP, the authors used the non-saturating logistic loss with R_1 regularization. They also trained the networks for a longer time on a bigger dataset with higher quality, and tuned some of the training hyperparameters.

An important property of the style-based architecture is that the influence of style is localized to the scale on which it is applied: the styles at the low resolutions affect high level features of the scene (such as gender, age, or pose in the case of human faces), and the styles at higher resolutions affect low level features (such as ethnicity, or skin and hair color). This effect can be demonstrated by using different latent vectors $\boldsymbol{w}$ for different parts of the network (see figure 2.10): the authors refer to this operation as style mixing. To encourage the locality of style and to ensure that the network produces meaningful images in this case, style mixing was performed even during training (this is called **mixing regularization** in the paper).

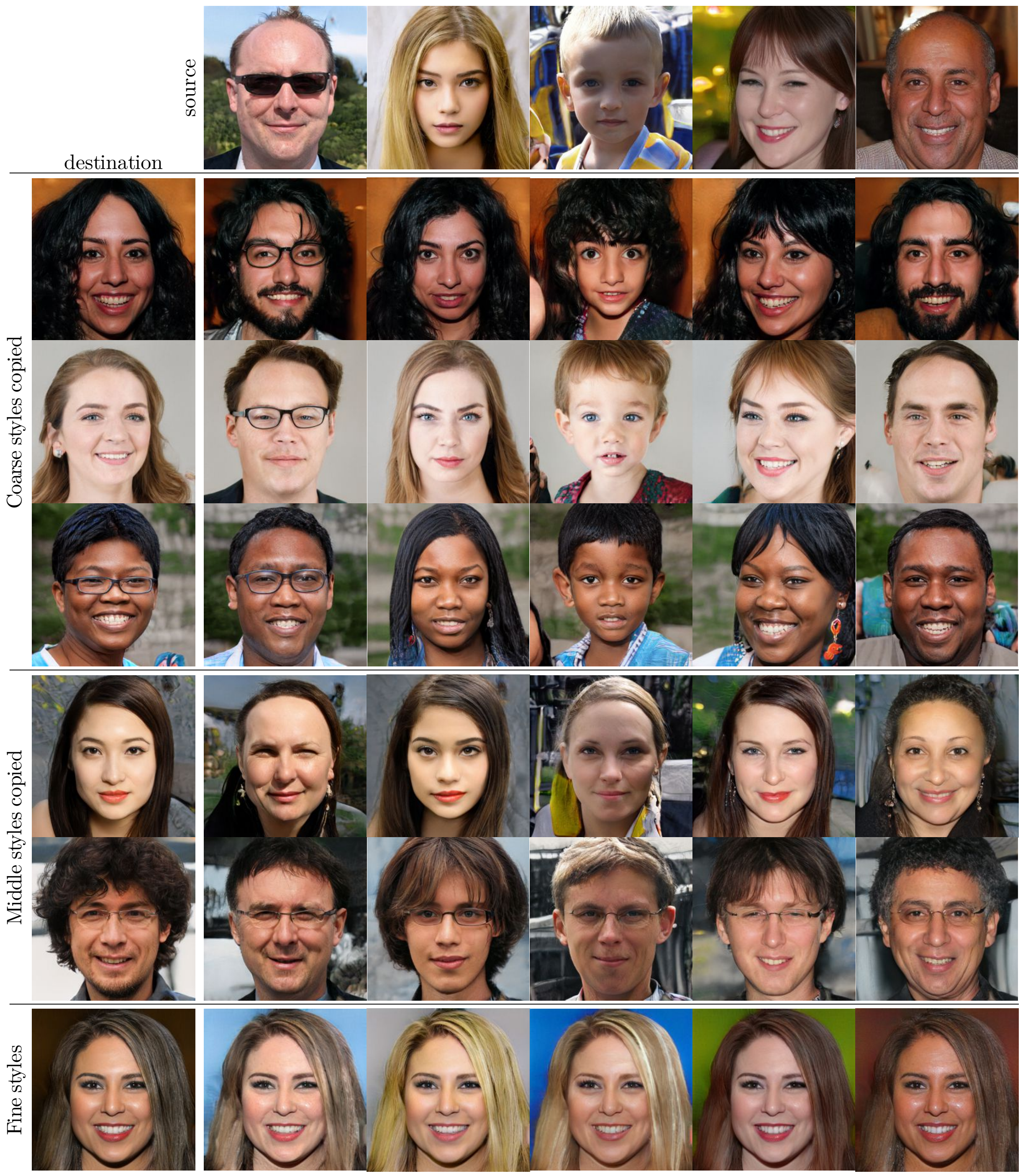


Figure 2.10: The effect of replacing styles of the destination images with the styles of the source images at varying resolutions: coarse (4×4 to 8×8), middle (16×16 to 32×32), fine (64×64 to 1024×1024). (Reproduced from figure 3 of [8].)

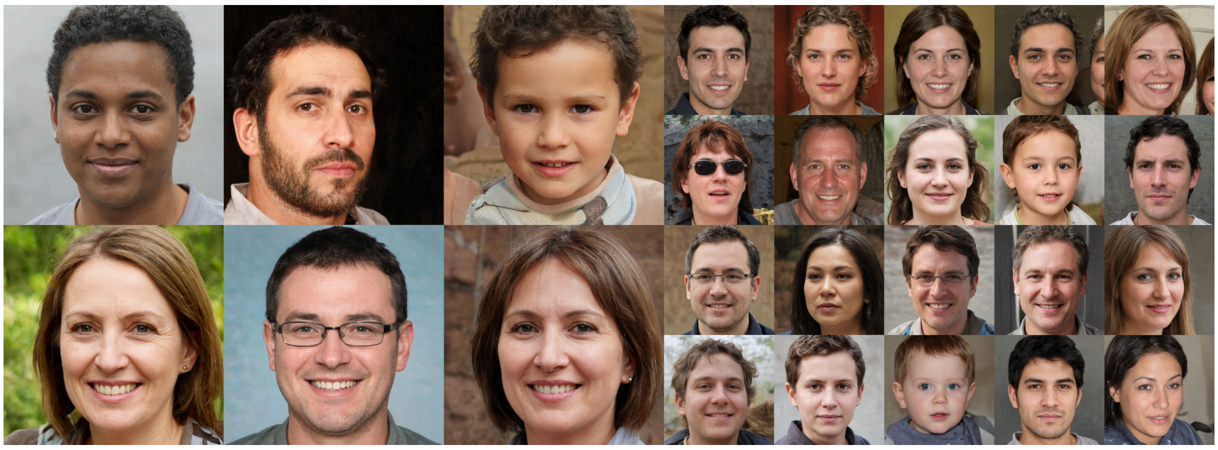


Figure 2.11: Uncurated set of samples generated by the StyleGAN 2 network. (Reproduced from figure 12 of [9].)

2.2.5 StyleGAN 2

Karras et al. improved their GAN architecture once more to yield StyleGAN 2 [9] (see figure 2.11 for examples).

Weight demodulation: The authors note that the instance normalization scheme in StyleGAN causes artifacts, so they replace it with a novel technique that fuses style modulation and normalization. Given style scaling factors $\boldsymbol{s}_{\text{scale}}$, we can fuse the multiplication by $\boldsymbol{s}_{\text{scale}}$ into convolution weights $\boldsymbol{w}$ as follows:

$$\boldsymbol{w}'(c_{\text{out}}, c_{\text{in}}, j, i) = \boldsymbol{w}(c_{\text{out}}, c_{\text{in}}, j, i) \cdot \boldsymbol{s}_{\text{scale}}(c_{\text{out}})$$

To approximately preserve the variation of input activations (and ensure that the activations do not vanish or blow up), we normalize the weights $\boldsymbol{w}'$ as follows:

$$\begin{aligned} \boldsymbol{w}''(c_{\text{out}}, c_{\text{in}}, j, i) &= \frac{\boldsymbol{w}'(c_{\text{out}}, c_{\text{in}}, j, i)}{\sigma(c_{\text{out}})} \\ \sigma(c_{\text{out}})^2 &= \sum_{c_{\text{in}}, i, j} \boldsymbol{w}'(c_{\text{out}}, c_{\text{in}}, i, j)^2 \end{aligned}$$

The weights $\boldsymbol{w}''$ are then used as the convolution kernel. This approach is reminiscent of equalized learning rate, but we rescale the convolution weights dynamically, not with a fixed constant.

Path length regularization: To make the mapping $\boldsymbol{w} \to G(\boldsymbol{w})$ smoother, the paper proposes a new regularization term:

$$\mathcal{L}_{\text{pl}} = \left(\left\|\boldsymbol{J}_{\boldsymbol{w}}^T \cdot \boldsymbol{y}\right\| - a\right)^2$$
$$\text{where } \boldsymbol{y} \sim \mathcal{N}(0, 1)$$

where $\boldsymbol{J}_{\boldsymbol{w}} = \frac{\partial G(\boldsymbol{w})}{\partial \boldsymbol{w}}$ is the Jacobian matrix of the generator with respect to the latent vector $\boldsymbol{w}$, $\boldsymbol{y}$ is a white noise image, and the constant a is computed from exponential running average of $\left\|\boldsymbol{J}_{\boldsymbol{w}}^{T} \cdot \boldsymbol{y}\right\|$. This regularization term ensures that small changes to the latent $\boldsymbol{w}$ correspond to small changes in the generated image $G(\boldsymbol{w})$.

Lazy regularization: Instead of adding the terms for R_1 regularization and path length regularization to the loss function in every iteration, we can compute them only once every K iterations and scale them with K. This saves significant amount of time and does not impact image quality.

Skip and residual connections: The authors experimented with adding skip connections and residual connections to both G and D. The best results were obtained with skip connections in G and residual connections in D; with these architectural modifications, progressive growing was no longer necessary and the networks were trained directly on the target resolution.

The authors demonstrate that the improved architecture achieves better results on several metrics and synthesizes images of higher quality. They show that the proposed path length regularization increases generator smoothness, which improves image quality and makes it easier to reconstruct a latent vector W that generates a particular image.

2.2.6 Other GANs

We will also briefly mention some other applications of GANs to create novel content in computer graphics.

- **pix2pix** [37] trains a GAN to translate images from one domain to another (such as edges to photos, day to night, facade labels to images, or RGB to grayscale in figure 2.12).
- **CycleGAN** [38] performs the same task, but does not need to be trained on paired images and can thus be trained on more domains (such as horses to zebras, or Cezanne to photos in figure 2.13).
- **TileGAN** [39] creates large textures by seamlessly tiling images generated by a modified ProGAN architecture (such as a satellite image in figure 2.14).
- **SinGAN** [40] trains a multi-scale generative model on a single image and uses it for various tasks, such as texture synthesis or super-resolution (see figure 2.15).

Some of these methods train the generator using multiple loss functions, for example pix2pix uses a combination of L^2 loss and adversarial loss.

2.3 Super-resolution

The goal of **super-resolution** is to transform input image into higher resolution. There are straightforward algorithms such as bilinear or bicubic interpolation, but more advanced algorithms can produce results that are much more visually pleasing. These methods are of special interest to us, because we would like to generate images in very high resolution.

SRCNN [41], the first application of convolutional networks to super-resolution, first upscales the input image to the target resolution using bilinear interpolation, transforms

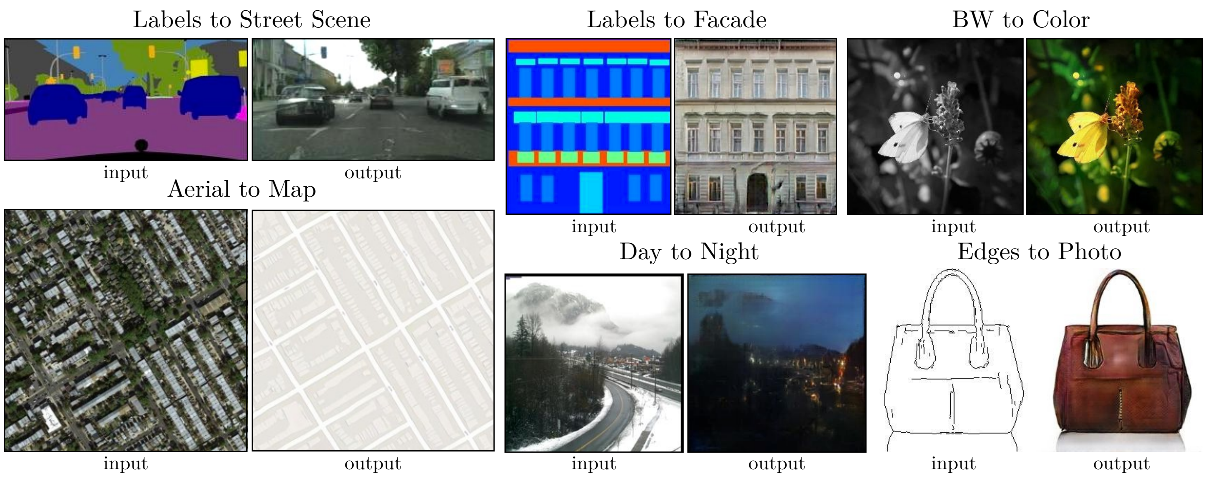


Figure 2.12: Image translations performed with pix2pix. (Reproduced from figure 1 of [37].)

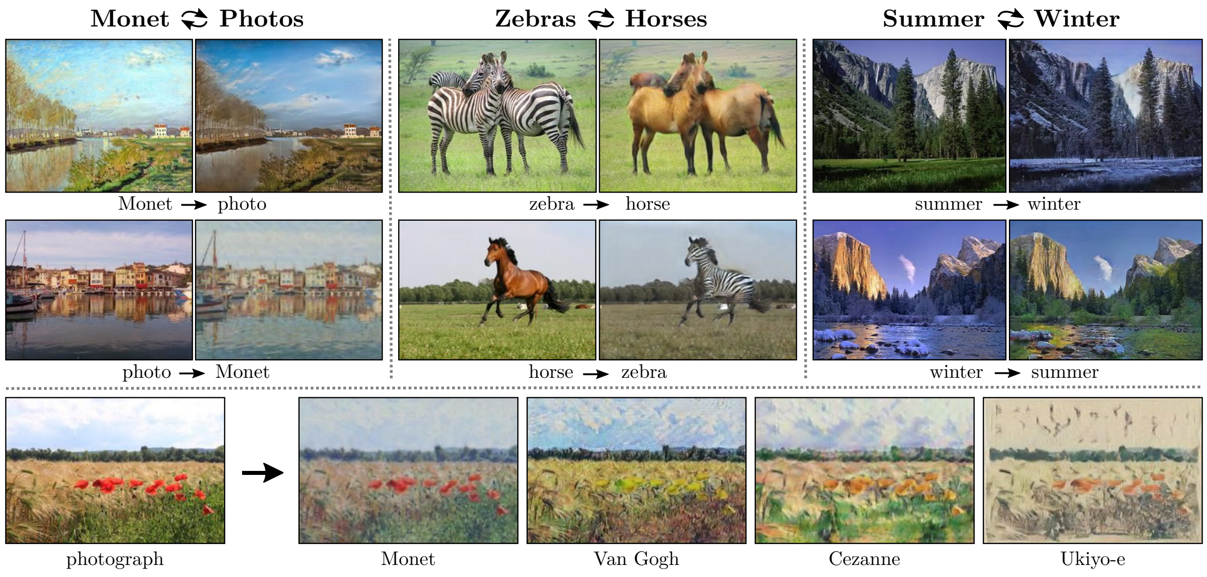


Figure 2.13: Image translation performed with CycleGAN. The model was trained with unpaired images, so it could learn mapping for which no paired samples are available (such as zebras and horses). (Reproduced from figure 1 of [38].)

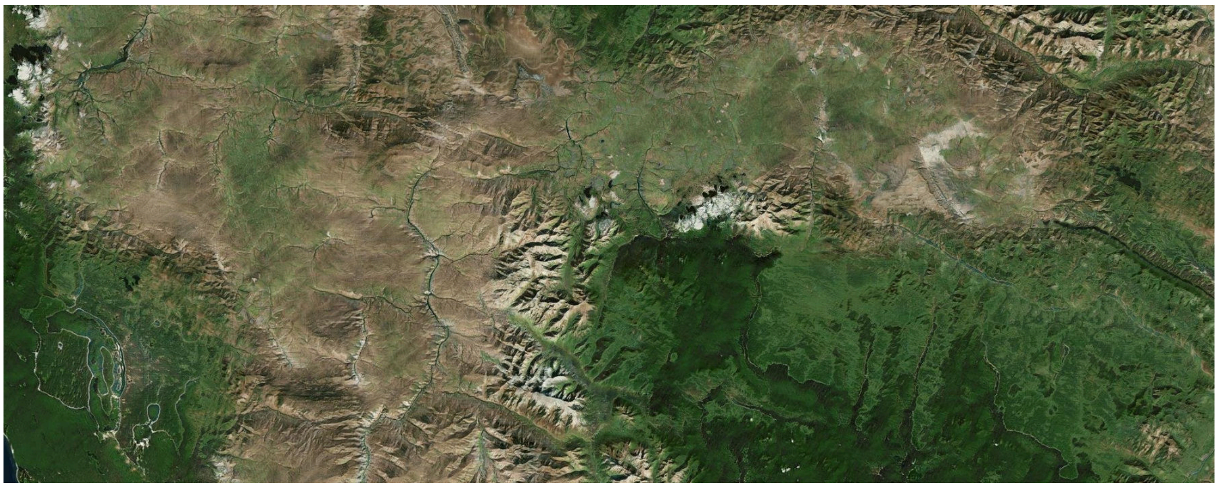


Figure 2.14: A crop from a high-resolution image generated by TileGAN. Note the tiled structure of this image. (Reproduced from supplementary material of [39].)

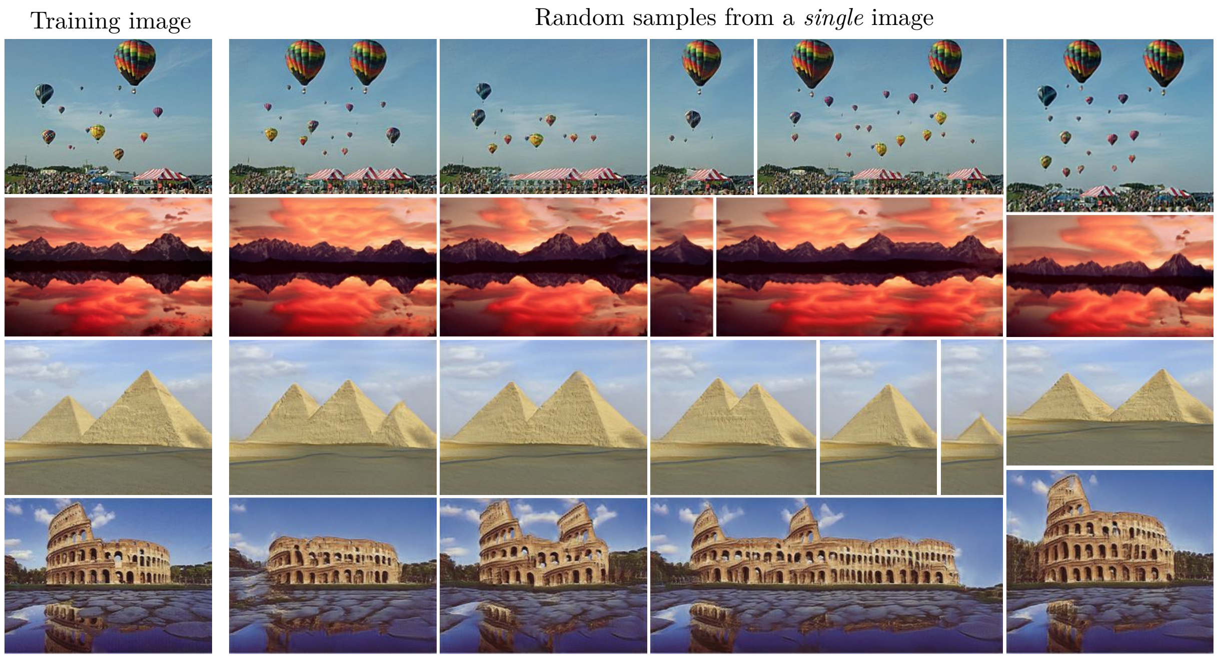


Figure 2.15: Samples from the SinGAN model trained on a single image. (Reproduced from figure 6 of [40].)

the RGB intensities into features with a convolution, processes the feature maps with a stack of convolutions, and finally converts them back to RGB with another convolution. The network was trained using mean-squared error. In effect, the convolutions learn to sharpen the blurred image. The network architecture was improved in **EDSR** [42].

EnhanceNet [43] applied **adversarial loss** to super-resolution: they train a discriminator D to distinguish real high-resolution images from fakes produced by the super-resolution network. They use two other losses (perceptual loss and texture matching loss) that use features from the VGG network and are directly related to the content and style representations in neural style transfer. A similar approach is used by **SRGAN** [44], which uses a different network architecture. Further improvements to the network architecture and loss function were introduced in **ESRGAN** [45]. See figure 2.16 for a sample of results achieved by these methods.

For a more detailed introduction to the field of neural super-resolution, we refer the reader to the survey [46]. We note that super-resolution is a well-studied technique that is already used commercially, such as in Nvidia DLSS[4] for increasing resolution of computer games.

[4]`https://developer.nvidia.com/dlss`

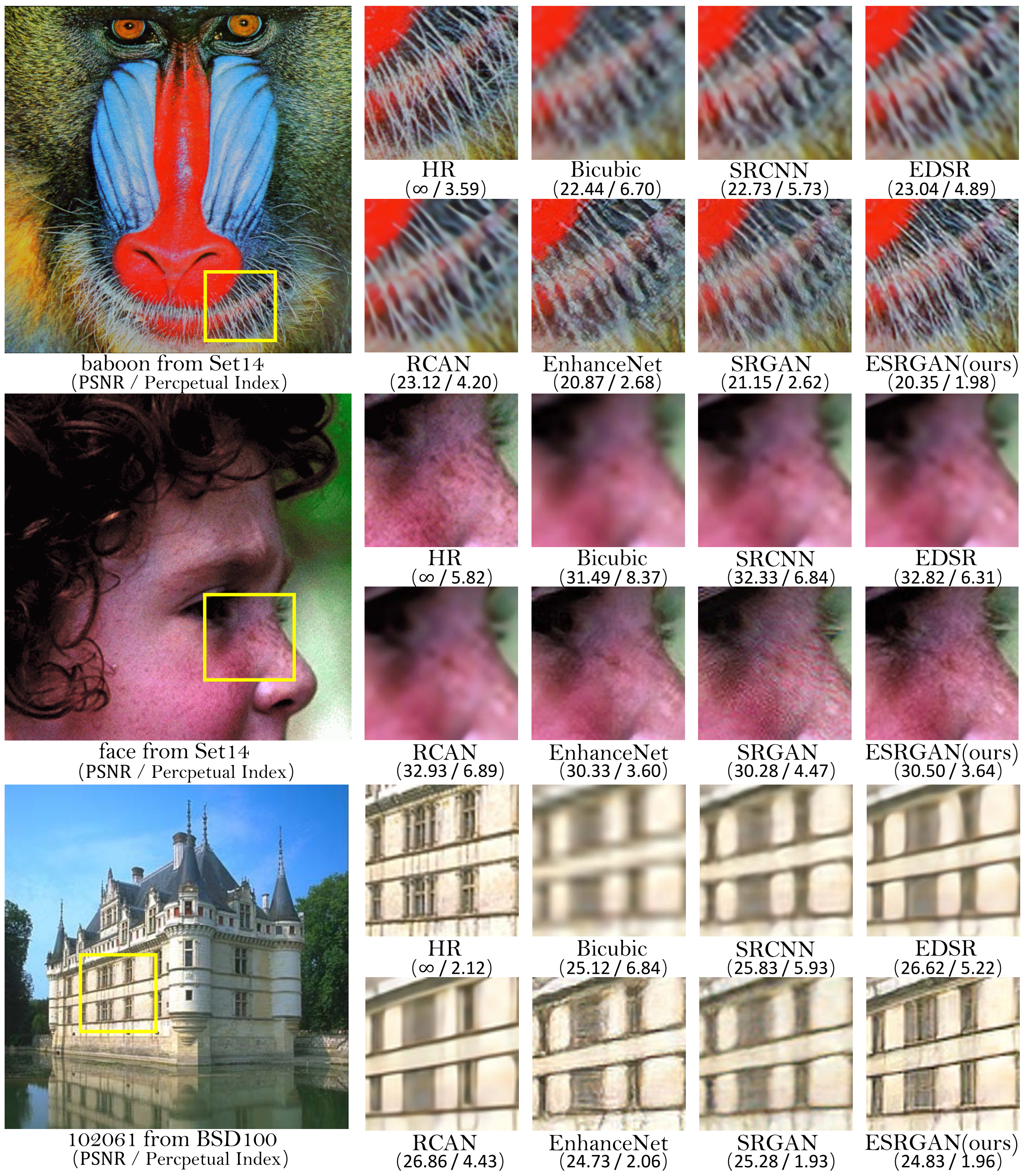


Figure 2.16: Comparison of various superresolution methods on a few image samples. HR is the ground truth, bicubic interpolation is the baseline analytic algorithm, remaining methods are deep convolutional networks. Note the amount of detail synthesized by the GAN-based methods. (Reproduced from figure 7 of [45].)

3. Prior work

In this chapter, we will describe the thesis of Štěpán Hojdar [10], because we directly build upon his work.

3.1 Dataset

The major contribution of Hojdar is the method of shooting photos and processing them into a format suitable for training. He uses a digital camera with a fisheye lens that can capture a full 180° view, attached to a tripod and pointing upward (see figure 3.1 for an example of captured image). In this setup, a single photo captures the whole sky, but we still need a bracketing sequence of several photos shot in quick succession with varying exposure in order to recover the high dynamic range. The camera was programmed to periodically capture the whole sequence, so it could automatically operate without any intervention from the operator. In this way, manual labor was reduced only to replacing the battery and downloading the photos from the SD card.

The RAW images from the camera are then processed into 16-bit TIFF images, using the open-source software RawTherape[1]. It performs demosaicing and partly compensates for camera artifacts such as vignetting and chromatic aberration.

The TIFF images from a bracketing sequence are then merged together to produce a single HDR image and converted from the lens projection into a standard fisheye projection. This step is performed via PtGui[2], a commercial software for creating panoramas.

In his work, Hojdar tested multiple projections and concluded that the most suitable one is the **stereographic fisheye** projection. This projection maps the spherical angles (θ, ϕ) (where θ is the polar distance and ϕ is the azimuthal angle) into the polar coordinates (r, ϕ) (where the radius r ranges from 0 to 1) using the formula:

$$r = \tan\left(\frac{\theta}{2}\right)$$

The end result of the whole dataset capturing pipeline is a set of HDR images in OpenEXR format, which show the sky in stereographic fisheye projection. In total, more than 30 000 images were captured, but only a part of this amount was fully processed. See figure 3.2 for example photos from this dataset.

3.2 Model

To train the GAN, Hojdar used the original ProGAN code[3] [7]. To process HDR images, he applied a standard log transform $\boldsymbol{x}_{\text{out}} = \log(\boldsymbol{x}_{\text{in}} + c)$ to the input HDR image $\boldsymbol{x}_{\text{in}}$, where $c = 10^{-4}$ is a small constant to avoid taking logarithm of zero. To ensure that the networks focus only on the inscribed circle in the fisheye images, a circular mask is applied

[1]`https://rawtherapee.com/`

[2]`https://www.ptgui.com/`

[3]`https://github.com/tkarras/progressive_growing_of_gans`

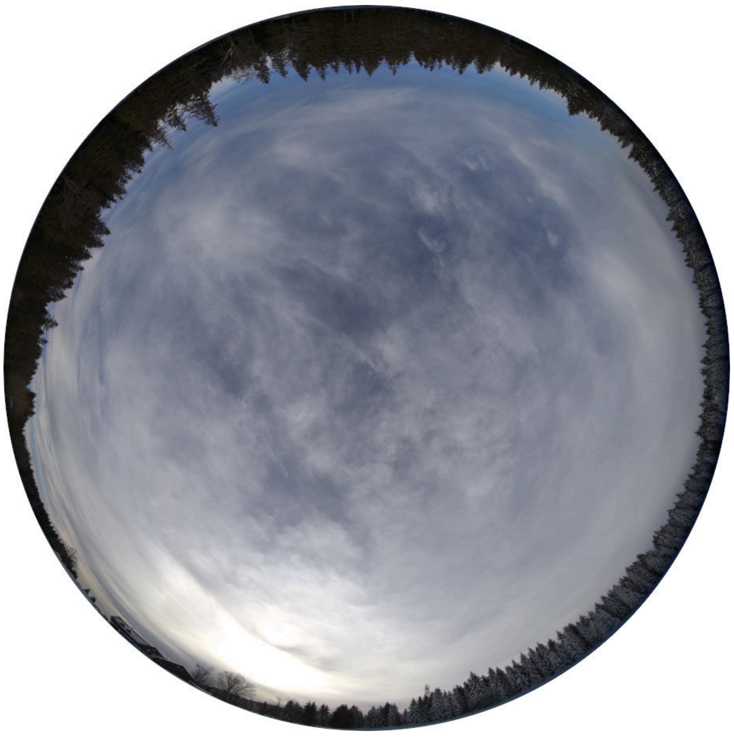


Figure 3.1: An image of the sky captured by the digital camera with fisheye lens used by [10]. (Reproduced from figure 3.1 of [10].)

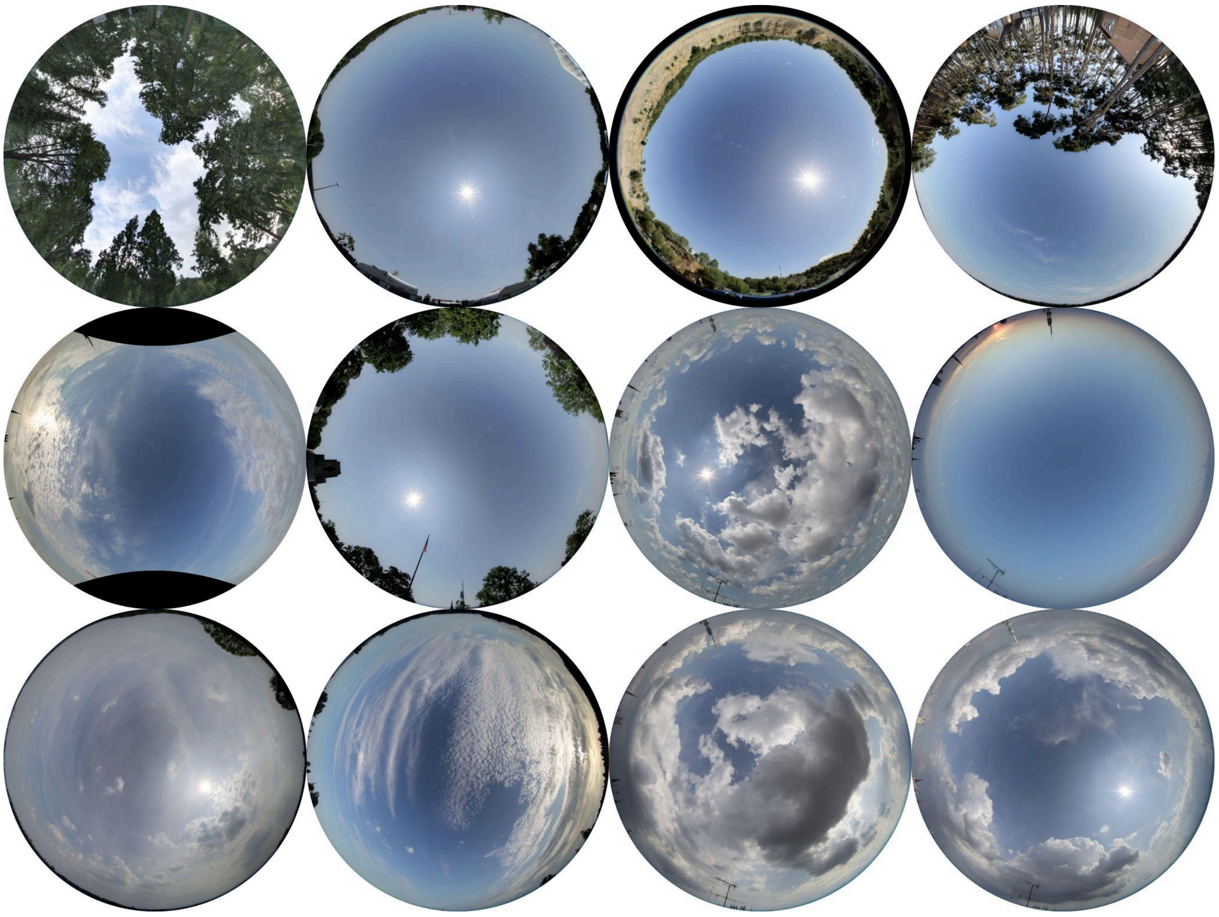


Figure 3.2: A curated sample of processed images from the sky dataset from [10] (this sample includes images captured after the thesis [10] was submitted).

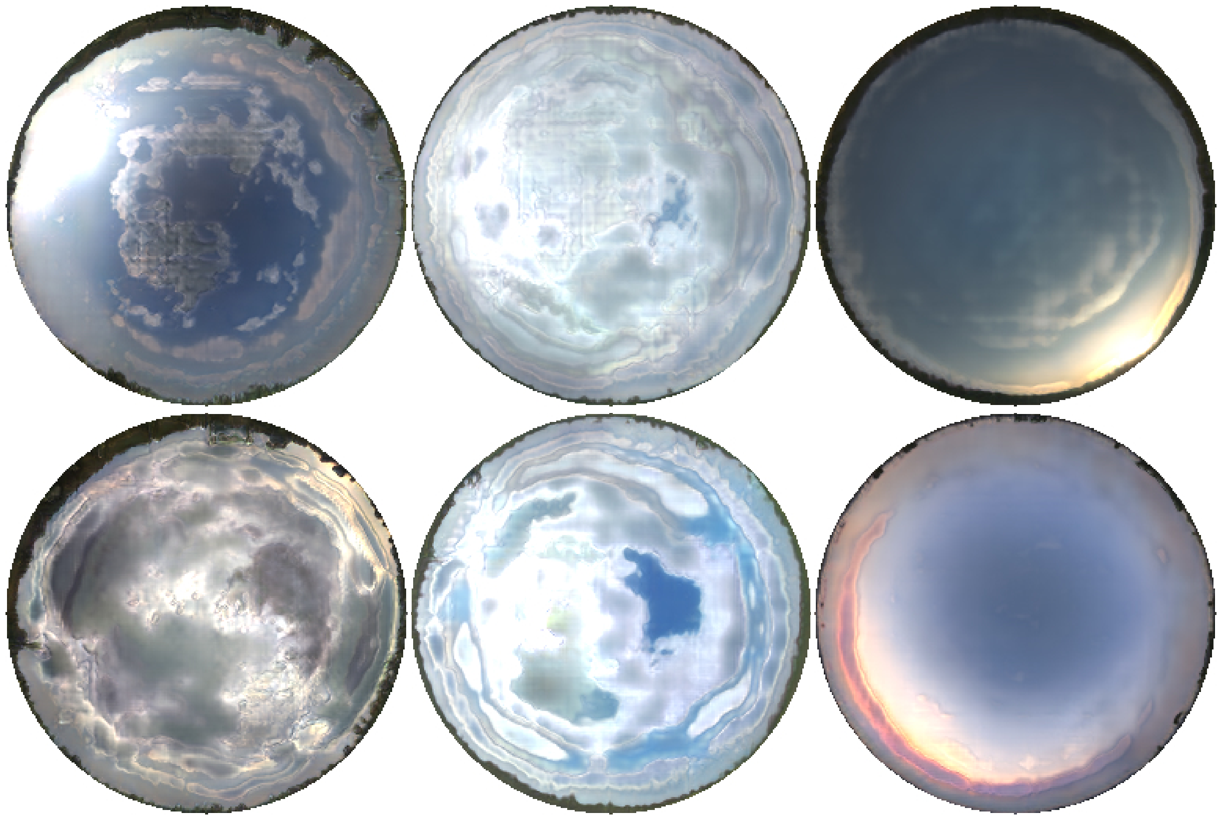


Figure 3.3: Sample of artifacts in images generated by the network of [10]. The first row shows “checkerboard” artifacts, the second row shows “oil stain” artifacts. (Reproduced from figures 5.30 and 5.31 of [10].)

to the output of the generator and the input of the discriminator, so all pixels outside of the unit circle are set to zero.

However, the results produced by this network were not of sufficient quality and contained many artifacts (see figures 3.4 and 3.3). The images contain many “oil stain”-shaped artifacts in place of clouds, suffer from checkerboarded areas, and sometimes are just completely broken.

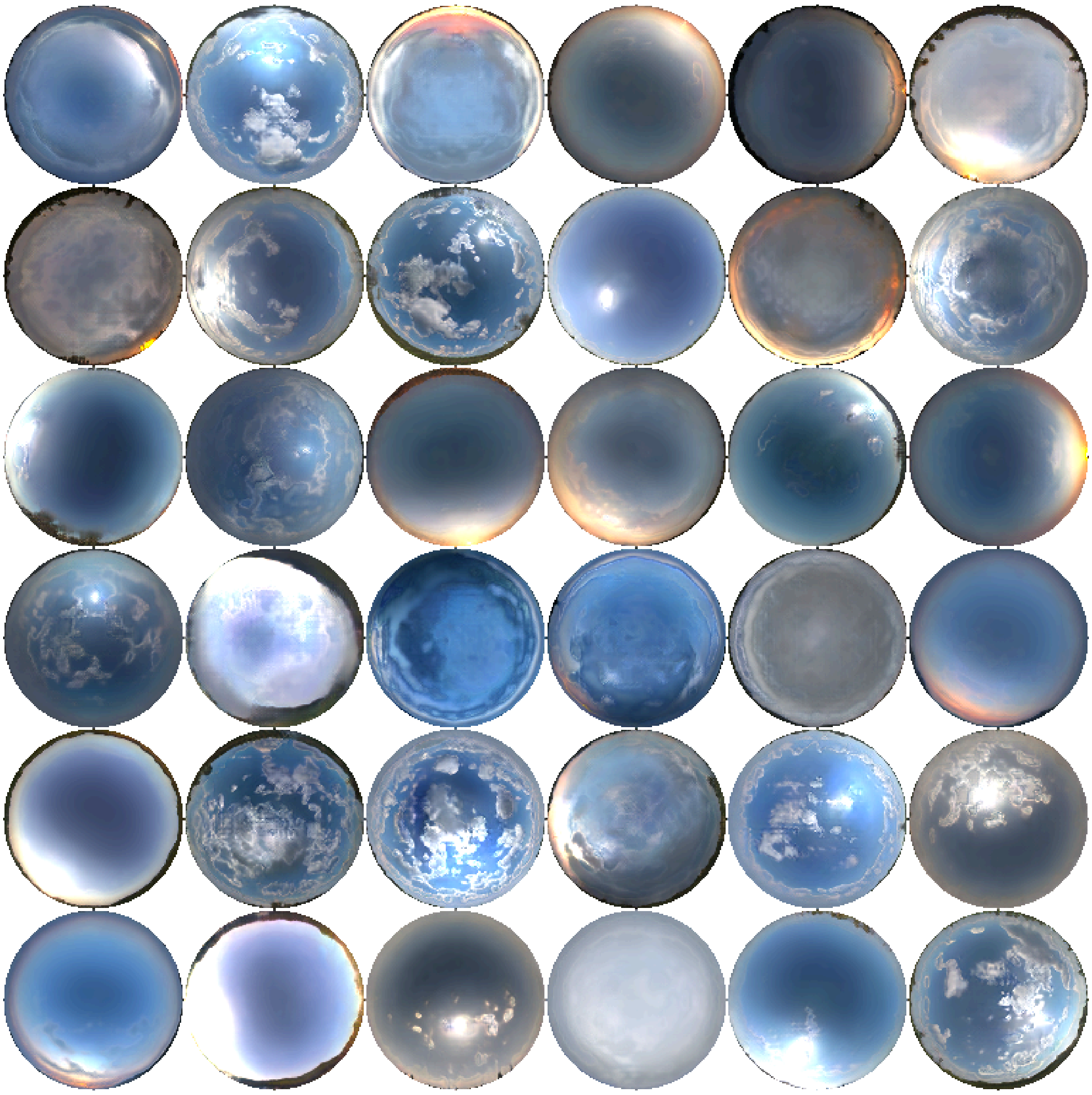


Figure 3.4: Random sample of images generated by the network of [10] in resolution 256×256. (Reproduced from figure 5.24 of [10].)

4. Methods

In this chapter we will describe our GAN models and the improvements that we made in order to increase quality and resolution of the generated images relative to the work by Hojdar [10].

To implement our models we use **PyTorch**[1] [47], a deep learning framework with a Python API. PyTorch provides a large library of operations on tensors and implements backpropagation. It also includes many optimization algorithms for training feedforward networks and provides lightweight APIs that simplify definition of neural networks, data loading and preprocessing, and other support for machine learning tasks.

Large deep networks require a huge amount of computation and must be trained on a GPU. PyTorch provides a unified interface for running the computations on any device, either the CPU or a GPU (multiple GPUs may also be used in parallel). The GPU support is implemented using CUDA[2] and the cuDNN[3] library, so only GPUs from Nvidia are supported.

4.1 ProGAN

We started with the ProGAN model [7] used by Hojdar [10]. We reimplemented the official ProGAN code[4] from scratch in PyTorch, which allowed us to deeply understand every detail of the models and of the training procedure, and also to significantly simplify the code, which was a necessary prerequisite for further experimentation.

We have made some minor improvements to the model:

- The original ProGAN code always generates images in the target resolution, even when training at lower resolutions, so the image is needlessly upscaled and then downscaled[5]. We fixed this and generate the image only at the resolution that is actually needed. However, this had two subtle implications:
  - The gradient penalty in the WGAN-GP loss pushes the norm of the gradient $\partial D(\boldsymbol{x}) \,/\, \partial \boldsymbol{x}$ to be close to 1. However, the input $\boldsymbol{x}$ changes size during training, which changes the scale of this gradient, so we must multiply the norm by $N \,/\, N_{\mathrm{tgt}}$, where N is the current resolution and N_{tgt} is the target resoution.
  - The circular mask was always applied at the target resolution in [10]. We found that applying the mask at the lower resolutions worked well only if the masks were created by downsampling of the binary mask at the target resolution, so they have smooth edges.
- We center the log-transformed HDR images to have zero mean. This is a standard preprocessing step for convolutional networks and we found that it noticeably improved the generated results.

[1] `https://pytorch.org/`
[2] `https://docs.nvidia.com/deeplearning/sdk/index.html`
[3] `https://docs.nvidia.com/deeplearning/sdk/index.html`
[4] `https://github.com/tkarras/progressive_growing_of_gans`
[5] This is probably because the output of a network in TensorFlow must have fixed size.

- To augment the dataset, Hojdar [10] precomputes and stores 10 fixed rotations for every input image. We rotate the images at runtime, so we reduce disk usage by 90 % and can do arbitrary rotations. We also support images stored in float16 (half-precision) and fast compression with the Zstandard algorithm[6], which further reduces disk usage.

Section 5.1 presents extensive experiments that we performed with this architecture to try to improve the quality of generated images.

4.2 StyleGAN

We were unable to produce high-quality images with ProGAN, so we switched to the improved StyleGAN architecture [8, 9]. Again, we did not use the original code[7] directly, but reimplemented the model in PyTorch; however, we could reuse much of the ProGAN code. We did not use all improvements from StyleGAN 2, only the demodulated convolution weights in the generator (this corresponds to configuration B from [9]).

We will now describe the networks, the loss functions and the training procedure. Even details that may seem insignificant are often crucial when training GANs, so we will describe the architecture thoroughly, including code snippets from the actual implementation.

4.2.1 Networks

PyTorch provides a lighweight API for organizing code that implements neural networks. This API is based on **modules**, which are standard Python classes derived from `torch.nn.Module`. Modules may contain named parameters and named submodules, so they can be hierarchically composed. A parameter of a module is registered by wrapping a `Tensor` object (with the values of the parameters) in a `Parameter` object and assigning it as an attribute (instance variable) of the module, typically in the constructor method `__init__` using the syntax `self.name = value`. The same applies to submodules, which are also automatically registered when assigned to an attribute.

The PyTorch API is located in the namespace `torch`, but we import several parts of this interface using shortened names for convenience:

```
import torch
import torch.nn as nn
import torch.nn.functional as F
```

In the code, we use `nf` as a shorthand for “number of features”[8], for example parameter `nf_in` is the number of input features, `nf_out` is the number of output features, and `nf_latent` is the number of features in the latent vector. We also use the name `res` as a shorthand for resolution as a logarithm with base 2, for example `res=6` means resolution $2^6 = 64$. `max_res` is the maximum resolution of the network.

The presented code snippets are simplified and omit some technical details of our implementation. However, all aspects that affect the models and their training are preserved.

[6]`https://github.com/facebook/zstd`

[7]`https://github.com/NVlabs/stylegan2`

[8]We inherited this convention from the original ProGAN code.

4.2.1.1 Building blocks

Fully-connected layer `Dense`

The `Dense` (fully-connected) module uses equalized learning rate, so the weight matrix is initialized from $\mathcal{N}(0, 1)$ and appropriately scaled at runtime.

```
class Dense(nn.Module):
    def __init__(self, nf_in, nf_out, lr_mul=1):
        self.weight = nn.Parameter((1/lr_mul) * torch.randn(nf_out, nf_in))
        self.weight_scale = lr_mul / math.sqrt(nf_in)

    def forward(self, x):
        return self.weight.matmul(x) * self.weight_scale
```

Convolution `Conv2d`

The `Conv2d` (two-dimensional convolution) module also uses equalized learning rate and adds zero padding to ensure that the size of the output matches the size of the input.

```
class Conv2d(nn.Module):
    def __init__(self, nf_in, nf_out, kernel):
        self.weight = nn.Parameter(torch.randn(nf_out, nf_in, kernel, kernel))
        self.weight_scale = 1 / math.sqrt(nf_in * kernel**2)
        self.padding = (kernel - 1) // 2

    def forward(self, x):
        weight = self.weight * self.weight_scale
        return F.conv2d(x, weight, padding=self.padding)
```

Modulated convolution `ModulatedConv2d`

The `ModulatedConv2d` module implements the weight demodulation scheme from StyleGAN 2. The parameter `self.weight` stores the convolution kernel, and `self.mod_weight` with `self.mod_bias` are parameters of the affine transform that maps the latent vector `latent` into style modulation scaling factors `mod`. Note that both `self.weight` and `self.mod_weight` use equalized learning rate and must be scaled with `self.weight_scale` and `self.mod_weight_scale`, respectively.

```
class ModulatedConv2d(nn.Module):
    def __init__(self, nf_in, nf_out, nf_latent, kernel, demodulate=True):
        super(ModulatedConv2d, self).__init__()
        self.weight = nn.Parameter(torch.randn(nf_out, nf_in, kernel, kernel))
        self.weight_scale = 1 / math.sqrt(nf_in * kernel**2)
        self.mod_weight = nn.Parameter(torch.randn(nf_in, nf_latent))
        self.mod_weight_scale = 1 / math.sqrt(nf_latent)
        self.mod_bias = nn.Parameter(torch.zeros(1, nf_in))
        self.padding = (kernel - 1) // 2
        self.demodulate = demodulate
```

First, we get the sizes of the various tensors; `n` is the minibatch size, `h` and `w` are the height and width of the input feature map, `kh` and `kw` are the height and width of the convolution kernel:

```
    def forward(self, x, latent):
        n, nf_latent = latent.size()
        n, nf_in, h, w = x.size()
        nf_out, nf_in, kh, kw = self.weight.size()

        # latent.size() == (n, nf_latent)
        # x.size() == (n, nf_in, h, w)
        # self.weight.size() == (nf_out, nf_in, kh, kw)
```

Then we compute the style modulation scaling factors `mod`:

```
        mod_weight = self.mod_weight * self.mod_weight_scale
        mod = latent.matmul(mod_weight.t())
        mod = mod + (self.mod_bias + 1)
        # mod.size() == (n, nf_in)
```

The convolution weights (kernel) have size `(nf_out, nf_in, kh, kw)`, but each sample in the minibatch has its own scaling factors, so the modulated weights have shape `(n, nf_out, nf_in, kh, kw)`. Therefore, `self.weight` and `mod` must be reshaped using the `.view()` method before they are multiplied together[9].

```
        weight = self.weight.view(1, nf_out, nf_in, kh, kw)
        mod = mod.view(n, 1, nf_in, 1, 1)
        weight = weight * mod * self.weight_scale
        # weight.size() == (n, nf_out, nf_in, kh, kw)
```

The weights are now demodulated to ensure that weighs flowing into each output feature for every sample in the minibatch have unit norm. The function `torch.rsqrt(x)` computes the reciprocal square root `1/sqrt(x)` (elementwise) and is more efficient than square root followed by a division. The `demodulate` parameter may be used to turn off the demodulation.

```
        if self.demodulate:
            demod = torch.rsqrt(weight.pow(2).sum(dim=(2,3,4), keepdim=True) + 1e-8)
            # demod.size() == (n, nf_out, 1, 1, 1)
            weight = weight * demod
```

When we modulate the convolution weights, we have a different convolution kernel for each sample in the minibatch, but the standard convolution primitive supports only a single kernel. We can work around this limitation by using grouped convolution: to convolve `n` kernels in tensor of size `(n, nf_out, nf_in, kh, kw)` with `n` feature maps in tensor of size `(n, nf_in, h, w)` to get `n` feature maps in tensor of size `(n, nf_out, h, w)`, we use a grouped convolution with `n` groups and reshape the kernel to size `(n*nf_out, nf_in, kh, kw)` and the input to size `(1, n*nf_in, h, w)`. We will get output of size `(1, n*nf_out, h, w)`, which can be easily reshaped to size `(n, nf_out, h, w)`.

```
        x = x.view(1, n*nf_in, h, w)
        weight = weight.view(n*nf_out, nf_in, kh, kw)
        x = F.conv2d(x, weight, groups=n, padding=self.padding)
        # x.size() == (1, n*nf_out, h, w)
        x = x.view(n, nf_out, h, w)
        return x
```

Activation, upscaling and downscaling

The `lrelu` (Leaky ReLU) function is used as the activation function in all networks. The activations are scaled to approximately preserve input variance (we have $\mathrm{var}(\mathrm{lrelu}(x)) =$

[9]Dimensions of size 1 are automatically broadcasted in the multiplication, so we get a tensor of size `(n, nf_out, nf_in, kh, kw)` by multiplying `(1, nf_out, nf_in, kh, kw)` with `(n, 1, nf_in, 1, 1)`.

$\frac{1}{2} \cdot \mathrm{var}(x) \cdot (1 + \alpha^2)$ for any random variable x with a distribution that is symmetric around zero).

```
def lrelu(x, alpha=0.2):
    scale = math.sqrt(2.0 / (1 + alpha**2))
    return F.leaky_relu(x, alpha, inplace=True) * scale
```

Upscaling is performed by `upscale2d` using bilinear interpolation:

```
def upscale2d(x, factor=2):
    return F.interpolate(x, mode="bilinear", scale_factor=factor, align_corners=False)
```

Downscaling is performed by `downscale2d` with average pooling:

```
def downscale2d(x, factor=2):
    return F.avg_pool2d(x, (factor, factor))
```

Minibatch standard deviation

The minibatch standard deviation layer was introduced in the ProGAN discriminator to encourage variation in the generator outputs. It computes standard deviation for every channel and every spatial position over the minibatch, averages the standard deviations into a single scalar, and broadcasts it over all samples in the minibatch and all spatial positions to form an extra feature, which is then concatenated to the feature map. Instead of the whole minibatch, the statistics may be computed over a group of a smaller size, given by the parameter `group_size`.

The layer is implemented in function `minibatch_stddev`. First, we reshape the input feature map of size `(n, nf, h, w)` into `(g, n/g, nf, h, w)`, where `g` is the size of the group:

```
def minibatch_stddev(x, group_size):
    n, nf, h, w = x.size()
    g = min(group_size, n)
    y = x.view(g, -1, nf, h, w)
```

then we compute per-group standard deviation of size `(n/g, nf, h, w)`:

```
    y = y - y.mean(dim=0, keepdim=True)
    y = y.pow(2).mean(dim=0)
    y = (y + 1e-8).sqrt()
    # y.size() == (n//g, nf, h, w)
```

average the deviations to get tensor of per-group deviations of size `(n/g, 1, 1, 1)`:

```
    y = y.mean(dim=(1,2,3), keepdim=True)
    # y.size() == (n//g, 1, 1, 1)
```

and broadcast these values over the groups:[10]

```
    y = y.repeat_interleave(g, dim=0)
    y = y.expand(n, 1, h, w)
    return torch.cat((x, y), dim=1)
```

[10]The method `.repeat_interleave(g, dim=0)` repeats each value in the first dimension `g` times.

4.2.1.2 Latent mapping network `F_Mapper`

The network `F_Mapper` transforms latents $\boldsymbol{z} \sim \mathcal{N}(0, 1)$ to latents $\boldsymbol{w}$ used by the synthesizer network. It is a straightforward sequence of fully-connected layers. To stabilize training, the parameters of this network are learned with a smaller learning rate than other parameters in the generator. This is accomplished by dividing all parameters with `lr_mul`, which is equivalent to multiplying the learning rate by `lr_mul` when the Adam optimizer is used, because Adam effectively normalizes the optimization step.

One block of the `F_Mapper` network is implemented in the `F_Block` module:

```python
class F_Block(nn.Module):
    def __init__(self, nf_in, nf_out, lr_mul):
        self.dense = Dense(nf_in, nf_out, lr_mul=lr_mul)
        self.bias = nn.Parameter(torch.zeros(1, nf_out))
        self.bias_scale = lr_mul

    def forward(self, x):
        return lrelu(self.dense(x) + self.bias*self.bias_scale)
```

The module `F_Mapper` then applies these blocks in sequence:

```python
class F_Mapper(nn.Module):
    def __init__(self, nf, blocks, lr_mul):
        self.blocks = nn.ModuleList([
            F_Block(nf, nf, lr_mul=lr_mul)
            for __ in range(blocks)
        ])
        self.nf = nf

    def forward(self, latent_z):
        w = latent_z
        for block in self.blocks:
            w = block(w)
        return w
```

4.2.1.3 Synthesizer network `G_Synthesizer`

The network `G_Synthesizer` is the main part of the generator that transforms the mapped latent $\boldsymbol{w}$ into a fake image.

Module `G_Layer`

The basic building block is the `G_Layer` module, which applies a modulated convolution and adds noise and bias. The original StyleGAN supports only a single noise channel per layer, which is scaled with a feature-wise scaling factor before being broadcasted to all features. However, we found that the network is able to learn richer images if it has access to multiple noise channels, which are converted to feature maps with a 1×1 convolution. The parameter `nf_noise` determines which scheme to use.

```python
class G_Layer(nn.Module):
    def __init__(self, nf_in, nf_out, nf_latent, nf_noise):
        self.conv = ModulatedConv2d(nf_in, nf_out, nf_latent, 3, demodulate=True)
        self.bias = nn.Parameter(torch.zeros(1, nf_out, 1, 1))
        if nf_noise == 1:
            self.noise_weight = nn.Parameter(torch.zeros(1))
        else:
            self.noise_weight = nn.Parameter(torch.zeros(nf_out, nf_noise, 1, 1))

    def forward(self, x, latent_w, noise):
        if self.noise_weight.dim() == 1:
            n = self.noise_weight * noise
        else:
            n = F.conv2d(noise, self.noise_weight)
        x = self.conv(x, latent_w)
        return lrelu(x + n + self.bias)
```

Modules `G_BlockFirst` and `G_Block`

The first block, implemented as module `G_BlockFirst`, starts the convolution stack with a learned constant and transforms it with a single instance of `G_Layer`:

```
class G_BlockFirst(nn.Module):
    def __init__(self, nf_in, nf_out, nf_latent, nf_noise):
        self.const = nn.Parameter(torch.randn(1, nf_in, 4, 4))
        self.layer = G_Layer(nf_in, nf_out, nf_latent, nf_noise)

    def forward(self, latent_w, noise):
        x = self.const.expand(latent_w.size(0), -1, -1, -1)
        return self.layer(x, latent_w, noise)
```

The remaining blocks are implemented as module `G_Block`. The input feature map is first upscaled and then processed with two instances of `G_Layer`:

```
class G_Block(nn.Module):
    def __init__(self, nf_in, nf_out, nf_latent, nf_noise):
        self.layer0 = G_Layer(nf_in, nf_out, nf_latent, nf_noise)
        self.layer1 = G_Layer(nf_out, nf_out, nf_latent, nf_noise)

    def forward(self, x, latent_w, noise0, noise1):
        x = upscale2d(x)
        x = self.layer0(x, latent_w, noise0)
        x = self.layer1(x, latent_w, noise1)
        return x
```

Module `G_ToRgb`

The feature maps at each resolution are projected to RGB intensities (in the log domain) with `G_ToRgb`, which applies a single modulated convolution followed by bias:

```
class G_ToRgb(nn.Module):
    def __init__(self, nf_in, nf_latent):
        self.conv = ModulatedConv2d(nf_in, 3, nf_latent, 1, demodulate=False)
        self.bias = nn.Parameter(torch.zeros(1, 3, 1, 1))

    def forward(self, x, latent_w):
        return self.conv(x, latent_w) + self.bias
```

Synthesizer network `G_Synthesizer`

The synthesizer network `G_Synthesizer` contains a `G_BlockFirst`/`G_Block` and a `G_ToRgb` for all resolutions:

```
class G_Synthesizer(nn.Module):
    def __init__(self, max_res, nf_latent, nf_noise, block_nfs):
        self.nf = lambda res: block_nfs[res-1]
        self.nf_latent = nf_latent
        self.nf_noise = nf_noise
        self.blocks = nn.ModuleList()
        self.to_rgbs = nn.ModuleList()

        for res in range(2, max_res + 1):
            nf_in, nf_out = self.nf(res-1), self.nf(res)
            if res == 2:
                block = G_BlockFirst(nf_in, nf_out, self.nf_latent, self.nf_noise)
            else:
                block = G_Block(nf_in, nf_out, self.nf_latent, self.nf_noise)
            to_rgb = G_ToRgb(nf_out, self.nf_latent)
            self.blocks.append(block)
            self.to_rgbs.append(to_rgb)
```

`G_Synthesizer` is able to generate the RGB image for a given latent $\boldsymbol{w}$ at any resolution `res`, with optional mixing of the lower resolution controlled by the parameter `mix`: when `mix=0`, the high resolution is returned without any modifications, but higher values of `mix` will blend a corresponding amount of the lower resolution into the returned image. The parameter `noise` contains a list of noise channels.

```
    def forward(self, latent_w, res, mix, noise):
        x = self.blocks[0](latent_w, noise[0])
        for r in range(3, res+1):
            x_down = x
            x = self.blocks[r-2](x, latent_w, noise[2*r-5], noise[2*r-4])

        rgb = self.to_rgbs[res-2](x, latent_w)
        if mix > 0:
            rgb_down = self.to_rgbs[res-3](x_down, latent_w)
            rgb_down = upscale2d(rgb_down)
            rgb = torch.lerp(rgb, rgb_down, mix)
        return rbg
```

4.2.1.4 Generator network `G_Net`

The generator network `G_Net` simply binds together `F_Mapper` and `G_Synthesizer`. It also implements sampling of random latents (both $\boldsymbol{z}$ and $\boldsymbol{w}$):

```
class G_Net(nn.Module):
    def __init__(self, max_res, nf_latent, nf_noise, block_nfs,
            mapper_blocks, mapper_lr_mul):
        self.synthesizer = G_Synthesizer(max_res,
            nf_latent=nf_latent, nf_noise=nf_noise, block_nfs=block_nfs)
        self.mapper = F_Mapper(nf_latent, blocks=mapper_blocks, lr_mul=mapper_lr_mul)
        self.max_res = max_res

    def random_latent_z(self, batch_size):
        z = torch.randn(batch_size, self.mapper.nf)
        return z * torch.rsqrt(z.pow(2).mean(dim=1, keepdim=True) + 1e-8)

    def random_latent_w(self, batch_size):
        z = self.random_latent_z(batch_size)
        return self.mapper(z)

    def forward(self, latent_z, res, mix):
        w = self.mapper(latent_z)
        return self.synthesizer(w, res, mix)
```

4.2.1.5 Discriminator network D_Net

The architecture of the discriminator is very similar to the generator, with modulated convolutions replaced by plain convolutions.

Modules D_Block and D_BlockLast

The convolution block at every resolution is implemented in module D_Block, which downscales the feature map after the second convolution:

```
class D_Block(nn.Module):
    def __init__(self, nf_in, nf_out):
        self.conv0 = Conv2d(nf_in, nf_in, 3)
        self.bias0 = nn.Parameter(torch.zeros(1, nf_in, 1, 1))
        self.conv1 = Conv2d(nf_in, nf_out, 3)
        self.bias1 = nn.Parameter(torch.zeros(1, nf_out, 1, 1))

    def forward(self, x):
        x = lrelu(self.conv0(x) + self.bias0)
        x = lrelu(downscale2d(self.conv1(x)) + self.bias1)
        return x
```

The last convolutional block, D_BlockLast, applies the minibatch standard deviation layer before the first convolution. The 4×4 feature map is then flattened to a vector and processed by two dense layers to produce the scalar score.

```
class D_BlockLast(nn.Module):
    def __init__(self, nf_in, nf_out, mbstd_group_size):
        self.mbstd_group_size = mbstd_group_size
        self.conv0 = Conv2d(nf_in + 1, nf_in, 3)
        self.bias0 = nn.Parameter(torch.zeros(1, nf_in, 1, 1))
        self.dense1 = Dense(4*4*nf_in, nf_out)
        self.bias1 = nn.Parameter(torch.zeros(1, nf_out))
        self.dense2 = Dense(nf_out, 1)
        self.bias2 = nn.Parameter(torch.zeros(1, 1))

    def forward(self, x):
        x = minibatch_stddev(x, self.mbstd_group_size)
        x = lrelu(self.conv0(x) + self.bias0)
        x = x.view(x.size(0), -1)
        x = lrelu(self.dense1(x) + self.bias1)
        x = self.dense2(x) + self.bias2
        return x
```

Module D_FromRgb

The module D_FromRgb projects RGB image into feature map with a 1×1 convolution:

```
class D_FromRgb(nn.Module):
    def __init__(self, nf_out):
        self.conv = Conv2d(3, nf_out, 1)
        self.bias = nn.Parameter(torch.zeros(1, nf_out, 1, 1))
    def forward(self, rgb):
        return lrelu(self.conv(rgb) + self.bias)
```

Discriminator network D_Net

The module D_Net then puts all pieces together to form the discriminator network. It evaluates the appropriate part of the network based on the resolution of the input image, optionally merging the feature map from lower resolution according to the parameter `mix`:

```
class D_Net(nn.Module):
    def __init__(self, max_res, block_nfs, mbstd_group_size):
        self.nf = lambda res: block_nfs[res-1]
        self.mbstd_group_size = mbstd_group_size
        self.blocks = nn.ModuleList()
        self.from_rgbs = nn.ModuleList()

        for res in range(2, max_res+1):
            nf_in, nf_out = self.nf(res), self.nf(res-1)
            if res == 2:
                block = D_BlockLast(nf_in, nf_out, self.mbstd_group_size)
            else:
                block = D_Block(nf_in, nf_out)
            self.blocks.append(block)
            self.from_rgbs.append(D_FromRgb(nf_in))

    def forward(self, rgb, res, mix):
        x = self.from_rgbs[res-2](rgb)
        for r in reversed(range(2, res+1)):
            x = self.blocks[r-2](x)
            if r == res and mix > 0:
                rgb_down = downscale2d(rgb)
                x_down = self.from_rgbs[r-3](rgb_down)
                x = torch.lerp(x, x_down, mix)
        return x
```

4.2.2 Losses

The loss functions implement the non-saturating logistic loss with R_1 regularization of the discriminator.

4.2.2.1 Discriminator loss D_loss

The loss function for the discriminator `D` is implemented as function `D_loss`. A minibatch of real images is given as parameter `real_imgs`, and the fake images `fake_imgs` are generated from random latents $\boldsymbol{z}$. If the networks are growing and the output of generator `G` is a mix between the low resolution and high resolution, we apply the same mix to the real images using the function `fade_reals` (given below). We also multiply both images with the circular mask of appropriate resolution and use the `.detach()` method on `fake_imgs` and `real_imgs` to make PyTorch forget the operations that produced these images and treat them as leaves of the computation graph.

```
def D_loss(G, D, real_imgs, res, mix, circular_masks, r1_gamma):
    latent_z = G.random_latent_z(real_imgs.size(0))
    fake_imgs = G(latent_z, res, mix)
    real_imgs = fade_reals(real_imgs, mix)
    fake_imgs = (fake_imgs * circular_masks[res]).detach()
    real_imgs = (real_imgs * circular_masks[res]).detach()
```

As the next step, we evaluate scores for both real and fake images using `D` and compute the non-saturating logistic loss:

```
    fake_scores = D(fake_imgs, res, mix)
    real_scores = D(real_imgs, res, mix)

    fake_loss = F.softplus(fake_scores)
    real_loss = F.softplus(-real_scores)
    loss = fake_loss + real_loss
```

Finally, we compute the R_1 regularization term and add it to the loss:

```
    r1_reg = D_r1_reg(D, real_imgs, res, mix, r1_gamma)
    return loss + r1_reg
```

The function `fade_reals`, applied to the real images before they are given to the discriminator `D`, simply mixes the downscaled image with the original:

```
def fade_reals(x, mix, upscale_mode):
    x_down = upscale2d(downscale2d(x), upscale_mode)
    return torch.lerp(x, x_down, mix)
```

4.2.2.2 R_1 penalty `D_r1_reg`

The R_1 gradient penalty for the discriminator `D` is computed in function `D_r1_reg`. The function `torch.autograd.grad` computes the gradient of the sum of scores `real_score_sum` with respect to the input images `real_imgs`[11]:

```
def D_r1_reg(D, real_imgs, res, mix, r1_gamma):
    real_scores = D(real_imgs, res, mix)
    real_score_sum = real_scores.sum()

    r1_grads = torch.autograd.grad(
        output = real_score_sum,
        input = real_imgs,
    )
```

We can now compute the squared L^2 norm and rescale it to account for the lower resolution of the images:

```
    r1_norms = r1_grads.pow(2).sum(dim=(1,2,3))
    r1_norms_scaled = r1_norms / 2**(D.max_res - res)
```

The R_1 regularization term is then weighted by the parameter `r1_gamma`:

```
    return r1_norms_scaled * (0.5*r1_gamma)
```

4.2.2.3 Generator loss `G_loss`

The generator loss, computed in `G_loss`, is quite straightforward. We generate a minibatch of random fake images using the generator `G` and multiply them with the circular mask:

```
def G_loss(G, D, batch_size, res, mix, device, circular_masks):
    latent_z = G.random_latent_z(batch_size, device=device)
```

[11]We have simplified the real function `torch.autograd.grad`, which in fact takes a list of inputs, a list of outputs, and a list of gradients w.r.t. the inputs (which are 1 by default).

```
    fake_imgs = G(latent_z, res, mix)
    fake_imgs = fake_imgs * circular_masks[res]
```

We then score the images using the discriminator D and compute the non-saturating logistic loss:

```
    fake_scores = D(fake_imgs, res, mix)
    loss = F.softplus(-fake_scores)
    return loss
```

4.2.3 Training

We start the training with 600k images at resolution 16×16, and then grow the network in phases. At the beginning of each phase, we gradually mix the high resolution into the upscaled low resolution for 600k training images, and then train only at the high resolution for another 600k images. The last phase at the target resolution runs until it is stopped manually. Batch size is decreased with increasing resolution to fit into the available memory.

The training code periodically dumps a sample of fake images and network snapshots to disk. If the training process is terminated, it can be resumed from this snapshot. We log training statistics into TensorBoard[12], so we can easily visualize them. Following the practice from ProGAN, we maintain a "stabilized" generator `Gs` with weights that are exponential moving averages of weights from the true generator `G`, and generate images from `Gs` instead of `G`.

4.2.3.1 Data preprocessing and augmentation

To ensure that data loading does not become a performance bottleneck, we preprocess the EXR images and store them in a format that can be loaded more quickly: we rescale every image to resolutions $2^2, 2^3, \ldots, 2^{10}$, and store each resolution directly as an array of floats of size $(H, W, 3)$. These arrays are compressed with a fast compression algorithm to save disk space and I/O bandwidth, and stored in a single binary file.

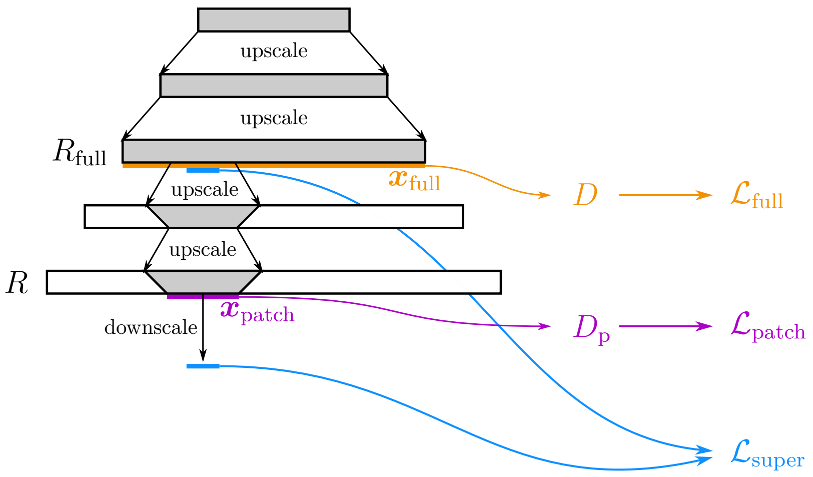
During training, we map this binary file into memory and let the operating system manage the transfers between disk and memory[13]. Training samples are randomly drawn from the dataset and each image is processed as follows:

1. Random rotation is applied to the image using the OpenCV library[14].
2. The image is transposed from shape $(H, W, 3)$ and BGR order (used by OpenCV) into shape $(3, H, W)$ and RGB order (used by our networks).
3. Random vertical flip and random horizontal flip are applied with probability 0.5.
4. The image is transformed from linear intensities $\boldsymbol{x}_{\text{lin}}$ into the log domain $\boldsymbol{x}_{\text{log}}$ using the relationship
$$\boldsymbol{x}_{\text{log}} = \log(\boldsymbol{x}_{\text{lin}} + c) - \mu_{\text{log}}$$
where $c = 10^{-4}$ is a small constant to avoid taking logarithm of 0, and $\mu_{\text{log}} = -4.2$ is the average logarithm value of all images in the dataset.

[12]https://www.tensorflow.org/tensorboard

[13]In fact, our training machines have enough RAM to store the whole dataset in memory.

[14]https://opencv.org/



upscale
upscale
R_{full}
$\boldsymbol{x}_{\text{full}}$
D
$\mathcal{L}_{\text{full}}$
upscale
upscale
R
$\boldsymbol{x}_{\text{patch}}$
downscale
D_{p}
$\mathcal{L}_{\text{patch}}$
$\mathcal{L}_{\text{super}}$

Figure 4.1: In SuperGAN, the full-size image $\boldsymbol{x}_{\text{full}}$ is generated at resolution R_{full} and scored with D to compute loss $\mathcal{L}_{\text{full}}$ (highlighted in orange). A smaller patch $\boldsymbol{x}_{\text{patch}}$ is then computed at resolution R and scored with D_{patch} to compute loss $\mathcal{L}_{\text{patch}}$ (highlighted in purple). The patch $\boldsymbol{x}_{\text{patch}}$ is then downscaled to resolution R_{full} and compared with the corresponding patch in $\boldsymbol{x}_{\text{full}}$ to compute loss $\mathcal{L}_{\text{super}}$ (highlighted in blue).

Augmenting the dataset with arbitrary rotations is very important for generalization, because convolutional networks are not rotation-independent, so every rotation is processed in a unique way and the effective size of the augmented dataset is much bigger than its real size.

4.3 SuperGAN

The architectures from the DCGAN family (which includes both ProGAN and StyleGAN) cannot be directly applied at resolutions beyond 1K, because they must always generate the whole image. At resolution 32K, a single feature map with 16 channels (which is already a small number) consumes 64 GiB of memory, so we would need hundreds of gigabytes of memory just to generate and discriminate a single image, which is well beyond the capabilities of current hardware.

SuperGAN is our modification of the StyleGAN architecture, inspired by super-resolution methods, which aims to synthesize images at resolutions higher than 1K. Because it is not possible to generate high-resolution images during training due to memory limitations, we generate the full image only up to given resolution R_{full} and generate only patches at higher resolutions $R > R_{\text{full}}$.

Our model is trained in the same way as StyleGAN at resolutions $R \leq R_{\text{full}}$, using the same generator G and discriminator D. At resolutions $R > R_{\text{full}}$, we compute the full generated image $\boldsymbol{x}_{\text{full}}$ at resolution R_{full}:

$$\boldsymbol{x}_{\text{full}} = \boldsymbol{x}_{R_{\text{full}}} = G_{R_{\text{full}}}(\boldsymbol{z})$$

Then we generate a random patch $\boldsymbol{x}_{\text{patch}}$ of the generated image at resolution R:

$$\boldsymbol{x}_{\text{patch}} = \boldsymbol{x}_R(y_0{:}y_1, x_0{:}x_1) = G_R(\boldsymbol{z})(y_0{:}y_1, x_0{:}x_1)$$

where $\boldsymbol{x}(y_0{:}y_1, x_0{:}x_1)$ represents the "slicing" operation on image $\boldsymbol{x}$ and the extent $y_0{:}y_1, x_0{:}x_1$ is sampled randomly. Due to the convolutional nature of the generator G, we can compute $\boldsymbol{x}_{\text{patch}}$ efficiently, because each pixel of the output image $\boldsymbol{x}_R$ depends only on a few neighboring pixels in the preceding feature maps (see figure 4.1 for an illustration). Moreover, we can reuse the feature maps that we computed when we have generated $\boldsymbol{x}_{\text{full}}$, so we need to compute only a small patch from all feature maps between R_{full} and R.

To train the generator, we use a weighted combination of three losses:

$$\mathcal{L} = \mathcal{L}_{\text{full}} + \mathcal{L}_{\text{patch}} + \lambda_{\text{super}}\mathcal{L}_{\text{super}}$$

$\mathcal{L}_{\text{full}}$ is the usual non-saturating logistic loss computed from $D(\boldsymbol{x}_{\text{full}})$, which ensures that the low-resolution image $\boldsymbol{x}_{\text{full}}$ is realistic.

$\mathcal{L}_{\text{patch}}$ is the same non-saturating logistic loss, but applied to the patch $\boldsymbol{x}_{\text{patch}}$ and computed with the **patch discriminator** D_{patch}. This loss guides the generator to synthesize realistic details at the high resolution R.

$\mathcal{L}_{\text{super}}$ is L^2 distance between the patch $\boldsymbol{x}_{\text{patch}}$ downscaled from R to R_{full} and the corresponding patch from $\boldsymbol{x}_{\text{full}}$. This ensures that $\boldsymbol{x}_R$ is actually an upscaled version of $\boldsymbol{x}_{R_{\text{full}}} = \boldsymbol{x}_{\text{full}}$. The constant λ_{super} is a weight that influences the balance between the superresolution loss $\mathcal{L}_{\text{super}}$ and the adversarial losses $\mathcal{L}_{\text{full}}$ and $\mathcal{L}_{\text{patch}}$.

The discriminator D is trained as in StyleGAN to separate $\boldsymbol{x}_{\text{full}}$ and the real images at resolution R_{full}, using the logistic loss and R_1 regularization. The patch discriminator D_{patch} is trained in the same way, but it learns to separate the patches $\boldsymbol{x}_{\text{patch}}$ and patches from real images.

The networks G and D have the same architecture as in our modified StyleGAN. The discriminator D_{patch} uses the same convolutional blocks as D, but it is fully-convolutional, so it can be applied to patches of arbitrary size and the output is a two-dimensional map of scores, which is averaged to produce the final score. However, we add zero padding to each convolution block in D_{patch}, so the output of the network does depend on the size and position of the input patch, so D_{patch} is not a pure fully-convolutional network. In our experiments, D_{patch} had 5 convolution blocks with 128 or 256 channels.

5. Results

For all our experiments, we used training images captured by Hojdar using the methods from [10]. We used only images from a single site, a rooftop in Prague (see figure 5.1 for an example image), because images from other sites contain large number of trees and other objects that occlude the sky.

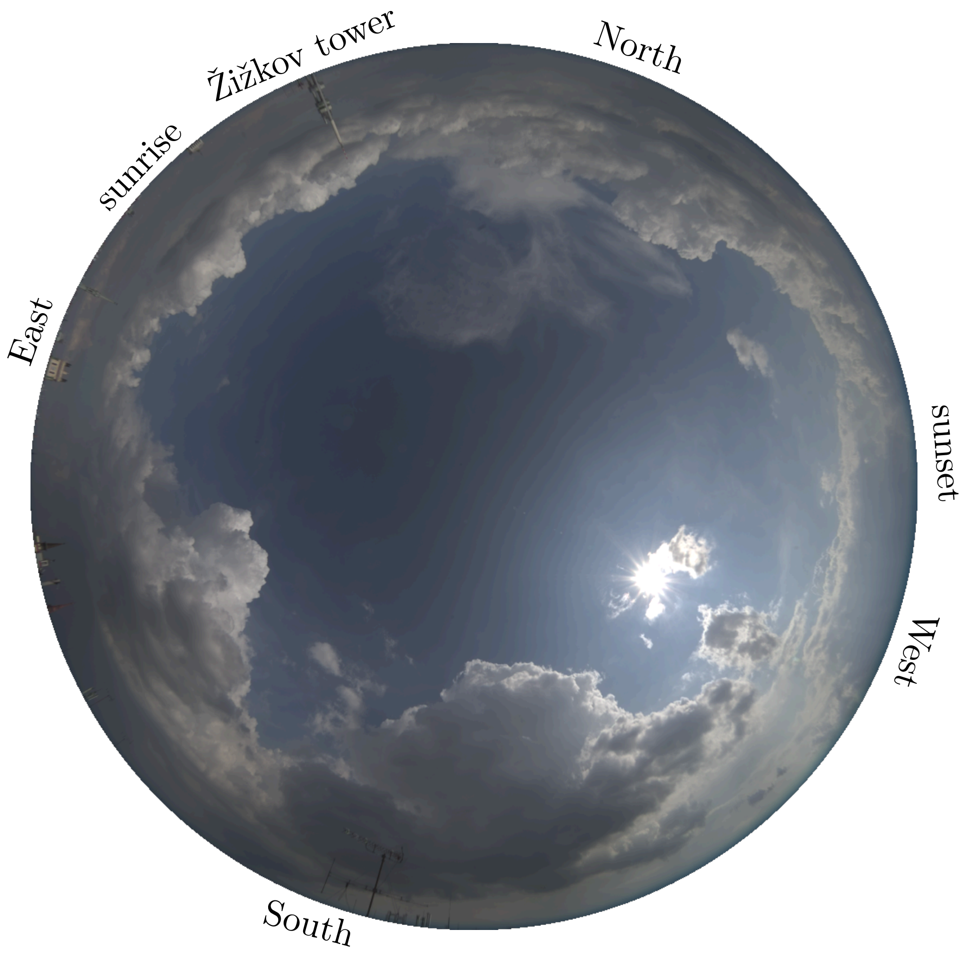


Figure 5.1: An example image from the Prague rooftop with directions for North, East, South, and West. Note that the camera points upward, so the directions are reversed relative to the usual map view. We also show the Žižkov tower in the background and approximate sun position at sunrise and sunset.

5.1 ProGAN

We performed many experiments with the ProGAN network described in section 4.1. We knew from [10] that the network is able to memorize a single image perfectly, but it is unable to fit a larger dataset without severe artifacts. We therefore experimented with various dataset sizes and augmentation methods, trying to find the “phase transition” between overfitting (memorization) and underfitting (artifacts). All experiments were performed at resolution 256×256.

5.1.1 Single day

We created a small dataset of 714 images captured on a single day at the Prague site (figure 5.2 shows sample images from this dataset). When the networks are trained on these images without augmentation, the synthesized images do not suffer from artifacts, but they are almost perfect copies of the training images (see figure 5.3). However, when augmentation is enabled, the generated images contain serious artifacts (see figure 5.4).

These experiments show that the generator is able to memorize and perfectly reproduce hundreds of similar images. Augmentation with arbitrary rotations increases the effective size of the dataset so much that the model is no longer able to fit the data well.

5.1.2 Single image

The previous experiment suggests that rotations are hard to model for our network. This is not surprising, because convolutional networks are not rotation-invariant. We trained three models on a single image augmented with arbitrary rotations and flips (see figure 5.5) and found that the network was able to learn rotations of a single image quite well. We also verified that interpolations of the latent vector resulted in smooth rotations with few abrupt changes, so the networks succeeded in learning some representation of rotation.

5.1.3 Larger dataset

Even though rotations provide a useful augmentation method, we do not really require our networks to learn arbitrary rotations of the sky. If the training images are always rotated in the same way relative to the Earth, they should be easier to generate, because the set of positions where the Sun can possibly appear is limited. We therefore trained the networks using a larger dataset of 7809 images from the Prague rooftop site, without any augmentation. We found that the generated images are again of very low quality, suffering from the usual artifacts.

5.1.4 Conclusion

We tried to improve the ProGAN architecture in multiple ways: cutout regularization [48] in the discriminator, injection of guiding features to the generator (such as the angle θ between the given pixel and the zenith), or producing the discriminator output from all resolutions. None of these modifications had a significant effect. We conclude that the ProGAN architecture does not seem to be powerful enough for our purposes.

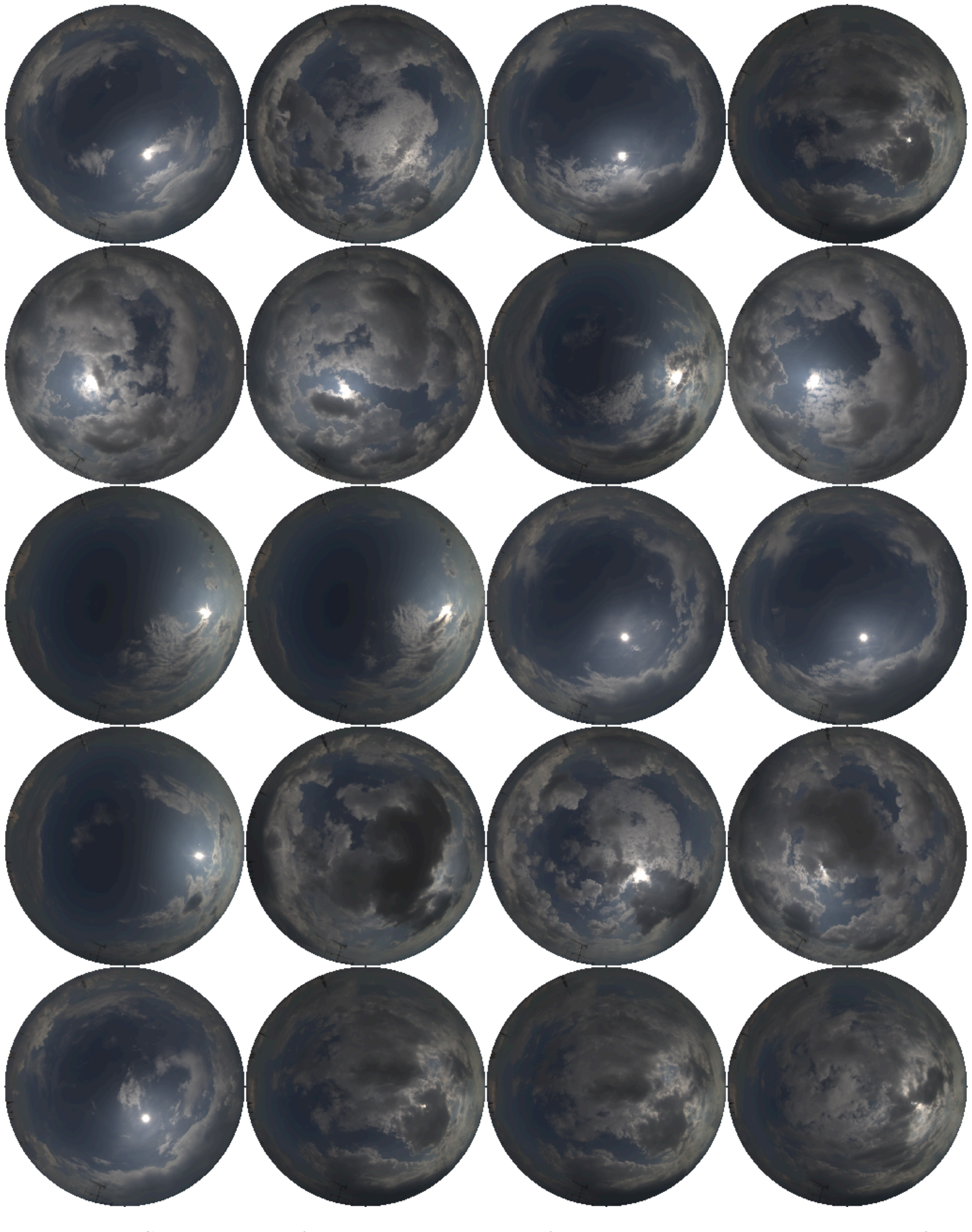


Figure 5.2: Sample images from the small dataset of 714 images captured on 2019-07-14 from the Prague rooftop. We picked a sunny day with rapidly moving cumulus clouds, so there is large variation in the cloud positions, but the appearance of the clouds is very consistent.

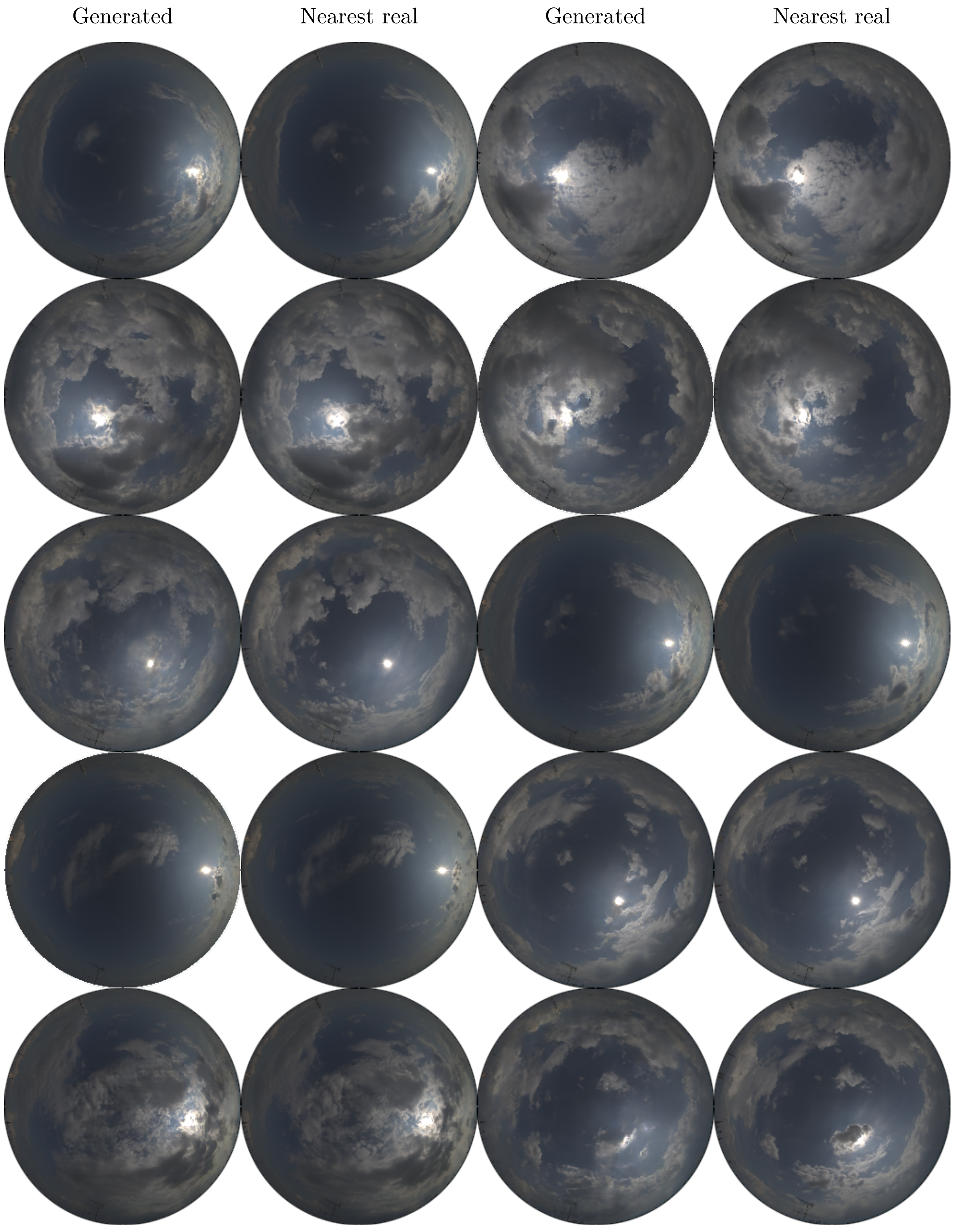


Figure 5.3: When trained on just 714 images 256×256 without augmentation, the ProGAN generator simply memorized the training images. We show 10 random generated images and their nearest neighbors in the training data.

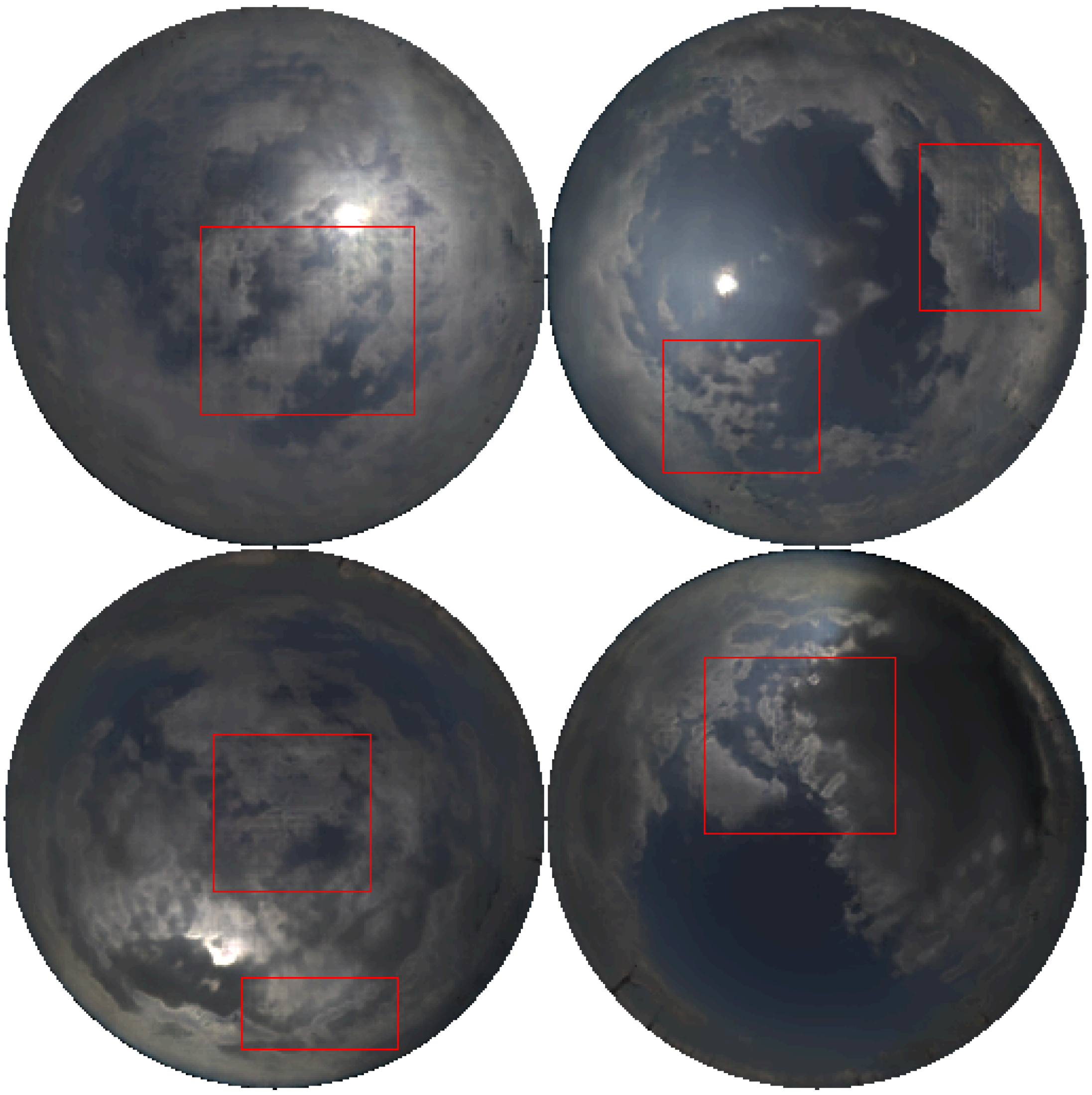


Figure 5.4: When trained on the 714 images 256×256 with arbitrary rotations and flipping, the images generated by ProGAN suffer from serious artifacts. We highlighted some of the artifacts in red rectangles and encourage the reader to zoom in.

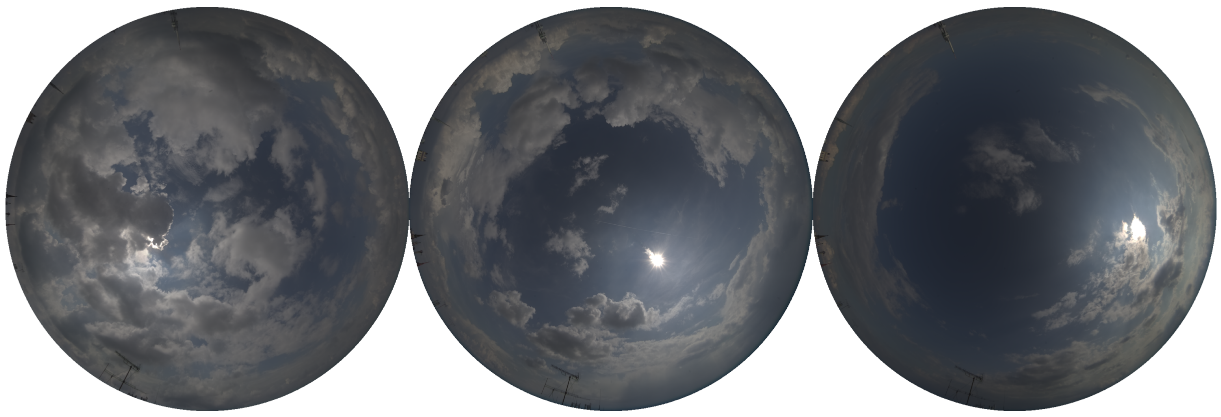


Figure 5.5: The three images that we used to test whether the ProGAN network is able to learn rotations of a single image. We picked images with "interesting" cloud cover.

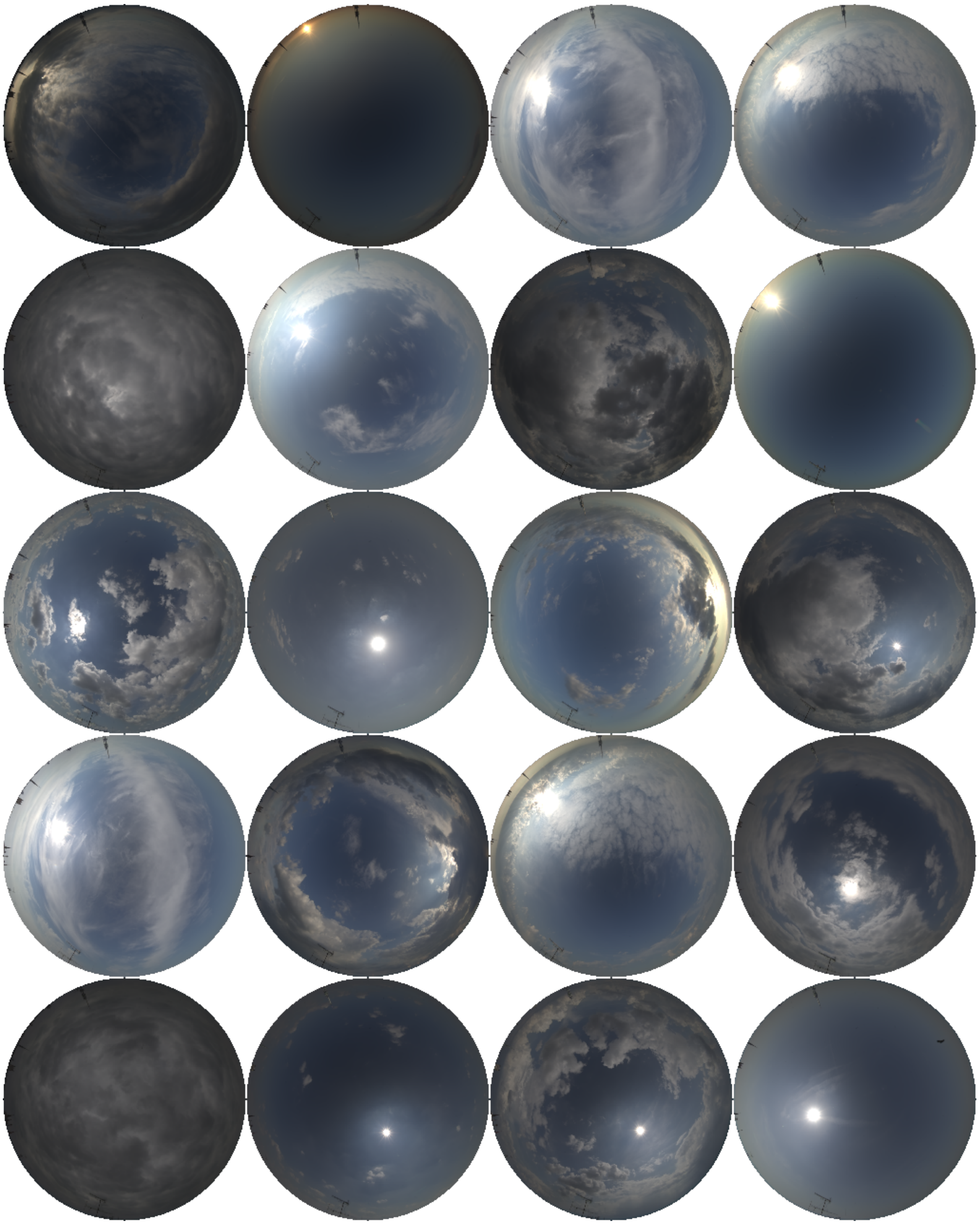


Figure 5.6: Sample from the larger dataset of 7809 images from the Prague rooftop.

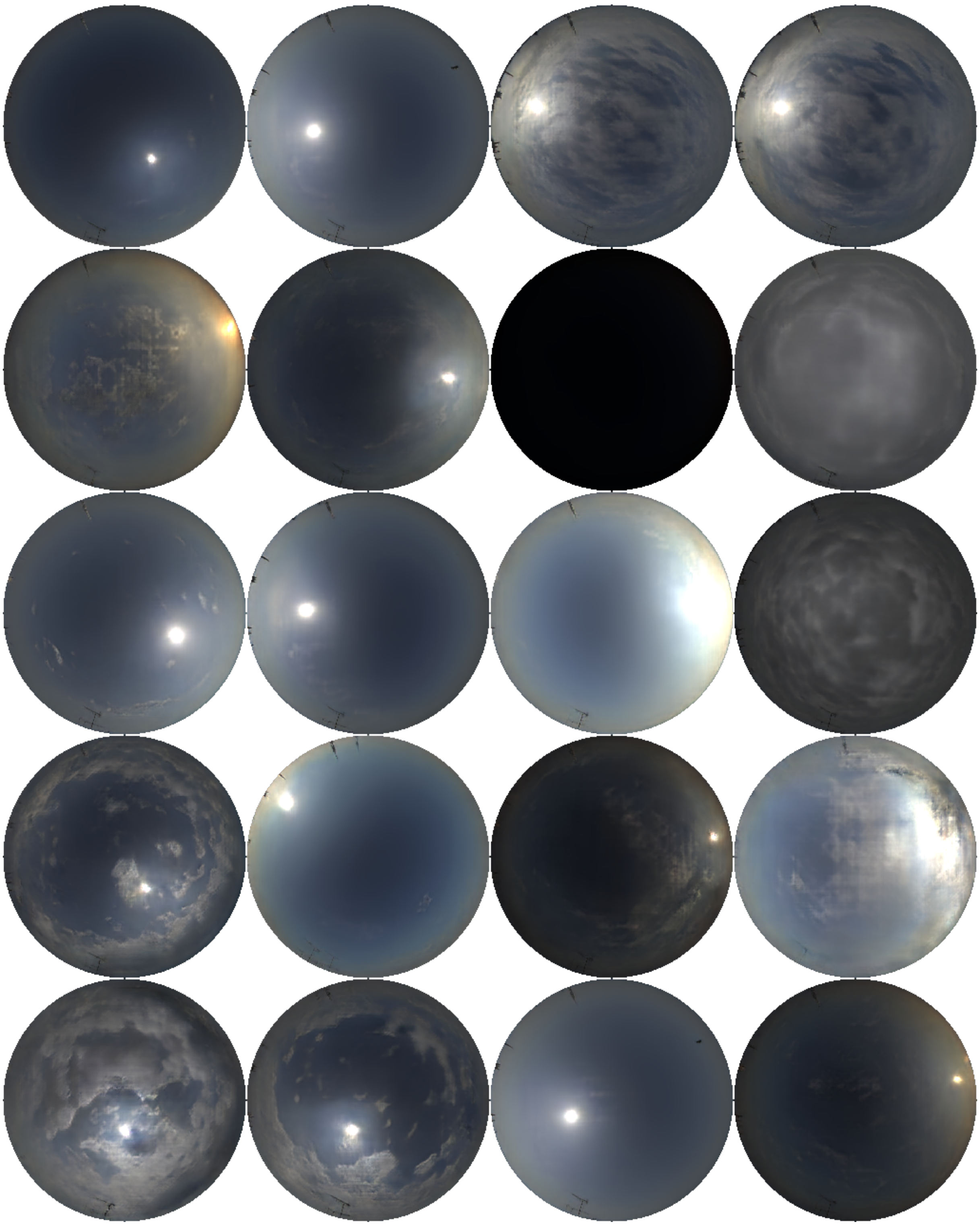


Figure 5.7: Generated images produced by the ProGAN model trained on the larger dataset of 7809 images without augmentation.

5.2 StyleGAN

Disappointed with ProGAN, we turned our attention to StyleGAN. This architecture has significantly higher computational requirements, so we reduced the number of features in both networks from 512 to 128, our StyleGAN networks therefore have approximately $10\times$ less parameters than our ProGAN networks. Initial experiments were quite promising, and optimizing several aspects of the networks (such as the R_1 penalty weight and usage of bilinear interpolation for upsampling in the generator) improved the results even further.

In our initial experiments, we have made a mistake that caused the generator to completely ignore the latent vector and rely solely on the injected noise for all variation. However, the generator was still able to generate quite good images.

5.2.1 Resolution 512×512

Final results from our StyleGAN network at resolution 512×512 are presented in figures 5.8 and 5.9. The images are of high quality, with few artifacts. We also projected a few images from figure 5.8 into equirectangular projection in figure 5.10 to be better able to assess the appearance of generated clouds near the horizon.

To verify that the generator did not just memorize training samples, in figure 5.11 we show 10 randomly generated images together with their nearest neighbor in the training data. The distances were evaluated with a simplified LPIPS metric [49]: we compute features after the fifth convolution from the VGG-19 network, normalize the features in each pixel, and compute L^2 distance between the normalized feature maps. Using a perceptual metric is more appropriate than simple L^2 distance between pixel values, because some image modifications significantly increase L^2 distance without changing the content (small translations), or significantly change the appearance but do not affect L^2 distance much (slight blur). We rotate each training image 12 times to account for the arbitrary rotations used as augmentation.

We also attempted to "invert" the generator and for a random sample of training images find their nearest neighbor in the space of possible synthetic images. We tried the "projection" method from [9], but this did not produce reasonable results, perhaps because we did not use the path-length regularization. In the end, we simply generated 16 000 synthetic images and selected the one with smallest simplified LPIPS distance to each training image. The result of this experiment in figure 5.12 shows that the generator learned to cover the training data very well.

5.2.2 Resolution 1024×1024

We also trained the StyleGAN network at resolution 1024×1024 (see figure 5.13 for example results). We found that the deeper network did not learn as well as the shallower network at resolution 512×512. This might be caused by the greater depth of the network, which makes it harder to train. Using skip or residual connections, as described in StyleGAN 2 [9], should improve the results; we will return to this topic in chapter 6.

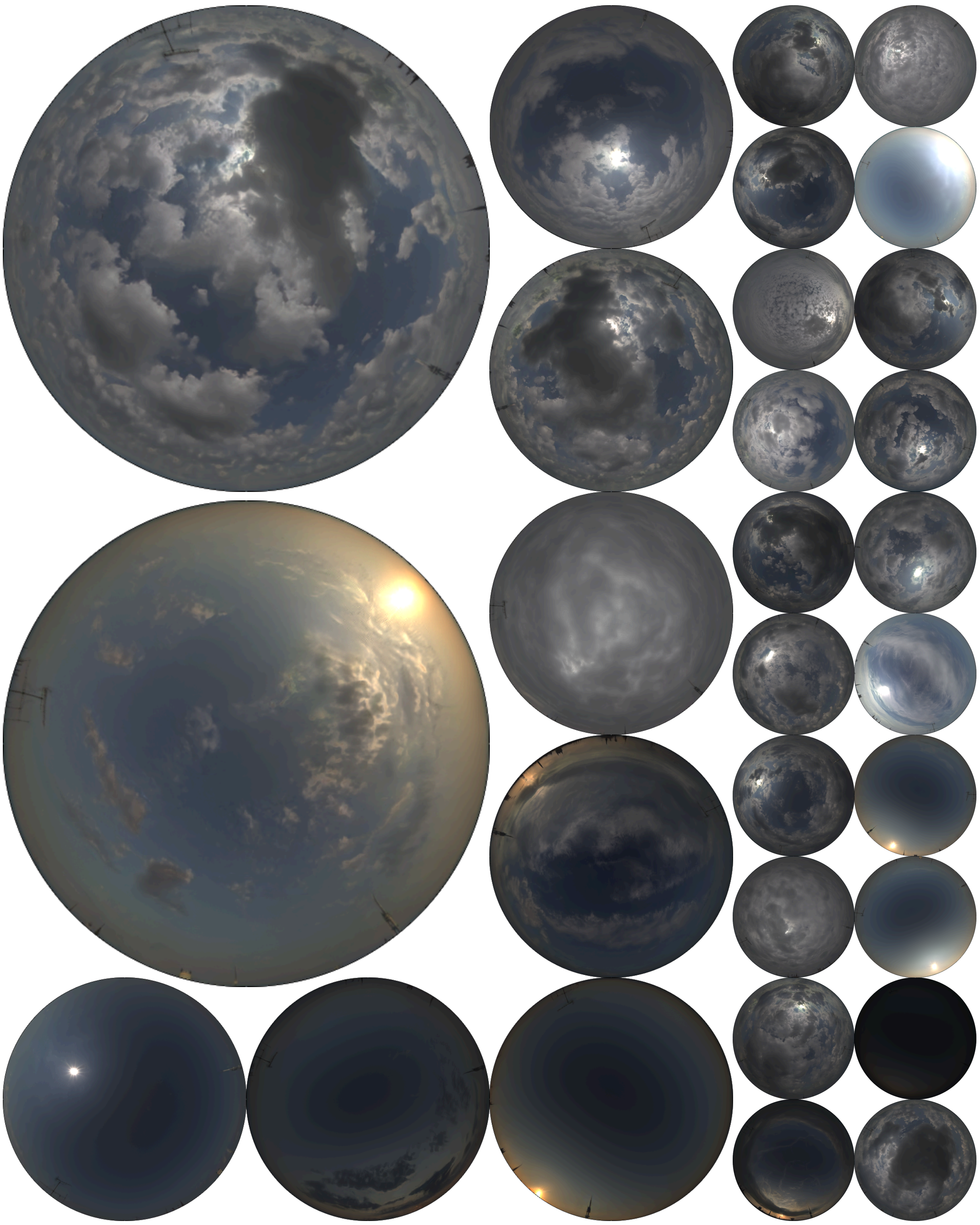


Figure 5.8: Curated sample of images generated by our StyleGAN generator at resolution 512×512.

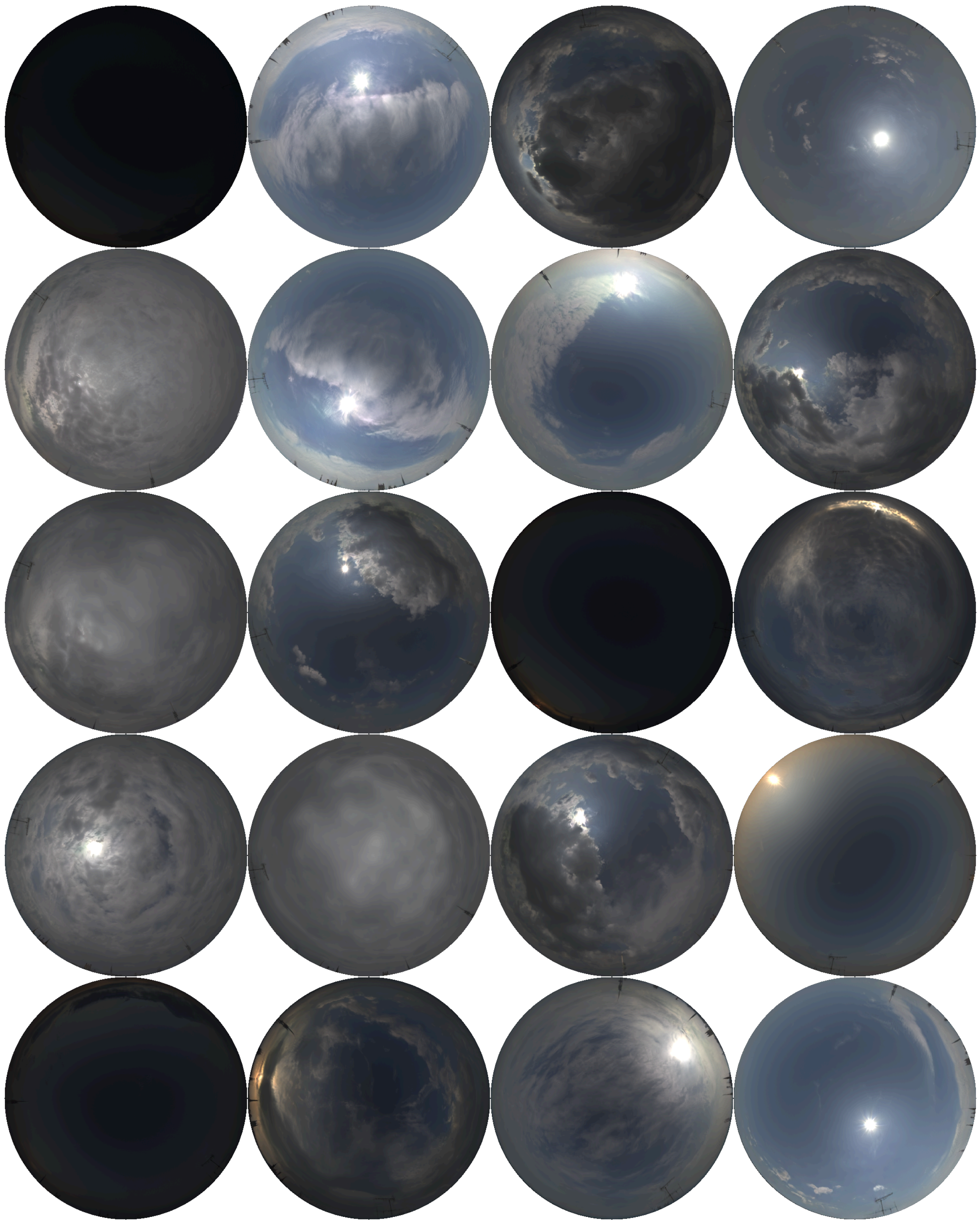


Figure 5.9: Random sample of images generated by our StyleGAN generator at resolution 512×512.

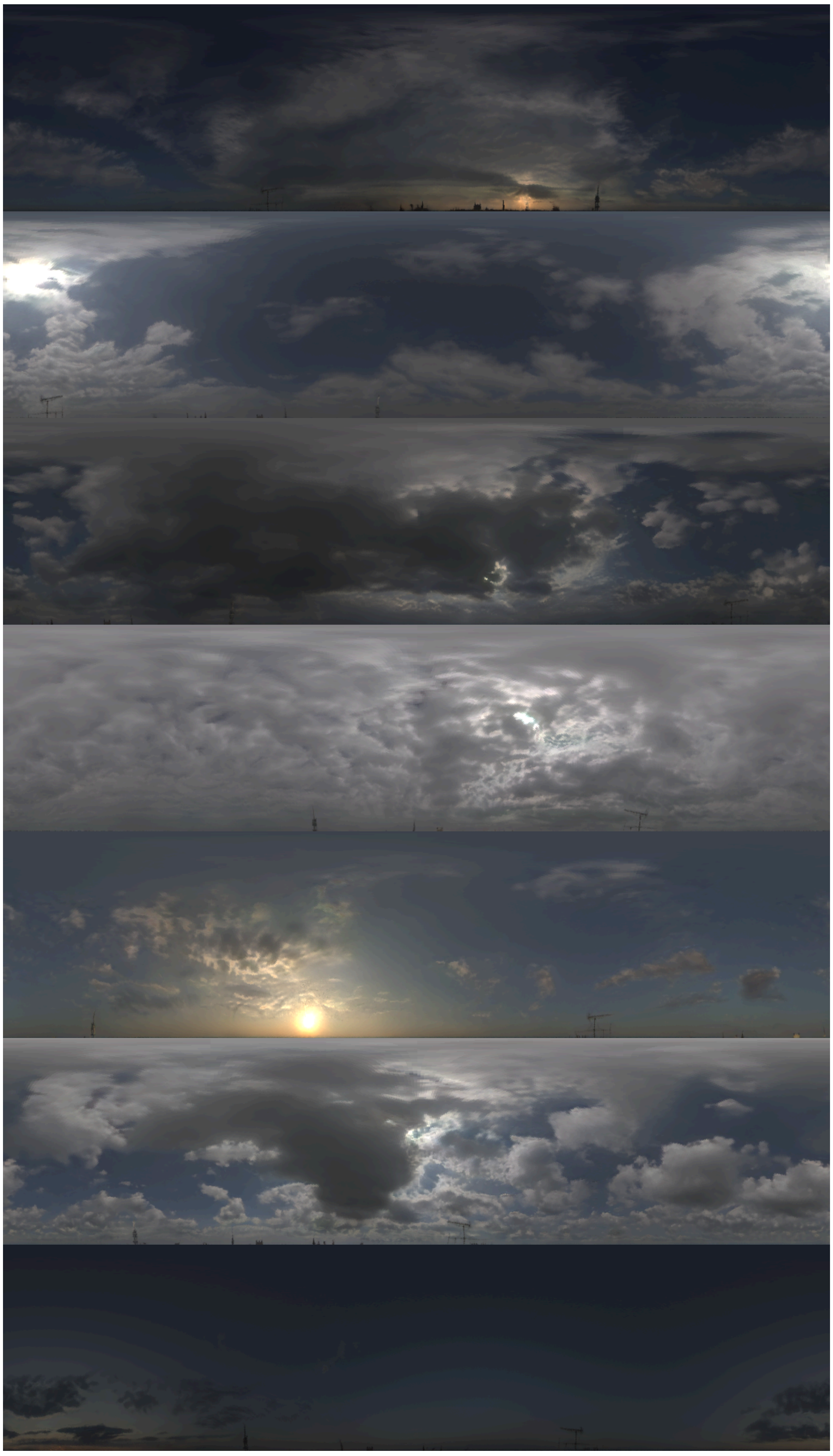


Figure 5.10: Some 512×512 fisheye images synthesized by StyleGAN from figure 5.8 projected into equirectangular projection, which is more natural for assessing the appearance near horizon. We used a simple projection method without antialiasing, so the projections are not of the highest possible quality.

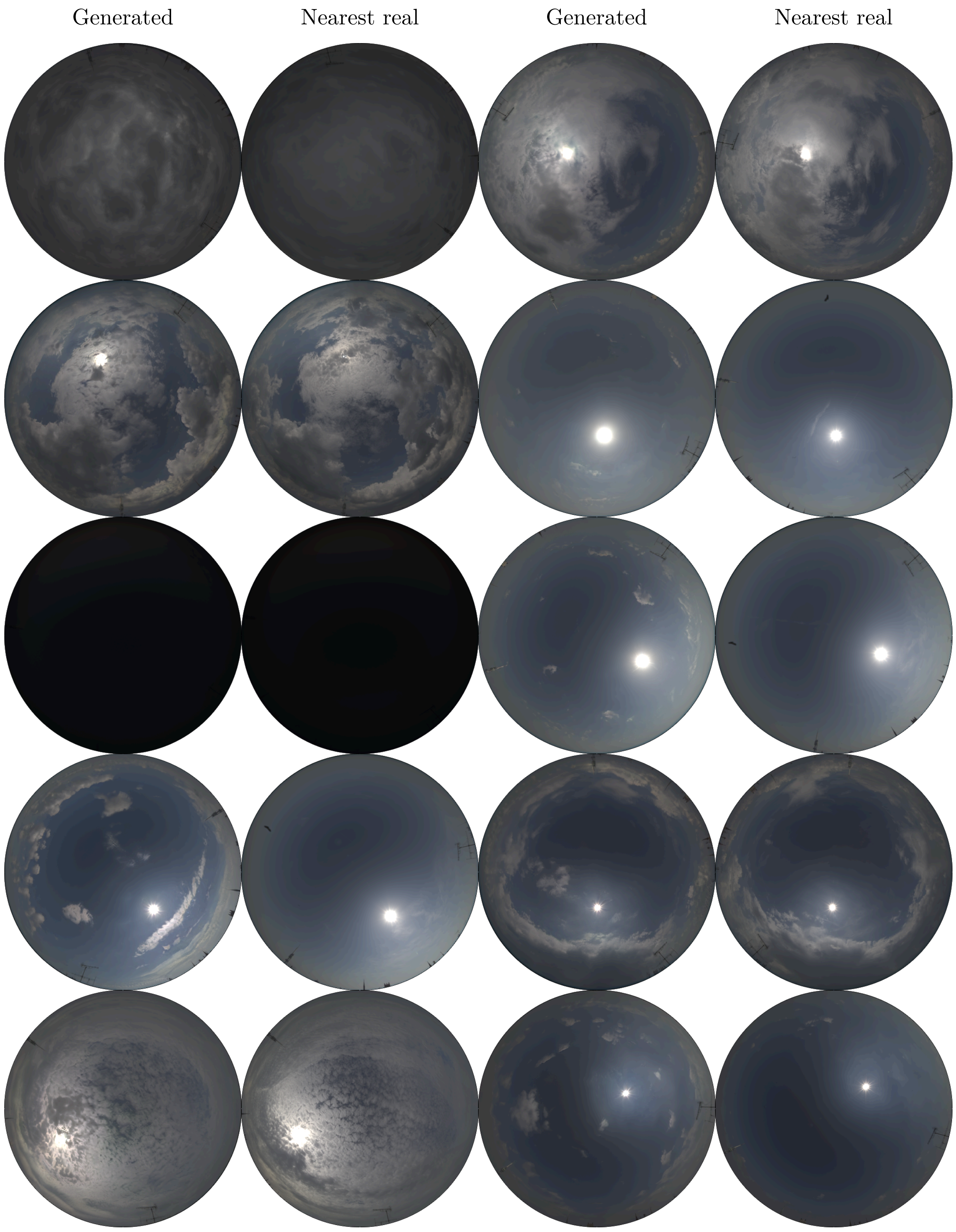


Figure 5.11: To demonstrate that the StyleGAN generator trained on 7809 images at resolution 512×512 did not just memorize training data, we show 10 random generated images and their nearest neighbors in the training data, evaluated using the simplified LPIPS metric described in the main text.

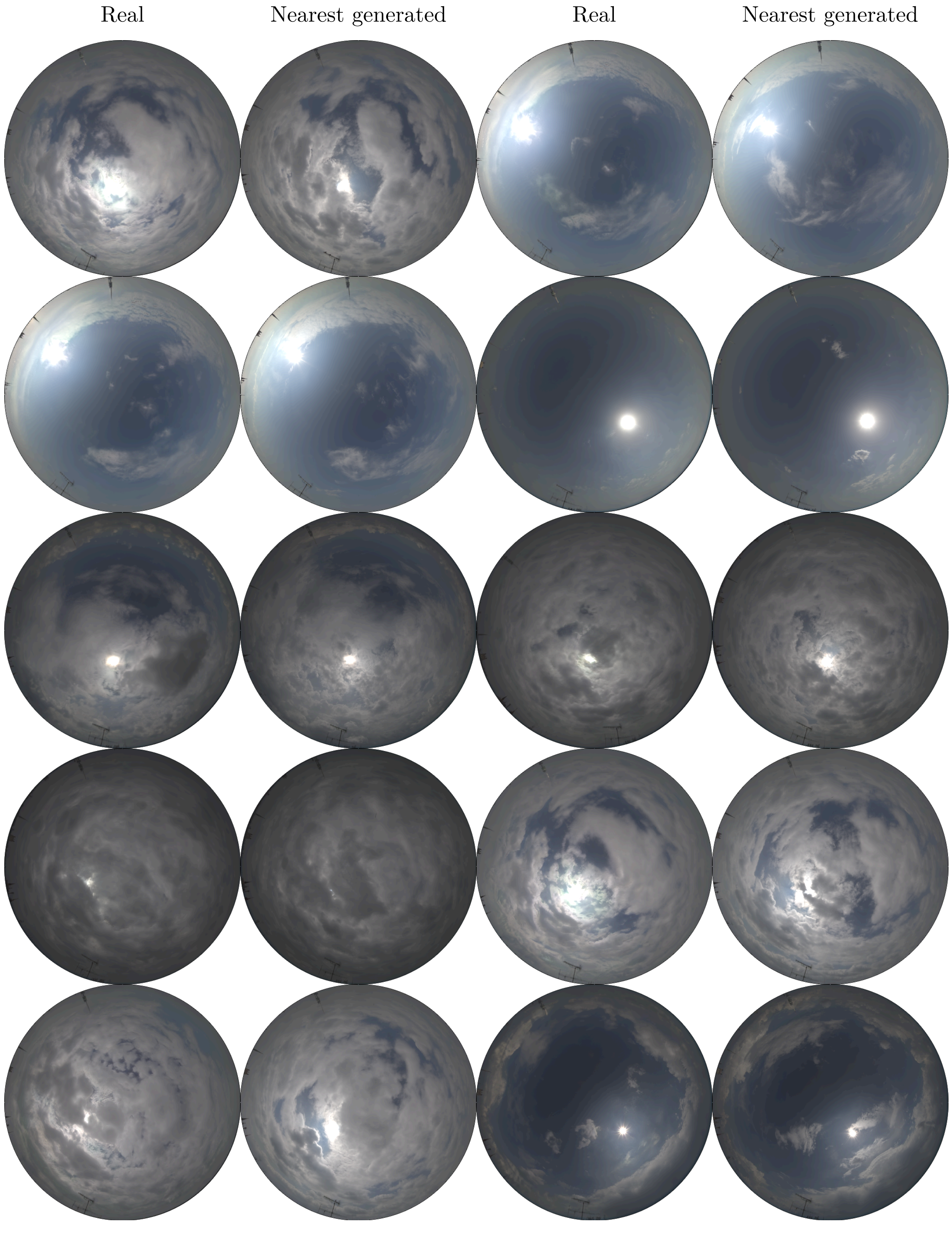


Figure 5.12: To show that the StyleGAN generator trained on 7809 images at resolution 512×512 covers the training data well, we randomly sampled 10 images from the training dataset and found the nearest synthetic images that the generator is able to produce.

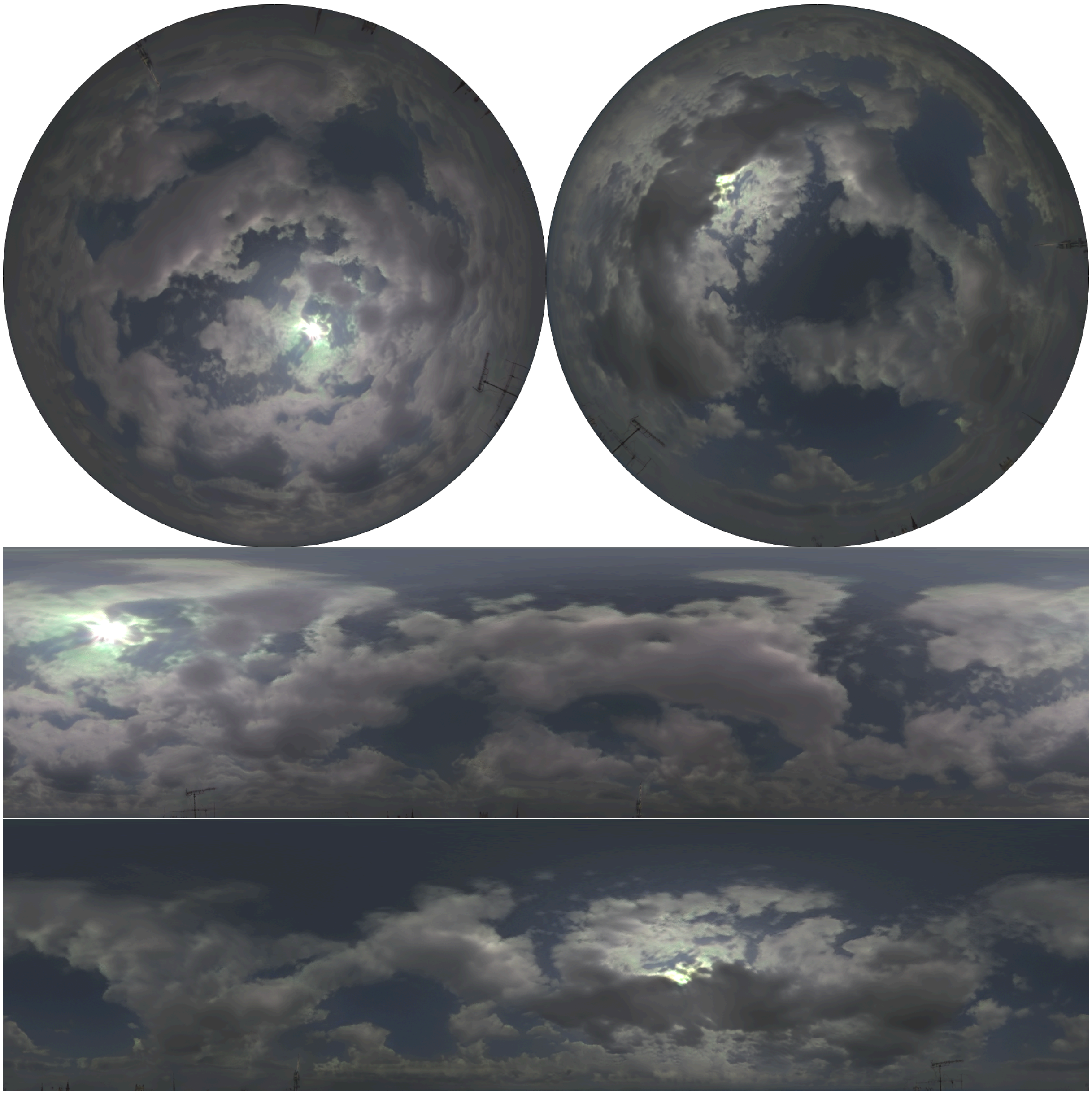


Figure 5.13: Two sample images at resolution 1024×1024 generated by our StyleGAN network. We also include equirectangular projections of these images.

5.2.3 Conclusion

Our StyleGAN network is able to synthesize images in high quality in resolutions up to 512×512. We discuss possible improvements of the architecture that could increase the quality and resolution in chapter 6.

5.3 SuperGAN

We have made some preliminary experiments with our SuperGAN architecture at $4\times$ upscaling, which seemed promising (see figure 5.14). However, at higher upscaling ratios, the quality deteriorates, probably because the patches $\boldsymbol{x}_{\text{fake}}$ evaluated by the patch discriminator D_{patch} become too small relative to the full-size image $\boldsymbol{x}_{\text{full}}$. A possible solution to this problem would be to use multiple patch discriminators at different resolutions. Moreover, the generator G becomes too deep and hard to train without skip connections. Further work is therefore needed to improve the quality of the synthesized images.

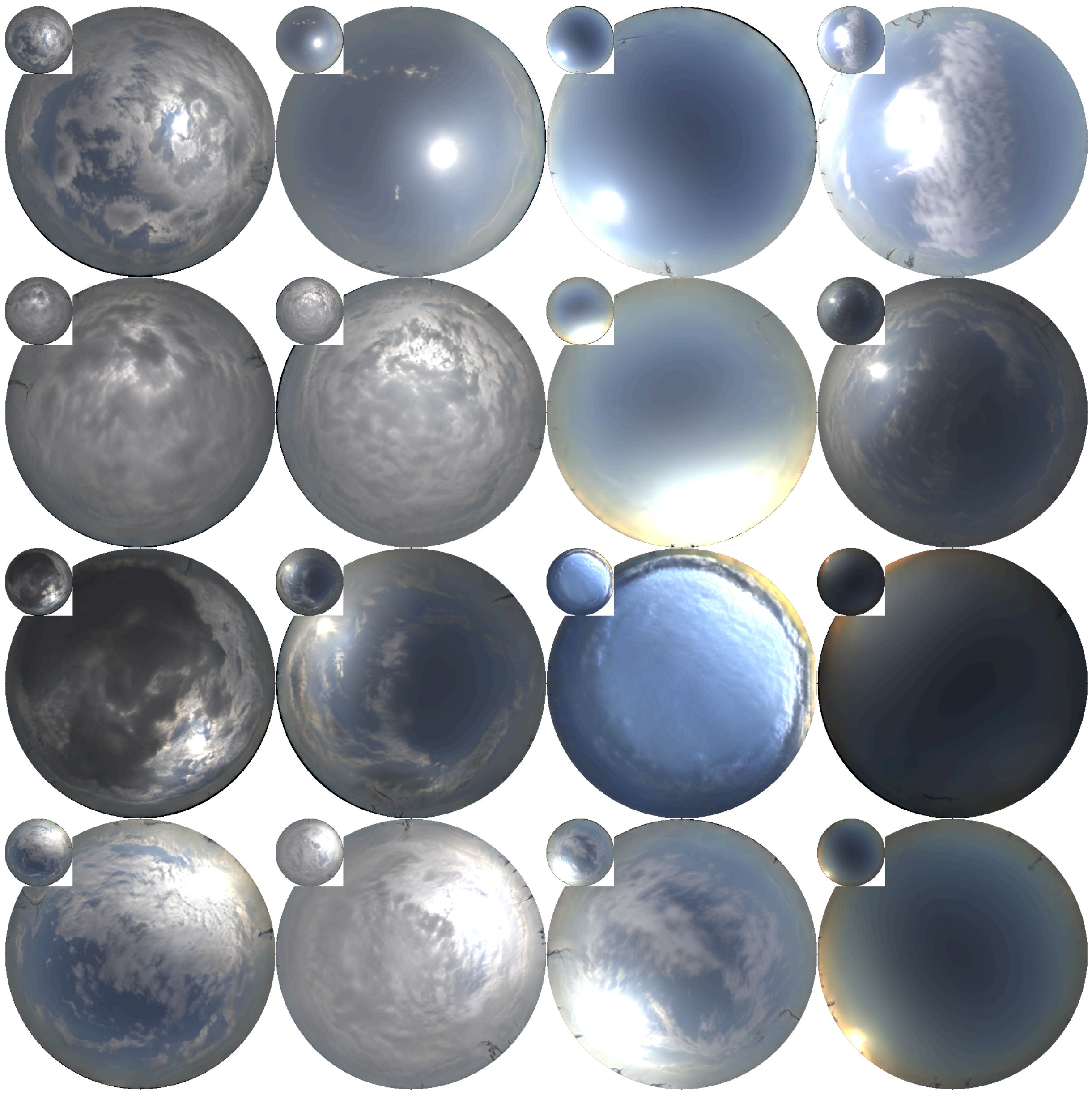


Figure 5.14: Images generated by our SuperGAN network with $R_{\text{full}} = 128$, $R_{\max} = 512$, patch size 96.

6. Conclusion and further work

We moved closer to our goal of generating high-resolution skydome images by significantly improving the quality of images at resolutions up to 512×512 relative to prior work [10]. We achieved this by using the StyleGAN architecture [8, 9], modified to generate HDR fisheye images. We found that the ProGAN architecture [7] used in [10] is significantly weaker, either overfitting to our dataset of producing severe artifacts, perhaps because the network architecture is quite weak. We also experimented with a novel architecture for synthesizing images at very high resolutions, which overcomes the memory and compute limitations that prevent ProGAN and StyleGAN from reaching resolutions beyond 1K.

We identified four areas of research that would have to be explored before the sky model could be used as a useful tool by 3D artists:

High resolution architecture: We successfully generated images at resolution 512×512, but further research is needed to develop an architecture that could synthesize images at resolutions 16K-32K. A model using the super-resolution approach proposed in this work may be able to scale to these resolutions if the architecture is improved.

As we mentioned in section 5.3, multiple patch discriminators will probably be needed to achieve high upscaling rations. Moreover, it will probably be necessary to use skip connections in the generator and residual connections in the discriminator (as described in StyleGAN 2 [9]) to successfully train deeper networks.

High resolution data: To train at very high resolutions, we need high-resolution training data. The dataset collection approach of Hojdar [10] can capture images only at resolutions up to 4K, so a different approach is needed for higher resolutions:

- Train D_{patch} on patches from standard photos that capture only a small section of the sky. Such photos can be shot very easily without any specialized equipment[1]. However, it is unclear whether a discriminator D_{patch} trained on such patches could successfully guide the generation of high-resolution fisheye images.
- Train D_{patch} on images in lower resolution. Clouds have a high degree of self-similarity, so we hypothesize that patches at resolutions 4K-32K are very similar regardless of the precise resolution. This would mean that D_{patch} could be successfully trained on 4K images, which can be captured in large amounts with the fisheye lens using the Hojdar method. However, if this hypothesis was false and patches from 4K images were not similar to patches from 32K images, the discriminator would not perform well and the generated images would be of low quality.
- Train D_{patch} on a small number of high-resolution images. It is possible to stitch a large fisheye image of the whole sky from many small images, each capturing only a part of the sky. This is the approach that artists currently use for obtaining environment maps, but it takes a lot of manual labor and special equipment. However, a trained person should be able to acquire at least a hundred of such images in

[1]We even collected a small dataset of such details as part of the work on this thesis, but did not use it in the end.

a matter of a few weeks. Large number of patches can be extracted from a single 32K image, so the patch discriminator D_{patch} could be successfully trained even on a small dataset.

Artifacts in dataset: Images in our dataset contain some undesired objects near the horizon, such as antennas, towers, buildings, or trees). The number of such disturbances can be reduced by capturing the dataset from a suitable place high above the ground, but some objects will inevitably remain. If these objects are present in the training data, the generator will learn to synthesize them, which is undesirable. A possible solution would be to use some image segmentation technique to detect these objects in the photos and remove them with some image inpaiting method. However, any artifacts produced by the inpainting method would be learned by the model.

Another issue with the dataset is that the sun is severely overexposed, leading to some artifacts and clipped dynamic range of generated images. It is unclear how this could be solved, because capturing the sun correctly with a digital camera requires a neutral-density filter, so the shooting process cannot be easily automated.

Control: To make our method useful to 3D artists, the generator cannot produce completely random images, but there has to be a way to control the synthesis with a small number of intuitive parameters, such as position of the sun, atmospheric conditions, or the current season. The generator may learn some semantic representation of skies in an unsupervised way, as demonstrated in the early work on DCGAN [34] and in StyleGAN [8]. However, we can probably obtain better results if we train the generator in a supervised way to synthesize images conditioned on these parameters. Extending the architecture to model conditional probability distribution $P(\boldsymbol{x} \mid \boldsymbol{y})$ is straightforward [50], but the dataset would have to be annotated with the parameters $\boldsymbol{y}$.

Another way to customize the generated images, which could perhaps complement the parameter-based approach described above, is to provide a "semantic paintbrush": the user creates a segmentation map and the generator synthesizes images that follow this segmentation. Such approach was demonstrated in GauGAN [51], but it requires segmentation maps of the training images.

References

[1] Erik Reinhard et al. *High dynamic range imaging: acquisition, display, and image-based lighting.* Morgan Kaufmann, 2010 (cit. on p. 4).

[2] *HDRI Haven.* https://hdrihaven.com/ (cit. on pp. 4–5).

[3] Lukas Hosek and Alexander Wilkie. "An Analytic Model for Full Spectral Sky-Dome Radiance". In: *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2012)* 31.4 (July 2012). https://cgg.mff.cuni.cz/projects/SkylightModelling/ (cit. on p. 4).

[4] Alexander Wilkie and Lukas Hosek. "Predicting Sky Dome Appearance on Earth-like Extrasolar Worlds". In: *29th Spring conference on Computer Graphics (SCCG 2013).* https://cgg.mff.cuni.cz/projects/SkylightModelling/. Smolenice, Slovakia, May 2013 (cit. on p. 4).

[5] Jan Novák et al. "Monte Carlo methods for physically based volume rendering". In: *ACM SIGGRAPH Courses.* Aug. 2018. ISBN: 978-1-4503-5809-5. DOI: 10/c5fj (cit. on p. 4).

[6] Yoshinori Dobashi, Yusuke Shinzo, and Tsuyoshi Yamamoto. "Modeling of clouds from a single photograph". In: *Computer Graphics Forum.* Vol. 29. 7. Wiley Online Library. 2010, pp. 2083–2090 (cit. on p. 4).

[7] Tero Karras et al. "Progressive growing of GANs for improved quality, stability, and variation". In: *arXiv preprint arXiv:1710.10196* (2017) (cit. on pp. 4, 23, 26–27, 37, 41, 71).

[8] Tero Karras, Samuli Laine, and Timo Aila. "A style-based generator architecture for generative adversarial networks". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition.* 2019, pp. 4401–4410 (cit. on pp. 4, 28–29, 31, 42, 71–72).

[9] Tero Karras et al. "Analyzing and improving the image quality of StyleGAN". In: *arXiv preprint arXiv:1912.04958* (2019) (cit. on pp. 4, 32, 42, 62, 71).

[10] Štěpán Hojdar. *Using neural networks to generate realistic skies.* https://is.cuni.cz/webapps/zzp/detail/213909/. 2019 (cit. on pp. 4, 37–42, 55–56, 71).

[11] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning.* http://www.deeplearningbook.org. MIT Press, 2016 (cit. on p. 6).

[12] Wikipedia contributors. *Static single assignment form — Wikipedia, The Free Encyclopedia.* https://en.wikipedia.org/w/index.php?title=Static_single_assignment_form&oldid=946746964. [Online; accessed 10-April-2020]. 2020 (cit. on p. 6).

[13] Jorge Nocedal and Stephen Wright. *Numerical optimization.* Springer Science & Business Media, 2006 (cit. on p. 7).

[14] Stephen Boyd, Stephen P Boyd, and Lieven Vandenberghe. *Convex optimization.* Cambridge university press, 2004 (cit. on p. 7).

[15] Yoshua Bengio. "RMSProp and equilibrated adaptive learning rates for nonconvex optimization". In: *corr abs/1502.04390* (2015) (cit. on p. 8).

[16] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization.* 2014. arXiv: 1412.6980. URL: https://arxiv.org/abs/1412.6980 (cit. on p. 8).

[17] Sergey Ioffe and Christian Szegedy. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift.* 2015. arXiv: 1502.03167. URL: https://arxiv.org/abs/1502.03167 (cit. on p. 10).

[18] Ilya Loshchilov and Frank Hutter. *Decoupled Weight Decay Regularization.* 2017. arXiv: `1711.05101` (cit. on p. 12).

[19] Sheldon Axler. *Linear algebra done right.* Third edition. Springer, 2015 (cit. on p. 15).

[20] Dmitry Ulyanov, Andrea Vedaldi, and Victor Lempitsky. "Instance normalization: The missing ingredient for fast stylization". In: *arXiv preprint arXiv:1607.08022* (2016) (cit. on p. 15).

[21] Kaiming He et al. "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition.* 2016, pp. 770–778 (cit. on pp. 16–17).

[22] Robert Geirhos et al. *ImageNet-trained CNNs are biased towards texture; increasing shape bias improves accuracy and robustness.* 2018. arXiv: `1811.12231` (cit. on p. 17).

[23] Olga Russakovsky et al. "ImageNet Large Scale Visual Recognition Challenge". In: *International Journal of Computer Vision (IJCV)* 115.3 (2015), pp. 211–252. DOI: `10.1007/s11263-015-0816-y`. URL: `https://arxiv.org/abs/1409.0575` (cit. on p. 17).

[24] J. Deng et al. "ImageNet: A Large-Scale Hierarchical Image Database". In: *CVPR09.* 2009. URL: `http://www.image-net.org/papers/imagenet_cvpr09.pdf` (cit. on p. 17).

[25] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "ImageNet classification with deep convolutional neural networks". In: *Advances in neural information processing systems.* 2012, pp. 1097–1105 (cit. on p. 17).

[26] Karen Simonyan and Andrew Zisserman. *Very Deep Convolutional Networks for Large-Scale Image Recognition.* 2014. arXiv: `1409.1556` (cit. on p. 17).

[27] Kaiming He et al. "Identity mappings in deep residual networks". In: *European conference on computer vision.* Springer. 2016, pp. 630–645 (cit. on p. 17).

[28] Ian Goodfellow et al. "Generative adversarial nets". In: *Advances in neural information processing systems.* 2014, pp. 2672–2680 (cit. on pp. 18–20, 22).

[29] Xudong Mao et al. "Least squares generative adversarial networks". In: *Proceedings of the IEEE International Conference on Computer Vision.* 2017, pp. 2794–2802 (cit. on p. 20).

[30] Xudong Mao et al. "On the effectiveness of least squares generative adversarial networks". In: *IEEE transactions on pattern analysis and machine intelligence* 41.12 (2018), pp. 2947–2960 (cit. on p. 20).

[31] Martin Arjovsky, Soumith Chintala, and Léon Bottou. "Wasserstein GAN". In: *arXiv preprint arXiv:1701.07875* (2017) (cit. on p. 21).

[32] Ishaan Gulrajani et al. "Improved training of Wasserstein GANs". In: *Advances in neural information processing systems.* 2017, pp. 5767–5777 (cit. on p. 21).

[33] Lars Mescheder, Andreas Geiger, and Sebastian Nowozin. "Which training methods for GANs do actually converge?" In: *arXiv preprint arXiv:1801.04406* (2018) (cit. on p. 22).

[34] Alec Radford, Luke Metz, and Soumith Chintala. "Unsupervised representation learning with deep convolutional generative adversarial networks". In: *arXiv preprint arXiv:1511.06434* (2015) (cit. on pp. 22, 24–25, 72).

[35] Tomas Mikolov et al. "Efficient estimation of word representations in vector space". In: *arXiv preprint arXiv:1301.3781* (2013) (cit. on p. 23).

[36] Kaiming He et al. “Delving deep into rectifiers: Surpassing human-level performance on imagenet classification”. In: *Proceedings of the IEEE international conference on computer vision.* 2015, pp. 1026–1034 (cit. on pp. 23, 25).

[37] Phillip Isola et al. “Image-to-image translation with conditional adversarial networks”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition.* 2017, pp. 1125–1134 (cit. on pp. 33–34).

[38] Jun-Yan Zhu et al. “Unpaired image-to-image translation using cycle-consistent adversarial networks”. In: *Proceedings of the IEEE international conference on computer vision.* 2017, pp. 2223–2232 (cit. on pp. 33–34).

[39] Anna Frühstück, Ibraheem Alhashim, and Peter Wonka. “TileGAN: synthesis of large-scale non-homogeneous textures”. In: *ACM Transactions on Graphics (TOG)* 38.4 (2019), pp. 1–11 (cit. on pp. 33–34).

[40] Tamar Rott Shaham, Tali Dekel, and Tomer Michaeli. “SinGAN: Learning a generative model from a single natural image”. In: *Proceedings of the IEEE International Conference on Computer Vision.* 2019, pp. 4570–4580 (cit. on pp. 33, 35).

[41] Chao Dong et al. “Image super-resolution using deep convolutional networks”. In: *IEEE transactions on pattern analysis and machine intelligence* 38.2 (2015), pp. 295–307 (cit. on p. 33).

[42] Bee Lim et al. “Enhanced deep residual networks for single image super-resolution”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition workshops.* 2017, pp. 136–144 (cit. on p. 35).

[43] Mehdi SM Sajjadi, Bernhard Scholkopf, and Michael Hirsch. “EnhanceNet: Single image super-resolution through automated texture synthesis”. In: *Proceedings of the IEEE International Conference on Computer Vision.* 2017, pp. 4491–4500 (cit. on p. 35).

[44] Christian Ledig et al. “Photo-realistic single image super-resolution using a generative adversarial network”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition.* 2017, pp. 4681–4690 (cit. on p. 35).

[45] Xintao Wang et al. “ESRGAN: Enhanced super-resolution generative adversarial networks”. In: *Proceedings of the European Conference on Computer Vision (ECCV).* 2018 (cit. on pp. 35–36).

[46] Zhihao Wang, Jian Chen, and Steven CH Hoi. “Deep learning for image super-resolution: A survey”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2020) (cit. on p. 35).

[47] Adam Paszke et al. “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: *Advances in Neural Information Processing Systems 32.* Ed. by H. Wallach et al. Curran Associates, Inc., 2019, pp. 8024–8035 (cit. on p. 41).

[48] Terrance DeVries and Graham W Taylor. “Improved regularization of convolutional neural networks with cutout”. In: *arXiv preprint arXiv:1708.04552* (2017) (cit. on p. 56).

[49] Richard Zhang et al. “The unreasonable effectiveness of deep features as a perceptual metric”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition.* 2018, pp. 586–595 (cit. on p. 62).

[50] Mehdi Mirza and Simon Osindero. “Conditional generative adversarial nets”. In: *arXiv preprint arXiv:1411.1784* (2014) (cit. on p. 72).

[51] Taesung Park et al. “Semantic image synthesis with spatially-adaptive normalization”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition.* 2019, pp. 2337–2346 (cit. on p. 72).