A Siamese Architecture for Neural Style Transfer

Matthias Wright

Master of Science in Machine Learning and Autonomous Systems

Department of Computer Science
The University of Bath

September 2019

A Siamese Architecture for Neural Style Transfer

submitted by

Matthias Wright

for the degree of Master of Science in Machine Learning and Autonomous Systems

of the

University of Bath

Department of Computer Science

September 2019

DECLARATION

This dissertation is submitted to the University of Bath in accordance with the requirements of the degree of Master of Science in Machine Learning and Autonomous Systems in the Department of Computer Science. No portion of the work in this dissertation has been submitted in support of an application for any other degree or qualification of this or any other university or institution of learning. Except where specifically acknowledged, it is the work of the author.

Signature of Author .....

Matthias Wright

Abstract

In this thesis, we introduced a dual autoencoder architecture that aimed to perform image style transfer without explicitly using RGB-pixel representations. Similar to the CycleGAN architecture (Zhu et al. [38]), the goal was to learn a mapping between two image domains. The purpose of the autoencoders was to learn the most important features of the image domains, thus removing non-essential information. The hypothesis was that the low-dimensional representations, learned through the autoencoders, would model images on a level that is semantically higher than pixel representations.

The mapping between the low-dimensional representations was performed by two deep neural networks (generators), one for each direction. Both networks were paired up with a discriminator to form a Generative Adversarial Network (GAN). We tried both fully-connected neural networks (FCNNs) and convolutional neural networks (CNNs) for the generators. While using FCNNs did not yield visually recognizable style transfer results, we were able to achieve some form of style transfer using CNNs. However, the stylized images were not visually indistinguishable from the images in the corresponding domain.

The results were evaluated quantitatively using the Fréchet Inception Distance (Heusel et al. [55]).

We concluded that the GANs were likely unable to capture the distribution of the low-dimensional representation of the corresponding domain.

Acknowledgements

I am very grateful to my supervisor, Professor Peter Hall, for guiding me through this project. Furthermore, I would like to thank my parents for their support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The motivation for this project lies in the observation that image style transfer is, in reality, texture transfer [1, p.2414]. Until more recently, texture transfer algorithms were limited to the use of low-level image features [1, p.2414]. However, style transfer algorithms should be capable of extracting the semantic content from an image in order to render it into a new style [1, p.2414]. This would necessitate a separation between style and content, which remains a difficult problem [1, p.2415]. This is because it requires the ability to recognize objects regardless of their depiction, which is not yet possible [3]. Furthermore, recognition of image style has been explored very little in computer vision research [2, p.1].
The goal of this project is to perform image style transfer without explicitly using RGB-pixel representations. For this purpose, a dual autoencoder architecture is proposed (see Fig. 1-1). Each autoencoder produces a low-dimensional representation of images from a specific domain (e.g. photographs and oil paintings). The low-dimensional representation aims to efficiently model both the style and the content of an image.

The main purpose of this architecture is to define a mapping between two domains X and Y without using paired training data. This is achieved by defining two unidirectional mappings: $F_X : X \to Y$ and $F_Y : Y \to X$. The aim is that an image $\hat{y} = F_X(x), x \in X$ is visually indistinguishable from images in Y with respect to style. In practice, F_X would define a mapping from domain X to domain $\hat{Y}$, where $\hat{Y}$ is distributed identically to Y [38, p.2243]. However, as stated by Zhu et al., "such a translation does not guarantee that the individual

inputs and outputs x and y are paired up in a meaningful way - there are infinitely many mappings (...) that will induce the same distribution over $\hat{y}$." [38, p.2243] Therefore, we will require the mapping between X and Y to be "cycle consistent" [38, p.2243]. This means that $F_Y(F_X(x)) \approx x$ and $F_X(F_Y(y)) \approx y$.
As stated above, the mappings F_X and F_Y are not directly translating from X to Y and vice versa. Instead, an encoder transforms an input image into its low-dimensional representation; then G_V or G_U will map this representation to the counterpart representation. Afterwards, a decoder will map the image back to the counterpart domain. Mathematically, this means that $\hat{y} = F_X(x) = D_Y(G_V(E_X(x)))$ and $\hat{x} = F_Y(y) = D_X(G_U(E_Y(y)))$. The architecture is described in detail in Sec. 3.1.

1.1 Hypothesis

As stated above, the purpose of the low-dimensional representations is to efficiently model style and content of an image. We argue that a key part of modeling is abstraction. Abstraction is essential in enabling us to reason about high-level image features [40, p.148]. We hypothesize that, by *forcing* the autoencoders to find image representations in much lower dimensions, we remove non-essential information from the images. This would indicate that the low-dimensional representations model images on a level that is semantically *higher* than pixel representations. One can even argue that this kind of representation offers some separation of style and content. However, in this thesis we will not attempt to explicitly perform this separation and instead focus on style transfer.

1.2 Novelty

Although style transfer has already been successfully carried out, we believe that our approach is novel because we attempt style transfer by first finding low-dimensional representations for images. Furthermore, many contemporary style transfer methods [1, 5] use two images: one providing style and one providing content. Our approach uses a single image providing content and then aims to transfer the style from an entire dataset onto it.

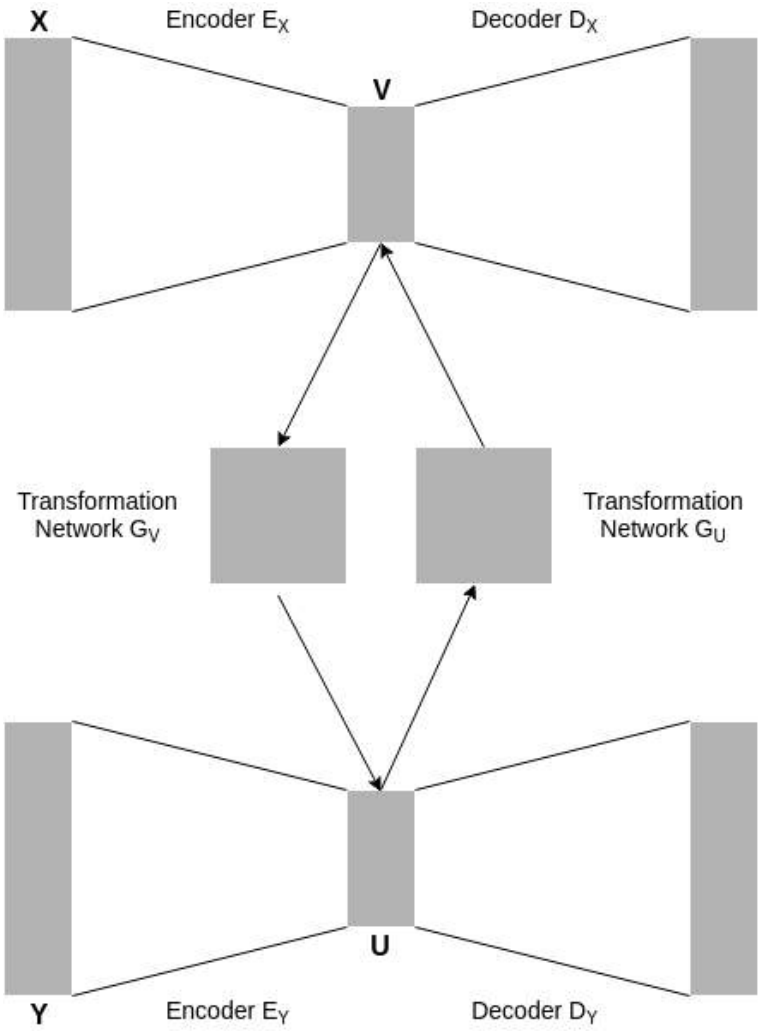

Figure 1-1: Dual autoencoder architecture.

Chapter 2

Background and Related Work

2.1 Neural Networks

The goal of a neural network is to approximate a function f^* [6, p.167]. This is achieved by defining a mapping $y = f(\mathbf{x}; \theta)$ and then *learning* the value of the parameter θ that yields the optimal approximation of f^* [6, p.167]. Neural networks receive raw data input and find multiple levels of representation that can be used for detection or classification [7, p.436]. The representations become more abstract with each further level [7, p.436]. These levels are generally referred to as layers [6, p.167]. The layer which takes in the data is often referred to as the *first layer*, the final layer is called the *output layer*, and the layers in between are called *hidden layers* [6, pp.167-168]. In theory, a neural network with as few as one hidden layer can approximate any Borel measurable function from one finite-dimensional space to another with arbitrary accuracy given that the layer has enough neurons [8, 9].

2.1.1 Learning

The process of finding the value of θ, or the *weights* of the neural network, can loosely be divided into three steps: forward propagation, back-propagation, and updating the weights. The goal of forward propagation is to compute the output of the neural network $\hat{y}$ for a corresponding input $\mathbf{x}$ [6, p.200]. Afterwards, a cost J is computed [6, p.200], which gives measure to the prediction error. The goal of back-propagation [10] is to route the cost information through the network

backwards in order to compute the gradient [6, p.200]. The weights are then updated by using some variation of gradient descent [6, p.200].

2.2 Convolutional Neural Networks

Convolutional neural networks (CNNs) [11] are a special form of neural networks that use convolution in at least one of their layers [6, p.330]. CNNs are designed to process data that is represented by multiple arrays, for example RGB images [7, p.439].
For fully-connected neural networks, the outputs for the current layer are computed by multiplying a weight matrix with the outputs of the previous layer [6, p.335] (and applying a non-linear activation). The weight matrix contains an element for the interaction of every neuron in the current layer and every neuron in the previous layer [6, p.335].
In contrast, the connections between two convolutional layers are *sparse* [6, p.335]. The neurons in these layers are organized in feature maps. A set of weights (called filter) connects each neuron in a feature map to local patches in the feature maps of the previous layer [7, p.439]. While different feature maps within a layer use different weights, neurons in the same feature map share the same weights [7, p.439]. This allows us to detect features in an image consisting of thousands of pixels by only using tens or hundreds of weights and thus reducing the memory consumption as well as the number of operations [6, p.335].
There is another special kind of layer that can be found in most CNN architectures: a *pooling layer*. A neuron in a pooling layer is computed by taking the maximum of a local patch of neurons in a feature map in the previous layer [7, p.439]. Note that pooling layers do not use any weights.
The process of learning the weights of the network works analogously to that of a regular neural network [7, p.439], as described in Sec. 2.1.1.

2.3 Autoencoders

An *autoencoder* is a neural network whose goal it is to recreate its input as its output [6, p.504]. It consists of two parts: an encoder $\mathbf{h} = E(\mathbf{x})$ and a decoder $\hat{\mathbf{x}} = D(\mathbf{h})$ [6, p.504]. An autoencoder is trained by minimizing a loss function

$\mathcal{L}(\mathbf{x}, D(E(\mathbf{x})))$ that measures the difference between $\mathbf{x}$ and $D(E(\mathbf{x}))$. It may seem pointless to train a network that simply copies the input to the output. However, we are generally more interested in the latent representation $\mathbf{h}$ than we are in the output [6, p.505]. The goal is that $\mathbf{h}$ attains useful properties, which may be achieved by choosing $\mathbf{h}$ to have a smaller dimension than $\mathbf{x}$ [6, p.505]. So-called *undercomplete* autoencoders are *forced* to capture the most important features of the data [6, p.505].

2.4 Generative Adversarial Networks

Generative Adversarial Networks (GANs) are a framework for estimating generative models by employing an adversarial training process, in which a generative model G and a discriminative model D are simultaneously trained [12]. Generally, both G and D are neural networks [12, p.3]. A prior distribution on input noise variables $p_z(z)$ is defined and $G(z)$ represents a mapping to the data space [12, p.3]. $D(\mathbf{x})$ gives measure to the probability that $\mathbf{x}$ originates from the data rather than from G [12, p.3]. During training D aims to maximize the probability of assigning the correct label to both training examples as well as generated samples from G [12, p.3]. G is trained to minimize $\log(1 - D(G(z)))$ [12, p.3].
Since GANs were proposed by Goodfellow et al. in 2014, several different variations have emerged. Among them is the Wasserstein GAN proposed by Arjovsky et al. [13]. It employs the Wasserstein distance for the objective function in order to improve the learning stability [13].

2.5 Related Work

2.5.1 Texture Synthesis and Transfer

Image style transfer can be regarded as a problem of texture transfer [1, p.2414]. The goal of texture transfer is to synthesize a texture from a source image and to transfer it onto a target image while preserving the semantic content of the target image by constraining the texture synthesis [1, p.2414].
Texture synthesis has been actively researched in the field of computer vision [15, p.1033]. In 1999, Efros and Leung performed texture synthesis by modeling

texture as a Markov random field [15, p.1034]. They assumed the probability distribution of brightness values for a pixel p to be independent of the rest of the image given the brightness values of its spatial neighborhood $w(p)$ [15, p.1034]. The value of a pixel p was synthesized by approximating the conditional probability distribution $P(p|w(p))$ and then sampling from it [15, p.1034]. Similar approaches were introduced by Paget and Longstaff [16] and Zhu et al. [17].
Although Markov random fields have been shown to be a good approximation for many textures types, they have the disadvantage that sampling from them is computationally expensive [18, p.480]. Wei and Levoy proposed an acceleration for this synthesis process by employing tree-structured vector quantization [18]. However, all of these methods "operate within the greedy single-pixel-at-a-time paradigm" and are thus prone to falling into the wrong part of the search space [19, p.342].
Motivated by the observation that - for "one-pixel-at-a-time" texture synthesis algorithms - most pixel values are entirely determined by what has been synthesized so far, Efros and Freeman proposed a texture synthesis algorithm which uses image patches as the unit of synthesis [19, p.342]. An image is synthesized by repeatedly picking an image patch (called block) that satisfies some overlap constraints and by then minimizing the error surface between the newly chosen block and the previously chosen blocks at the overlap region [19, p.342]. The authors augmented this algorithm to perform texture transfer by additionally requiring each new block to satisfy a desired correspondence map, a spatial map of a corresponding quantity over both the texture image and the target image [19, p.345]. However, as stated by Gatys et al., although these algorithms achieve notable results, they are bounded by the fundamental limitation that they only make use of low-level image features of the target image in order to conduct the texture transfer [1, p.2414].

2.5.2 Non-Photorealistic Rendering

Artistic image generation is not a particularly new research area in computer graphics. In this section, we will consider Non-photorealistic Rendering (NPR), a research field of computer graphics that "brings together art and science, concentrating less on the process and more on the communication content of an image."

[20, p.2] Photorealistic rendering generally aims to achieve the highest possible level of detail, even if it makes the image cluttered or confusing [20, p.2]. In contrast, artists use their artwork to communicate "information that may not be readily apparent in photographs or real life." [20, p.1] Using computer-generated images for the same purpose is the motivation for the field of NPR [20, p.1].

Early works in the field include an interactive program by Haeberli, which was designed to create static and animated abstract images of photographs and synthetic scenes [21]. The program follows the cursor across the canvas, obtains a color by point sampling the source image, and then paints a brush of that color [21, p.208]. Instead of drawing a line, the program generates a scattering of brush strokes by creating brush stroke locations from a probability distribution in the neighborhood of the cursor [21, p.208]. The author's intention was to convert a synthetic or natural scene into an abstract impressionist image [21, p.207].

Hertzmann extended this line of research to "automated painting and drawing without human intervention" [22, p.453] by proposing a method for creating an image with a "hand-painted appearance" from a photograph, which is capable of describing a wide range of visual styles [22] (see Fig. 2-3, Fig. 2-1, and Fig. 2-4). The algorithm receives a source image and a list of brush sizes (expressed as radii $R_1, ..., R_n$) as input and then, starting from a constant color image, proceeds to create a series of layers, one for each brush size [22, p.454]. For each layer R_i, a reference image is created by blurring the source image, a process that is performed by convolution with a Gaussian kernel of standard deviation $f_\sigma R_i$, where f_σ is a constant factor [22, p.454]. This reference image is then approximated by finding areas of the image that differ from the reference image and covering them with new brush strokes of the current brush size R_i [22, p.454]. The key idea is that each brush is supposed to capture only those details of the source image that have at least the same size as the brush [22, p.454]. Furthermore, curved brush strokes are incorporated by modeling brush strokes as anti-aliased cubic B-splines [22, p.455]. Hertzmann and Perlin extended this algorithm to painterly rendering of videos [23].

Gooch et al. proposed a similar approach that takes an input image and produces a painting-like image composed of brush strokes [24] (see Fig. 2-2). Their algorithm first segments the input image and then smooths the boundaries of each segmented region, which are used to calculate brush stroke paths [24, p.84]. Then,

the central axis (called the ridge set) of each segment is approximated discretely [24, p.84]. The elements of the ridge set are pieced together into tokens that can be spatially ordered into lists [24, p.84]. The brush stroke paths are then rendered as brush strokes [24, p.84]. A key difference to the method by Hertzmann [22] is that Gooch et al. made use of image segmentation.

In 2001, Hertzmann et al. introduced the concept of images analogies [25] (see Fig. 2-5). In contrast to the methods by Haeberli [21], Hertzmann [22], and Gooch et al. [24], their algorithm does not explicitly perform brush strokes and instead takes three images as input: the unfiltered source image A, the filtered source image A', and the unfiltered target image B [25, p.329]. The output is the filtered target image B' [25, p.329], which relates to B in "the same way" as A' relates to A [25, p.327]. This is achieved by constructing the Gaussian pyramid [26] representations of A, A', and B, and then for each level, the statistics of each pixel in the target pair are compared to statistics for each pixel in the source pair in order to find the closest-matching pixel [25, pp.329-330]. This approach is similar to the aforementioned texture synthesis algorithms proposed by Efros and Leung [15] and Wei and Levoy [18].

Other work in the field of NPR include the generation of artificial classic mosaics based on principled global optimization [27] or stylized abstraction of photographs using flow-based filtering [28].

Similar to the methods in Sec. 2.5.1, many of these methods have the limitation that they only use low-level image features [29, p.1]. Furthermore, these algorithms are constructed to render a source image in one specific style [1, p.2421].

2.5.3 Neural Style Transfer

In 2016, Gatys et al. proposed a novel algorithm that enabled them to reproduce different styles from famous paintings onto natural images [29, p.1] (see Fig 2-6). This has lead to the emergence of a new research field called *Neural Style Transfer* (NST), which is based around the process of using neural networks to render image content in different styles [29, p.2]. The algorithm by Gatys et al. poses image style transfer as an optimization problem within a single neural network [1, p.2416]. The neural network used by the authors was the VGG network [14], a CNN trained to perform object recognition and localization [1, p.2416]. The loss

function is composed of two weighted components: a content loss and a style loss [1, p.2419].

Let us first look at the content loss: an input image is encoded in each layer of a CNN by the filter responses to that image [1, p.2416]. Given an original image p, gradient descent can be used to find another image x that causes the same feature responses [1, p.2416]. Let $P^{[l]}$ and $F^{[l]}$ denote the respective feature representation of p and x in layer l and $\mathcal{L}_{content}$ the squared-error loss between $P^{[l]}$ and $F^{[l]}$. The gradient of $\mathcal{L}_{content}$ with respect to the image x is then computed using back-propagation, which allows us to iteratively update x until it causes the same response in a certain layer as the original image p [1, p.2416].

For the style loss, the Gram matrix consisting of the correlations between different filter responses is used [1, p.2416]. Given an input image a, we aim to find an image x that matches the style representation of a using gradient descent [1, p.2417]. Let $A^{[l]}$ and $G^{[l]}$ denote the respective Gram matrix of a and x in layer l [1, p.2417]. The contribution E_l of layer l to the overall style loss is given by the sum of the squared distances between the entries of $A^{[l]}$ and $G^{[l]}$ [1, p.2417]. The style loss $\mathcal{L}_{style}$ is then given by the weighted sum of E_l for all layers [1, p.2417]. The gradient of $\mathcal{L}_{style}$ with respect to x can be computed using back-propagation, which allows us to iteratively update x until it matches the style representation of the original image a [1, p.2417].

In order to transfer the style of an image a onto an image p, a new image is synthesized that matches the content representation of p and the style representation of a [1, p.2417].

While the NST algorithm by Gatys et al. has produced remarkable results, it is subject to some limitations. Gatys et al. have stated that CNNs are capable of capturing high-level content "in terms of objects and their arrangement in the input image" [1, p.2416] and that style transfer algorithms should be able to "extract the semantic image content from the target image." [1, p.2414] However, Gatys et al. themselves have stated about their method [1] that it is "based on a parametric texture model defined by summary statistics" [42, p.3731], which is a contradiction to the earlier claim.

The algorithm is also subject to some technical limitations. The dimension of the optimization problem and the number of neurons in the CNN grow linearly as a function of the number of pixels [1, p.2421]. Therefore, the duration of the

transfer process depends heavily on the image resolution, which prohibits online and interactive applications [1, p.2421]. Moreover, the synthesized images are sometimes subject to low-level noise [1, p.2421]. Another issue is that CNN features inevitably neglect some low-level information, which is why the algorithm does not perform well in preserving the coherence of details during stylization [29, p.5]. Additionally, brush stroke variations and the depth information contained in the content image are not considered [29, p.5] and when both the content image and the style image are photographs, the output image still tends to look like a painting [5, p.6697].

In order to improve photographic style transfer, Luan et al. [5] augmented the algorithm proposed by Gatys et al. [1] (see Fig. 2-9). The authors introduced a "photorealism regularization term" into the style transfer objective function [5, p.6999]. By constraining the transformation that is applied to the content image, the output image is ensured to only be represented by locally affine color transformations [5, pp.6999-7000]. In addition, the style transfer process is augmented by semantic segmentation of the inputs [5, p.6999]. Although this method preserves the detail structures of the content image and produces images that are photorealistic, the "stylishness" is greatly reduced [34, p.1718].

Johnson et al. proposed a similar algorithm that is capable of performing style transfer without having to solve an optimization problem at test time [30]. This enables a drastically improved speed-up compared to the method by Gatys et al. [30, p.707]. The architecture by Johnson et al. consists of an image transformation network f_W and a loss network ϕ [30, p.698]. The image transformation network f_W transforms input images x into output images $\hat{y}$, where $\hat{y} = f_W(x)$ [30, p.697]. The loss network ϕ is the 16-layer VGG network [14] pretrained on the ImageNet dataset [31] for image classification [30, p.699]. ϕ is used to define a feature reconstruction loss l^{ϕ}_{feat} and a style reconstruction loss l^{ϕ}_{style} that give measure to the differences between content and style between two images [30, p.699]. l^{ϕ}_{feat} and l^{ϕ}_{style} are defined similarly to $\mathcal{L}_{content}$ and $\mathcal{L}_{style}$ in Gatys et al. [1], respectively [30, pp.700-701]. f_W is trained using stochastic gradient descent to minimize the weighted combination of l^{ϕ}_{feat} and l^{ϕ}_{style} [30, p.697]. For each image x, there is a content target y_c and a style target y_s [30, p.698]. The output image $\hat{y}$ should combine the content target y_c with the style of y_s, where $y_c = x$ [30, p.698].

Ulyanov et al. [32] proposed an architecture that is similar to that of Johnson et al. (see Fig. 2-8). Instead of using the image transformation network f_W, they used a generator network g, which takes a noise sample and a content image as input and then outputs a new image depicted in the new style [32, p.1351]. The objective function that is used to assess the quality of the generated image is similar to the one introduced by Gatys et al. [32, p.1352].
Ulyanov et al. later augmented this approach by applying normalization to every single image instead of a badge of images to improve the performance of the generator [33, p.4106]. Additionally, they introduced a new objective function that learns generators that uniformly sample from the Julesz ensemble [33, p.4106].
In order to address the loss of detail that may occur during stylization, Li et al. propose to incorporate additional constraints related to the content detail structures into the optimization objective of the style transfer [34, p.1716]. The authors obtain the Laplacian matrix $D(x)$ of an image x by convolving the image with the Laplacian filter D [34, p.1719]. The Laplacian filter is the discrete approximation of the two dimensional Laplacian operator [34, p.1719]. The Laplacian operator is commonly used for the extraction of detail structures in an image [34, p.1719]. The Laplacian loss $\mathcal{L}_{lap}$ between the content image and the synthesized image is then defined to be the Euclidean distance between their respective Laplacian matrices [34, p.1719]. The style transfer procedure is then formulated as proposed by Gatys et al. [1] with the difference that the overall objective function is no longer the weighted combination of $\mathcal{L}_{content}$ and $\mathcal{L}_{style}$ but the weighted combination of $\mathcal{L}_{content}$, $\mathcal{L}_{style}$, and $\mathcal{L}_{lap}$ [34, p.1719].
Although the methods proposed by Johnson et al. [30], Ulyanov et al. [32, 33], and Li et al. [34] are capable of performing style transfer two orders of magnitude faster than the method by Gatys et al. [1], they require that a separate neural network is trained for each style [29, p.7]. To address this limitation, Xu et al. proposed a feed-forward network for arbitrary style transfer that can handle unseen content and style image pairs [35]. The authors utilize an encoder-decoder architecture for their transformation network [35, p.3] that takes in a content and a style image as input and then outputs the stylized image [35, p.5]. The encoder uses convolutional layers from the VGG network [14] that was pretrained on the ImageNet dataset [31]. The decoder is designed "almost symmetric" to the encoder and is trained from scratch [35, p.4]. Unlike the authors of the aforemen-

tioned approaches, the authors of this approach introduce a GAN into their model [35, p.3]. The transformation network serves as the conditional generator and is paired up with a discriminator that is conditioned on the style category labels [35, p.3]. The discriminator in this GAN is different from those in traditional GANs in that it has two tasks. First, it is supposed to distinguish between real and fake images and, secondly, it is supposed to predict the style category [35, p.3]. The generator is updated by minimizing the weighted combination of adversarial loss, style classification loss, content loss, and style loss [35, p.5]. Content loss and style loss are defined as proposed by Gatys et al. [1]. The discriminator is trained to maximize the probability of correctly identifying real and fake examples as well as the probability of correctly predicting the style category [35, p.6]. The architecture is trained with diverse multi-domain images [35].

Another approach for transferring arbitrary visual styles onto content images was contributed by Li et al. [36]. In contrast to the previous methods mentioned in this section, the authors employ an autoencoder to formulate style transfer as an image reconstruction problem combined with feature transformation [36, p.3]. The pretrained VGG network [14] serves as the encoder and a decoder network is trained to reconstruct the VGG features back to the input image [36, p.3]. The decoder is constructed to be symmetrical to the VGG network (up to the Relu_X_1 layer), using up-sampling to enlarge the feature maps [36, p.3]. Five different feature maps from five different layers are selected to train five separate decoders [36, p.3]. The objective function is given by the weighted combination of the reconstruction loss and the content loss [36, p.3]. The content loss is defined as proposed by Gatys et al. [1]. For a given content image I_c and style image I_s, the corresponding feature maps f_c and f_s are computed with the encoder [36, p.3]. Before the reconstruction of I_c from f_c, a whitening and coloring transform is applied to adjust f_c with respect to the statistics of f_s [36, p.4].

Closely related to style transfer is the concept of *image-to-image translation* that addresses the problem of learning a mapping from input image to output image [37]. Zhu et al. extended this concept in order to translate an image from a source domain to a target domain without paired training data [38]. Their approach can also be used to perform style transfer [38, p.2247]. The model includes two mappings: $G : X \rightarrow Y$ and $F : Y \rightarrow X$ with corresponding training data $\{x_i\}_{i=1}^{N} \in X$ and $\{y_j\}_{j=1}^{M} \in Y$ [38, p.2244]. Two adversarial discriminators D_X

and D_Y are introduced, where D_X distinguishes between images from $\{x_i\}_{i=1}^{N}$ and translated images $\{F(y_j)\}_{j=1}^{M}$ and D_Y distinguishes between images from $\{y_j\}_{j=1}^{M}$ and translated images $\{G(x_i)\}_{i=1}^{N}$ [38, p.2244]. In addition to the regular GAN loss function, a "cycle consistency loss" is introduced that encourages $F(G(x)) \approx x$ and $G(F(y)) \approx y$ [38, p.2243]. The architecture for the generators is taken from Johnson et al. [30] and the discriminators use PatchGANs as described by Isola et al. [37] [38, p.2245].

2.5.4 Comparing Neural Style Transfer and Non-Photorealistic Rendering

This section aims to compare NPR and NST first with respect to their methodology and then their results. When one compares the two fields, it is important to note that many of the NPR methods discussed in Sec. 2.5.2 were proposed in the late 1990s and the early 2000s, while the NST methods discussed in Sec. 2.5.3 were proposed from 2016 onward.

While both NPR and NST approaches generally perform a similar task, the specific algorithms are very different. A key difference between NPR and NST algorithms is that, while NST algorithms generally *learn* how to perform style transfer by training on image data, NPR algorithms do not.

Most NPR algorithms are designed for a particular style and cannot be extended to other styles [29, p.1]. All of the methods reviewed in Sec. 2.5.2 receive one image as input (the content image) and then perform the transformations. The only exception is the image analogies framework by Hertzmann [25]. In contrast, all of the NST methods reviewed in Sec. 2.5.3, except the work by Zhu et al. [38], receive two images as input: content image and style image. This enables the transfer of many different styles because the style information is not explicitly encoded in the algorithm but rather provided as an image. In this regard, the image analogies method by Hertzmann [25] has similarities to the contemporary NST methods.

NST algorithms generate the stylized image either directly through gradient descent [1, 30] or by first training a generator network that accomplishes this task in a feed-forward process [32, 33, 34, 35, 36, 38]. In contrast, NPR algorithms often perform brush strokes explicitly [21, 22, 24]. One might argue that this is

closer to the process an artist goes through when creating artwork. However, as Hertzmann stated with respect to NPR algorithms: "Few, if any, algorithms can be said to model the process by which artists operate." [40, p.148]

In the following paragraphs, we will compare some of the stylized images produced by NPR and NST methods. This leads us to the question of how to evaluate methods designed for artistic purposes [40, p.150]. Hall and Lehmann have argued against the use of the Turing test or quantitative measures [39, pp.335-336]. We will therefore qualitatively compare stylized images from NPR and NST against each other and then against artwork produced by humans [39, p.338]. When comparing and evaluating images we will use three criteria: consistency, content preservation, and authenticity. We define a generated image as consistent when the imposed style is applied to the whole image. Content preservation will be judged by the extent to which the original image content is still present in the stylized image. Authenticity will be gauged in comparison to artwork produced by humans. This is, admittedly, the most subjective of the three categories.
At the heart of NPR methods is the concept of abstraction. As stated by Gooch, "NPR images convey information more effectively by omitting extraneous detail, focusing attention on relevant features." [41, p.165] And further, "much of NPR research is not about what we add, but what we can take away." [41, p.165] Two abstract art movements that immediately come to mind when observing NPR artwork are Expressionism and Impressionism. This claim is in line with the goal formulated by Haeberli to take a synthetic or natural scene and transform it into an "abstract impressionistic image." [21, p.207] Hertzmann even intentionally emulated the categories "Impressionist", "Expressionist", "Colorist Wash", and "Pointillist" in his experiments [22, p.458]. Incidentally, many of the style images used to inform the style transfer methods reviewed in Sec. 2.5.3 are images of artwork that belongs to the movements of Expressionism and Impressionism [1, 30, 32, 33]. Therefore, we will use artwork created by humans from those very movements for the comparison.

Consistency Both NPR and NST methods perform very well in this category. The style is consistently applied to all areas of the image in all of the NPR (see Fig. 2-1, 2-2, 2-3, 2-4, 2-5) and NST (see Fig. 2-6, Fig. 2-7, Fig. 2-8, and Fig.2-9)

examples.

Content Preservation NPR methods perform well in this category. Fig. 2-3 and Fig. 2-5 show that even content from the background of the content image is retained in the stylized image. Note that for the other NPR examples, the authors did not include the original image in their publication.
Not every NST method performs well in this category. While the approaches by Ulyanov et al. [33] (see Fig. 2-8) and Luan et al. [5] (see Fig. 2-9) preserve the majority of the image content, the same cannot be stated for the method by Gatys et al. [1]. Fig. 2-6 shows that only the content in the front of the content image is preserved. This limitation has been addressed in Sec. 2.5.3. As stated by Jing et al., the algorithm by Gatys et al. "does not consider the variations of brush strokes and the semantics and depth information contained in the content image, which are important factors in evaluating the visual quality." [29, p.5]

Authenticity We will compare the examples against the two art pieces illustrated in Fig. 2-10 and Fig. 2-11.
Not all NPR methods perform well in this category. However, we regard Fig. 2-1, Fig. 2-2, Fig. 2-4, and Fig. 2-5 as authentic artwork. Fig. 2-1 and Fig. 2-5 could be seen as Impressionist artwork and Fig. 2-2 and Fig. 2-4 could be seen as Expressionist artwork. However, we regard Fig. 2-3 as not authentic. This is because the image looks blurred but also contains many tiny points. One could argue that it is an artwork from the Pointillist movement. We argue, however, that it does not convey the same atmosphere.
NST methods perform well in this category. Even though the examples by Gatys et al. [1] did not preserve all of the content, this does not hurt their authenticity. The examples in Fig. 2-6 and Fig. 2-8 could be seen as Expressionist artwork.

Figure 2-1: Result from the NPR method proposed by Hertzmann [22]. Input image is left and output image is right.

Figure 2-2: Result from the NPR method proposed by Gooch et al. [24].

Figure 2-3: Result from the NPR method proposed by Hertzmann et al. [22].

Figure 2-4: Result from the NPR method proposed by Hertzmann et al. [22].

Figure 2-5: An image analogy from the NPR method proposed by Hertzmann et al. [25]. B' relates to B in "the same way" as A' relates to A [25].

Figure 2-6: Style transfer results by Gatys et al. [1]. The original photograph depicting the Neckarfront in Tübingen, Germany, is shown in **A** (Photo: Andreas Praefcke). The painting that provided the style for the respective generated image is shown in the bottom left corner of each panel. **B** *The Shipwreck of the Minotaur* by J.M.W. Turner, 1805. **C** *The Starry Night* by Vincent van Gogh, 1889. **D** *Der Schrei* by Edvard Munch, 1893. **E** *Femme nue assise* by Pablo Picasso, 1910. **F** *Composition VII* by Wassily Kandinsky, 1913.

Figure 2-7: Style transfer results by Gatys et al. [1].

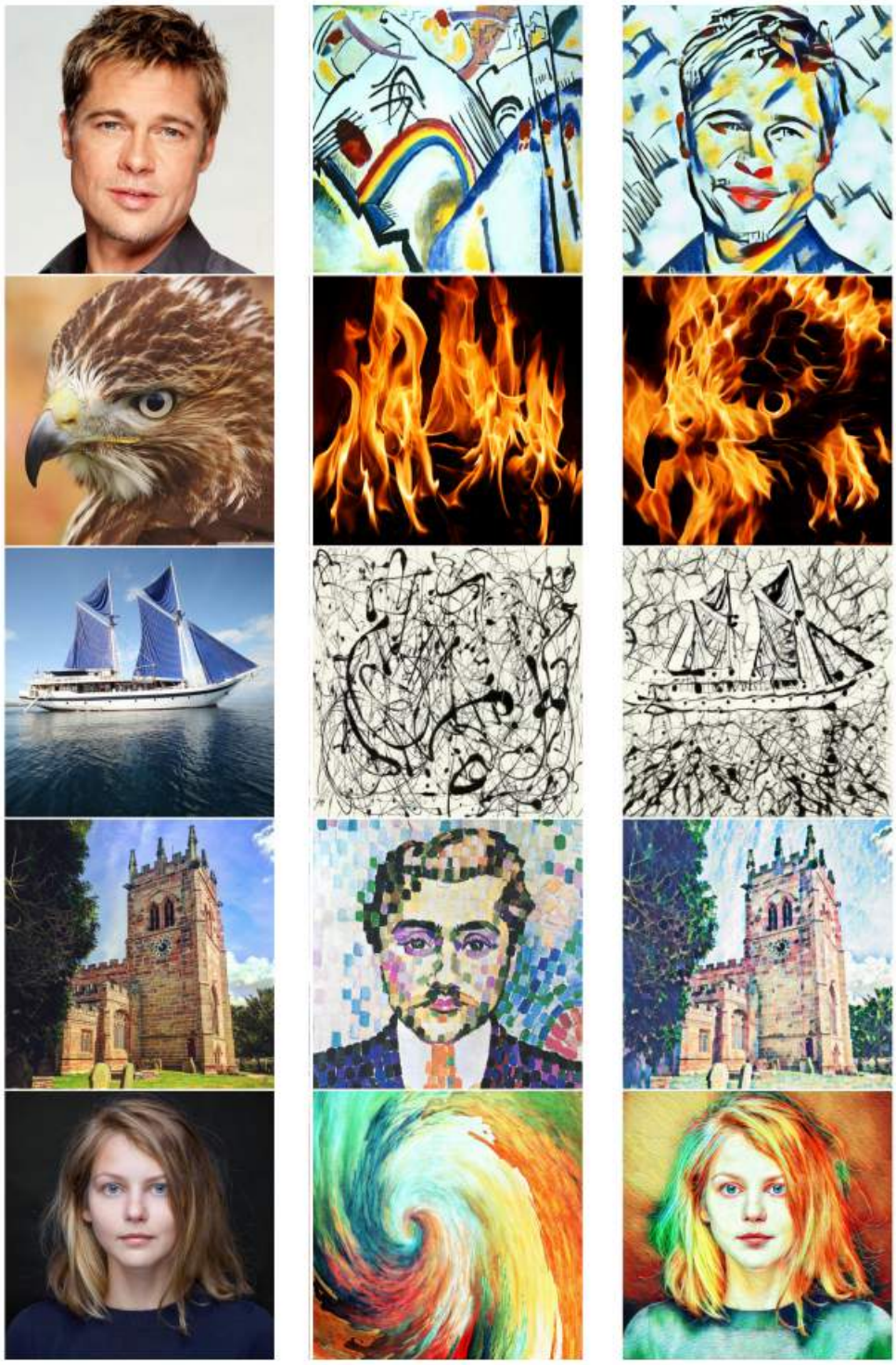

Figure 2-8: Style transfer results by Ulyanov et al. [33]. The left column contains the content images, in the middle column are the style images, and the right column consists of the output images [33].

Figure 2-9: Style transfer results by Luan et al. [5]. Left is the content image, in the middle is the style image, and right is the output image [5].

Figure 2-10: *The Starry Night* by Vincent van Gogh, 1889.

Figure 2-11: *Autumn on the Seine, Argenteuil* by Claude Monet, 1873.

2.6 Our Architecture

As stated above, the purpose of the architecture we propose is to perform image style transfer between two domains. The architecture draws on many of the previous methods reviewed in Sec. 2.5. As opposed to Gatys et al. [1] and Johnson et al. [30] and similar to Ulyanov et al. [32, 33], Xu et al. [35], Li et al. [36], and Zhu et al. [38], our method generates the stylized image through a feed-forward process. Our approach has similarities to the encoder-decoder architecture proposed by Xu et al. [35]. The difference is that we employ two autoencoders. Furthermore, we do not explicitly perform transformations on the low-dimensional representation. Similarly to Zhu et al. [38], we define a mapping between two image domains. A key difference is that we are not directly mapping between two domains but rather between low-dimensional representations of the domains.

Chapter 3

Materials and Methods

3.1 Architecture

As stated in Ch. 1, the goal of the proposed architecture is to define a mapping between two domains X and Y without using paired training data. This mapping is realized by two unidirectional mappings: $F_X : X \to Y$ and $F_Y : Y \to X$. The aim is for the mappings F_X and F_Y to perform a style transfer, which means that an image $\hat{y} = F_X(x), x \in X$ is visually indistinguishable from images in Y with respect to style and vice versa. In order to ensure that the inputs and outputs x and y "are paired up in a meaningful way", we will enforce a *cycle consistency* constraint for the mapping between X and Y [38, p.2243]. This means that $F_Y(F_X(x)) \approx x$ and $F_X(F_Y(y)) \approx y$.
As stated in Ch. 1, an autoencoder A_X produces a low-dimensional representation v of images in X. A_X consists of an encoder E_X and a decoder D_X, defined as $E_X : X \to V$ and $D_X : V \to X$, respectively. Analogously, the autoencoder A_Y produces a low-dimensional representation u of images in Y and consists of an encoder E_Y and a decoder D_Y, defined as $E_Y : Y \to U$ and $D_Y : U \to Y$, respectively. The aforementioned mappings (transformation networks) between the low-dimensional representations are defined as $G_V : V \to U$ and $G_U : U \to V$. The mappings F_X and F_Y are not directly translating from X to Y and vice versa. Instead, an encoder transforms an input image into its low-dimensional representation; then G_V or G_U will map this representation to the counterpart representation. Afterwards, a decoder will map the image back to the counterpart

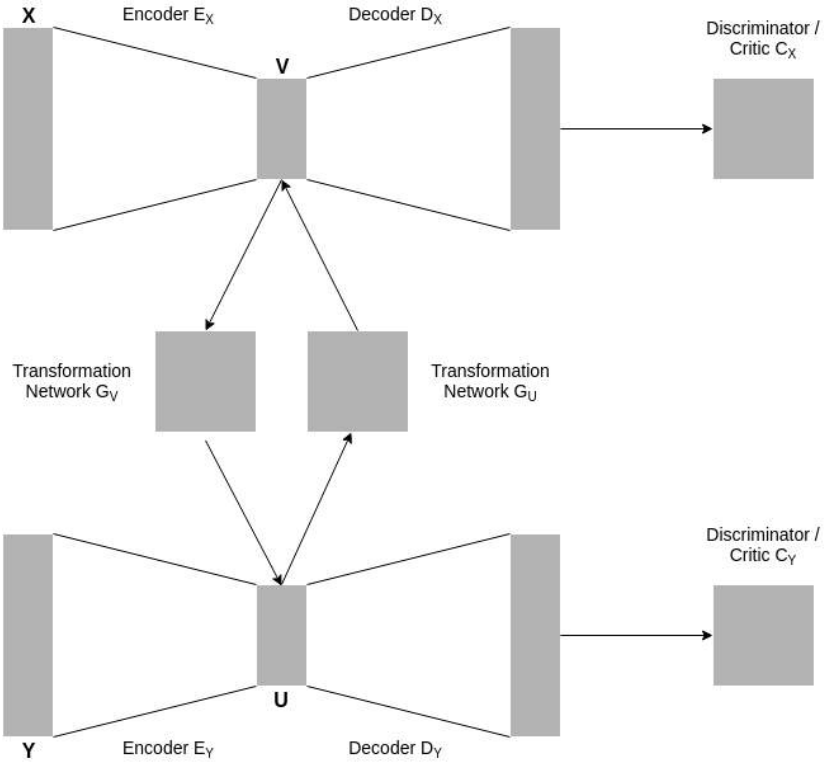

Figure 3-1: Dual autoencoder architecture with two discriminators.

domain. Mathematically, this means that $\hat{y} = F_X(x) = D_Y(G_V(E_X(x)))$ and $\hat{x} = F_Y(y) = D_X(G_U(E_Y(y)))$.
Similarly to Zhu et al. [38, p.2243], we introduced two discriminators C_X and C_Y into the architecture. The goal of C_X is to distinguish between images from $x \in X$ and transformed images $\hat{x} = F_Y(y), y \in Y$. Conversely, the goal of C_Y is to distinguish between images from $y \in Y$ and transformed images $\hat{y} = F_X(x), x \in X$. Fig. 3-1 shows a diagram of the architecture.

3.1.1 Training Procedure

It is important to note that in all experiments both autoencoders were pretrained and fixed during the training procedure of the architecture.

Cycle Consistency Loss

The cycle consistency losses encourage images that are translated to the counterpart domain and then translated back to look approximately like the original. The losses are given by:

$$\begin{aligned}\mathcal{L}_X(G_V, G_U) &= \mathbb{E}_{x\sim p_X}[||x - F_Y(F_X(x))||_1] \\ &= \mathbb{E}_{x\sim p_X}[||x - D_X(G_U(E_Y(D_Y(G_V(E_X(x))))))||_1]\end{aligned} \tag{3.1}$$

$$\begin{aligned}\mathcal{L}_Y(G_V, G_U) &= \mathbb{E}_{y\sim p_Y}[||y - F_X(F_Y(y))||_1] \\ &= \mathbb{E}_{x\sim p_X}[||y - D_Y(G_V(E_X(D_X(G_U(E_Y(y))))))||_1]\end{aligned} \tag{3.2}$$

We use L1 distance instead of L2 distance, because the L1 distance is less prone to blurring [37, p.5969].

Adversarial Losses

The transformation networks G_V and G_U can be thought of as adversarial generators in the architecture. The transformation network G_V and the discriminator C_Y as well as the transformation network G_U and the discriminator C_X form two GANs. The difference with respect to ordinary GANs is that G_V and G_U do not generate images directly. They merely map one latent representation to the other, which is then passed through the decoder of the corresponding autoencoder to generate an image. This is a key difference to the work by Zhu et al. [38]. The GANs were trained as Wasserstein GANs as proposed by Arjovsky et al. [13]. However, as proposed by Gulrajani et al. [43], instead of clipping the weights of the discriminators, we used a gradient penalty to satisfy the Lipschitz constraint. The loss function for G_V is given by:

$$\mathcal{L}_G(G_V) = \mathbb{E}_{x\sim p_X}[-\log(C_Y(\hat{y}))] \tag{3.3}$$

where $\hat{y} = D_Y(G_V(E_X(x)))$ and the loss function for G_U is given by:

$$\mathcal{L}_G(G_U) = \mathbb{E}_{y\sim p_Y}[-\log(C_X(\hat{x}))] \tag{3.4}$$

where $\hat{x} = D_X(G_U(E_Y(y)))$. The loss function for C_X is given by:

$$\mathcal{L}_C(C_X) = \mathbb{E}_{y\sim p_Y}[C_X(\hat{x})] - \mathbb{E}_{x\sim p_X}[C_X(x)] + \lambda\mathbb{E}_{y\sim p_Y}[(||\nabla_{\tilde{x}}C_X(\tilde{x})||_2 - 1)^2] \quad (3.5)$$

where $\hat{x} = D_X(G_U(E_Y(y)))$, $\tilde{x} = \epsilon x + (1-\epsilon)\hat{x}$, and $\epsilon \sim U(0,1)$. The loss function for C_Y is given by:

$$\mathcal{L}_C(C_Y) = \mathbb{E}_{x\sim p_X}[C_Y(\hat{y})] - \mathbb{E}_{y\sim p_Y}[C_Y(y)] + \lambda\mathbb{E}_{x\sim p_X}[(||\nabla_{\tilde{y}}C_Y(\tilde{y})||_2 - 1)^2] \quad (3.6)$$

where $\hat{y} = D_Y(G_V(E_X(x)))$, $\tilde{y} = \epsilon y + (1-\epsilon)\hat{y}$, and $\epsilon \sim U(0,1)$. The parameter λ is the *penalty coefficient*. Gulrajani et al. [43] propose to set $\lambda = 10$, which we adopted.

Alg. 1 describes the training procedure for training the architecture.

Algorithm 1 Training procedure of the architecture

Models to train: G_V, G_U, C_X, C_Y
Data: $\{x^{(i)}\}_{i=1}^m \sim p_X$ and $\{y^{(i)}\}_{i=1}^m \sim p_Y$
Input: pretrained autoencoders $A_X = \{E_X, D_X\}$ and $A_Y = \{E_Y, D_Y\}$

for each epoch **do**
  for $i = 1, ..., m$ **do**
    Compute $\hat{y}^{(i)} = D_Y(G_V(E_X(x^{(i)})))$
    Update G_V with respect to $\mathcal{L}_G(G_V)$ ▷ see Eq. 3.3
    Update C_Y with respect to $\mathcal{L}_C(C_Y)$ ▷ see Eq. 3.6
    Map $\hat{y}^{(i)}$ back to X domain with $D_X(G_U(E_Y(\hat{y}^{(i)})))$
    Update G_V and G_U with respect to $\beta\mathcal{L}_X(G_V, G_U)$ ▷ see Eq. 3.1

    Compute $\hat{x}^{(i)} = D_X(G_U(E_Y(y^{(i)})))$
    Update G_U with respect to $\mathcal{L}_G(G_U)$ ▷ see Eq. 3.4
    Update C_X with respect to $\mathcal{L}_C(C_X)$ ▷ see Eq. 3.5
    Map $\hat{x}^{(i)}$ back to Y domain with $D_Y(G_V(E_X(\hat{x}^{(i)})))$
    Update G_V and G_U with respect to $\beta\mathcal{L}_Y(G_V, G_U)$ ▷ see Eq. 3.2
  end for
end for

Note that for the sake of simplicity, Alg. 1 only describes updates using individual

training examples. In our implementation we used mini-batches of size 50.
Similar to Zhu et al. [38], we scaled the cycle consistency losses with a factor β, which aims to balance the two different types of losses for the transformation networks G_V and G_U. Note that Zhu et al. [38] denote this factor with λ, however, we chose to use β to avoid confusion with the penalty coefficient of the Wasserstein GAN.

3.1.2 Autoencoders

As mentioned above, the autoencoders were pretrained and fixed during the training procedure of the architecture. This section contains implementation details concerning the training of the autoencoders, both of which use the same architecture.
As stated in Sec. 1.1, the purpose of finding a low-dimensional representation for an image is the removal of non-essential information. This was achieved by setting the latent space of the autoencoders to be lower-dimensional than the input space. Learning an undercomplete representation forces autoencoders to capture the most important features of the data [6, p.505] (see Sec. 2.3).
The encoder consists of convolutional layers that shrink the input image into its low-dimensional representation. From there, the decoder scales it back up to its original dimension. Initially, this was achieved using deconvolution (also called transposed convolution) as proposed by Zeiler et al. [44]. However, after observing *checkerboard artifacts*[1] in the output image, we replaced all deconvolutional layers with the combination of nearest-neighbor interpolation followed by a convolutional layer (as proposed by Odena et al. [45]). Fig. 3-2 shows the architecture that was used for both autoencoders.

[1] Checkerboard artifacts are checkerboard patterns that occur in generated images and are caused by the deconvolution operation [45].

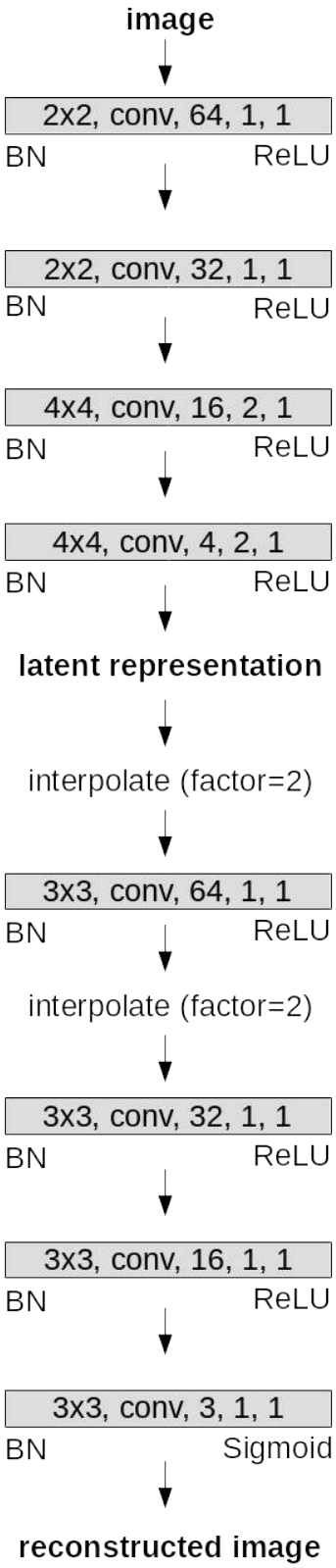

Figure 3-2: Network architecture of the autoencoders. See Fig. 3-3 for an explanation of the notation.

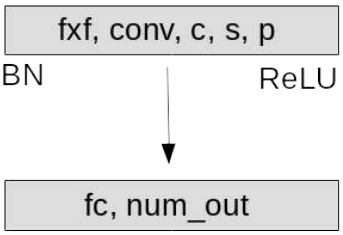

Figure 3-3: Notation for neural network architectures. fxf denotes the kernel size of a convolutional layer, c denotes the number of channels in the layer, s denotes stride, and p denotes padding. BN denotes a Batch Normalization layer after the layer. ReLU denotes the activation function. For a fully-connected layer, num_out denotes the number of output neurons for that layer.

3.2 Data

The oil painting images were extracted from the BAM dataset [4] and the photograph images were extracted from the COCO dataset [46]. All image were cropped and scaled to have the same size. Fig. 3-4 shows example images from the COCO dataset [46] and Fig. 3-5 shows example images from the BAM dataset.
The high-resolution images that we used for testing were extracted from the DIVerse 2K resolution image dataset (DIV2K) [47].

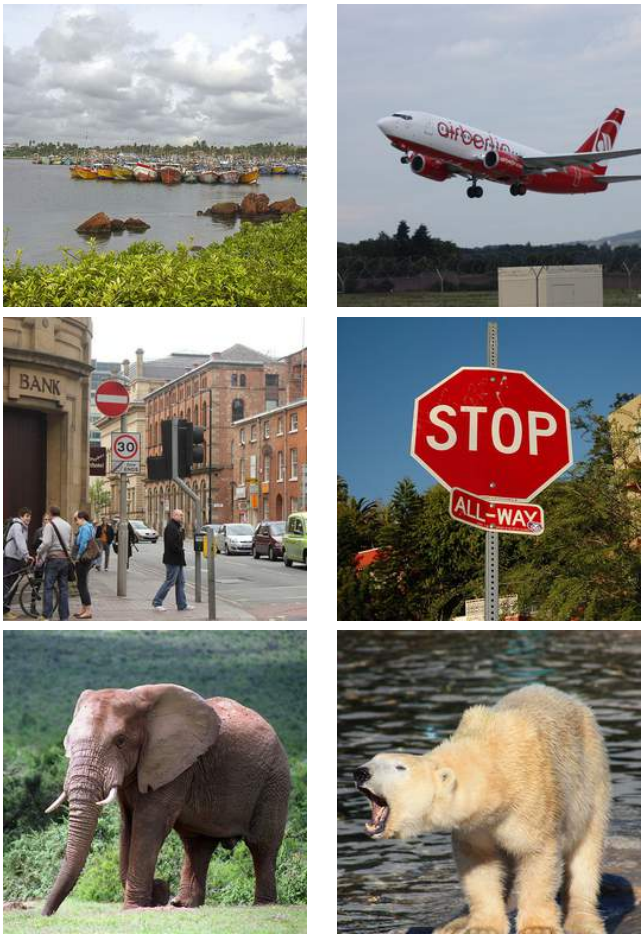

Figure 3-4: Example images from the COCO dataset.

Figure 3-5: Example images from the BAM dataset.

3.3 Experiments

We implemented two different variations of the architecture. In one variation, the transformation networks G_V and G_U were implemented as fully-connected neural networks and in the other variation they were implemented as CNNs. Table 3.1 contains the hyperparameter settings that we used for our experiments. The Adam optimization algorithm [49] was used throughout all experiments.

Hyperparameter	Value
Batch Size	50
Learning Rate	0.0002
β_1 (Adam)	0.5
β_2 (Adam)	0.9
λ (Wasserstein GAN)	10

Table 3.1: Hyperparameter settings used for training the architecture.

3.3.1 Fully-Connected Transformation Networks

The images were scaled down to 128x128. The reason is that fully-connected neural networks are generally more memory-consuming than CNNs and we were operating under somewhat limited computational resources. An RGB image of size 128x128 has 49,152 dimensions. The dimension of the latent space was 4096. Both transformation networks were given the same architecture: a 6-layer fully-connected network, each layer followed by a Batch Normalization layer [48] and a ReLU activation function [50].

As stated in Sec. 3.1.1, the two transformation networks do not generate images directly. They map between the latent representations, which are then propagated through a pretrained decoder network to generate an image. In order to test the general capabilities of such an uncommon GAN architecture, we trained it separately. Fig. 3-6 shows the model. The decoder was part of an autoencoder that was trained on oil painting images from the BAM dataset. The transformation network used the same fully-connected architecture as described above. The random noise was Gaussian-distributed.

We repeated this experiment, but instead of feeding the transformation net-

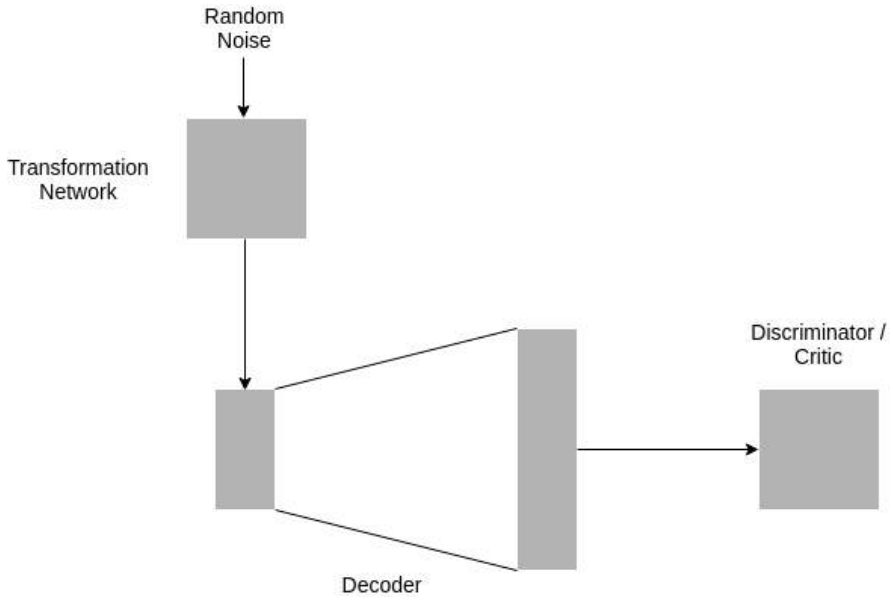

Figure 3-6: Feeding random noise into the transformation network. The transformation network serves as the generator. The decoder is pretrained and fixed during the training of the GAN.

work random noise, we used the latent representation of photograph images. Fig. 3-7 shows the model. This is precisely the mapping F_X as defined in Sec. 3.1. For testing purposes, we trained this separately without the other direction F_Y. We also tried to normalize the low-dimensional representations before feeding them into the transformation network.

3.3.2 Convolutional Transformation Networks

We used images of size 256x256. An RGB image of size 256x256 has 196,608 dimensions. The dimension of the latent space was 16,384. Both transformation networks were given the same architecture. Similar to Zhu et al. [38], we chose transformation networks that consisted of several residual blocks (as proposed by He et al. [51]). Fig. 3-8 shows a residual block. The transformation networks consisted of 4 blocks each (8-layers).

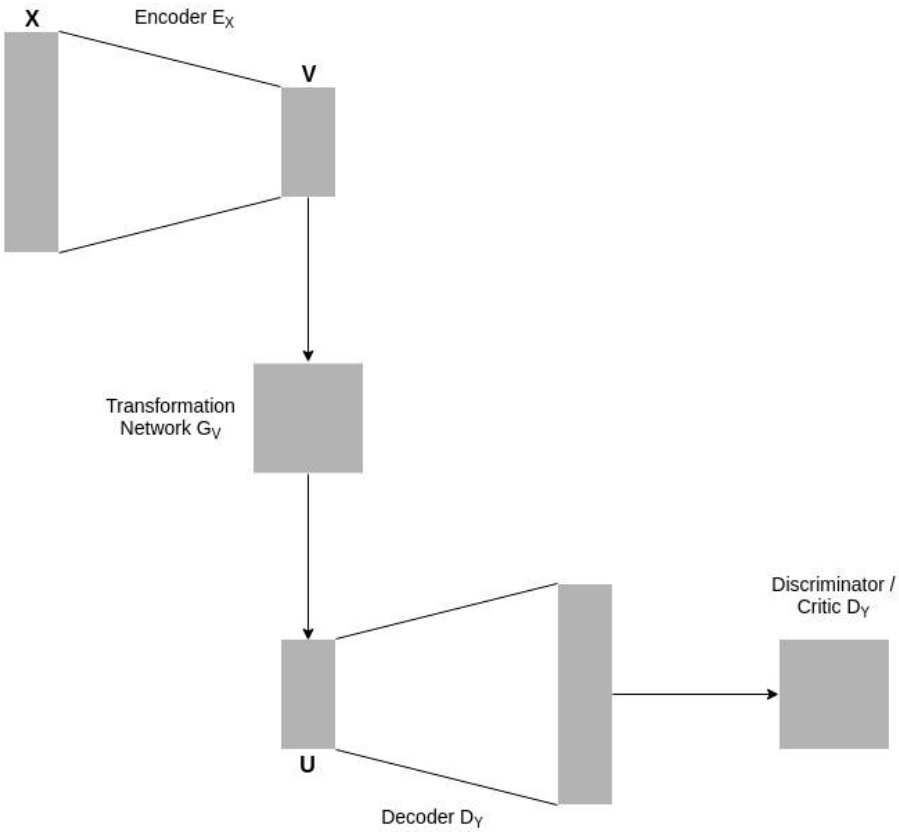

Figure 3-7: Translating in one direction only. The transformation network serves as the generator, both the encoder and the decoder are pretrained and fixed during the training of the GAN.

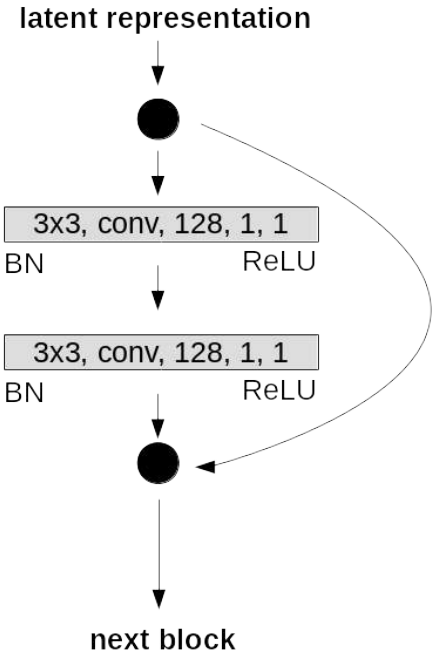

Figure 3-8: A Residual block.

3.4 Semantic Segmentation

During the experiments, we noticed that sometimes the main subjects (large objects or large beings) would blend in with the background in the stylized images. In order to improve the quality of those examples, we used semantic segmentation to crop out the main objects of an image. For each object, we scaled up the cropped-out image patch to the size of the original image. Then we separately put each patch containing an object through the style transfer model. Afterwards, we scaled down the patches again and inserted the object back into the stylized image.
We used the Mask R-CNN model (He et al. [52]) for semantic segmentation [2].

[2]We used a pretrained model from the official TensorFlow [53] GitHub repository, which can be found here: `https://github.com/tensorflow/tpu/blob/master/models/official/mask_rcnn/mask_rcnn_demo.ipynb`

3.5 Performance Measure

As stated in Ch. 1, the goal for F_X is to produce images that follow the same distribution as images in Y and vice versa for F_Y. However, measuring the performance of generative models is not a trivial task [54]. Heusel et al. [55] proposed the Fréchet Inception Distance (FID) to measure the similarity between generated images and real images. To calculate the distance, first, activations of a hidden layer of the Inception network [56] are computed for both generated images and real images. The mean and covariance are computed for both sets of activations, which are assumed to follow a multidimensional Gaussian distribution. Finally, the Fréchet distance [57] between the two Gaussians yields the FID. Heusel et al. state that the FID "is consistent with increasing disturbances and human judgment" [55, p.6].

Chapter 4

Results

In this chapter, we will present our empirical results in form of images and quantitative measurements.

4.1 Fully-Connected Transformation Networks

Fig. 4-1 and 4-2 show the style transfer results for the full architecture with fully-connected transformation networks. Note that the other examples we tested were very similar in appearance.

Fig. 4-3 shows generated oil paint images where we fed Gaussian noise into a transformation network (see Fig. 3-6).

Fig. 4-4 shows the style transfer results for one direction (without cycle consistency constraints) with fully-connected transformation networks (see Fig. 3-7).

Fig. 4-5 shows the same as Fig. 4-4, except that we also normalized the low-dimensional representations before feeding them into the transformation network.

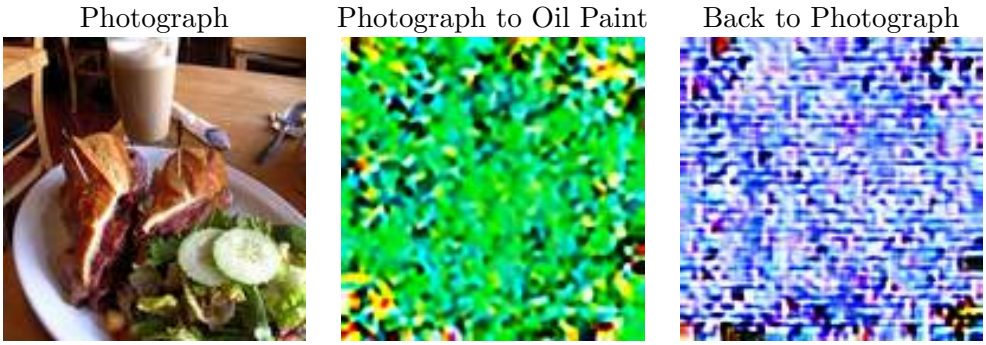

Figure 4-1: Translating photograph to oil paint and back with the full architecture where the transformation networks are fully-connected neural networks.

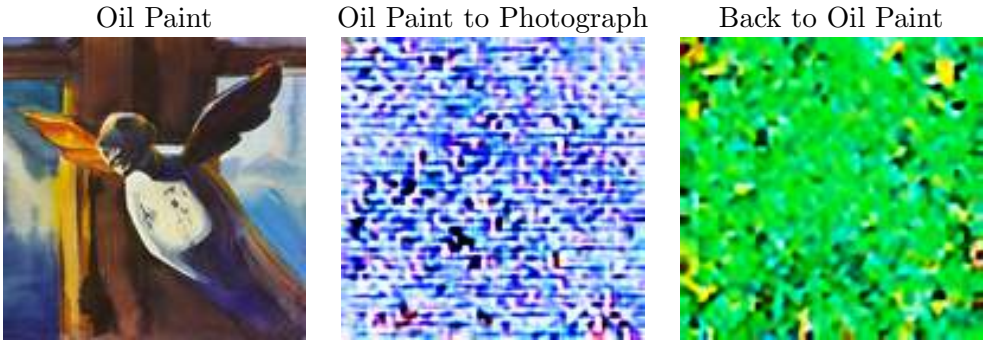

Figure 4-2: Translating oil paint to photograph and back with the full architecture where the transformation networks are fully-connected neural networks.

Figure 4-3: Translating Gaussian noise to oil paint with a fully-connected transformation network.

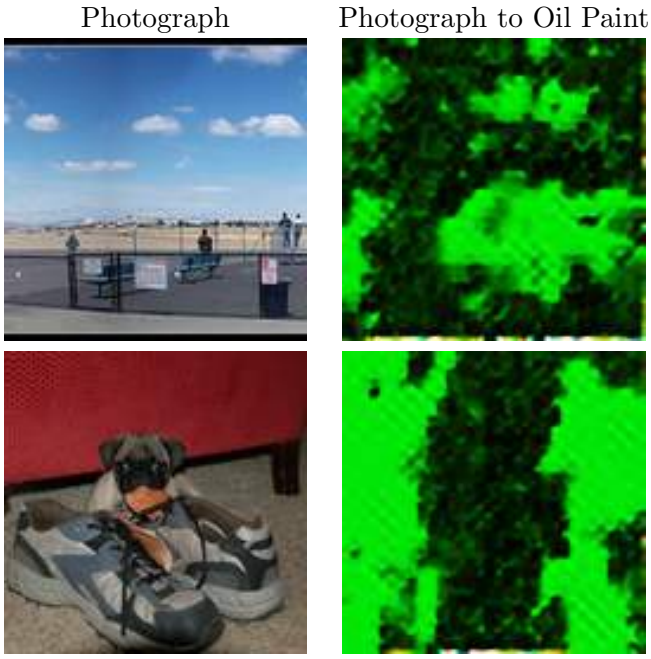

Figure 4-4: Translating photograph to oil paint without cycle consistency constraints with fully-connected transformation networks.

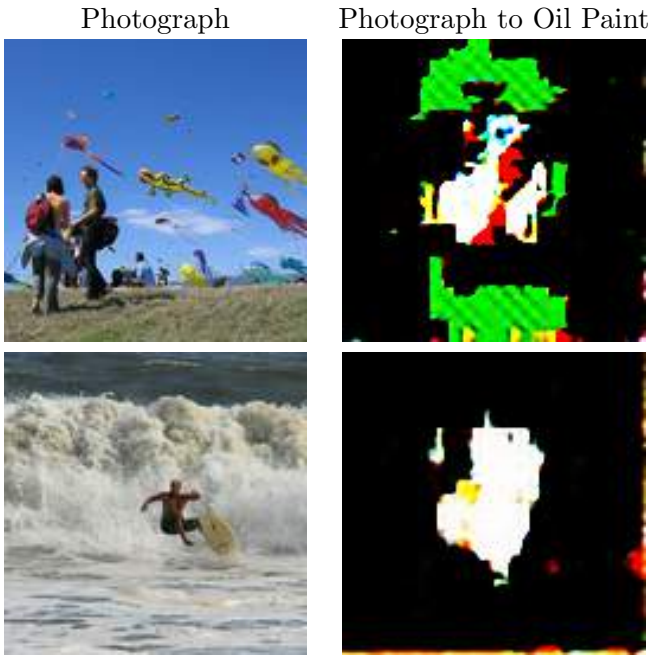

Figure 4-5: Translating photograph to oil paint without cycle consistency constraints with fully-connected transformation networks with normalized low-dimensional representations.

4.2 Convolutional Transformation Networks

Fig. 4-6 and Fig. 4-7 show the style transfer results for the full architecture with convolutional transformation networks for photographs to oil paint and oil paint to photographs, respectively. Fig. 4-8, Fig. 4-9, Fig. 4-10, and Fig. 4-11 show additional style transfer results from photographs to oil paint with different values for the cycle consistency loss factor β.

4.2.1 High-Resolution Images

Fig. 4-12 and Fig. 4-13 show photograph to oil paint style transfer results with high resolution images.

4.2.2 Semantic Segmentation

Fig. 4-14 and Fig. 4-15 show style transfer results from photograph to oil with the aid of semantic segmentation as described in Sec. 3.4.

4.2.3 Fréchet Inception Distance

In order to evaluate our style transfer results, we used the FID (see Sec. 3.5). To establish a baseline, we computed the FID between two samples of real images. Then we computed the FID between a sample of real images and a sample of reconstructed images as well as a sample of real images and a sample of generated images. Table 4.1 contains the FIDs for photographs and Table 4.2 contains the FIDs for oil paint images.

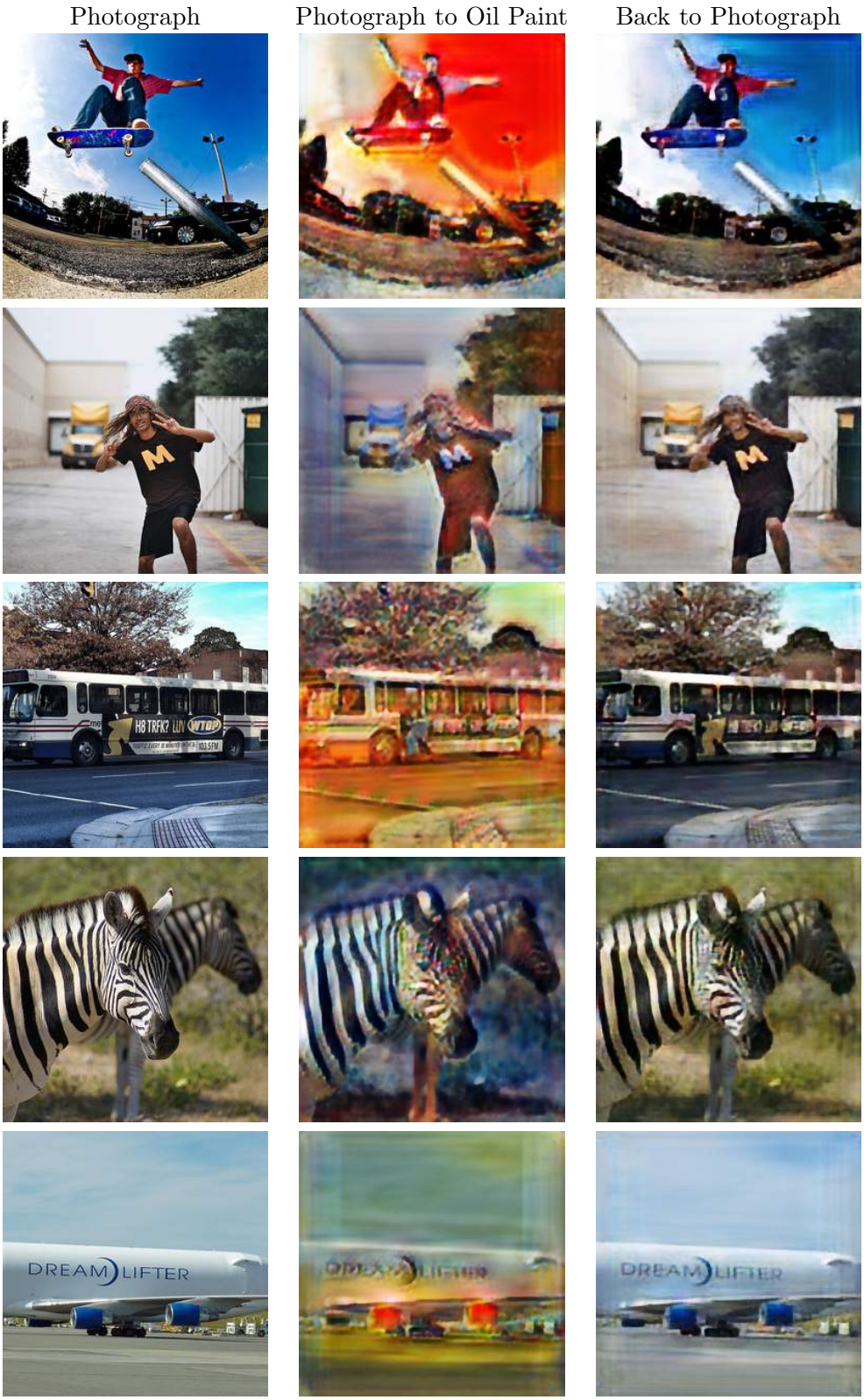

Figure 4-6: Translating photograph to oil paint and back with the full architecture where the transformation networks are convolutional neural networks. The cycle consistency loss factor β was set to 10.

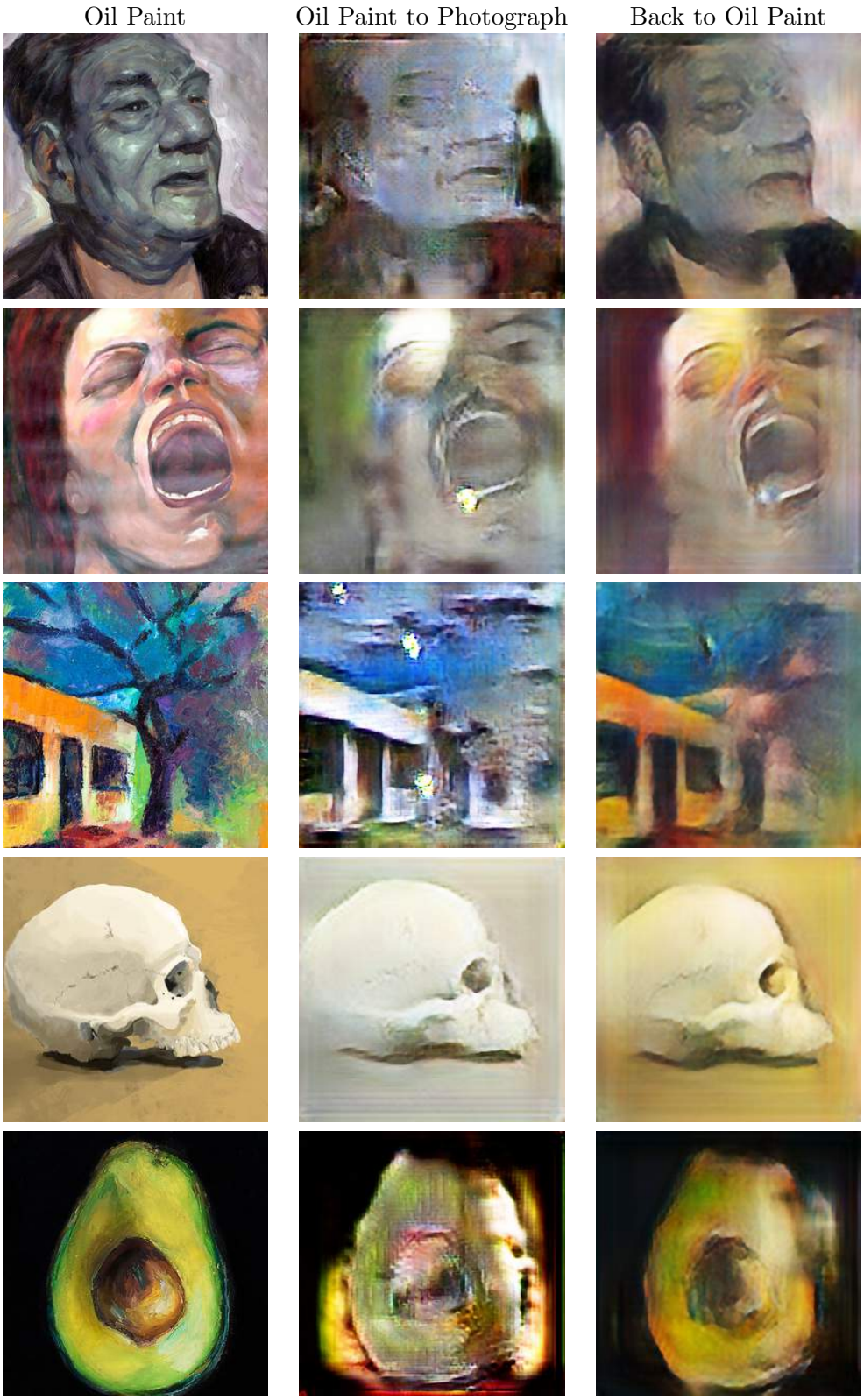

Figure 4-7: Translating oil paint to photograph and back with the full architecture where the transformation networks are convolutional neural networks. The cycle consistency loss factor β was set to 10.

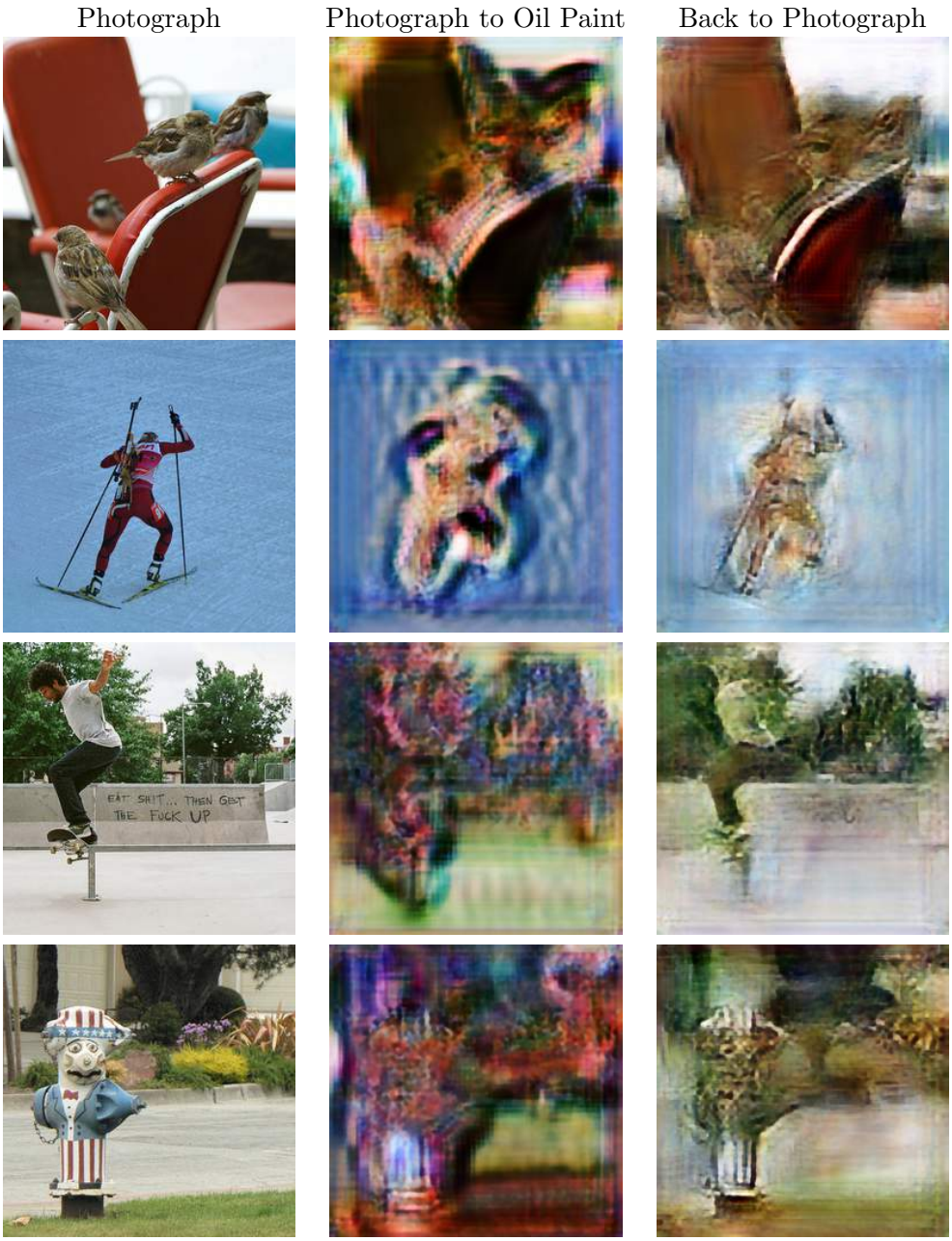

Figure 4-8: Translating photograph to oil paint and back with the full architecture where the transformation networks are convolutional neural networks. The cycle consistency loss factor β was set to 0.5.

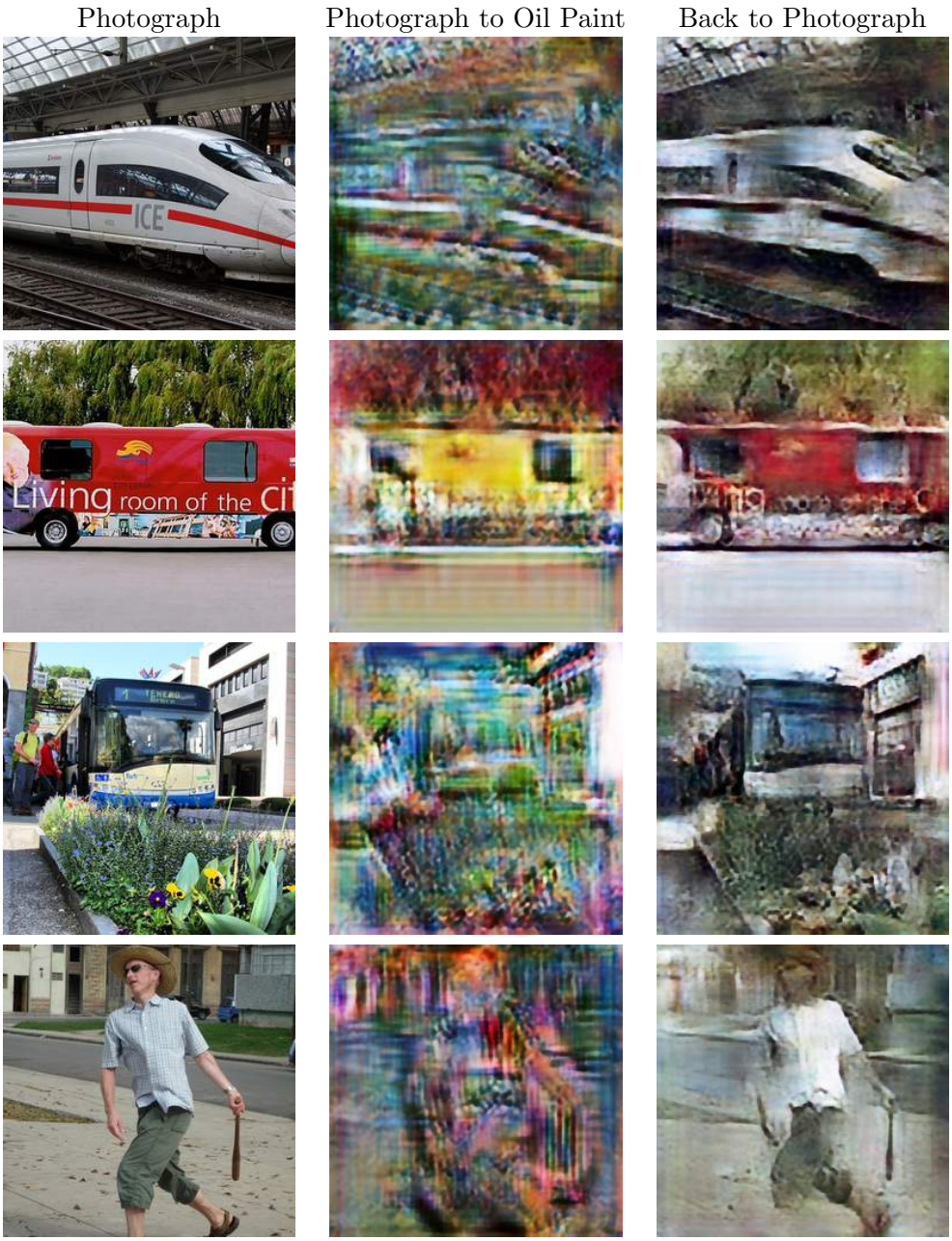

Figure 4-9: Translating photograph to oil paint and back with the full architecture where the transformation networks are convolutional neural networks. The cycle consistency loss factor β was set to 1.

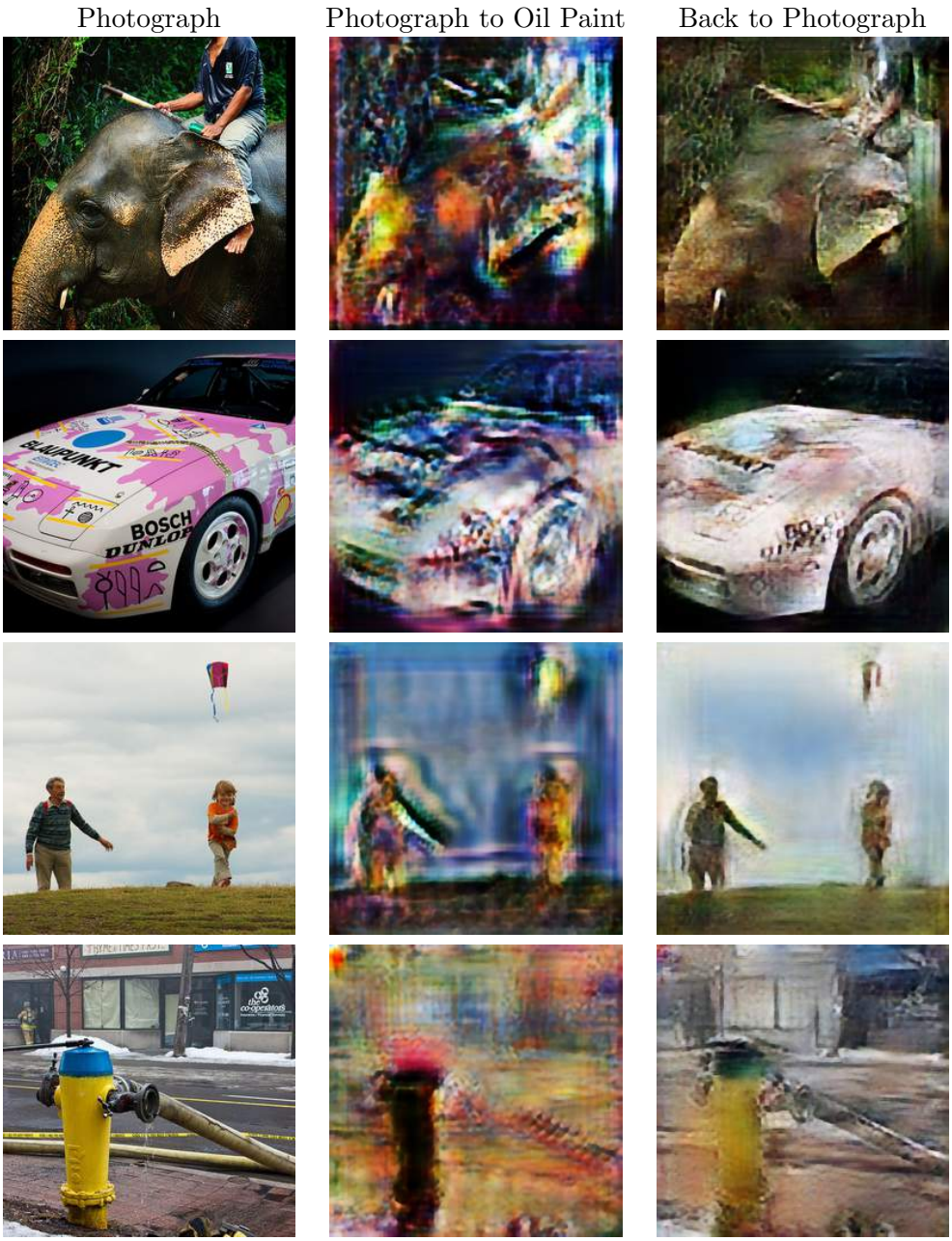

Figure 4-10: Translating photograph to oil paint and back with the full architecture where the transformation networks are convolutional neural networks. The cycle consistency loss factor β was set to 5.

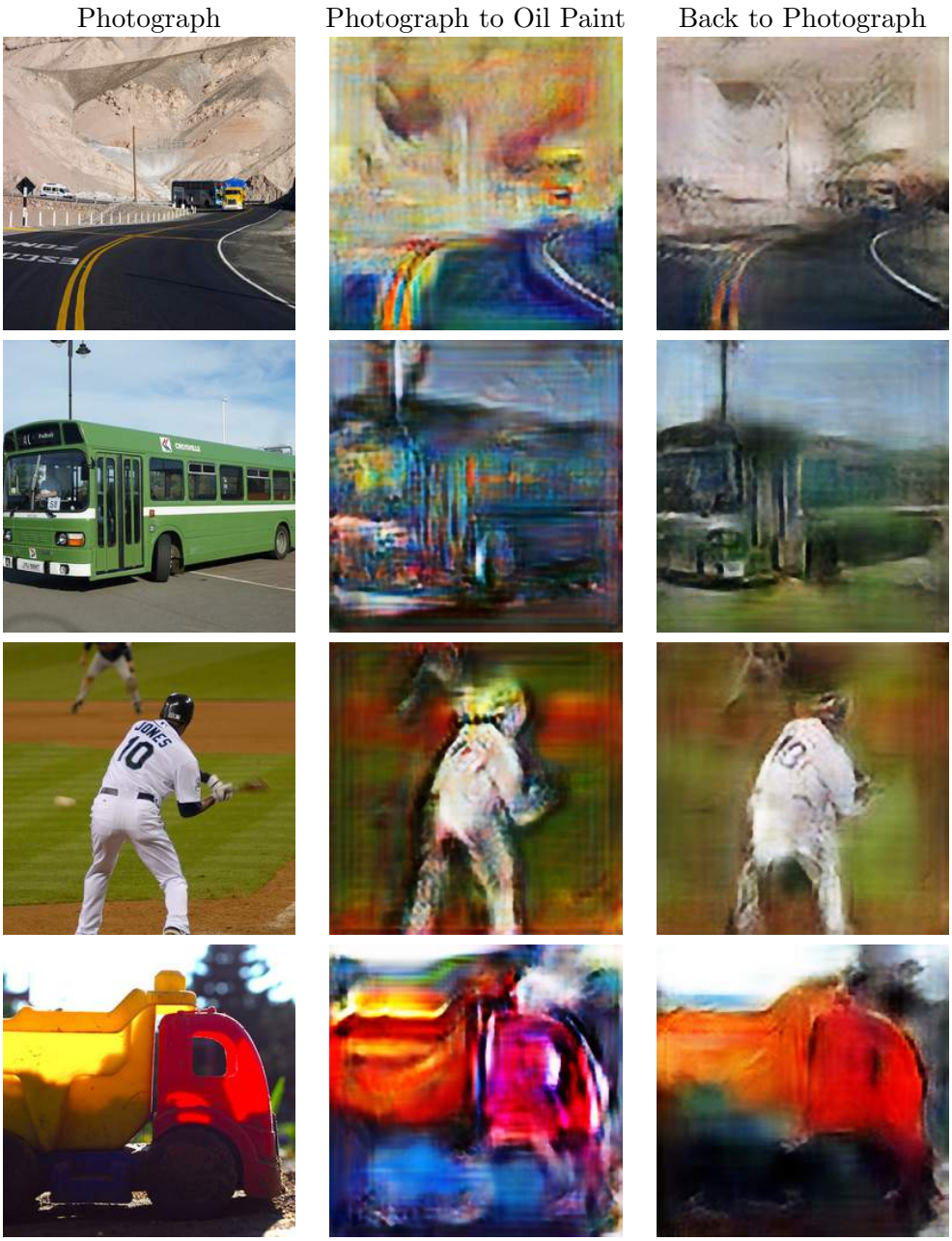

Figure 4-11: Translating photograph to oil paint and back with the full architecture where the transformation networks are convolutional neural networks. The cycle consistency loss factor β was set to 10.

(a) Photograph.

(b) Photograph to Oil Paint.

Figure 4-12: High-Resolution image style transfer from photograph to oil paint.

(a) Photograph.

(b) Photograph to Oil Paint.

Figure 4-13: High-Resolution image style transfer from photograph to oil paint.

(a) The segmented image.

(b) Style transfer from photograph to oil paint without semantic segmentation.

(c) Style transfer from photograph to oil paint with semantic segmentation.

Figure 4-14: Improving style transfer results with semantic segmentation.

(a) The segmented image.

(b) Style transfer from photograph to oil paint without semantic segmentation.

(c) Style transfer from photograph to oil paint with semantic segmentation.

Figure 4-15: Improving style transfer results with semantic segmentation.

Comparison	FID
Real Photographs vs Real Photographs (Baseline)	20.6816
Real Photographs vs Reconstructed Photographs	131.237
Generated Photographs vs Real Photographs	199.562

Table 4.1: FIDs between reconstructed photographs, generated photographs, and real photographs. Note that the baseline was established using two different samples of real photographs.

Comparison	FID
Real Oil Paint vs Real Oil Paint (Baseline)	20.0616
Real Oil Paint vs Reconstructed Oil Paint	82.1227
Generated Oil Paint vs Real Oil Paint	138.761

Table 4.2: FIDs between reconstructed oil paint images, generated oil paint images, and real oil paint images. Note that the baseline was established using two different samples of real oil paint images.

Chapter 5

Discussion

5.1 Fully-Connected Transformation Networks

The results in Sec. 4.1 show that using fully-connected transformation networks does not yield visually recognizable style transfer results. Fig. 4-1 and Fig. 4-2 show that both style transfer from photograph to oil paint and vice versa result in noisy images. However, Fig. 4-3 shows that the architecture is capable of translating Gaussian noise into images that have similarities with oil paint images, although the generated images do not contain recognizable content either.
Yet performing the style transfer only in one direction without the cycle consistency loss (as described in Fig. 3-7) leads to similar results as achieved by the full architecture. Normalizing the low-dimensional representations to be approximately Gaussian-distributed did not improve the results (see Fig. 4-5).

5.2 Convolutional Transformation Networks

Using convolutional transformation networks instead of fully-connected networks leads to improved style transfer results.
Fig. 4-7 shows how translating photographs to oil paint images results in stylized images that contain visually recognizable content and how the reconstructed images that are translated back to photographs look similar to the original. Although the translated images are stylized in a way that makes them vaguely similar to oil paint images, they lack the colorfulness and the brush strokes that are key

characteristics of oil paintings [58, p.19].
Translating from oil paint images to photographs also results in stylized images that contain visually recognizable content and the reconstructed images that are translated back to oil paint look similar to the original as well. However, the translated images do not bear many similarities with photographs (see Fig. 4-6). Instead, they still represent many of the oil paint characteristics.

The results in Sec. 4.2.2 show that the style transfer works better for high-resolution images as opposed to using the same image size as was used for the training images.
Furthermore, we were able to use semantic segmentation to create separation between the background and the main objects of the stylized images (Sec. 4.2.2).

In Sec. 2.5.4, we introduced three criteria that guided the comparison between NPR and NST images: consistency, content preservation, and authenticity. In the following paragraphs, we will apply these criteria to our stylized images presented in Sec. 4.2.

Consistency The images that were translated from oil paint to photograph (Fig. 4-7) and the images that were translated from photograph to oil paint (Fig. 4-6) are generally consistent. The same is true for the high-resolution images that were translated from photograph to oil paint (Fig. 4-12 and Fig. 4-13).

Content Preservation The images that were translated from oil paint to photograph (Fig. 4-7) lost content in the process. The contents of the images that were translated from photograph to oil paint (Fig. 4-6) has mostly been retained although some details were lost. The high-resolution images that were translated from photograph to oil paint (Fig. 4-12 and Fig. 4-13) retained a majority of their content as well as the details.

Authenticity Neither the images that were translated from oil paint to photograph (Fig. 4-7) nor the images that were translated from photograph to oil paint (Fig. 4-6) do well in this category. The produced examples are unlikely to be confused with artwork created by humans. The high-resolution images that were translated from photograph to oil paint (Fig. 4-12 and Fig. 4-13) turned

out better. However, they lack the brush strokes that are typical of oil paintings created by humans.

5.3 Weighing the Cycle Consistency Loss

In Sec. 3.1.1 we described the two loss functions that are used to update the parameters of the transformation networks. While the cycle consistency loss encourages images that are translated to the other domain and back to look approximately like the original, the adversarial loss encourages images that are translated to the other domain to look more like the images from that other domain. As stated in Sec. 3.1.1, in order to balance the relationship between the two losses, we scaled the cycle consistency loss with a factor β. Fig. 4-8, Fig. 4-9, Fig. 4-10, and Fig. 4-11 show the influence of β on the style transfer.
We can observe that the cycle consistency loss is necessary to preserve the image content. The translated images contain almost unrecognizable image content for $\beta = 0.5$ and $\beta = 1$ (see Fig. 4-8 and Fig. 4-9). For $\beta = 5$ and $\beta = 10$ we can observe that the main content of images is also present in the translated images (see Fig. 4-10 and Fig. 4-11).

5.4 Fréchet Inception Distance

In Sec. 3.5 we introduced the FID, which we used to quantify the style transfer results. If we compare the FIDs in Table 4.1 and Table 4.2 with the style transfer results in Fig. 4-6 and Fig. 4-7, we can make three observations:

1. The FID between real photographs and real photographs (20.6816) and the FID between real oil paint and real oil paint (20.0616) are almost identical.

2. The FIDs between real images and reconstructed images (82.1227 and 131.237) are lower than the FIDs between generated images and real images (199.562 and 138.761). These measurements can be verified visually. The reconstructed images are generally of higher quality than the generated images.

3. The FID between generated oil paint and real oil paint (138.761) is lower than the FID between generated photographs and real photographs (199.562).

These measurements can also be verified visually. Generally, generated oil paint images looked more similar to real oil paint images than generated photographs looked to real photographs.

These observations indicate that the FID is consistent with human judgment (as stated by Heusel et al. [55, p.6]).
For comparison, Table 5.1 contains FIDs for a Wasserstein GAN (with gradient penalty instead of gradient clipping) trained on different datasets. These results were reported by Lucic et al. [59, p.7]. Our FID measurements (Table 4.1 and Table 4.2) do not come close to the FIDs reported by Lucic et al. However, it should be noted that Lucic et al. trained a regular Wasserstein GAN which is quite different from our architecture.

Dataset	FID
MNIST [11]	20.3
Fashion-MNIST [60]	24.5
CIFAR-10 [61]	55.8
CelebA [62]	30.0

Table 5.1: FIDs obtained from training a Wasserstein GAN (with gradient penalty instead of gradient clipping) on different datasets, as reported by Lucic et al. [59, p.7].

5.5 Style Transfer and Texture Transfer

Most of the contemporary style transfer algorithms perform texture style transfer without considering the geometric style [63, p.12]. However, for artwork from most art movements texture transfer alone is not sufficient for successful style transfer.
Cubism is a good example. Fig. 5-1 shows the painting *Portrait of Daniel-Henry Kahnweiler* by Pablo Picasso, which exemplifies Analytical Cubism [64, p.138]. We can observe that Picasso hardly makes use of linear perspective and it is "as if we see the subject from multiple, contradictory viewpoints simultaneously." [64, p.138] Additionally, the shape of the subject is broken, making it almost unrecognizable [64, p.138]. This makes it impossible for style transfer methods that

Figure 5-1: *Portrait of Daniel-Henry Kahnweiler* by Pablo Picasso, 1910.

merely perform texture transfer to render a photograph of a human into this particular style. The same is true for the painting *Guernica* by Pablo Picasso (see Fig. 5-2), whose contents are heavily distorted and fragmented [64, p.29].
The medieval period provides us with further examples of artwork where texture transfer is not sufficient. Fig. 5-3 shows the altarpiece *Maestà Altarpiece* by Duccio di Buoninsegna. We can observe that the altarpiece does not convey a strong illusion of depth. The lack of perspective requires geometric transformations in order for a successful style transfer to be conducted.

Figure 5-2: *Guernica* by Pablo Picasso, 1937.

Figure 5-3: *Maestà* by Duccio di Buoninsegna, 1308-1311.

5.6 The Role of Perception

"Drawing on the Right Side of the Brain" by Betty Edwards [66] is one of the most successful books for learning how to draw [65, p.172]. In this book, the author argues that "drawing is not a skill of hand, paper or pencil, but a skill of perception." [65, p.172] Our creativity is highly influenced by how we perceive and understand the world [65, p.172].
Heath and Ventura argue that the same applies for computational creativity [65, p.174]. Thus, a machine has to be able to perceive the world before it can draw. This means that current style transfer techniques are not able to perform style transfer for artworks that rely on pareidolia [1]. To illustrate this, let us consider the painting *The Librarian* by Giuseppe Arcimboldo (Fig. 5-4). Current object detection algorithms might be able to detect the books in the painting but are likely to miss the underlying structure of a person. Performing texture transfer or even geometric transformations to a photograph of a person is also unlikely to result in such a painting because a person is not made out of books.

[1] Pareidolia is "the tendency to perceive a specific, often meaningful image in a random or ambiguous visual pattern." [67]

Figure 5-4: *The Librarian* by Giuseppe Arcimboldo, 1566.

5.7 Hypothesis and Limitations

As stated in Sec. 1.1, the purpose of the low-dimensional representations, produced by the autoencoders, is to efficiently model style and content of an image. We hypothesized that by *forcing* the autoencoders to find image representations in much lower dimensions, we would remove non-essential information from the images.

The style transfer results presented in Sec. 4.2 indicate that the low-dimensional representations do contain essential image information. The translated images contain the essential content from the original images.

However, the inability of the transformation networks to generate recognizable image content without the cycle consistency loss being weighted stronger than the adversarial loss indicates that the GANs may not be able to capture the distribution of the low-dimensional representation of the corresponding domain. This theory is supported by FID measurements. The FID between generated images and real images was significantly higher than the FID between real images and real images.

Furthermore, our model is not capable of performing geometric transformations.

Chapter 6

Conclusion

The goal of this thesis was to perform style transfer between two image domains without explicitly using RGB-pixel representations. We were able to achieve some form of style transfer, but the stylized images were not visually indistinguishable from the images in the corresponding domain. This leads us to the conclusion that the GANs may not be able to capture the distribution of the low-dimensional representations.

6.1 Future Work

6.1.1 Geometry-Aware Style Transfer

Recently, Yaniv et al. [63] proposed a technique for geometry-aware style transfer of human faces. Although there does not exist a straight-forward approach for incorporating their technique into our proposed architecture, there might be some other way of including geometric transformations.

6.1.2 Incorporating Randomness

As mentioned above, oftentimes the stylized images lacked colorfulness and variety. A possible extension of the architecture could be the incorporation of random noise.
In a regular GAN, the generator receives random noise as input and generates a sample. In our architecture, the generators (transformation networks) receive the

low-dimensional representation of an image from one domain and generate the corresponding low-dimensional representation for the other domain.
One approach could be to perturb the low-dimensional representation with random noise before or after feeding it through the transformation network. During training, one could also try to alternate between feeding images through the architecture and feeding random noise through the architecture. That way the transformation networks could learn to generate samples from random noise. Some of our experiments have shown that fully-connected transformation networks produce much better results when they receive Gaussian noise instead of low-dimensional image representations.

Bibliography

[1] Gatys LA, Ecker AS, and Bethge M. 2016. *Image Style Transfer Using Convolutional Neural Networks*. IEEE Conference on Computer Vision and Pattern Recognition, pp.2414-2423. pages 6, 9, 10, 14, 15, 17, 18, 19, 20, 21, 22, 23, 24, 27, 28, 32

[2] Karayev S, Trentacoste M, Han H, Agarwala A, Darrell T, Hertzmann A, and Winnemoeller H. 2014. *Recognizing Image Style*. Proceedings British Machine Vision Conference 2014. pages 9

[3] Hall P, Cai H, Wu Q, and Corradi T. 2015. *Cross-depiction problem: Recognition and Synthesis of Photographs and Artwork*. Computational Visual Media, 1(2), pp.91-103. pages 9

[4] Wilber MJ, Fang C, Jin H, Hertzmann A, Collomosse J, and Belongie S. 2017. *BAM! The Behance Artistic Media Dataset for Recognition Beyond Photography*. IEEE International Conference on Computer Vision, pp.1211-1220. pages 39

[5] Luan F, Paris S, Shechtman E, and Bala K. 2017. *Deep Photo Style Transfer*. IEEE Conference on Computer Vision and Pattern Recognition, pp.6997-7005. pages 6, 10, 19, 24, 30

[6] Goodfellow I, Bengio Y, and Courville A. 2016. *Deep Learning*. MIT Press. pages 12, 13, 14, 37

[7] LeCun Y, Bengio Y, and Hinton G. 2015. *Deep learning*. Nature, 521, pp.436-444. pages 12, 13

[8] Hornik K, Stinchcombe M, and White H. 1989. *Multilayer feedforward networks are universal approximators*. Neural Networks, 2(5), pp.359-366. pages 12

[9] Cybenko G. 1989. *Approximation by superpositions of a sigmoidal function*. Mathematics of Control, Signals and Systems, 2(4), pp.303-314. pages 12

[10] Rumelhart DE, Hinton GE, and Williams RJ. 1986. *Learning internal representations by backpropagating errors*. Nature, 323, pp.533-536. pages 12

[11] LeCun Y, Bottou L, Bengio Y, and Haffner P. 1998. *Gradient Based Learning Applied to Document Recognition*. Proceedings of the IEEE, 86(11), pp.2278-2324. pages 13, 67

[12] Goodfellow I, Pouget-Abadie J, Mirza M, Xu B, Warde-Farley D, Ozair S, Courville A, and Bengio Y. 2014. *Generative Adversarial Nets*. Advances in Neural Information Processing Systems 27. pages 14

[13] Arjovsky M, Chintala S, and Bottou L. 2017. *Wasserstein Generative Adversarial Networks*. International Conference on Machine Learning. pages 14, 35

[14] Simonyan K and Zisserman A. 2015. *Very Deep Convolutional Networks for Large-Scale Image Recognition*. International Conference on Learning Representations. pages 17, 19, 20, 21

[15] Efros AA and Leung TK. 1999. *Texture Synthesis by Non-parametric Sampling*. IEEE International Conference on Computer Vision, pp.1033-1038. pages 14, 15, 17

[16] Paget R and Longstaff ID. 1998. *Texture Synthesis via a Noncausal Nonparametric Multiscale Markov Random Field*. IEEE Transactions on Image Processing, 7(6), pp.925-931. pages 15

[17] Zhu SC, Wu Y, and Mumford D. 1998. *Filters, Random Fields and Maximum Entropy (FRAME): Towards a Unified Theory for Texture Modeling*. International Journal of Computer Vision, 27(2), pp.107-126. pages 15

[18] Wei L-Y and Levoy M. 2000. *Fast Texture Synthesis using Tree-structured Vector Quantization*. 27th annual conference on Computer graphics and interactive techniques, pp.479-488. pages 15, 17

[19] Efros AA and Freeman WT. 2001. *Image Quilting for Texture Synthesis and Transfer*. 28th annual conference on Computer graphics and interactive techniques, pp.341-346. pages 15

[20] Gooch B and Gooch A. 2001. *Non-Photorealistic Rendering*. A K Peters. pages 16

[21] Haeberli P. 1990. *Paint By Numbers: Abstract Image Representations*. 17th annual conference on Computer graphics and interactive techniques, pp.207-214. pages 16, 17, 22, 23

[22] Hertzmann A. 1998. *Painterly Rendering with Curved Brush Strokes of Multiple Sizes*. 25th annual conference on Computer graphics and interactive techniques, pp.453-460. pages 6, 16, 17, 22, 23, 25, 26

[23] Hertzmann A and Perlin K. 2000. *Painterly Rendering for Video and Interaction*. 1st international symposium on Non-photorealistic animation and rendering, pp.7-12. pages 16

[24] Gooch B, Coombe G, and Shirley P. 2002. *Artistic Vision: Painterly Rendering Using Computer Vision Techniques*. 2nd international symposium on Non-photorealistic animation and rendering, pp.83-90. pages 6, 16, 17, 22, 25

[25] Hertzmann A, Jacobs CE, Oliver N, Curless B, and Salesin DH. 2001. *Image Analogies*. 28th annual conference on Computer graphics and interactive techniques, pp.327-340. pages 6, 17, 22, 26

[26] Adelson EH, Anderson CH, Bergen JR, Burt PJ, and Ogden JM. 1984. *Pyramid methods in image processing*. RCA Engineer, 29(6), pp.33-41. pages 17

[27] Liu Y, and Veksler O, and Juan O. 2010. *Generating Classic Mosaics with Graph Cuts*. Computer Graphics Forum, 29(8), pp.2387-2399. pages 17

[28] Kang H, Lee S, and Chui CK. 2009. *Flow-Based Image Abstraction*. IEEE Transactions on Visualization and Computer Graphics, 15(1), pp.62-76. pages 17

[29] Jing Y, Yang Y, Feng Z, Ye J, Yu Y, and Song M. 2017. *Neural Style Transfer: A Review*. arXiv preprint arXiv:1705.04058. pages 17, 19, 20, 22, 24

[30] Johnson J, Alahi A, and Fei-Fei L. 2016. *Perceptual Losses for Real-Time Style Transfer and Super-Resolution*. European Conference on Computer Vision, pp.694-711. pages 19, 20, 22, 23, 32

[31] Russakovsky O, Deng J, Su H, Krause J, Satheesh S, Ma S, Huang Z, Karpathy A, Khosla A, Bernstein M, Berg AC, Fei-Fei L. 2015. *ImageNet Large Scale Visual Recognition Challenge*. International Journal of Computer Vision, 115(3), pp.211-252. pages 19, 20

[32] Ulyanov D, Lebedev V, Vedaldi A, and Lempitsky V. 2016. *Texture Networks: Feed-forward Synthesis of Textures and Stylized Images*. International Conference on Machine Learning, pp.1349-1357. pages 20, 22, 23, 32

[33] Ulyanov D, Vedaldi A, and Lempitsky V. 2017. *Improved Texture Networks: Maximizing Quality and Diversity in Feed-forward Stylization and Texture Synthesis*. IEEE Conference on Computer Vision and Pattern Recognition, pp.4105-4113. pages 6, 20, 22, 23, 24, 29, 32

[34] Li S, Xu X, Nie L, and Chua T-S. 2017. *Laplacian-Steered Neural Style Transfer*. Proceedings of the 25th ACM international conference on Multimedia, pp.1716-1724. pages 19, 20, 22

[35] Xu Z, Wilber MJ, Fang C, Hertzmann A, and Jin H. 2018. *Beyond Textures: Learning from Multi-domain Artistic Images for Arbitrary Style Transfer*. arXiv preprint arXiv:1805.09987. pages 20, 21, 22, 32

[36] Li Y, Fang C, Yang J, Wang Z, Lu X, and Yang M-H. 2017. *Universal Style Transfer via Feature Transforms*. Advances in Neural Information Processing Systems 30. pages 21, 22, 32

[37] Isola P, Zhu J-Y, Zhou T, and Efros AA. 2017. *Image-to-Image Translation with Conditional Adversarial Networks*. IEEE Conference on Computer Vision and Pattern Recognition, pp.5967-5976. pages 21, 22, 35

[38] Zhu J-Y, Park T, Isola P, and Efros AA. 2017. *Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks*. IEEE International Conference on Computer Vision, pp.2242-2251. pages 1, 9, 10, 21, 22, 32, 33, 34, 35, 37, 43

[39] Hall P and Lehmann A-S. 2013. *Don't Measure - Appreciate! NPR Seen Through the Prism of Art History*. Image and Video-Based Artistic Stylisation, pp.333-351. pages 23

[40] Hertzmann A. 2010. *Non-Photorealistic Rendering and the Science of Art*. 8th International Symposium on Non-Photorealistic Animation and Rendering, pp.147-157. pages 10, 23

[41] Gooch AA, Long J, Ji L, Estey A, and Gooch B. 2010. *Viewing Progress in Non-photorealistic Rendering through Heinlein's Lens*. 8th International Symposium on Non-Photorealistic Animation and Rendering, pp.165-171. pages 23

[42] Gatys LA, Ecker AS, Bethge M, Hertzmann A, and Shechtman E. 2017. *Controlling Perceptual Factors in Neural Style Transfer*. IEEE Conference on Computer Vision and Pattern Recognition, pp.3730-3738. pages 18

[43] Gulrajani I, Ahmed F, Arjovsky M, Dumoulin V, and Courville A. 2017. *Improved Training of Wasserstein GANs*. Advances in Neural Information Processing Systems 30. pages 35, 36

[44] Zeiler MD, Krishnan D, Taylor GW, and Fergus R. 2010. *Deconvolutional networks*. IEEE Computer Society Conference on Computer Vision and Pattern Recognition, pp.2528-2535. pages 37

[45] Odena A, Dumoulin V, and Olah C. 2016. *Deconvolution and Checkerboard Artifacts*. Distill. `http://distill.pub/2016/deconv-checkerboard`. pages 37

[46] Lin T-Y, Maire M, Belongie S, Hays J, Perona P, Ramanan D, Dollár P, and Zitnick CL. 2014. *Microsoft COCO: Common Objects in Context*. European Conference on Computer Vision, pp.740-755. pages 39

[47] Agustsson E and Timofte R. 2017. *NTIRE 2017 Challenge on Single Image Super-Resolution: Dataset and Study*. IEEE Conference on Computer Vision and Pattern Recognition (Workshops), pp.1122-1131. pages 39

[48] Ioffe S and Szegedy C. 2015. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. International Conference on Machine Learning, pp.448-456. pages 42

[49] Kingma DP and Ba JL. 2015. *Adam: A Method for Stochastic Optimization*. International Conference for Learning Representations. pages 42

[50] Nair V and Hinton GE. 2010. *Rectified Linear Units Improve Restricted Boltzmann Machines*. International Conference on Machine Learning, pp.807-814. pages 42

[51] He K, Zhang X, Ren S, and Sun J. 2016. *Deep Residual Learning for Image Recognition*. IEEE Conference on Computer Vision and Pattern Recognition, pp.770-778. pages 43

[52] He K, Gkioxari G, Dollár P, and Girshick R. 2017. *Mask R-CNN*. IEEE International Conference on Computer Vision, pp. 2980-2988. pages 45

[53] Abadi M, Agarwal A, Barham P, Brevdo E, Chen Z, Citro C, Corrado GS, Davis A, Dean J, Devin M, Ghemawat S, Goodfellow I, Harp A, Irving G, Isard M, Jozefowicz R, Jia Y, Kaiser L, Kudlur M, Levenberg J, Mané D, Schuster M, Monga R, Moore S, Murray D, Olah C, Shlens J, Steiner B, Sutskever I, Talwar K, Tucker P, Vanhoucke V, Vasudevan V, Viégas F, Vinyals O, Warden P, Wattenberg M, Wicke M, Yu Y, and Zheng X. 2015. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from `tensorflow.org`. pages 45

[54] Theis L, van den Oord A, and Bethge M. 2016. *A note on the evaluation of generative models*. International Conference for Learning Representations. pages 46

[55] Heusel M, Ramsauer H, Unterthiner T, Nessler B, and Hochreiter S. 2017. *GANs Trained by a Two Time-Scale Update Rule Converge to a Local Nash Equilibrium*. Advances in Neural Information Processing Systems 30. pages 1, 46, 67

[56] Szegedy C, Liu W, Jia Y, Sermanet P, Reed S, Anguelov A, Erhan D, Vanhoucke V, and Rabinovich A. 2015. *Going Deeper with Convolutions*. IEEE Conference on Computer Vision and Pattern Recognition, pp.1-9. pages 46

[57] Fréchet M. 1957. *Sur la distance de deux lois de probabilité*. Comptes rendus de l'Académie des Sciences, 244(6), pp.689-692. pages 46

[58] Dai Y. 2019. *Diversity of Color and Semantic Communication in Oil Painting*. Argos, 36(75), pp.19-26. pages 65

[59] Lucic M, Kurach K, Michalski M, Gelly S, and Bousquet O. 2018. *Are GANs Created Equal? A Large-Scale Study*. Advances in Neural Information Processing Systems 31. pages 67

[60] Xiao H, Rasul K, and Vollgraf R. 2017. *Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms*. arXiv preprint arXiv:1708.07747. pages 67

[61] Krizhevsky A. 2009. *Learning Multiple Layers of Features from Tiny Images*. Technical Report. pages 67

[62] Liu Z, Luo P, Wang X, and Tang X. 2015. *Deep Learning Face Attributes in the Wild*. IEEE International Conference on Computer Vision, pp.3730-3738. pages 67

[63] Yaniv J, Newman Y, and Shamir A. 2019. *The Face of Art: Landmark Detection and Geometric Style in Portraits*. ACM Transactions on Graphics, 38(4), Article No. 60. pages 67, 73

[64] Penny Huntsman. 2015. *Thinking About Art: A Thematic Guide to Art History*. Wiley-Blackwell. pages 67, 68

[65] Heath D and Ventura D. 2016. *Before A Computer Can Draw, It Must First Learn To See*. International Conference on Computational Creativity, pp.172-179. pages 70

[66] Edwards B. 1989. *Drawing on the Right Side of the Brain*. Tarcher. pages 70

[67] "pareidolia". In Merriam-Webster.com. Retrieved: 14. August 2019. `https://www.merriam-webster.com/dictionary/pareidolia`. pages 70

Appendices

Appendix A

Software

A.1 Requirements

The following software packages have to be installed in order to run the code.

- Python 3.5.2+
- NumPy 1.11.0+
- PyTorch 0.4.1+
- PIL 1.1.7+
- Matplotlib 3.0.1+

A.2 Hardware

All experiments were performed on a Nvidia Tesla K40 GPU (12GB memory) with Cuda 8.0. A GPU is not required but highly recommended.

A.3 Instructions

A.3.1 Training

```
import siamese

data_photo = '...' # file path to the photograph images (RGB) folder
data_oil = '...' # file path to the oil images (RGB) folder
```

```
siam_model = siamese.Siamese(in_path_photo=data_photo,
                             in_path_oil=data_oil,
                             autoencoder_path='saves/autoencoders',
                             num_epochs=100, batch_size=50,
                             learning_rate=0.0002,
                             recon_loss_weight=10,
                             penalty_coef=10, verbose=True)

siam_model.train()
```

A.3.2 Resume Training

```
import siamese

data_photo = '...' # file path to the photograph images (RGB) folder
data_oil = '...' # file path to the oil images (RGB) folder
siam_model = siamese.Siamese(in_path_photo=data_photo,
                             in_path_oil=data_oil,
                             autoencoder_path='saves/autoencoders',
                             num_epochs=100, batch_size=50,
                             learning_rate=0.0002,
                             recon_loss_weight=10,
                             penalty_coef=10, verbose=True)

siam_model.load(epoch=18, path='saves')
siam_model.train()
```

A.3.3 Style Transfer for a Single Image

```
import util

siam_model = siamese.Siamese(autoencoder_path='saves/autoencoders')
siam_model.load(epoch=18, path='saves')

x_photo = util.image_to_tensor('image.jpg')
```

```
x_oil = siam_model.translate_photo_to_oil(x_photo)
x_oil = util.tensor_to_numpy(x_oil)
x_oil = util.numpy_to_image(x_oil)
x_oil.save('stylized_image.png')
```

A.4 Submission

The source code is contained in the following directory:

```
.
├── autoencoder.py
├── model.py
├── saves
│   ├── autoencoders
│   │   ├── autoencoder_oil_20.pth
│   │   └── autoencoder_photo_20.pth
│   ├── discriminator_oil_18.pth
│   ├── discriminator_photo_18.pth
│   ├── map_u_to_v_18.pth
│   └── map_v_to_u_18.pth
├── siamese.py
└── util.py
```

A.5 Source Code

This section contains the source code.

siamese.py

```
import torch
import torchvision
import torch.autograd as autograd
import model
import autoencoder
import util
import os


class Siamese:

    def __init__(self, in_path_photo=None, in_path_oil=None, autoencoder_path=None, num_epochs=100, batch_size=50,
                 learning_rate=0.0002, recon_loss_weight=10, penalty_coef=10, verbose=True):
        """
        This class implements the siamese architecture.
        :param in_path_photo: (string) the file path indicating the location of the training data for photographs.
        :param in_path_oil: the file path indicating the location of the training data for oil.
        :param autoencoder_path: (string) the path where the save files of the autoencoders are stored.
        :param num_epochs: (int) the number of epochs.
        :param batch_size: (int) the batch size.
        :param learning_rate: (int) the learning rate for the Adam optimizer.
        :param recon_loss_weight: (float) the parameter that scales the cycle consistency / reconstruction loss (beta)
        :param penalty_coef: (float) the penalty coefficient for the Wasserstein GAN (lambda)
        :param verbose: (boolean) if true, the training process is printed to console
        """
        self.device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
        self.use_cuda = torch.cuda.is_available()
        self.in_path_photo = in_path_photo
        self.in_path_oil = in_path_oil
        self.num_epochs = num_epochs
        self.batch_size = batch_size
        self.learning_rate = learning_rate
        self.recon_loss_weight = recon_loss_weight
        self.penalty_coef = penalty_coef
        self.verbose = verbose

        self.start_epoch = 1
        self.d_photo_losses = []
        self.d_oil_losses = []
        self.g_photo_losses = []
        self.g_oil_losses = []
        self.photo_rec_losses = []
        self.oil_rec_losses = []

        self.auto_photo = autoencoder.Autoencoder()
        self.auto_photo.load(autoencoder_path + 'autoencoder_photo_20.pth')
        self.auto_photo.eval()
        self.auto_oil = autoencoder.Autoencoder()
        self.auto_oil.load(autoencoder_path + 'autoencoder_oil_20.pth')
        self.auto_oil.eval()

        self.map_v_to_u = model.Map().cuda() if self.use_cuda else model.Map()
        self.map_u_to_v = model.Map().cuda() if self.use_cuda else model.Map()

        self.v_to_u_optimizer = torch.optim.Adam(self.map_v_to_u.parameters(), lr=learning_rate, betas=(0.5, 0.9))
        self.u_to_v_optimizer = torch.optim.Adam(self.map_u_to_v.parameters(), lr=learning_rate, betas=(0.5, 0.9))

        self.discriminator_photo = model.Discriminator().cuda() if self.use_cuda else model.Discriminator()
        self.discriminator_oil = model.Discriminator().cuda() if self.use_cuda else model.Discriminator()

        self.d_photo_optimizer = torch.optim.Adam(self.discriminator_photo.parameters(), lr=learning_rate,
                                                  betas=(0.5, 0.9))
        self.d_oil_optimizer = torch.optim.Adam(self.discriminator_oil.parameters(), lr=learning_rate, betas=(0.5, 0.9))

    def train(self):
        """
        Trains the architecture.
        """
        training_data_photo = torchvision.datasets.ImageFolder(self.in_path_photo, torchvision.transforms.Compose([
                                                               torchvision.transforms.ToTensor()]))

        data_loader_photo = torch.utils.data.DataLoader(training_data_photo, batch_size=self.batch_size, shuffle=True,
                                                        num_workers=2)

        training_data_oil = torchvision.datasets.ImageFolder(self.in_path_oil, torchvision.transforms.Compose([
                                                               torchvision.transforms.ToTensor()]))

        data_loader_oil = torch.utils.data.DataLoader(training_data_oil, batch_size=self.batch_size, shuffle=True,
                                                      num_workers=2)

        l1_criterion = torch.nn.L1Loss().cuda() if self.use_cuda else torch.nn.L1Loss()

        progress_bar = util.ProgressBar()

        for epoch in range(self.start_epoch, self.num_epochs + 1):
            print('Epoch {}/{}'.format(epoch, self.num_epochs))
            for i, (photo_batch, oil_batch) in enumerate(zip(data_loader_photo, data_loader_oil), 1):
                x_photo, _ = photo_batch
                x_oil, _ = oil_batch
                x_photo = x_photo.to(self.device)
                x_oil = x_oil.to(self.device)

                if x_photo.shape[0] != x_oil.shape[0]:
                    continue

                z_oil, x_rec_photo = self._train_photo_to_oil_and_back(x_photo, l1_criterion)

```

siamese.py

```
                v = self.auto_photo.encode(x_photo)
                u_dash = self.map_v_to_u(v)
                z_oil = self.auto_oil.decode(u_dash)

                self._update_discriminator(self.discriminator_oil, self.d_oil_optimizer, x_real=x_oil, x_fake=z_oil,
                                           losses=self.d_oil_losses)

                z_photo, x_rec_oil = self._train_oil_to_photo_and_back(x_oil, l1_criterion)

                u = self.auto_oil.encode(x_oil)
                v_dash = self.map_u_to_v(u)
                z_photo = self.auto_photo.decode(v_dash)

                self._update_discriminator(self.discriminator_photo, self.d_photo_optimizer, x_real=x_photo,
                                           x_fake=z_photo, losses=self.d_photo_losses)

                if self.verbose:
                    info_str = 'd_photo: {:.4f}, d_oil: {:.4f}, g_photo: {:.4f}, g_oil: {:.4f}'.\
                               format(self.d_photo_losses[-1], self.d_oil_losses[-1], self.g_photo_losses[-1],
                                      self.g_oil_losses[-1])

                    progress_bar.update(max_value=len(data_loader_oil), current_value=i + 1, info=info_str)

            if not os.path.exists('saves/'):
                os.makedirs('saves/')
            self.save(epoch, path='saves')

            progress_bar.new_line()

    def translate_photo_to_oil(self, x_photo):
        """
        Translates the given image of a photograph
        to an image of an oil paint.
        :param x_photo: (tensor) image of a photograph.
        :return: (tensor) image translated to oil paint.
        """
        v = self.auto_photo.encode(x_photo.detach())
        u_dash = self.map_v_to_u(v.detach())
        z_oil = self.auto_oil.decode(u_dash.detach())
        return z_oil

    def translate_oil_to_photo(self, x_oil):
        """
        Translates the given image of an oil painting
        to an image of a photograph.
        :param x_oil: (tensor) image of an oil painting.
        :return: (tensor) image translated to photograph.
        """
        u = self.auto_oil.encode(x_oil.detach())
        v_dash = self.map_u_to_v(u.detach())
        z_photo = self.auto_photo.decode(v_dash.detach())
        return z_photo

    def _train_photo_to_oil_and_back(self, x_photo, criterion):
        # forward photo
        v = self.auto_photo.encode(x_photo.detach())
        u_dash = self.map_v_to_u(v)

        z_oil = self.auto_oil.decode(u_dash)
        d_fake = self.discriminator_oil(z_oil)
        adversarial_loss = -torch.mean(d_fake)
        self.v_to_u_optimizer.zero_grad()
        adversarial_loss.backward()
        self.v_to_u_optimizer.step()
        self.g_oil_losses.append(adversarial_loss.item())

        u = self.auto_oil.encode(z_oil.detach())
        v_dash = self.map_u_to_v(u)
        x_rec_photo = self.auto_photo.decode(v_dash)
        loss_rec_photo = criterion(x_rec_photo, x_photo)
        loss_rec_photo *= self.recon_loss_weight
        self.u_to_v_optimizer.zero_grad()
        self.v_to_u_optimizer.zero_grad()
        loss_rec_photo.backward()
        self.u_to_v_optimizer.step()
        self.v_to_u_optimizer.step()
        self.photo_rec_losses.append(loss_rec_photo.item())

        return z_oil, x_rec_photo

    def _train_oil_to_photo_and_back(self, x_oil, criterion):
        # forward oil
        u = self.auto_oil.encode(x_oil.detach())
        v_dash = self.map_u_to_v(u)

        z_photo = self.auto_photo.decode(v_dash)
        d_fake = self.discriminator_photo(z_photo)
        adversarial_loss = -torch.mean(d_fake)
        self.u_to_v_optimizer.zero_grad()
        adversarial_loss.backward()
        self.u_to_v_optimizer.step()
        self.g_photo_losses.append(adversarial_loss.item())

        v = self.auto_photo.encode(z_photo.detach())
        u_dash = self.map_v_to_u(v)
        x_rec_oil = self.auto_oil.decode(u_dash)
        loss_rec_oil = criterion(x_rec_oil, x_oil)
```

siamese.py

```
        loss_rec_oil *= self.recon_loss_weight
        self.v_to_u_optimizer.zero_grad()
        self.u_to_v_optimizer.zero_grad()
        loss_rec_oil.backward()
        self.v_to_u_optimizer.step()
        self.u_to_v_optimizer.step()
        self.oil_rec_losses.append(loss_rec_oil.item())

        return z_photo, x_rec_oil

    def _update_discriminator(self, discriminator, optimizer, x_real, x_fake, losses):
        """
        Performs a single optimization step for the discriminator.
        :param x_real: (tensor) batch of images from the training set.
        """
        d_real = discriminator(x_real)
        d_fake = discriminator(x_fake)
        gradient_penalty = self._gradient_penalty(discriminator, x_real, x_fake)
        d_loss = torch.mean(d_fake) - torch.mean(d_real) + self.penalty_coef * gradient_penalty

        optimizer.zero_grad()
        d_loss.backward()
        optimizer.step()

        losses.append(d_loss.item())

    def _gradient_penalty(self, discriminator, x_real, x_fake):
        """
        Returns the gradient penalty for the discriminator.
        This function was taken from this GitHub repository: https://github.com/EmilienDupont/wgan-gp
        :param x_real: (tensor) batch of images from the training set.
        :param x_fake: (tensor) batch of generated images.
        :return: (float) gradient penalty.
        """
        alpha = torch.rand(x_real.shape[0], 1, 1, 1)
        alpha = alpha.expand_as(x_real)
        alpha = alpha.to(self.device)

        interpolates = alpha * x_real + (1 - alpha) * x_fake
        interpolates = autograd.Variable(interpolates, requires_grad=True)
        interpolates = interpolates.to(self.device)
        d_interpolates = discriminator(interpolates)

        gradients = autograd.grad(outputs=d_interpolates, inputs=interpolates,
                                  grad_outputs=torch.ones(d_interpolates.shape).to(self.device),
                                  create_graph=True, retain_graph=True, only_inputs=True)[0]

        gradients = gradients.view(x_real.shape[0], -1)

        norm = torch.sqrt(torch.sum(gradients ** 2, dim=1) + 1e-12)
        gradient_penalty = torch.mean((norm - 1) ** 2)
        return gradient_penalty

    def save(self, epoch, path):
        """
        Saves the parameters of all the trainable networks to file.
        :param epoch: (int) the current epoch.
        :param path: (string) a file path indicating the location where the files should be stored.
        :return:
        """
        torch.save({
            'epoch': epoch,
            'model_state_dict': self.discriminator_photo.state_dict(),
            'optimizer_state_dict': self.d_photo_optimizer.state_dict(),
            'losses': self.d_photo_losses
        }, path + '/discriminator_photo_' + str(epoch) + '.pth')
        torch.save({
            'epoch': epoch,
            'model_state_dict': self.discriminator_oil.state_dict(),
            'optimizer_state_dict': self.d_oil_optimizer.state_dict(),
            'losses': self.d_oil_losses
        }, path + '/discriminator_oil_' + str(epoch) + '.pth')
        torch.save({
            'epoch': epoch,
            'model_state_dict': self.map_v_to_u.state_dict(),
            'optimizer_state_dict': self.v_to_u_optimizer.state_dict(),
            'g_oil_losses': self.g_oil_losses,
            'oil_rec_losses': self.oil_rec_losses
        }, path + '/map_v_to_u_' + str(epoch) + '.pth')
        torch.save({
            'epoch': epoch,
            'model_state_dict': self.map_u_to_v.state_dict(),
            'optimizer_state_dict': self.u_to_v_optimizer.state_dict(),
            'g_photo_losses': self.g_photo_losses,
            'photo_rec_losses': self.photo_rec_losses
        }, path + '/map_u_to_v_' + str(epoch) + '.pth')

    def load(self, epoch, path):
        """
        Loads parameters for all the trainable networks from files.
        :param epoch: (int) the epoch from the saves.
        :param path: (string) a file path indicating the location of the saves.
        :return:
        """
        checkpoint = torch.load(path + '/map_v_to_u_' + str(epoch) + '.pth', map_location=self.device)
        self.map_v_to_u.load_state_dict(checkpoint['model_state_dict'])
        self.v_to_u_optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
```

siamese.py

```
292         self.g_oil_losses = checkpoint['g_oil_losses']
293         self.oil_rec_losses = checkpoint['oil_rec_losses']
294         self.start_epoch = checkpoint['epoch'] + 1
295         checkpoint = torch.load(path + '/map_u_to_v_' + str(epoch) + '.pth', map_location=self.device)
296         self.map_u_to_v.load_state_dict(checkpoint['model_state_dict'])
297         self.u_to_v_optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
298         self.g_photo_losses = checkpoint['g_photo_losses']
299         self.photo_rec_losses = checkpoint['photo_rec_losses']
300         checkpoint = torch.load(path + '/discriminator_photo_' + str(epoch) + '.pth', map_location=self.device)
301         self.discriminator_photo.load_state_dict(checkpoint['model_state_dict'])
302         self.d_photo_optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
303         self.d_photo_losses = checkpoint['losses']
304         checkpoint = torch.load(path + '/discriminator_oil_' + str(epoch) + '.pth', map_location=self.device)
305         self.discriminator_oil.load_state_dict(checkpoint['model_state_dict'])
306         self.d_oil_optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
307         self.d_oil_losses = checkpoint['losses']
308
```

model.py

```
"""
This file contains the implementations for all of the networks used in the siamese architecture.
"""

import torch.nn as nn


class ResidualBlock(nn.Module):
    """
    Implements a residual block as proposed by He et al.
    """

    def __init__(self, in_channels, out_channels, kernel_size, padding=1, stride=1, dilation=1, groups=1):
        super(ResidualBlock, self).__init__()
        self.conv_res1 = nn.Conv2d(in_channels=in_channels, out_channels=out_channels, kernel_size=kernel_size,
                                   padding=padding, stride=stride, dilation=dilation, groups=groups, bias=False)
        self.conv_res1_bn = nn.BatchNorm2d(num_features=out_channels, eps=1e-5, momentum=0.9, affine=True,
                                           track_running_stats=True)
        self.conv_res2 = nn.Conv2d(in_channels=out_channels, out_channels=out_channels, kernel_size=kernel_size,
                                   padding=padding, dilation=dilation, groups=groups, bias=False)
        self.conv_res2_bn = nn.BatchNorm2d(num_features=out_channels, eps=1e-5, momentum=0.9, affine=True,
                                           track_running_stats=True)

        if stride != 1:
            # in case stride is not set to 1, we need to downsample the residual so that
            # the dimensions are the same when we add them together
            self.downsample = nn.Sequential(
                nn.Conv2d(in_channels=in_channels, out_channels=out_channels, kernel_size=1, stride=stride,
                          dilation=dilation, groups=groups, bias=False),
                nn.BatchNorm2d(num_features=out_channels, eps=1e-5, momentum=0.9, affine=True, track_running_stats=True)
            )
        else:
            self.downsample = None

        self.relu = nn.ReLU(inplace=False)

    def forward(self, x):
        residual = x

        out = self.relu(self.conv_res1_bn(self.conv_res1(x)))
        out = self.conv_res2_bn(self.conv_res2(out))

        if self.downsample is not None:
            residual = self.downsample(residual)

        out = self.relu(out)
        out += residual
        return out


class Map(nn.Module):
    """
    Implements the network architecture that maps between the low-dimensional representations.
    Referred to as 'transformation networks' or 'generators' in the thesis. We used the same architecture
    for both domains.
    """

    def __init__(self):
        super(Map, self).__init__()

        self.net = nn.Sequential(
            nn.Conv2d(in_channels=4, out_channels=64, kernel_size=3, stride=1, padding=1),
            ResidualBlock(in_channels=64, out_channels=64, kernel_size=3, stride=1, padding=1),
            ResidualBlock(in_channels=64, out_channels=64, kernel_size=3, stride=1, padding=1),
            ResidualBlock(in_channels=64, out_channels=64, kernel_size=3, stride=1, padding=1),
            ResidualBlock(in_channels=64, out_channels=64, kernel_size=3, stride=1, padding=1),
            nn.Conv2d(in_channels=64, out_channels=4, kernel_size=3, stride=1, padding=1)
        )

    def forward(self, x):
        out = self.net(x)
        return out


class Discriminator(nn.Module):
    """
    Implements a discriminator network. We used the same architecture for both domains.
    """

    def __init__(self):
        super(Discriminator, self).__init__()

        self.conv = nn.Sequential(
            nn.Conv2d(in_channels=3, out_channels=16, kernel_size=5, stride=2),
            nn.LeakyReLU(negative_slope=0.2),
            nn.Conv2d(in_channels=16, out_channels=32, kernel_size=5, stride=2),
            nn.BatchNorm2d(num_features=32),
            nn.LeakyReLU(negative_slope=0.2),
            nn.Conv2d(in_channels=32, out_channels=64, kernel_size=5, stride=2),
            nn.BatchNorm2d(num_features=64),
            nn.LeakyReLU(negative_slope=0.2),
            nn.Conv2d(in_channels=64, out_channels=128, kernel_size=5, stride=2),
            nn.BatchNorm2d(num_features=128),
            nn.LeakyReLU(negative_slope=0.2),
            nn.Conv2d(in_channels=128, out_channels=128, kernel_size=5, stride=2),
            nn.BatchNorm2d(num_features=128),
```

model.py

```
            nn.LeakyReLU(negative_slope=0.2),
        )
        self.linear = nn.Sequential(
            nn.Linear(in_features=128, out_features=1),
        )

    def forward(self, x):
        out = self.conv(x)
        out = out.view(-1, 128)
        out = self.linear(out)
        return out


class Interpolate(nn.Module):
    """
    Downsamples or upsamples the input according to the given scale factor.
    """
    def __init__(self, scale_factor, mode):
        """
        :param scale_factor: (float) the factor by which the input is downsampled / upsampled.
        :param mode: the method that is used for sampling. Possible ptions: 'nearest', 'linear',
                     'bilinear', 'bicubic', 'trilinear', 'area'.
        """
        super(Interpolate, self).__init__()
        self.interpolate = nn.functional.interpolate
        self.scale_factor = scale_factor
        self.mode = mode

    def forward(self, x):
        return self.interpolate(x, scale_factor=self.scale_factor, mode=self.mode)


class AutoEncoder(nn.Module):
    """
    Implements an Autoencoder network. We used the same architecture for both domains.
    """
    def __init__(self):
        super(AutoEncoder, self).__init__()
        self.encode = nn.Sequential(
            nn.ReplicationPad2d(padding=1),
            nn.Conv2d(in_channels=3, out_channels=64, kernel_size=2, stride=1, bias=False),
            nn.BatchNorm2d(num_features=64),
            nn.LeakyReLU(),
            nn.ReplicationPad2d(padding=1),
            nn.Conv2d(in_channels=64, out_channels=32, kernel_size=2, stride=1, bias=False),
            nn.BatchNorm2d(num_features=32),
            nn.LeakyReLU(),
            nn.ReplicationPad2d(padding=1),
            nn.Conv2d(in_channels=32, out_channels=16, kernel_size=4, stride=2, bias=False),
            nn.BatchNorm2d(num_features=16),
            nn.LeakyReLU(),
            nn.ReplicationPad2d(padding=1),
            nn.Conv2d(in_channels=16, out_channels=4, kernel_size=4, stride=2),
            nn.LeakyReLU(),
        )

        self.decode = nn.Sequential(
            Interpolate(scale_factor=2, mode='nearest'),
            nn.ReplicationPad2d(padding=1),
            nn.Conv2d(in_channels=4, out_channels=64, kernel_size=3, stride=1, bias=False),
            nn.BatchNorm2d(num_features=64),
            nn.LeakyReLU(negative_slope=0.2),
            Interpolate(scale_factor=2, mode='nearest'),
            nn.ReplicationPad2d(padding=1),
            nn.Conv2d(in_channels=64, out_channels=32, kernel_size=3, stride=1, bias=False),
            nn.BatchNorm2d(num_features=32),
            nn.LeakyReLU(negative_slope=0.2),
            nn.ReplicationPad2d(padding=1),
            nn.Conv2d(in_channels=32, out_channels=16, kernel_size=3, stride=1, bias=False),
            nn.BatchNorm2d(num_features=16),
            nn.LeakyReLU(negative_slope=0.2),
            nn.ReplicationPad2d(padding=1),
            nn.Conv2d(in_channels=16, out_channels=3, kernel_size=3, stride=1),
            nn.Sigmoid()
        )

    def encoder(self, x):
        out = self.encode(x)
        return out

    def decoder(self, z):
        out = self.decode(z)
        return out

    def forward(self, x):
        z = self.encoder(x)
        r = self.decoder(z)
        return r
```

autoencoder.py

```
import torch
import torchvision
import model
import util
import os


class Autoencoder:

    def __init__(self, in_path=None, num_epochs=100, batch_size=50, learning_rate=0.0001, name=None, verbose=False):
        """
        This class implements the training procedure for the autoencoders.
        :param in_path: (string) the file path indicating the location of the training data.
        :param num_epochs: (int) the number of epochs.
        :param batch_size: (int) the batch size.
        :param learning_rate: (int) the learning rate for the Adam optimizer.
        :param name: (string) the name of the model (used when saving the parameters to file)
        :param verbose: (boolean) if true, the training process is printed to console
        """
        self.device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
        self.use_cuda = torch.cuda.is_available()
        self.in_path = in_path
        self.num_epochs = num_epochs
        self.batch_size = batch_size
        self.name = name

        self.net = model.AutoEncoder().cuda() if self.use_cuda else model.AutoEncoder()
        self.optimizer = torch.optim.Adam(self.net.parameters(), lr=learning_rate, weight_decay=1e-5)
        self.losses = []
        self.verbose = verbose
        self.start_epoch = 1

    def train(self):
        """
        Train the autoencoder.
        """
        training_data = torchvision.datasets.ImageFolder(self.in_path, torchvision.transforms.Compose([
                                                         torchvision.transforms.ToTensor()]))

        data_loader = torch.utils.data.DataLoader(training_data, batch_size=self.batch_size, shuffle=True,
                                                  num_workers=2)

        criterion = torch.nn.L1Loss().cuda() if self.use_cuda else torch.nn.L1Loss()

        progress_bar = util.ProgressBar()

        if not os.path.exists('saves/'):
            os.makedirs('saves/')

        for epoch in range(self.start_epoch, self.num_epochs + 1):
            print('Epoch {}/{}'.format(epoch, self.num_epochs))
            for i, data in enumerate(data_loader, 1):
                x, _ = data
                x = x.to(self.device)

                output = self.net(x)

                loss = criterion(output, x)

                self.optimizer.zero_grad()
                loss.backward()
                self.optimizer.step()

                self.losses.append(loss)

                if self.verbose and (i % 10 == 0 or i == len(data_loader) - 1):
                    info_str = 'loss: {:.4f}'.format(self.losses[-1])
                    progress_bar.update(max_value=len(data_loader), current_value=i + 1, info=info_str)

            progress_bar.new_line()

            self.save(epoch=epoch, path='saves/' + self.name + '_' + str(epoch) + '.pth')

    def encode(self, x):
        return self.net.encoder(x)

    def decode(self, z):
        return self.net.decoder(z)

    def save(self, epoch, path):
        """
        Saves the autoencoder to the specified file path.
        :param epoch: (int) current epoch
        :param path: (string) file path
        """
        torch.save({
            'epoch': epoch,
            'model_state_dict': self.net.state_dict(),
            'optimizer_state_dict': self.optimizer.state_dict(),
            'losses': self.losses
        }, path)

    def load(self, path):
        """
        Loads the weights of the autoencoder from the file path.
        :param path: (string) file path
        """
```

```
        checkpoint = torch.load(path, map_location=self.device)
        self.net.load_state_dict(checkpoint['model_state_dict'])
        self.optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
        self.losses = checkpoint['losses']
        self.start_epoch = checkpoint['epoch']

    def eval(self):
        """
        Sets the autoencoder to evaluation mode.
        This is not necessary but it saves time for evaluations.
        """
        self.net.eval()
        for param in self.net.parameters():
            param.requires_grad = False

```

util.py

```
 1
 2 """
 3 This file contains auxiliary functions and classes that do not fit anywhere else.
 4 """
 5
 6 import matplotlib.pyplot as plt
 7 from matplotlib import pylab
 8 import sys
 9 import numpy as np
10 from PIL import Image
11 import torch
12
13 plt.style.use('ggplot')
14
15
16 def plot_training_losses(d_losses, g_losses):
17     """
18     Creates a plot for the training losses of the discriminator and the generator
19     and saves the figure as a PNG file.
20     :param d_losses: (list) the losses of the discriminator.
21     :param g_losses: (list) the losses of the generator.
22     """
23     plt.plot(d_losses, label='Discriminator')
24     plt.plot(g_losses, label='Generator')
25     plt.title('Wasserstein GAN: training losses')
26     plt.ylabel('Loss')
27     plt.xlabel('Iteration')
28     pylab.legend(loc=9, bbox_to_anchor=(0.5, -0.15), ncol=3)
29     plt.tight_layout()
30     plt.savefig('training_losses.png')
31     plt.clf()
32
33
34 def image_to_tensor(path):
35     """
36     Reads an image from disk space and converts it to a PyTorch tensor.
37     :param path: (string) the file path specifying the location of the image.
38     :return: (tensor) PyTorch tensor representing the image (normalized to [0, 1]).
39     """
40     img = np.asarray(Image.open(path))
41     im_norm = img / 255
42     im_norm = np.moveaxis(im_norm[np.newaxis], 3, 1)
43     return torch.from_numpy(im_norm).float()
44
45
46 def tensor_to_numpy(x):
47     """
48     Converts a PyTorch tensor representing an image to a NumPy array.
49     :param x: (tensor) PyTorch tensor.
50     :return: (array) NumPy array.
51     """
52     return np.moveaxis(x.data.cpu().numpy(), 1, 3).squeeze()
53
54
55 def numpy_to_image(x):
56     """
57     Converts a NumPy array representing an image (normalized to [0, 1])
58     to a PIL Image object.
59     :param x: (array) NumPy array.
60     :return: (object) PIL Image object.
61     """
62     return Image.fromarray(np.uint8(x * 255))
63
64
65 def pil_to_tensor(img):
66     """
67     Converts a PIL Image object to a PyTorch tensor.
68     :param img: (object) PIL Image object.
69     :return: (tensor) PyTorch tensor (normalized to [0, 1]).
70     """
71     img = np.asarray(img)
72     im_norm = img / 255
73     im_norm = np.moveaxis(im_norm[np.newaxis], 3, 1)
74     return torch.from_numpy(im_norm).float()
75
76
77 class ProgressBar:
78     """
79     Creates a progress bar in the command line that illustrates
80     the training progress. This progress bar was inspired by Keras,
81     but the code was not taken from Keras.
82     """
83
84     def __init__(self, width=30):
85         self.width = width
86
87     def update(self, max_value, current_value, info):
88         """
89         Update the progress bar.
90         :param max_value: (int) the maximum value of the progress bar.
91         :param current_value: (int) the current value of the progress bar.
92         :param info: (string) an info string that will be displayed to the right of the progress bar.
93         """
94         progress = int(round(self.width * current_value / max_value))
95         bar = '=' * progress + '.' * (self.width - progress)
96         prefix = '{}/{}'.format(current_value, max_value)
97
```

util.py

```
 98          prefix_max_len = len('{}/{}'.format(max_value, max_value))
 99          buffer = ' ' * (prefix_max_len - len(prefix))
100
101          sys.stdout.write('\r {} {} [{}] - {}'.format(prefix, buffer, bar, info))
102          sys.stdout.flush()
103
104      def new_line(self):
105          print()
106
107
108
```