Artificially Painting Pictures & Artworks

Universität Heidelberg

Fakultät für Physik

Artificially Painting Pictures & Artworks

Masters Thesis

Arthur Willy Heimbrecht

June 2nd 2020

supervised by Professors Björn Ommer and Tilman Plehn

Abstract

This is where the abstract is going to be.

Contents

Introduction | 1

The American Cambridge dictionary defines the word "picture" as

picture: a drawing, painting, photograph, etc.

(As submitted to the Cambridge American Dictionary)

add phonetic signs

which right away separates between different media that can make up a picture. If one were to look up the definitions of "drawing", 'painting" and "photograph" as well, this is what distinguishes the three words from one another.

Paintings are only pictures that are made with paint.

A drawing confines oneself to pictures drawn with a pencil or pen.

And finally, a photograph is a picture taken with a camera.

All of these three methods could be used to picture the same image, but will always give different results, as each medium comes with its own limitations.

The painting will almost always deviate in its details from the original image, as the paint tends to mix and flow. The drawing will present large evenly colored areas often in a grainy way due to the structure of the paper which it has been drawn on. The photograph will limit an artist in showing anything more than the raw visual input.

Yet, all of these techniques also have their own characteristics that other techniques do not possess.

When comparing paintings to photographs this becomes especially clear. The layered texture of paintings which varies vastly in thickness within a single painting is something that photographs will never be able to capture and display, as photographs are just the projection of a 3D world onto a 2D plane. This will not only be confirmed by art historians or the like, but can be seen when looking at the industry of painting replicas of masterpieces with oil such as the Starry Night by van Gogh or most famously the Mona Lisa by DaVinci.

There can only be a market for such replicas can only if there is enough of a difference between an actual painting and a very high resolution photo of a painting.

This does not hold as well for drawings on the other hand. As drawings are also inherently a 2D representation with just an infinitesimal height to them, they CAN be captured by photos quite well. Also, as technology has become more and more advanced, it is possible to capture and print a drawing in such a high quality and fidelity to the original drawing, that it is hard to tell the original drawing and the print apart.

This shows also in the popularity of drawing applications on computers and tablets that have been useful tools to digital artists for over 2 decades now. And further development it has now become a feasible challenge to mimic the feel of pen on paper with styluses on glass tablets.

All of this makes it clear, that the art of drawing has been object to the digital revolution as much as the art of taking photographs.

But what about paintings? Are there not also applications that try to imitate painting techniques, or 3D printed replicas of famous paintings? Yes there are, but it is clear that this comes with a lot of limitations or huge effort, as 3D scans and prints and real-time fluid simulations are cutting-edge technology. And even then the majority of digital content is consumed though 2D screens which are not able to display what makes a painting unique.

So why even bother then, if all of these limitations are in place?

As it already has been hinted, there is existing technology that is capable of taking paintings into the digital realm. At the same time Augmented Reality (AR) and Virtual Reality (VR) gain more traction every year, which will only strengthen this development.

Thus, the question arises: Is it possible to digitalize paintings? One way to achieve this are the mentioned 3D scans and Gigapixel photographs of paintings. Another way are painting applications which simulate brush strokes virtually. But could these two approaches be combined such that a painting can be made up of digital brush strokes that replicate the original brush strokes in a painting?

Such a combined approach would help to promote the current development in this area and bring interesting applications in other fields. Two exemplary applications would be the conservation of images and the study of paintings.

As far as conservation goes, an image that is disassembled into its individual brush strokes will store information about the original painting in a more efficient manner than Gigapixel images or 3D scans. Both of these archiving methods generate huge amounts of data, which is why only a few hundred images in the world are archived in this way [**googleartproject**]. Having extracted brush stroke information at hand could store this information more efficiently and improve reconstruction as simulation techniques improve as well.

Considering the study of paintings, there are already computer assisted methods to distinguish forgeries from real artworks. A way of extracting brush strokes from paintings would open up new possibilities to study the style and techniques of an artists that goes beyond visual inspection.

At last there are other fields that deal with artworks and could benefit from from such an approach such as artistic style transfer in computer vision. Artistic style transfer relies exclusively on images of paintings when it comes to learning the style of an artist. As it has been laid out images do not capture everything that makes up a painting and it would be interesting if style transfer could benefit from more advanced information about a painting.

This thesis will perform an experimental evaluation whether it is possible to extract individual brush strokes from a photograph of a painting. Furthermore, it shall make a first attempt at performing style transfer using brush strokes.

As there is a vast realm of different painting techniques and materials, this work shall only be evaluated on oil painting brush strokes, as there is little data available on other techniques and it is quite possibly the easiest to replicate as well. Even though there are many artists which created oil paintings in the past, this work will mainly focus on later works by van Gogh. These works are well known and thus there are many high quality photographs of his works. Also, most of these paintings show very clear brush strokes which is decisive of van Gogh's style.

introduce the appraoch broadly

reformulate

Structure of this work First, the theoretical background for such an attempt along with related work will be presented. Then the approach itself will be inferred in two parts that describe the training of a differentiable renderer and the subsequent retrieval of brush strokes from an image. This is followed by experiments as well as an ablation study of the approach. Lastly, there will be a discussion about the results and the future direction of research.

Allocation of individual contributions This work has been

Theoretical Background & Related Work

Machine Learning & Computer Vision

2

In this chapter the theoretical backgrounds and motivation of Neural Networks ought to be explored.

First, a physical/biological motivation will set the foundation for the theoretical background accompanied by a more mathematical motivation. Thereafter, more advanced lines of thought will be introduced that finally lead to convolutional neural networks as the current driving force in computer vision .

2.1 Artificial Neural Networks

Neurobiological Inspiration

Many human achievements have at least been partially inspired by studying nature. A very popular example is that of airplanes and birds. The study of how birds are able to fly found that the shape of their wing is the essential element to flying. Inventors and engineers have taken this inspiration and slowly but steadily created planes and the like from it. Yet, planes are not birds, as they are not flapping their wings (yet?), but integrate of other inventions like jet engines to get the same (or better) capability of flying as birds.

Quite similarly, the brain features a lot of insights, how intelligence or something that seems like it, can be constructed. In the same way humans studied birds to understand flying, researchers are now studying the brain to create better artificial intelligence.

In the earliest stages of this they would try to imitate the brain, as people have tried imitating birds at first and failed similarly.

There is current research on replicating the brains structure to the neuron level on hardware with more success [**brainscales**].

Warren McCulloch and Walter Pitts proposed one of the earliest models of a neuron, which targeted simplifying the real Neuron.

Notably McCulloch and Pitts (1943) even preceded Hodkins and Huxley's (1952) Nobel price winning description of a neuron.

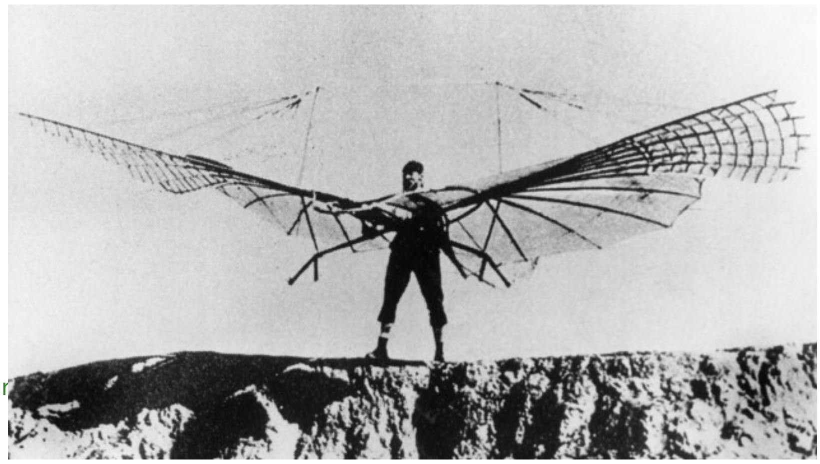


Figure 2.1: Otto Lilienthal with his flying apparatus. https://www.deutschlandfunkkultur.de/geschichte-der-fliegerei-wie-der 976.de.html?dram:article_id=308043

Biological Neuron

For this the functioning of a neuron shall be described shortly.

Human cells in that make up the brain are called **neurons**. They connected to each other through dendrites, synapses and axons and they use these connections to send signals.

A neuron receives signals through its dendrites, which all lead to the cell body (soma). At the cell body the signals are accumulated. If the summed value of the signals' potential reaches a certain threshold, an action potential is generated. The new signal then travels along the axon and its branches towards other neurons. The axon ends in synapses which connect the axon to other neurons' dendrites. Synaptic transmissions are usually mediated by chemicals and not by electrical signals. The chemical nature of the synapse allows it to forward either an excitatory or an inhibitory signal. Excitatory signals will bring the cell potential closer to the threshold, while inhibitory do the opposite [**coloratlas**].

Cell potentials are usually decreases by excitatory signals, as the action threshold sits below the resting potential.

What makes the brain so powerful though, is not the neuron itself with its arguably simple structure but the vast network of billions of these neurons. Each neuron is connected to thousands of other neurons with which it constantly communicates. How a signal is transported between the neurons depends on the interplay of synaptic weights, neuron connections and the threshold of each neurons.

Artificial Neuron

McCulloch and Pitts saw that powerful things can be achieved when connecting lots and lots of simple structures. Thus, they proposed an even simpler model of a neuron:

They restricted their neuron to a binary state (on or off). Each neuron then has a number of incoming signals from other neurons which would either be positive or negative. A neuron would only become active if the number of incoming positive signals minus the number of negative signals exceeds the neuron's threshold. McCulloch and Pitts then also changed the highly parallel and complex nature of biological neural networks to a single layer feed-forward network architecture.

add equation with activation function, when neuron fires

In a feed-forward network neurons are grouped into layers and operate in parallel within a layer. Each neuron in a layer is fed the same input signal (often described as an input layer). The neuron's state is then computed according to an equation such as equation **??**. The activation value of each neuron then defines an output.

As this models deviates from nature quite a bit, these structures a better referred to as **units** instead of neurons.

In a single layer architecture a layer often consists of a single unit (see Figure **??**. Basically, input signals come from one side and output signal go out the other side, which can be expressed in a simple equation like equation **??**. This is not only done for practical reasons but also inspired by observation of layered neuron structures in the brain.

The McCulloch-Pitts model is capable of emulating simple logical relations (AND, OR, NOT) but nor XOR which will be explained later.

Perceptron

This McCulloch's and Pitts' is very much simplified yet comes with some weaknesses. Thus, the **perceptron** has been introduced by Frank Rosenblatt in 1957. Instead of having a binary unit, he came up with a linear threshold unit (LTU). The LTU allows for numeric instead of binary signals which can be weighted with independent factors. Also they use the 'bias trick' to parametrize the threshold or bias of the unit as yet another weight for an input that is constantly one. The resulting equation for each LTU then reads: Equation equation **??** can then be formulated in a vectorized form such that:

equation for LTU with bias trick

vectorized LTU

In this form the calculations for each unit become mathematically and computationally relatively easy. Each layer can be expressed as a vector of activations $\mathbf{x}$. Multiplying this vector with the weight matrix W and applying the element-wise activation function returns the activations for the subsequent layer. This simple yet capable description of a unit built the base for a first surge of interest in neural networks.

The word perceptron describes the whole function f in equation equation **??** which can consist of many LTUs

Mathematical Interpretation

The very simplistic mathematical formulation of the perceptron suggests that there might be a mathematical meaning to it besides the biological analogy. Indeed, the perceptron is equal to the definition of a **binary linear classifier**.

A binary linear classifier, classifies inputs into two classes, hence binary. It does so by drawing a virtual hyperplane in input space and predicts the class for each input, depending on whether it lies above or below this hyperplane.

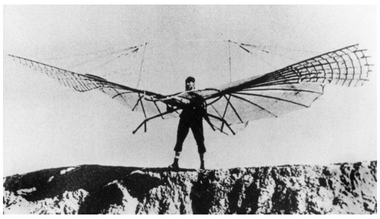


Figure 2.2: As a very simple example, this binary classifier has data on how often the words 'weightloss' and 'invest' appear in an email. Any time these two words appears too often and the data point is above the decision boundary, the email is classified as 'spam'

As the hyperplane (or decision boundary) is quantified by a linear function, the classifier is described as linear.

A classic problem would be classifying email as spam. Given the two data inputs (*i.e.* frequency of of the words 'weight loss' and 'invest') the classifier has to make a decision. For this reason the binary classifier defines a **decision boundary**. Any data point that lies above this decision boundary is classified as 'spam', any point beneath is classified as 'not spam' This makes sense as normal email rarely use the two words. For other people like a nutritionist, the word weightloss can come up more frequently. This means the decision boundary differs for different users and their mail.

To compute where a point lies relative to the decision boundary, a data point's value along each axis x_1, x_2 is weighted individually w_1, w_2 and summed with a bias b. The result is checked whether it is above or below a threshold t.

$$z = w_1 x_1 + w_2 x_2 + b \tag{2.1}$$

With the classification 'spam' if $z > r$ and 'not-spam' $z \leq r$ (*in dubio pro reo*). Equally $z - r > 0$ holds for spam as well such that the threshold can be brought into the equation and z is checked against 0. r can then be absorbed into the bias.

$$\rightarrow z = w_1 x_1 + \ldots + w_D x_D + b - r = w_1 x_1 + \ldots + w_D x_D + b' \tag{2.2}$$

For arbitrary dimensions D this becomes

$$z = w_1 x_1 + \ldots + w_D x_D + b \tag{2.3}$$

where $\mathbf{x}$ and $\mathbf{w}$ can be defined by vectors

$$z = w_1 x_1 + \ldots + w_D x_D + b = \mathbf{w}^T \mathbf{x} + b \tag{2.4}$$

The decision boundary can be easily derived from this, since $\mathbf{w}$ is the orthogonal vector to the hyperplane and $\frac{b}{l} \|\mathbf{w}\|$ is the displacement of the plane along $\mathbf{w}$.

For a simpler notation one can define an additional 'virtual' input which has a constant value of 1 as x_0, the bias b can then be elegantly included into $\mathbf{w}$.

$$z = w_0 b + \mathbf{w}^T \mathbf{x} = w_0 b + w_1 x_1 + \ldots + w_D x_D = \hat{\mathbf{w}}^T \mathbf{x} \tag{2.5}$$

with $x_0 = 1$ and $w_0 = b$. This is called the **bias trick**.

This description also holds for perceptrons with more than one unit. In that case, the input vector $\mathbf{x}$ and the weights $\mathbf{w}$ become matrices with multiple column vectors.

$$\mathbf{x} \rightarrow (\mathbf{x}_1, \mathbf{x}_2, \ldots) = \boldsymbol{X} \mathbf{w} \rightarrow (\mathbf{w}_1, \mathbf{w}_2, \ldots) = \boldsymbol{W} \rightarrow \mathbf{z} = \hat{\boldsymbol{W}}^T \mathbf{X} \tag{2.6}$$

Loss Function

Now with a mathematical definition at hand the next step is to quantify the output. In order to train a classifier, an objective has to be formulated though a **loss function**. Usually (in the supervised case), there is already data set on which the network can be trained. In the given example this would be mails which were read beforehand and then declared either 'spam' $\tilde{y}_i = 1$ or 'not-spam' $\tilde{y}_i = -1$ $\tilde{y}_i$ is called the **label** for a sample x_i with index i.

With the binary linear classifier, the decision boundary has been introduced ($z_i > 0$?) to predict this label for any given sample. This is sufficient to predict a class but a lot of information is lost this way. For training and evaluation the information available in z should be used. *e.g.* a large z implies that the data point is far away from the decision boundary, thus, the classifier is very sure for this classification. For $z \approx 0$ the classifier is not that sure and for $z = 0$ the classifier is indecisive. Ultimately, z can be viewed as a score that is calculated for each data input.

The question then becomes how to quantify how well the classifier performs on the given data. For this reason a loss function is defined which measures the classifier's performance on the data. A straight forward choice is **least squares**, where the score's distance to the label is measured.

$$\mathcal{L}_{\mathrm{LS}} = \sum_i (y_i - z_i)^2 \tag{2.7}$$

The value of the loss function become minimal for $y_i = z_i$ for $i = 1, ..., N$. Yet, this score function is especially susceptible to outliers which will cause the decision plane to skew towards outliers with $z > 1$ or $z < -1$

For this reason **support vector machines (SVM)** employ a **maximum margin** classifier. A maximum margin classifier seeks to find a decision boundary which as far from the closest representatives of each class as possible (see Figure **??**). The maximum margin is defined as

$$\text{margin} = d_+ + d_- \tag{2.8}$$

with d_+ the distance to the nearest training sample with class +1 and d_1 the closest training sample for class −1. Noticeably, this requires for the data to be linearly separable, which means that there must exist a hyperplane which perfectly separates the data according to its class. The margin becomes maximal for $d_+ = d_-$. Since w is orthogonal to the hyperplane, $\mathbf{w}$ can always be rescaled such that

$$d_+ = d_- = \frac{1}{||\mathbf{w}||} \tag{2.9}$$

Additionally, $\mathbf{w}$ can be chosen such that $z = \mathbf{w}_i^T \mathbf{x}_i + b_i \geq +1$

for $\tilde{y}_i = +1$ and vice versa for $\tilde{y}_i = -1$. Thus,

$$\tilde{y}_i z_i \geq 1 \tag{2.10}$$

will hold, for all x_i with equality for points on the margin, as there is always at least one point on the margin.

Thus,

$$d_- = d_+ = \frac{1}{||\mathbf{w}||} \tag{2.11}$$

and the margin

$$d_- + d_+ = \frac{2}{||\mathbf{w}||} \tag{2.12}$$

is maximized when $||\mathbf{w}||$ is minimized.

Subsequently, the classification can be expressed as a relatively simple optimization problem.

$$\arg\min_{w,b} \frac{1}{2} ||\mathbf{w}||^2 \tag{2.13}$$

under the constraints

$$\tilde{y}_i z_i = \tilde{y}_i(\mathbf{w}_i^T \mathbf{x} + b) \geq 1 \forall i \tag{2.14}$$

How to solve this optimization problem shall be explained in Subsection **??**

2.2 Fully Connected Neural Networks

Multi-Layer Perceptron

With the perceptron at hand for which is capable of classification any linear separable data, the question becomes, what are the limitations to this? Minsky and Pappert found the limitations in 1969 with their book 'Perceptrons'. They oulined the limitations of perceptrons with the `XOR` problem. The problem becomes obvious when looking at the `XOR` problem in a 2D plane (see Figure 2.3)

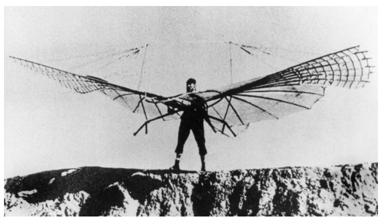


Figure 2.3: `OR` and `XOR` operations visualized. The `XOR` problem cannot be solved by drawing a single line.

As it has been explained in the previous section, the perceptron is equal to a binary linear classifier. As such, the perceptron can only classify linear separable data perfectly.

Since the XOR problem can obviously not be solved with a straight line separating the two classes, the perceptron is also not able to compute such an operation.

This realization led to the first decline in interest in artificial neural networks.

Since then there have been ways of solving this problem for SVMs by projecting the data into a higher dimensional space. Another approach kept the logic as is but went from the existing shallow network approaches to deep neural networks (DNN). **Deep Neural Networks (DNN)** are ANNs which consist of more than one hidden layer. A single perceptron may not be capable of computing XOR but it is capable of calculating AND, OR and their negated forms. By using one perceptron with two units to compute AND and OR, a second layer perceptron can use compute XOR

$$XOR(x, y) = AND(OR(x, y), NOT(AND(x, y))) \quad (2.15)$$

Now, the XOR problem becomes solvable.

The ability to stack multiple layers in ANNs stems from discovery of backpropagation which shall be explained in Subsection **??**.

As before only linear functions could be approximated by perceptrons, multiple layers of perceptrons allow for any higher degree function to be approximated as well. Thus, **Multi-Layer Perceptrons (MLP)** sparked new interest in the field of artificial neural networks.

This interest also originated in the similarly layered structure that has been found in the brain.

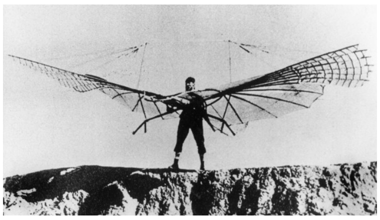


Figure 2.4: The brains structure under a microscope

Ultimately, MLPs really start to show the connected structure in a network that is typically expected.

MLPs are also called **fully connected networks** since each unit is connected to all unit in the previous layers as well as all units in the next layer.

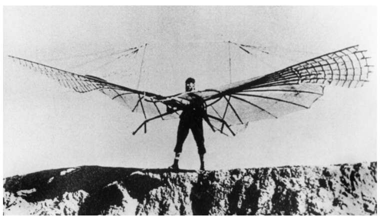


Figure 2.5: Layers of an MLP

Activation Functions

These newfound capabilities for MLPs are not only restricted to binary operations but will translate into continuous space. In this case the hidden-layer perceptrons get stripped of their activation function. The activation function of a perceptron has been used up until now to make a class prediction y from the score z, which is also called pre-activation.

Replacing the step function with a linear activation function (*i.e.* identity function), each hidden-layer's perceptron would initially seem to increase the capabilities of the MLP. Unfortunately, this is not the case as any subsequent perceptrons with linear activations can be reduced to a single preceptron.

$$\mathbf{z}_2 = \boldsymbol{W}_2^T \mathbf{z}_1(\mathbf{x}) = \boldsymbol{W}_2^T \boldsymbol{W}_1^T \mathbf{x} = \boldsymbol{W}' \mathbf{x} \tag{2.16}$$

Consequently, the step-function that was used in the `XOR` problem played an important role. The reason for this is the non-linear nature of the step-function in contrast to any linear activation function. It can easily be shown, that equation 2.16 does not hold for non-linear activation functions.

Thus, the question becomes which other activation functions there are that go beyond binary classification. An early popular choice are sigmoid functions (logistic function or tanh). Especially the logistic function is popular due to its similarity to the step-function amongst other things.

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{2.17}$$

Another popular choice are rectified linear units (ReLU) which are identical to a linear activation for $z > 0$ but mimic the step function for $z < 0$. Basically, a ReLU suppresses the signal of a perceptron until it reaches the threshold of 0 and then forwards the signal unaltered.

Another activation function 'leaky ReLU' attenuates the signal below the threshold with a factor α instead of fully suppressing it.

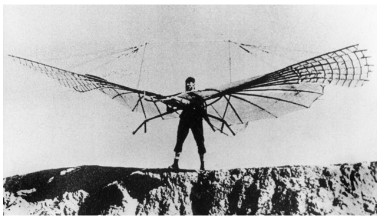


Figure 2.6: A sigmoid function. It saturates to 1 for very large inputs and 0 for very small inputs, similar to the step function.

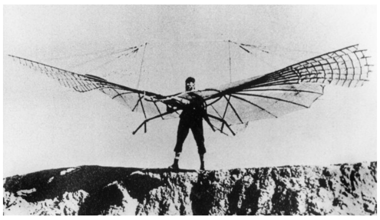


Figure 2.7: ReLu activation function

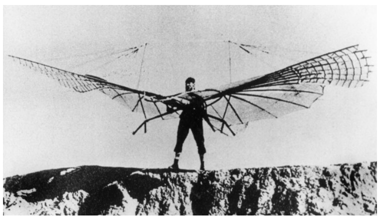


Figure 2.8: leaky ReLu activation function with $\alpha = 0.2$

2.3 Convolutional Neural Networks

With fully connected networks at hand, it seems as if any complex function could be solved by just stacking enough hidden layers. While this seems compelling at first, the issue of the computational burden arises very quickly. Especially for broad layers with many units the number of connections and weights becomes problematic. For two exemplary layers with $1,000$ units each, there would be $1,000,000$ connections as well as $1,000$ biases. This causes two kinds of problems.

1. Each connection represents a multiplication and summation to the activation value which introduces a computational burden.
2. All connections have separate weights which must be trained, which requires a vast number of training data.

Another problem is invariance for spatially or temporally distributed data. If a pixel is to be shifted by a single pixel, or an audio track delayed by a second, the content does not change, and as such the result should not either. An FCN would likely over-fit the data and be susceptible to such variations.

Thus, computer vision as a field is particularly affected by this issue. Since the input are often images with upwards of $64 \times 64 = 4096$ pixels. Hence, a different solution is needed.

Convolutions

A solution to the previously stated problem are convolutional layers.

A **convolution** is a mathematical operation which produces for any given point of a function f the weighted average of its surroundings. The way the surroundings are weighted is through a kernel function g. This principle can be translated into image space where a filter g is applied to each pixel and its surroundings. Specifically, many filters of size $k \times k$ are put on top of the image similar to tiles on a roof. Tiles may overlap but are evenly spaced over the area they cover. The spacing between the centers of each tile is called a stride s and will be assumed the same along each dimension. The return value of each filter then defines a new grid of values much like the original input. Typically, the output of such a convolution is smaller than the input, since the outermost filter must still fit fully into the image. Yet, it is possible to avoid this problem by padding an image with values such that the output has the same size.

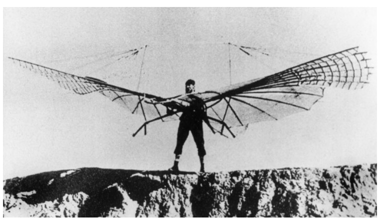


Figure 2.9: A convolution in 1D space

These filters may seem very simple but they are able to capture very interesting properties in an image and transform them as well.

The easiest example would be a 3x3 filter which blurs the image it is applied to. The filter given in Figure 2.10 results in such blurring since the signal of the pixel at the center is dominant but mixed with the signal of its surrounding pixels.

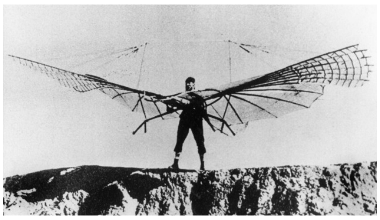


Figure 2.10: 3x3 filter for blurring.

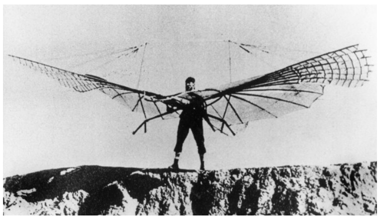


Figure 2.11: 3x3 filter for sharpening edges.

Another filter like Figure 2.11 will sharpen edges in an image since the signal of the surrounding pixels is subtracted from at the center pixel signal.

One can easily image that these kernels can also perform blurring only along one axis or sharpening gradients in one direction. As many such filters are imaginable, there are often many such filters per layer, which are called filter-banks. The output of each filter then defines an input channel for the next layer much like the three color channels in an image. With many channels convolutions can become even more complex as each filter takes all previous channels as input and combines them into a new channel.

Interestingly, the weights of each kernel may not be hand-crafted but can be learned similar to fully connected layers.

Pooling

A sub-type of convolutional layers are pooling layers. Pooling layers have the purpose of downsampling an image or layer to a lower resolution. Basically, downsampling can already be achieved by choosing a stride $s > 1$. Another way is to downsample an image without any learnable weights by using one of two specific hand-crafted kernels.

The first option is average pooling. By taking the mean of four neighboring pixels (each weight is $\frac{1}{4}$) with a stride of $s = 2$ in each channel, the input is scaled down by a factor of 2 along each axis. Figuartively, four pixels will be combined into a single pixel by taking their mean.

Max pooling on the other hand does not take the average but only takes the maximum value of the four inputs. The result will then propagate only the most dominant signals.

Either way, pooling will result in loss of information but is often necessary to reduce the computational load. Especially when the number of channels increases for deeper layers, it is preferable to reduce memory usage and computational load along with the spatial size.

Typical CNN architectures will often stack several convolutional layers then apply a single pooling layer before another stack of convolutional layers is applied. This way the spatial size of the input is gradually reduced while the number of channels is increased at the same time.

2.4 Recurrent Neural Networks

Long-term Short Memory

3 Optimization & Gradient Descent

Optimzers have played a very important role for the development of neural networks and their comeback after the so-called 'AI winter'.

The very early implementations of networks like the perceptron, had very straight forward rules for tuning the weights within the network.

These networks were inspired by nature and so was training them. In accordance to observations made on neurons, weights were

3.1 Optimization Problems

3.2 Gradient Descent

3.3 Backpropagation

3.4 Vanishing Gradients

Normalization

Residual Networks

Ioffe and Szegedy introduced BN to ease training of feedforward networks. affine parameters γ and β

3.5 Optimization Algorithms

Momentum

Stochastic Gradient Descent

AdaM

Other Algorithms

3.6 Reinforcemnt Learning

Generative Models 4

4.1 Decoders

4.2 Generative Adversarial Networks

Discriminators

4.3 Autoencoders

Artistic Computer Vision | 5

This chapter shall dive into related work that combines computer vision with artistic content.

Applying computer vision techniques to images with artistic content has been interesting and challenging at the same time. Due to the often observed change in appearance that can be observed for many artistic images, existing CV pipelines for *i.e.* classification usually do not work as intended. This makes it all the more interesting to see, how these pipelines are affected by this domain gap.

At the same time artistic images bring something into the mix which

5.1 Problem Formulation & Approach

The problem that style transfer might seem easy to capture at first, but it quickly becomes harder when trying to formulate it. As 'style' is simply defined as "a typical way of doing something", it includes actually all aspects

5.2 Brush Stroke Analysis

5.3 Style Transfer

This field of style transfer has its origins

Early Approaches

Hertzmann *et al.* proposed a way of stylizing images by using trainable filters. They call their technique 'image analogies' since the transformation between two training images is analogously applied to a test image.

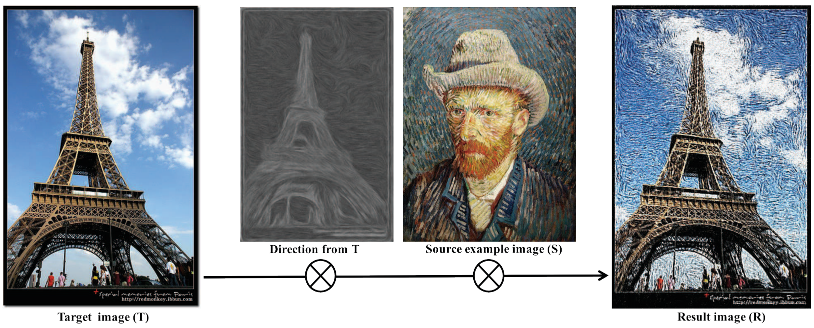


Figure 5.1: Target image T is combined with a dictional map specially obtained from T and a style image S. The result maintains the direction's flow while presenting the texture from S.

Basically, for creating the impression of brush strokes, a photograph A and a painting of that photograph A' are required. As this is rarely the case, **iamgeanalogies** showed that using anisotropic diffusion works also reasonably well to generate A from A'. Then their algorithm searches locally around a position p for the best filter parameters $F(p)$ that transform $A(p)$ into $A'(p)$. By using another search algorithm to match similar regions $B(p)$ and $A(p)$, $F(p)$ can be used to transform $B(p)$ into $B'(p)$. Thus B' – a stylized version of B – can be obtained by transforming it analogously to $A \rightarrow A'$. Other works like Lee *et al.* [2] have built on this principle idea which has led to a fast texture transfer algorithm [**fasttexturetransfer**]. Besides only matching local region according to their pixel values, Lee *et al.* obtain a flow map (which they call 'directions'), that is based on the content image's gradient. This flow is then also matched against the style image.

Neural Style Transfer

The field of style transfer has really gained traction in 2015 with the publication of **A Neural Algorithm for Artistic Style** by Gatys *et al.* It was the first approach to transfer the style of one image to another and at the same time maintaining a high contextual fidelity. In retrospective, this work really kicked off neural style transfer as a field.

Gatys *et al.* themselves pinpoint the novelty of their approach as 'manipulations in feature spaces' as opposed to previous approaches that 'directly manipulate the pixel representation of an image'[3]. They use existing neural architectures and extract information in two separate ways, such that content and style can be separated.

Previous works already used **perceptual loss** to accumulate information on the content in an image [**percep_loss**], or check whether two images have the same content [**other_percep_loss**]. Perceptual loss is based on the VGG-19 architecture [**VGG**] which is a deep CCN trained for object classification on ImageNet [**imagenet**]. By arguing that the network's layer activations increasingly respond to the content when following the networks' hierarchy. Some much even, that it is possible to reconstruct the content of an image by using the activations of one such layer.

For reconstruction of an image's content, gradient descent is performed on a white noise image. The gradient descent aims to minimize the perceptual distance between the reconstruction and the target image. Perceptual distance is defined as the L2-distance between the activations of two images in deep layer of the VGG-network.

For image vector $\mathbf{x}$ with $\mathbf{x} \in= \mathbb{R}^{M_0}, M_0 = H_x \dot{W}_x$, a layer l of the network has N_l feature maps of size M_l. In this case M_l is equal to the height times the width of the feature map of the l-th layer. The activations of the i-th filter ($i \in N_l$) at position j ($j \in M_l$) at layer l can then be represented by matrix $F(\mathbf{x})^l_{ij} \in \mathbb{R}^{N_l \times M_l}$. The perceptual distance is then defined as

$$d_{\text{percep}}(\mathbf{x}, \mathbf{y}) = \sum_{i,j} (F(\mathbf{x})^l_{ij} - F(\mathbf{y})^l_{ij})^2 = \left\| \mathbf{F}(\mathbf{x})^l - \mathbf{F}(\mathbf{y})^l \right\|_2^2 \quad (5.1)$$

, which allows to define the perceptual loss or content loss as

$$\mathcal{L}_{\text{content}} = \frac{1}{2} \sum_{i,j} (F(\mathbf{x})^l_{ij} - F(\mathbf{y})^l_{ij})^2 = \frac{1}{2} \left\| \mathbf{F}(\mathbf{x})^l - \mathbf{F}(\mathbf{y})^l \right\|_2^2 \quad (5.2)$$

The factor $\frac{1}{2}$ will cancel out when deriving $\mathcal{L}_{\text{content}}$ with respect to $F(\mathbf{x})^l_{ij}$.

Minimizing the content loss between two images, by using gradient descent the content of an image can be restored (see Figure 5.3).

As approximating the content of an image has now become possible, the question is whether this is possible with style as well. Gatys *et al.* again turned to the pre-trained VGG network for this. They explicitly reduce style to texture for this reason and thus search for a feature space that captures **texture** rather than content. Subsequently, Gatys *et al.* propose the use of **Gram matrices** as they capture the correlations of feature-activations over their spatial extent.

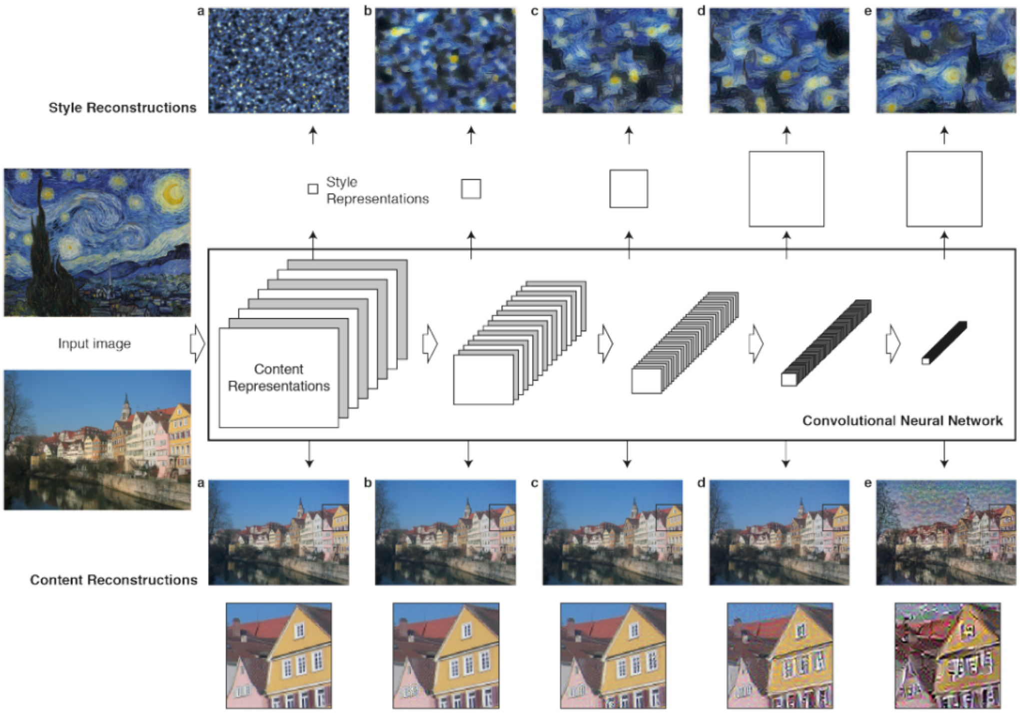


Figure 5.2: Reconstructions of content(bottom) and style(top) using different layers. [3]

The Gram matrix of a given matrix **A** is the inner product of all column vectors in **A**.

$$G = \langle a_i, a_j \rangle = \mathbf{A}^T\mathbf{A} \text{ if } a_1 ... a_j \text{ are column vectors of } \mathbf{A} \quad (5.3)$$

The resulting Gram matrix G now has the form $j \times j$ and captures texture information but no longer the global content.

Section **??** already mentioned that style is very complex and exists at various scales at the same time which Gatys *et al.* address by using many layers. As these layers sit at different depths their field of view varies and each layer captures information at a different scale. Early layers will tend to hold small scale information, later layers will hold larger scale information with every layer.

Gatys *et al.* first compute the Gram matrices each layer l for both the target style image and the current image. Then they use the L2 distance metric to measure the discrepancy between them.

$$G(\mathbf{x})^l = \frac{1}{(2N_x^l M_x^l)^2} F(\mathbf{x})^{lT} F(\mathbf{x})^l \tag{5.4}$$

$$G(\mathbf{y})^l = \frac{1}{(2N_y^l M_y^l)^2} F(\mathbf{y})^{lT} F(\mathbf{y})^l \tag{5.5}$$

$$d^l_{\text{style}}(\mathbf{x}, \mathbf{y}) = \left\| G(\mathbf{x})^l - G(\mathbf{y})^l \right\|_2^2 \tag{5.6}$$

$$\tag{5.7}$$

The style distances at each layer are then weighted and summed up to make up the style loss:

The denominator of $\frac{1}{4N_x^{l\,2} M_x^{l\,2}}$ is squared since the Gram matrix is the product of a matrix with itself transposed.

$$\mathcal{L}_{\text{style}} = \sum_l w_l d^l_{\text{style}}(\mathbf{x}, \mathbf{y}) \tag{5.8}$$

This style loss can again by used together with gradient descent in order to check whether it is possible to reconstruct the texture of an image much like the content of an image. Figure 5.3 shows that it is in fact possible to reconstruct the texture of the image at various scales. Specifically the local consistency of each texture becomes larger, the deeper the layer sits.

Gatys *et al.* now combine the losses for one content image **c** with a style image **s** and optimize **x** in the same way the reconstructions have been obtained.

$$\mathcal{L}_{\text{total}} = \lambda_{\text{content}} \mathcal{L}_{\text{content}}(\mathbf{x}, \mathbf{c}) + \lambda_{\text{style}} \mathcal{L}_{\text{style}}(\mathbf{x}, \mathbf{s}) \tag{5.9}$$

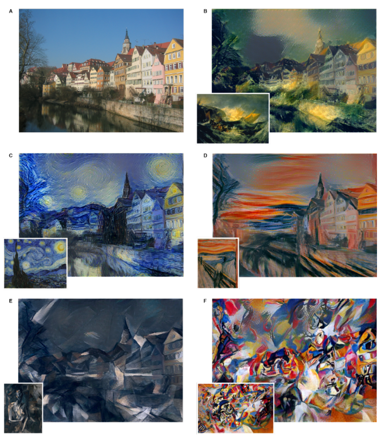


Figure 5.3: Style transfer examples by Gatys *et al.* [3]

The result of this can be seen in Figure **??**

Follow-Up Research

There has been some follow-up research on Gatys *et al.*'s work which addresses mainly how the style loss works.

Li *et al.* have shown that the style loss is equivalent to calculating the **maximum mean discrepancy (MMD)** between the features of each layer [4]. MMD is a test-statistic for a null hypothesis $p = q$ with the data $X = \{x_i\}_{i=1}^n$, sampled from p,

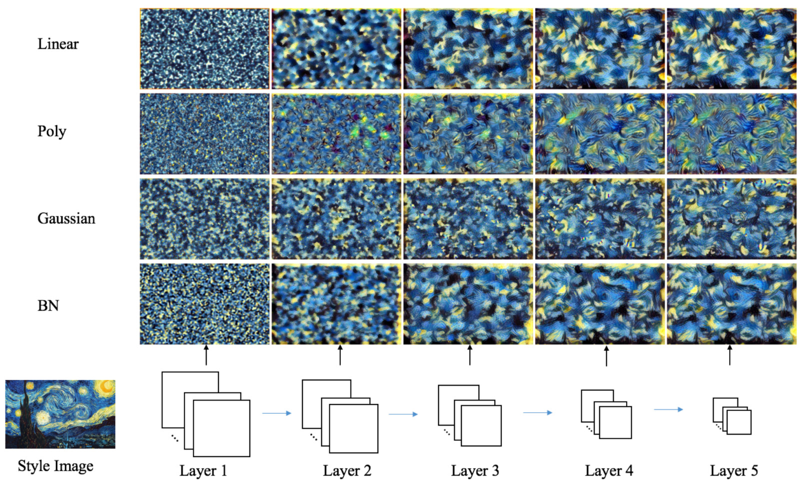


Figure 5.4: Reconstructed textures for Starry Night using different kernel functions k [4]

and $Y = \{y_i\}_{i=1}^{n}$, sampled from q, at hand. It can be used as a difference measure as well and vanishes only if $p = q$.

$$\mathrm{MMD}^2[X, Y] = \frac{1}{n^2} \sum_{i=1}^{n} \sum_{i'=1}^{n} k(\mathbf{x}_i, \mathbf{x}_{i'}) \tag{5.10}$$

$$+ \frac{1}{m^2} \sum_{j=1}^{m} \sum_{j'=1}^{m} k(\mathbf{y}_j, \mathbf{y}_{j'}) \tag{5.11}$$

$$- \frac{2}{nm} \sum_{i=1}^{n} \sum_{j=1}^{m} k(\mathbf{x}_i, \mathbf{y}_j) \tag{5.12}$$

MMD can be based on different kernel functions k and Li *et al.* have shown that the style loss is equivalent to the squared MMD with a polynomial kernel. Consequently they were able to show, that style transfer work with different kernel functions as well and even by explicitly matching the batch statistics (see Figure 5.4):

$$d^l_{\text{style}}(\mathbf{x}, \mathbf{y}) = \frac{1}{N_l} \sum_{i=1}^{N_l} \left((\mu^i_{F(\mathbf{x})^l} - \mu^i_{F(\mathbf{y})^l}) + (\sigma^i_{F(\mathbf{x})^l} - \sigma^i_{F(\mathbf{y})^l}) \right) \tag{5.13}$$

LenDu tested whether pre-trained weights play an important role when performing style transfer. He was able to show basic style transfer even with random initialized networks, but results vary wisely depending on the random initialization. Ultimately, it is possible so obtain some style transfer with this technique but the pre-trained weights seem to play an important role in stablizing the reconstruction process.

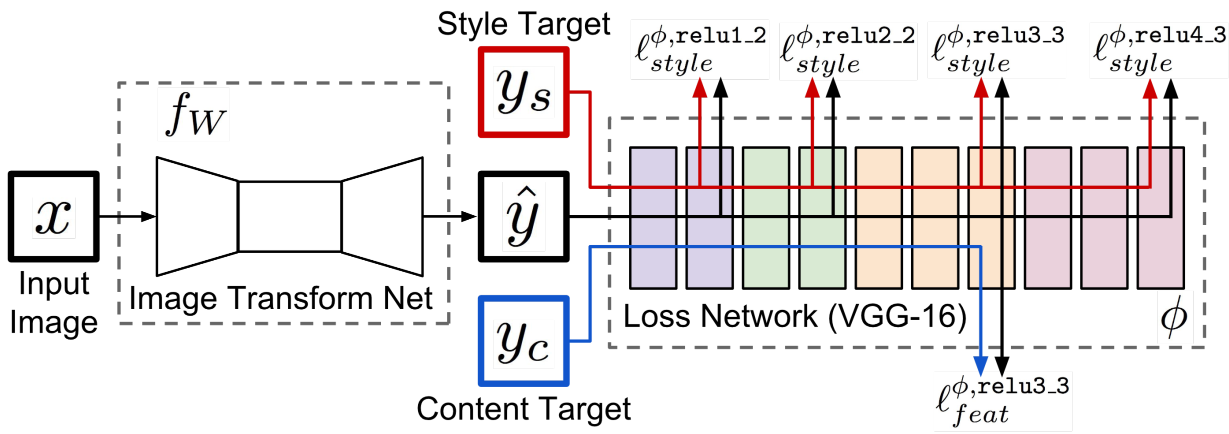


Figure 5.5: Training set-up by Johnson *et al.* [5]

State of the Art

Real-Time Style Transfer

Following Gatys *et al.* seminal work, other have followed suit in trying to stylize images with neural networks. **Johnson** were the first to use the same losses but train a feed-forward architecture with it [5]. They were able to significantly speed up the stylization process like this as stylization was performed in a single feed-forward pass instead of a lengthy gradient descent optimization. Ultimately, this enabled them to generate stylized images in real-time from a given content image, using a deep residual convolutional neural network.

Arbitrary & Universal Style Transfer

Huang *et al.* used a different feed-forward approach for arbitrary style. They first encode an arbitrary style image $\mathbf{s}$ as well as a content image $\mathbf{c}$ using a pre-trained VGG network. This allows them to obtain the activations at a very deep layer of the network $F^l(\mathbf{s})$ and $F^l(\mathbf{c})$. Then they compute the second order statistics for both $\mu_F^l(\mathbf{s}), \sigma_F^l(\mathbf{s})$ and $\mu_F^l(\mathbf{c}), \sigma_F^l(\mathbf{c})$. Using adaptive instance normalization (AdaIN), they rescale the content activations such that they match the statistics of the style activations.

$$F'^l = \sigma_F^l(\mathbf{s}) \frac{F^l(\mathbf{c} - \mu_F^l(\mathbf{c})}{\sigma_F^l(\mathbf{c})} + \mu_F^l(\mathbf{s}) \tag{5.14}$$

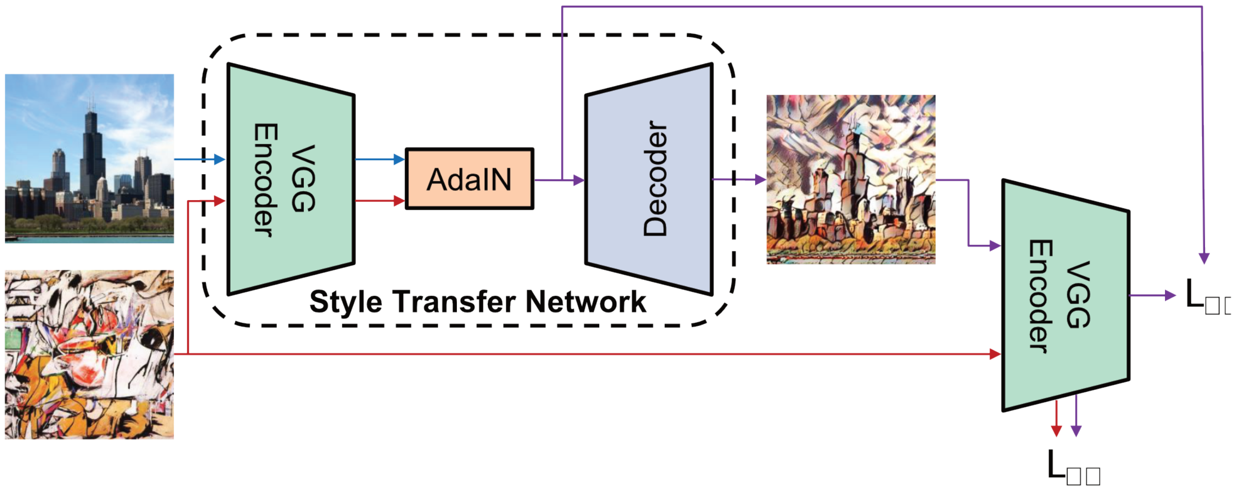


Figure 5.6: Training set-up by Huang *et al.* [6]

Finally, they train a decoder that minimizes the style and content loss, as they have been proposed by Gatys *et al.* They achieve comparable results to other style transfer approaches at a similar speed to Johnson *et al.* while allowing for any target style eventhough only training only once.

this is actually just PCA decompsition to a certain degree

A similar approach by Li *et al.* relies on matching the covariance and the mean of content and style activations. They do that through what they call 'whitening and coloring transform' [7]. First they whiten $F^l(\mathbf{c})$ into $\hat{F}^l$ such that $\hat{F}^l\hat{F}^{lT} = I$

$$\hat{F}^l = E_c D_c^{-\frac{1}{2}} E_c^T (F^l(\mathbf{c}) - \mu_c) \tag{5.15}$$

Where D_c is the diagonal matrix of eigenvalues of the covariance matrix and E_c is the respective orthogonal matrix of eigenvectors. such that

$$F^l(\mathbf{c})F^l(\mathbf{c})^T = E_c D_c E_c^T \tag{5.16}$$

Then coloring is performed by rescaling the whitened representation $\hat{F}^l$ into $\tilde{F}^l$

$$\tilde{F}^l = E_s D_s^{\frac{1}{2}} E_s^T \hat{F}^l + \mu_s \tag{5.17}$$

$$F^l(\mathbf{s})F^l(\mathbf{s})^T = E_s D_s E_s^T \tag{5.18}$$

The resulting $\tilde{F}^l$ is then decoded with a pre-trained decoder to render the final stylized result. Li *et al.* use pre-train the decoder solely on natural images and perceptual loss and reconstruction loss as objective. Additionally, they introduce a pipeline that performs style transfer on multiple scales sequentially. They achieve good results in real-time with just training the decoder once.

Bidirectional Style Transfer

Zhu *et al.* went a different way on style transfer and rely on a generative adversarial objective to identify style in an image. Specifically, they transform images between any two domains $x \in \mathcal{X}$ and $y \in mathcalY$, not just photos and artworks. For this, they use two discriminators (D_X and D_Y, one for each domain) as well as two transformation networks ($G : X \rightarrow Y$ and $H : Y \rightarrow X$). Each translation network then transforms a given sample from one domain into the other and the discriminator assesses the result.

$$\mathcal{L}_{\text{adv}} = \log D_Y(y) + \log(1 - D_Y(G(x))) \tag{5.19}$$

Additionally, the transformed image is then transformed *again* and compared to the original input, in what they call **cycle loss**.

$$\mathcal{L}_{\text{cycle}} = ||F(G(x)) - x||_2^2 \tag{5.20}$$

In the end, Zhu *et al.* are able to stylize and de-stylize images with their networks G and H. The main novelty here though is the good stylization quality they achieve without any of the previously introduces style losses. They have also shown one way, in which GANs are also capable of performing style transfer reasonably well. Other notable efforts were .

list GAN-based style transfer efforts

Adversarial Style Transfer

Building on these GAN-based approaches, Sanakoyeu *et al.* improved the quality of adversarial style transfer and extended it to abstract styles as well. They argue that ImageNet-based approaches inherently favor photorealistic styles through the data set that ImageNet has been trained on [9]. Furthermore, approaches like cycleGAN suffer a similar fate as the back-transformation with cycle consistency opposes loss of detail in more abstract styles.

In order to retain content and global structure of an image, they introduce a fixed-point loss, which requires the stylized image to stay as-is when being re-stylized.

$$\mathcal{L}_{\text{content}} = \|E(G(E(x))) - E(x)\|_2^2 \tag{5.21}$$

To minimize this loss, the encoder must understand original content and stylized content. They also implement a transformed reconstruction loss for better visual quality of the stylized image

$$\mathcal{L}_{\text{transformed}} = \|T(x) - T(G(E(x)))\|_2^2 \tag{5.22}$$

The results show good visual quality, especially concerning the details and loss of details for abstract styles. Also this approach focusses on stylizing not only for a single image but the style of an artist in general.

dima take this further and focus on stylizing different content specifically. This means, a person is differently depicted than a tree, considering level of detail, colorscheme *etc.*. , which holds with real-world experience. They achieve this by using the same fixed-point loss that Sanakoyeu *et al.* but combine it with a second update step. In this second update step they require similar scenes to be placed closely in feature space and dissimilar scene to lie further apart. They add a transformation block between encoder and decoder shape the feature space accordingly.

Others

There exist many other approaches that are capable of transferring style. Some focus very heavily on stylization of portraits using self-attention modules [**ugatit**]. Others choose an approach similar to cycleGAN but add a shared encoding space for content and separate attribute spaces where style is encoded [**unit**, **munit**, **drit**, **drit++**]. With the latter ones mainly focussing on separating shape and appearance of images and recombining them arbitrarily. One such example is taking the posture of a person in one image and combining it with the clothes and appearance of a person in another image.

The lines between these the applications and style transfer as it has been presented are blurry with many approaches that are capable of performing both.

5.4 Painterly Rendering

Painterly rendering is a field of computer vision that focusses stylization of images on giving the impression that certain tools were used. Most often this would be the looks of pencil drawings or brush stroke paintings as these looks are very distinctive. Thus, painterly renderings rely on a brush stroke or a pencil line as its smallest unit which generates the data. Style transfer, in contrast, still uses individual pixels as smallest unit.

The use of these larger units often comes with a level of abstraction which painterly rendering approaches often enforce explicitly though the coarseness said units. One could see this as a hard regularization. This is the reason why painterly rendering is rarely compared to style transfer (which has been described in the last section) as style transfer achieves this abstraction more implicitly. For instance, an impressionistic style is often achieved by limiting the length and width of brush strokes in painterly renderings.

There are several different key approaches how to achieve this, which shall be presented chronologically.

The earliest approaches were stroke-based renderings, which artificially generate single strokes. With such strokes at hand the challenge is to both improve the quality of these strokes and improve the algorithm which aligns them. This field is especially close to the approach of this thesis.

Early filter-based rendering approaches then appeared with superior computational efficiency and ultimately led to style transfer and texture transfer. This has already been explained int he previous section but shall be mentioned here to place it historically.

Lately, drawing networks revived stroke-based rendering with modern machine learning techniques. The focus of this field is really to remove any hand-tuning of parameters that stroke based rendering required and imitate the way humans would draw.

Stroke-Based Rendering

Stroke-based rendering focuses on generating real-looking brush strokes and composing images with these. Two aspects which are of importance in this thesis as well.

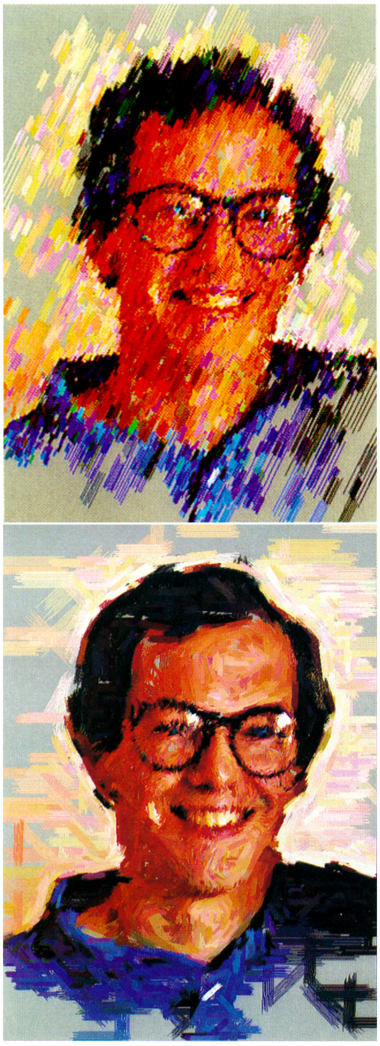


Figure 5.7: Interactively painted images using Haeberli's method with a hand-selected orientation (Figure **??**) and a gradient-driven orientation(Figure **??**).

Haeberli introduced stroke-based rendering with his seminal work 'Paint by numbers: abstract image representations' [10]. In this work he presents two methods which would allow for reconstructing images in an abstracted representation. His first approach is interactive and requires a user to click a certain points in an image to place a brush stroke there. Then his software would automatically select a brush stroke color and size. He proposed several ideas on how to align the brush strokes on the canvas. Either users could do this on their own, or the orientation would be perpendicular to either the global gradient (uniform alignment) or local gradient (non-uniform alignment) of the image.

Haeberli even introduced the use of a scanned brush stroke texture in his algorithm.

Additionally to his interaction-based approach, Haeberli also introduced a relaxation based approach [10]. In this approach a given set of 100 brush strokes is iteratively perturbed. If the perturbation minimizes the energy function (L2-distance), the perturbation is kept. If not, another perturbation is applied.

This is described by Hertzmann as trial-and-error algorithm and very similar to genetic algorithms which shall be explained later [11].

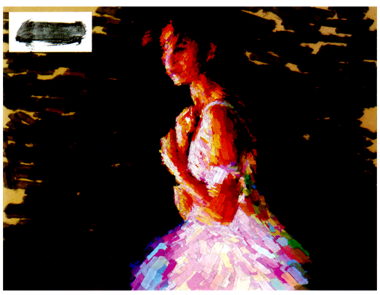


Figure 5.8: Rendered image with a brush stroke texture

Based on Haeberli's work, other authors automated the process of stroke positioning. At the same they improved various aspect of brush strokes which were well categorized in a review by Hegde *et al.* The relevant categories are position, path and orientation, length and width, ordering, and color [**PRveview**].

Litwinowicz proposed a straight-forward way of placing brush strokes evenly spaced over the entire image. Parameters are then inferred similar to Haeberli's approach. To give a more random feel to these brush strokes, the obtained parameters are randomly perturbed according to preset parameters. Also, a weakness of orienting brush strokes perpendicular to the gradient is dealt with. For large uniform areas with little to no gradient, the orientation could become arbitrary. Litwinowicz proposes to refine the gradient field by interpolating between the boundaries of large uniformly colored areas. Also, Litwinowicz introduced temporal coherence to brush strokes, which meant that brush strokes move with the optical flow between frame in a video.

Figure 5.9: Image approximated by relaxation.

hertzmann reformulated the problem as an energy minimization problem in two publications [**hertzmanreview**, **Hertzmann**].

$$E(R) = E_{\text{recon}}(R) + E_{\text{area}}(R) + E_{\text{nstr}}(R) + E_{\text{cov}}(R) \tag{5.23}$$

$$E_{\text{recon}}(R) = \sum_{x \in W, y \in H} w_{\text{recon}\,x,y} \left\| I_{x,y}(R) - I_{x,y} \right\|_2^2 \tag{5.24}$$

(5.25)

where R is a brush stroke representation, I is the target image, and $I(R)$ is the rendered representation. x and y are pixel positions. By adjusting the different weights, properties of the rendering can be altered. w_{recon} can vary spatially and dictate how well the reconstruction must fit the original image in certain areas.

Additionally, he added long strokes as B-splines with arbitrary control points. In contrast, Haeberli and Litwinowicz argued that short straight brush strokes would aid the perception of impressionistic style. Furthermore, **Hertzmann** added advanced rendering for brush strokes with synthetic textures in his work [**Hertzmann**].

Hertzmann combined all these aspects in his approach with advanced relaxation methods similar to Haeberli. Based on trial-and-error search, **Hertzmann** samples a local region along the many dimensions that represent a single brush stroke. The best set of parameters that minimizes the energy function E is then picked as new parameters.

In order to achieve better visual quality **Hertzmann** also employs a coarse-to-fine multi-layer rendering approach. Hereby, he blurred the image in the early iterations of his method and fixed the brush size at a large value. Blurring of the image would then be gradually reduced along with the brush size. The final implementation is further optimized to accelerate the relaxation algorithm and allow for more brush strokes than Haeberli's approach.

Ultimately, **Hertzmann** achieves respectable results with his approach and many ideas of this thesis can be found in his works as well.

Stroke Rendering

Parallel to the advancements in arranging brush strokes in stroke-based rendering, others improved the rendering quality for many different styles. Most notably for this work, **baxter** were the first to implement a full 3D simulation of a brush and the process of placing paint on a canvas. They were even able to simulate mixing of color and different levels of dryness this way. Ultimately, they were able to showcase a drawing by a real artist based on the simulations in real-time [**baxter**].

wetbrush took this even further by including more accurate simulations of fluid dynamics and even single bristles in a brush stroke. They were able to generate a 3D model of paint on canvas from this which they could make subject to different lighting [**wetbrush**]. **adobe** tried to imitate this with the goal to achieve real-time simulation of the methods presented by **wetbrush** Indeed, they were successful up to a point, where deviations to [**wetbrush**] became visible for too many stacked layers of paint.

Drawing Networks

Drawing networks belong to the realm of printerly rendering approaches, yet are rarely compared with their predecessors.

Some of the first approaches to rediscover this area used recurrent neural networks to imitate predefined stroke sequences [**sketchRNN**, **graves**]. Such approaches require paired data where the sequence for a sketch or handwriting is already known.

SPIRAL (SPIRAL) by **SPIRAL** were the first to break this barrier and learn sequences without such paired data [**SPIRAL**]. They used a graphics engine and deep reinforcement learning, to train their network. Based on the current state of the canvas, SPIRAL predicts an action in form of brush stroke parameters. This process is iteratively applied to the canvas N times. Only at the end the result is evaluated in an adversarial fashion against the target image. Notably, the renderer is seen as a black-box and SPIRAL is agnostic to the given interface. Results show that this approach works well for simple stroke-based data such aas handwriting data sets. Tests on images, such as the CelebA faces data set, show limitations as SPIRAL could only be trained for up to 20 strokes.

Other approaches followed suit, after **SPIRAL**'s seminal work. Many approaches were inpsired by **worldmodel**'s idea to encode an environment through an autoencoder into latent space. A model is then trained on this fewer-dimensional latentspace representation instead of the real-world data.

strokenet employ differentiable rendering combined with two encoders to predict a single brush stroke in a feed-forward manner [**strokenet**]. First they train a postion encoder (which actually is a decoder) that transforms an input position to a coordinate in a 64x64 matrix. Then, they train the renderer with n control points decoded by the positon encoder combined with additonal information about brush size and pressure. The dataset is randomly generated with a fixed number of control points from a custom render engine. In the end, the agent is trained, which encodes the current canvas as well as the target image to predict a set of parameters for a single brush stroke.

THeir method works notably better for predicting single brush strokes on empty canvas, that dollwing brush stroks for more complex shapes. **strokenet** also showed tests on images but got very poor results.

neuralpainters chose a different approach to differentiable rendering. He first trained a generator to decode action space input to brush stroke images in adversarial fashion. Then he used an LSTM-based agent to predict a sequence of brush stroke parameters which are rendered by the generator. As the whole pipeline is differentiable, **neuralpainters** was able to avoid the use of reinforcement learning. Results show better image reconstruction than SPIRAL when using the generator, but slightly worse results when combining the obtained action with the original renderer. **neuralpainters** also presents the capability of reconstructing content similar to Gatys, Ecker, and Bethge by minimizing the perceptual loss directly in action space with his network.

learning2paint also employ a pre-trained decoder and use more advanced reinforcement learning techniques [**learning2paint**]. This allows allows their network to predict strokes for an arbitrary number of steps, where SPIRAL could only predict up to 16 strokes. Their results show significantly more detail, especially as the number of strokes increases.

All previous approaches work only on very limited resolutions. This is why **paintbot** extend reinforcement-based approaches to higher resolutions by using more advanced reinforcemnt learning rechniques [**paintbot**, **LpaintB**]. They do not use a differentiable renderer but a sliding-window approach which allows them to generalize to images of larger scales. Notably, they train each network specifically for an image, as the training does not generalize well for different images. They argue that this can be seen as a form of style transfer but such reconstructions mainly show a simple color shift.

LpaintB's approach is the closest approach to this thesis, considering the goal of their approach. They are able to generate stroke-based renderings of high-resolution photographs as well as images of painting just shy of 1 megapixel. Each network is specifically trained for a single target image, which takes at least 1h to accomplish [**paintbot**].

Genetic Algorithms

Genetic algorithms are typically not closely associated with painterly rendering or drawing networks, even though they represent just a different approach to algorithms for this problem.

Genetic algorithms already perform a similar task in order to approximate images by other geometric shapes or even smaller photos (also known as the popular photo mosaic effect). Starting with a random set of circles that are parameterized by their position, radius, and color, it then chooses the most successful samples and resamples in a region around these. This process is repeated until a certain level of convergence is reached.

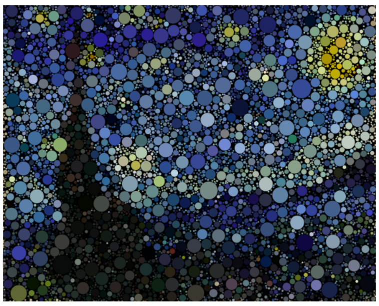


Figure 5.10: Starry Night approximated by a genetic algorithm using only circles. https://effyfan.com/2018/03/02/w6-van-gogh-flowfield/

Figure 5.11: Photo mosaic of Starry Night using only images by the Hubble Space Telescope. http://www.astro.uvic.ca/~alexhp/new/figures/starrynight_HST.001.jpg

As well as this does work, it is very much computationally expensive as most samples will not fit the image, thus searching for the small set of fitting shapes requires to evaluate all the wrong shapes as well. Considering artworks, brushstrokes have many more degrees of freedom, and artworks usually consist of upwards of a few thousand brushstrokes. Consequently, it would be considerably more challenging to apply to this problem until computational resources have become a few magnitudes more powerful.

5.5 Conclusion

Works like these show that there is interest in improving the missing details in style transfer. Ultimately, it it would be desirable that the fields of style transfer and painterly rendering converge further. Style transfer brings impressive perception of style. Painterly rendering brings realistic composition and/or low-level details.

This thesis aims to improve reconstruction quality for painterly renderings without reinforcement learning techniques while at the same time generating an interpretable brush stroke representation.

Contribution and Experiments

Approach | 6

6.1 Motivation

two stage approach be Ha and Schmidhuber 'world model"

The basic approach of this work consists of two steps:

1. A differentiable renderer which can generate images of brushstrokes from a parameter representation.
2. An optimization procedure that iteratively approximates an image through brushstrokes representations.

Having two separate steps can be motivated by comparing the optimization procedure to the actual process of painting an image. An artist will most likely not pick single color particles and then place them on canvas. Instead, an artist uses a brush or other utilities (see Pollock or others) to place more paint with a single action.

Figure 6.1: A typical set of brushes and spatulas used for oil paintings.

Doing so – of course – limits the control over each drop of paint but maintains enough control to still create very delicate details in paintings. This trade-off depends on the brush's size, such that an artist must choose the brush size depending on the content.

reformualte this

An example would be the painting of a uniformly colored sky. Using a large brush size, the artist can cover a lot of canvas in relatively little time as well as keep the color well distributed over the canvas because the brush spreads the color more or less evenly within the brushstroke. On the other hand, if one were to draw a sky with the smallest brush available, not only would it take forever to paint, it would also be hard to keep the paint evenly distributed over multiple strokes.

Now, translating this onto the given problem of recreating/approximating an image through brushstrokes, it would mean to limit the process to only use what we would describe as brushstrokes.

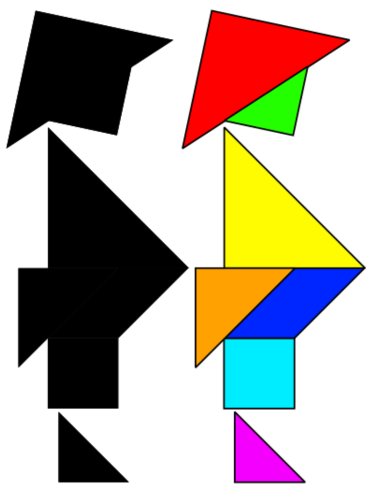


Figure 6.2: An Example of Tangram.

A comparable example is the game of Tangram.

Tangram is a Chinese puzzle game that has the objective of replicating a given silhouette only with a set of 7 unique shapes. The shapes may not overlay or be cut or anything. Quite similarly, the objective of an optimizer is to replicate an image by only using brushstrokes.

Genetic algorithms already perform a similar task in order to approximate images by other geometric shapes or even smaller photos (also known as the popular photo mosaic effect).

Genetic algorithms follow a random sampling approach that 'evolves' as genomes do. Starting with a random set of circles that are parameterized by their position, radius, and color, it then chooses the most successful samples and resamples in a region around these. This process is repeated until a certain level of convergence is reached.

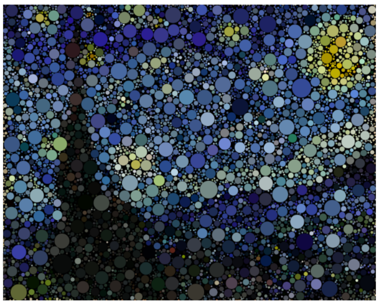


Figure 6.3: Starry Night approximated by a genetic algorithm using only circles. https://effyfan.com/2018/03/02/w6-van-gogh-flowfield/

As well as this does work, it is very much computationally expensive as most samples will not fit the image, thus searching for the small set of fitting shapes requires to evaluate all the wrong shapes as well. Considering artworks, brushstrokes have many more degrees of freedom, and artworks usually consist of upwards of a few thousand brushstrokes. Consequently, it would be considerably more challenging to apply to this problem until computational resources have become a few magnitudes more powerful.

This premise can be overcome, though, by using a differentiable renderer. A **differentiable renderer** is capable of creating/rendering shapes in the pixel domain. In contrast to conventional renderers, it does so by solely using differentiable operations. Thus, the previously described task becomes feasible, as random sampling can be replaced by gradient descent. Ordinary renderers usually do not rely on differentiable operations, as faster operations with more straightforward logic render images well enough already.

Nonetheless, it is theoretically possible to create a differentiable renderer [**something**].

When talking about rendering brushstrokes, it would be even harder to think of a differentiable pipeline to draw brushstrokes of reasonable quality from a set of parameters.

Neural networks turn this problem around. As neural networks are inherently differentiable, the question becomes

Figure 6.4: Photo mosaic of Starry Night using only images by the Hubble Space Telescope. http://www.astro.uvic.ca/~alexhp/new/figures/starrynight_HST.001.jpg

rather how to make an existing neural pipeline render images from parameters. Previous works have shown that neural networks are capable of conditionally generating high-resolution and high-quality images. Conditioning the generator on brushstroke parameters as well as some noise should then output an image of the respective brushstroke with some variability to it.

The basic idea was proposed by **japaneseneuralrenderer** [**japaneseneuralrenderer**] as it facilitates training of reinforcement learning based networks.

Inspired by this, the approach becomes more apparent. First, a neural network is trained as a differentiable renderer. Then the same renderer is used by a gradient descent-based optimization procedure to approximate an artwork as a set of renderer input parameters.

Both steps require some tricks to avoid pitfalls like computational limitations, which are outlined in the following two sections.

6.2 Neural Renderer

The neural renderer in **japanese neural renderer**'s [**japanese neural renderer**] work improves an architecture based on SPIRAL [**SPIRAL**]. **Learning2Paint**'s efforts used a differentiable renderer to

move this into the related work section and shorten it further facilitate deep reinforcement learning for drawing images [**Learning2Paint**]. As mentioned, the renderer is required to be differentiable and ideally require as few resources as possible.

Data Set

Unfortunately, there is no data set available for this task, which means that a data set must be created explicitly for this approach.

There are several sources for virtual and real brushstrokes, to choose from. These resources will be evaluated in the following part. The main focus lies on three qualities for each data source:

1. Suitable data format: As brushstrokes usually overlap in a painting, the data should already provide information about opacity or transparency, favorably in the common RGBA format, which has a fourth alpha channel to hold opacity information.
2. Data set size and variability: Only with enough different data available, it will be possible to train a renderer reliably.
3. Image quality. Brushstrokes should be as close to real-world brushstrokes as possible to ensure high-quality renderings later on.

Brushstroke Images

reference this

There are multiple sets of hand-drawn brushstrokes available online. Most notably, there is a set of various well-classified colors and brush styles created by 'zolee' on the platform `onlygfx.com`. It consists of approximately 1000 brushstrokes that mostly follow rather straight horizontal paths. These brushstrokes are mostly grouped by color and painting technique (oil, acrylic, watercolor...). All images are in the PNG format with the background made transparent in a post-editing step.

This data set has the advantage that it consists of real-world brushstrokes that were painted under presumably reproducible conditions. On the other side, brushstrokes are of mostly the same width throughout the data set and also do not come with information which path the brush took or any other non-visual information. Also, the data is very sparse. Many color shades are not represented, which means that the generator would have to interpolate them or simply would not be capable of rendering any brushstrokes in this color.

It seems that this data set would be suitable to replicate single real-world brushstrokes as images. However, limitations to the data make it unlikely that a generator could learn a coherent representation from this.

Painting Libraries

The mentioned work of SPIRAL [**SPIRAL**] relies on synthetic data instead of real-world images. It used the painting library 'libmypaint' [**libmypaint**] to generate brushstrokes from parameters in real-time during training.

The apparent advantage of this and other painting libraries is the fact that one can fully control the output through parameters. As the whole space of input parameters for the renderer can be covered, it is much easier to avoid pitfalls like they were described in Section **??**.

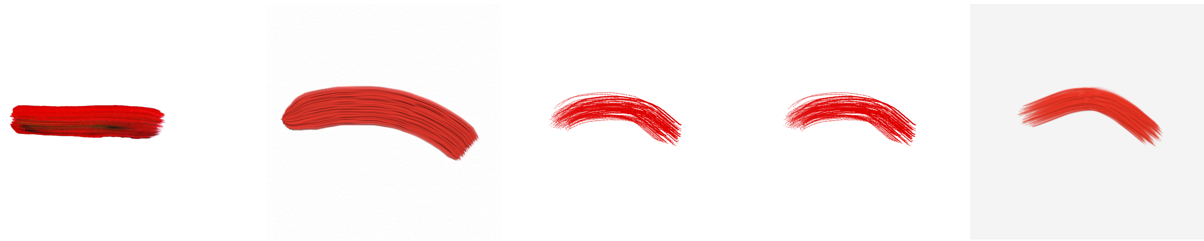
Still, this data set falls short regarding the authenticity of rendered strokes. Especially the inner area of the stroke shows a uniform color, which is far from what real brushstrokes would look like.

This data set is better suited for our task than the images from which were described in the previous seciton are. However, the synthetic data will tend to make all rendering look a bit 'cartoonish' or flat, which could, in turn, limit convergence during the latter optimization process.

Fluid Simulation

Fluid Paint is a project by David Li [**fluidpaint**] that uses simple fluid dynamics to give artificial brushstrokes a more plastic look. It has been implemented in JavaScript and OpenGL.

There is a C++-version in the git-repository of SPIRAL along with Python bindings created by Yaroslav Ganin.



(a) Photograph **(b)** ArtRage **(c)** Affinity Photo **(d)** libmypaint **(e)** FluidPaint

Figure 6.5: Comparison of similar brushstrokes in each data set

Using these Python bindings, it possible to generate brushstrokes locally outside of a web browser environment.

The quality and controllability of FluidPaint fall right in the middle of the two previously mentioned data sets. The generated brushstrokes look distinctively better than those generated with libmypaint, but still lack the quality of the real-world images. Concerning controllability, FluidPaint allows controlling the path of the paintbrush's handle rather than the path of the brushstroke itself. As a result, there occur some offsets to a given path as opposed to libmypaint.

As a reasonable compromise between controllability, variability, and render quality, FluidPaint emerges as a sensible choice. Although real-time data generation is not possible with this library, data generation can be parallelized. Such a set-up still allows for the creation of large data sets in a reasonable time frame.

Even though this data set still has some weaknesses, it is probably the best choice for training a differentiable renderer because of the noted reasons.

these two weird software stuff things

Other 'honorable mentions' are painting programs such as , which allow for even more authentic brushstrokes but lack any well-documented interface in order to generate a vast number of brushstrokes.

talk about RGBA advantages

Brushstroke Formalism

cut this joke

With the means of data set production seized , what is left is to formulate the parameters that define the brushstrokes. These parameters must quantify the following three properties of brushstrokes:

- color
- thickness
- path

The easiest of these three properties is quantifying the color. Naturally, computer vision relies on the RGB format, which defines color as a set of three 8-Bit integer values between 0 and 255. As for path and thickness, these two properties depend on the given coordinate system. FluidPaint represents the canvas as a 2D plane in the $[0, 1]$ range. Thus, it makes sense to follow the same representation.

Thickness can be any value in $[0, 1]$ for each brushstroke, where 0 is an infinitely small brushstroke, and 1 is a brushstroke as wide as the canvas. As both these edge cases do not make sense in this application, the range is constrained to $[.03, .2]$, which includes only brushstrokes that are visible and also do not cover the whole canvas.

Quantifying the path now is a little more tricky. The fluid dynamics simulation that FluidPaint uses relies on internal time steps at which the equations are evaluated and subsequently rendered. At the same time, each step allows only a linear motion of the brush handle between positions a and b. Subsequently, any curved paths must be split into linear/straight segments that together should resemble a curved line. As more steps mean longer simulation times and fewer steps mean edgy movement, a value of 20 time-steps per stroke emerged as a good compromise . The same number of steps has been used in SPIRAL's implementation.

this is actually different as the steps depend on the length of the path. Look this up

Another problem becomes how to express a curved path in numbers. The most straightforward representation would be a sequence of points that make up any curved path. Such an approach allows for the highest versatility, but at the same time introduces a noticeable amount of parameters as each point consists of 2 coordinates totaling to 40 values. These values are also not independent of one another but should follow a reasonable path as otherwise, the resulting brushstroke would look somewhat like a random walk than an actual brushstroke. Since works such as SPIRAL or 'Learning to Paint' face a quite similar obstacle, their solution should be applicable in this case as well. Both used so-called 'Bezier curves' which parametrize curved paths by a limited set of numbers.

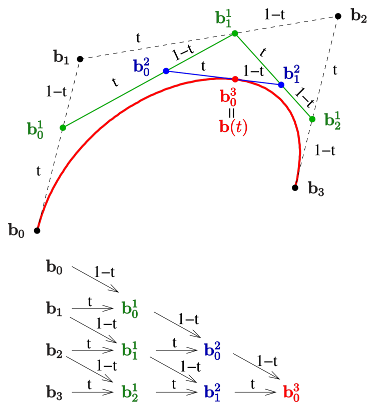


Figure 6.6: Sample of a 3rd degree Bezier curve, using the De-Casteljau-algorithm, https:\/\/de.wikipedia.org\/wiki\/Bézierkurve#\/media\/Datei:Bezier-cast-3.svg

bezier curves margin note

Bezier curves parametrize curved paths not as linear segments, but as analytical paths which can be used in computer models. They can be of different orders, which allows them to follow more complicated paths. In this case, the simplest form – the first order Bezier curve– is already sufficient. It defines a curve through its start- and endpoint, as well as a control point. The curve is then defined as the path of a point over the time interval $[0, T]$. First, one connects the start- and endpoints with the control point to get two lines in return. On these lines, lie two points that move linearly along their lines within a virtual time interval $[0, T]$. Then these two points are connected in the same way as before by a third line. Again, this line has a point moving along its path throughout $[0, T]$. As the first two points that define the third line will move, the line's orientation will change as well, thus translating the linear movement of the point into a complex curved path. The path that this point then takes in $[0, T]$ defines the Bezier curve. A first-order Bezier curve will only bend into one direction or follow a straight path. For higher orders, the displayed process can be applied iteratively and allows for more complex curves. As brushstrokes usually follow a quite simple path and fewer parameters are preferred, Bezier curves of first-order are suitable as parametrization.

Ultimately, this gives ten values that are sufficient to parametrize brushstrokes with certain constraints:

- Three 2D coordinates that define the Bezier curves (6 values).
- One thickness parameter.
- Three values in RGB space.

Data Constraints

Given the parameters listed in section 6.2, the data still needs further constraints to facilitate the generator's training even further.

Section 6.2 already hinted at the impracticality of online data generation. A rough estimation by timing the rendering of 100.000 FluidPaint brushstrokes reveals that a dedicated CPU server is capable of generating 300 strokes per second. A neural network with batch size 32 is limited to ≈ 10 iterations per second under these circumstances, which would mean a clear bottleneck. Thus, it seems advisable to generate data

experiment in appendix

beforehand with enough samples to cover the data space sufficiently. It will allow for much faster access to data, as individual data samples are relatively small and can be stored in a binary data file such as HDF5.

Besides this constraint to the amount of data available, another set of constraints will be introduced to reduce the data space to 'valid' brushstrokes only. 'Valid' brushstrokes will be defined as brushstrokes that resemble real-world brushstrokes to a certain degree. This primarily concerns two relations within a brushstroke:

- Its width-to-length ratio.
- Its curvature.

The width-to-length ratio will be restricted to brushstrokes that are at least two times as long as they are wide.

$$\|\mathbf{s} - \mathbf{e}\| \overset{!}{\leq} 2 \times (\text{brush size}) \tag{6.1}$$

Due to the simulation background of FluidPaint shorter brushstrokes will show some artifacts due to the bristles' length in the simulation which depends on the width of the stroke. Another reason for this is the intended use-case, which will focus on van Gogh paintings. As van Gogh did not practice pointillism, most of his strokes have a length to them, which brings such a constraint in line with some characteristics of van Gogh's style.

The same argumentation applies to the curvature: Most brushstrokes (especially those by van Gogh) have a certain 'flow' or 'smoothness' to them, which can be described by using strokes with large curvature radii and without any corners in the strokes' path. Thus, the data set will also be restricted to strokes that follow these descriptions. In order to achieve this with random sampling in mind, a multivariate Gaussian distribution is placed between start ($\mathbf{s}$) and end point ($\mathbf{e}$). The two axes are rotated such that the short axis is in line with the vector $\mathbf{a} = \mathbf{s} - \mathbf{e}$ while the other sits orthogonal. Then both axes are scaled with $\|a\|_2$ and also the handpicked values $\frac{1}{200}$ and $\frac{1}{25}$ for along a and orthogonal to it, respectively. Figure **??** shows samples from this distribution for an exemplary brushstroke.

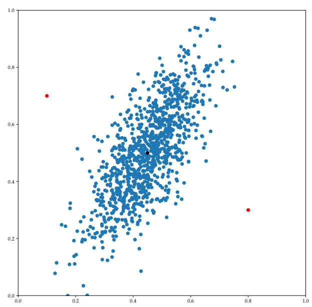


Figure 6.7: Exemplary scatter plot for given start and end point to visualize the covariance matrix

This distribution is intended to follow that of brushstrokes as they would appear in the real world. The majority of brushstrokes will be straight or just slightly bent due to the maximum of the PDF being at the center of s and e. Bent brushstrokes will mostly be symmetric as the long axis of the multivariate Gaussian is orthogonal to a. Still, there will be strokes that have their bent towards either end of the brushstroke as well as some strokes with a high curvature. The area of interest, though, will be densely populated as intended.

$$p(\mathbf{c}|\mathbf{s},\mathbf{e}) = \mathcal{N}(\mu,\Sigma) \tag{6.2}$$

$$\mu = \frac{\mathbf{s}+\mathbf{e}}{2} \tag{6.3}$$

$$\Sigma = \begin{pmatrix} a_x & 0 \\ 0 & a_y \end{pmatrix} \tag{6.4}$$

$$\mathbf{a} = \mathbf{s} - \mathbf{e} \tag{6.5}$$

The color of the brushstrokes is not constrained as the color distribution of the target data set is not known at this point.

why not van Gogh color distribution?

Data Set Creation

The data set will be created with 100.000 samples that follow the constraints that were presented in section 6.2. As an underlying distribution, the uniform distribution is chosen as it allows a more evenly coverage of the data space.

First, a set of start and end points, as well as brush size, is drawn and checked against (6.1). If the constraint is not met, the set will be redrawn entirely. In case the constraint is satisfied, a checkpoint is sampled according to (6.2). If **c** lies outside the render window, the checkpoint will be resampled. At last, an RGB set is sampled from a uniform distribution as a color.

The resulting tuple of start, end and control point, brush size, and RGB color is then added to the data set. Before rendering starts, the values of **s**, **e** and **c** are scaled with the handpicked factor of 0.7 to ensure the brushstrokes are rendered completely within the window and not cut by an edge of the render window. At last, the brushstrokes are rendered according to the data set and added as well.

The render canvas size was chosen to be 64x64 pixels for several reasons: First, even with such a small canvas size, training for the renderer takes about one day. Secondly, the larger the render canvas size becomes, the deeper the renderer needs to be, which results in more computational overhead in the optimization routine as well as more layer through which the gradient has to be propagated. Lastly, as there will be upwards of a thousand brushstrokes in a single image, increasing the canvas size to 128x128 would require four times as much memory per rendered image. As 1000 brushstrokes would already account for $1000x64x64x4\text{B} \approx$ 16.4GiB a fourfold increase would be significant.

As the last step, the data set is renormalized to the range $[-1, 1]$ for convenience and to facilitate training as well.

explain why zero-centered data is good

Architecture

The architecture of the brushstroke generator follows that of an inverse VGG network. It is widely used and has shown in previous works that it should be capable of handling this task. The architecture consists of three dense layers at the beginning, followed by a two-times upsampling layer as well as three convolutional layers. The same pipeline with a two-times upsampling layer, and three convolutional layers is repeated until the target size is reached. After the last convolutional layer, a hyperbolic tangent function is applied to restrict the output to the $[-1, 1]$ range. As part of the hyper-parameter search, different tweaks to the architecture have been tested:

check the details

- An additional noise input at every layer with a size equal to that of the existing signal.
- Additional information about the position in the pixel grid in every layer, so-called CoordConv [**coordconv**].
- Various combinations of activation and normalization functions.

The discriminator is designed after the same principles and resembles a VGG encoder network. First, three convolutional layers are applied, followed by a downsampling /pooling layer. This structure is repeated until a target resolution of 4x4 pixels is reached. Then a set of three dense layers is applied to give one final prediction per sample.

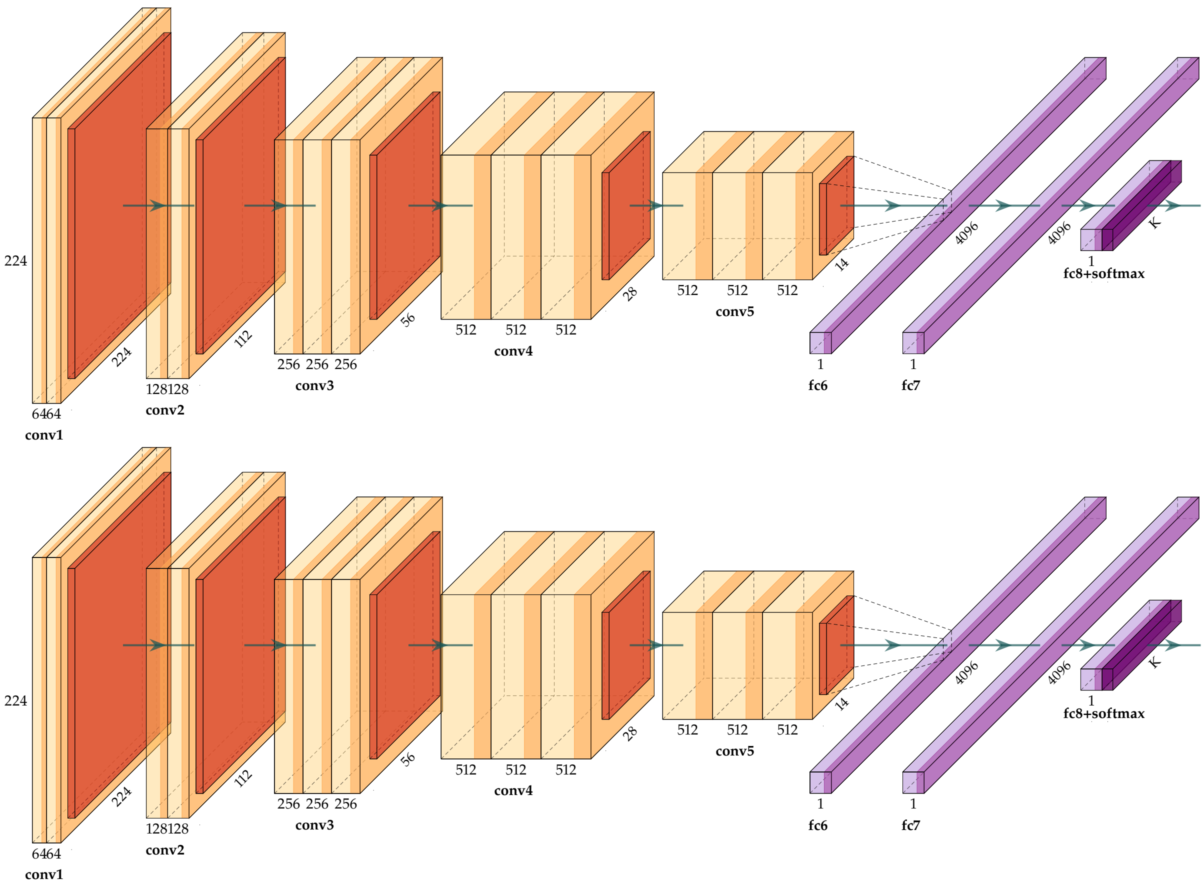


Figure 6.8: Visualization of the generator and discriminator architectures

Training

During training, the L2 distance and the FID score serve as evaluation metrics. The FID score becomes necessary as the visual comparison of the generated samples has proven difficult between different experimental runs. The L2 distance does not qualify as a sufficient metric for later training stages as the stochastic nature of the brushstrokes puts a lowerbound on the L2-distance.

A two-time-step update rule was implemented to stabilize training further.

write this down and look at the tricks that were used

Results

pick some results and present whether they are any good

6.3 Stroke Approximation

Data Set

The data set will be presented in this section due to the data defining demands for later networks. Also, the data set for the optimization task should meet a few requirements.

In order to focus on the brushstrokes in an image, the data set should consist of relatively high-resolution images. At the same time, most brushstrokes should fit into a 64x64 window. Any larger strokes can not be approximated with a single rendering window and thus would falsely be approximated by multiple brushstrokes.

Thus, more information per image is required, than merely the resolution. The scale of the image, therefore, becomes inquired information in this task. Ideally, each image should be accompanied by its measurement as this allows for rescaling the image accordingly, given that one knows how large a typical brushstroke is.

At last, the painting's technique must be oil on canvas or similar techniques as this is what the renderer has been trained for.

All these requirements on the data are met by data directly available from the Van Gogh Museum in Amsterdam. Each high-resolution image is categorized by its technique as well as the period in which it was painted. This is accompanied by information on the dimensions of the image.

Optimization Algorithm

The next stage of the approach takes the pre-trained neural renderer and uses it to approximate brushstrokes in images. In theory, the neural renderer should allow gathering meaningful gradients in parameter space even though the losses are calculated in image space, as explained in section 6.1. There, it was outlined already that the method of choice will be an optimization procedure that relies on the differentiability of the neural renderer.

Nevertheless, this is not the only possible approach to dissect images into sets of parameters. As explained in 5.4, there exist a variety of approaches for this kind of task. Some of these promise a fast inference of parameters from images, which makes the optimization approach in this work seem like a step back at first. The following section is dedicated to justifying the chosen approach by comparing the existing state-of-the-art approaches.

The targeted task is to approximate the representation of an image with roughly 1,000,000 pixels = 1MP by means of $\approx$ 10.000 brushstrokes (see section **??**).

Genetic Algorithms

put much of this into the related work section

Genetic algorithms are possibly the simplest approach. As it was laid out in sections 6.1 and **??**, genetic algorithms use random sampling and previous best results to approximate fitting solutions to the rendering problem, which was described in section **??**.

The current state-of-the-art solutions are capable of finding a solution in $\approx 1h$ when searching for an approximation of a 1MP image with simple geometric shapes like circles or triangles. This corresponds to roughly 1×10^9 sampling steps. The time frame and the number of sampling steps depend on target shape density, target accuracy, sampling rate per shape, and degrees of freedom (with the latter requiring more sampling).

Looking back at section **??**, each brushstroke has 10 degrees of freedom with some inter-dependencies between them. Also it is known that the obtained neural renderer is capable of generating $\approx 300\frac{\text{images}}{s}$. This means that rendering 10^9 sampled brushstrokes will take:

$$\frac{10^9\text{samples}}{300\frac{\text{images}}{s}} \approx 3.33 \times 10^6 s \approx 926h \approx 38.6d$$

Clearly, this is an absurd amount of time to spend **per image**. Furthermore, the sampling magnitude has not even been corrected for by the additional degrees of freedom that brushstrokes provide.

Ultimately, this means that genetic algorithms are not an option for this task, as they are too inefficient.

Brushstroke Extraction

Next in the line are algorithm-based approaches that extract brushstrokes from an image using standard computer vision techniques.

put much of this into the related work section

Besides texton based image characterization [**textons**] and pure filter based approaches [**filters**] there have been approaches to extract brushstrokes or some of their characteristics [**brushstrokecharacteristics**] [**brushstrokeextraction**].

The latter would pose a valid option for the task at hand. Unfortunately, the best existing techniques fail to detect brushstrokes reliably over the whole image but only identify the most significant ones (see figure **??**) /todoadd example image. Other approaches are only able to extract a few characteristics, like the orientation of a brushstroke, which proves insufficient as well.

Other approaches will usually not get parameters but again pixels, which they analyze.

show some of the extracted brushstrokes

Stroke Based Rendering

Stroke Based Rendering or Painterly Rendering also uses algorithms to approximate an image through brushstrokes with the original goal to achieve some stylization along the way. While early works relied on interactive approaches, later publications were then able to automate this process entirely. Judging from tests by the authors, such an approach would take between and hours per image. As figures **??** and **??** show, these approaches obtain similar results to genetic algorithms and tend to draw an image from a coarse scale to a more delicate scale over time instead of a locally coherent manner.

this number

this number

These approaches were not intended to be used to obtain brushstrokes from a painting but focus on stylizing images. This renders such an approach unfit for the goal of this thesis as well.

Drawing Networks

Lastly, drawing networks are the newest iteration of approaches in this field. Beginning with , there have been approaches that make use of feed-forward or recurrent neural networks combined with either supervised training or deep

put much of this into the related work section

which one was first?

reinforcement learning. The best results of these approaches are shown in figure **??**.

Noticeably, all of these images have a maximum resolution of 256 pixels along their longest edge. There have not been any approaches yet, which can go significantly beyond this limitation. Notably, the high computational costs of training recurrent neural networks seem to be an obstacle when shooting for higher resolutions. Such resolutions can not be deemed sufficient when looking at individual brushstrokes, as outlined in 6.3.

Consequently, drawing networks must also be ruled out.

Combined Approach

Even though there have been plenty of previous approaches to the task of extracting brushstrokes from images or similar tasks like rendering images through brushstrokes, none of these quite meet all the requirements of this task. Namely, high-resolution input images, brushstroke focussed rendering or retrieval, limited resources, and realistic depiction of brushstrokes.

Although brushstroke extraction is the topic of this chapter, stroke based rendering and drawing networks both seem capable of making brushstroke extraction more feasible with their inverse approach. Drawing networks introduced differentiable renderers as a tool to facilitate training. Stroke based renderers achieve a parametrization implicitly by focussing on replicating the entire image instead of extracting individual brushstrokes.

They can be combined into a unified approach that uses the differentiable renderer and the objective of recreating an entire image. The resulting approach does not use re-sampling like genetic algorithms but relies on gradient descent to converge to a solution significantly faster.

The limited capabilities of such a renderer would also guarantee that any approximation is composed of only valid brushstrokes instead of single pixels as it would be the case with normal generative models.

All in all, this approach is probably the best suited for approximating a target image only through brushstrokes as closely as possible.

The decision of whether to use an optimization-based approach is also linked to another question: Whether to sequentially or parallelly place brushstrokes.

As an example, most drawing networks and stroke-based renderers rely on a sequential approach. Intuitively, it makes sense to use a sequential approach as artists also place their brushstrokes sequentially on the canvas. However, there is a significant difference in how existing computer vision approaches place strokes compared to artists. As neural networks search for ways to reduce the loss function as much as possible and as fast as possible, images tend to be made up of large canvas-filling brushstrokes at early stages, which in turn reduce the L2 loss.

This is contrary to how an artist usually works. Artists tend to fill the background based on content (*e.g.* sea or sky is often painted first) and will often leave some blank canvas to paint foreground objects later on. They also do not use giant brushes for this but normal-sized brushes with which they place many strokes. Thus, sequential approaches differ significantly from real-world paint processes.

In contrast, parallel approaches predict the whole painting as one set of parameters. They are not as widespread due to the computational pitfalls that come with predicting brushstrokes for a whole painting at once compared to predicting only a few at a time. Also, parallel approaches still need to predict in which order brushstrokes should be placed on the canvas (this will be explained in section 6.3) such that brush strokes can overlap. Nonetheless, the advantage to sequential approaches is the fact, that foreground an background strokes can influence one another.

An example would be a stroke in the foreground, changing its path and revealing the canvas beneath. Then another stroke in the background should cover this up if the color matches. For sequential approaches, the background stroke can not be changed afterward, and it is very complicated to formulate a loss that propagates this information. As artists already plan their future brushstrokes when painting the background, it would require a drawing network to plan far ahead. At the same time, an optimization-based approach will not need to do that.

Another argument for parallel optimization is the focus on actually visible brushstrokes. As artists cover up previous strokes, again and again, it becomes a shot in the dark to guess what these first brush strokes might have looked like. Subsequently, any such guess is ill-posed and only introduces noise to the problem. A parallel approach does not care for how the background could have been drawn but only focus on what is visible in the given image.

Pitfalls of Feed-Forward Approaches

In the development of this thesis, there were experiments targeting a feed-forward approach before ultimately deciding on an optimization-based approach that is now presented. As it would exceed the scope of this thesis, the arising problems and cause for discarding this approach shall be discussed.

margin explain cmnist

add image that were drawn by ff network

First, the computational burden of a feed-forward approach is very high. Existing feed-forward drawing networks compromise image resolution to realize their implementation. As it was possible to implement this for small scale data like the cMNIST data set, such an approach seems feasible at first. However, since compromising image resolution is not a viable option (see Section **??**, one must find ways around this problem. Fully convolutional architectures are a popular solution, which allows training on small scale data and inference on large scale images. Unfortunately, the problem of predicting many spatially brushstrokes with complex parameters has proven too difficult for fully convolutional approaches.

This goes hand in hand with another problem that occurred: the placement of brushstrokes on the canvas. As artists are not bound to the same pixel grid as computers typically are, they can place brushstrokes freely on the canvas. More so, they can pack brushstrokes densely in one area while distributing them broadly in another. Classic CNNs are not able to allow for similar behavior as they always require a grid layout. Experiments with either displaceable grid cells or stacked grids have proven did work on a small scale. However, together with a fully convolutional architecture, the approach did not seem to scale to larger images.

Thus, an optimization-based approach became favorable as it offers good approximations at high resolutions with manageable computational overhead.

Optimization Procedure

The previous section **??** already specified why an optimization-based approach is preferable over a feed-forward or recurrent approach. This section aims to give more details about the optimization procedure.

Rendering Layout

Fundamentally, the optimization procedure is inspired by stroke-based rendering procedures. It could also be compared to the style transfer approach by ~gatys [3], with parameters optimized instead of pixels. The difference to normal stroke-based renderers, though, is the limited size of a rendered brushstroke in this work's renderer.

This poses a significant challenge that might not be obvious at first.

Ideally, the optimization procedure should be able to place strokes freely on the canvas, as this allows for an unbiased approximation. Furthermore, it would allow allocating many small strokes in areas where the artist placed many strokes and use fewer and wider strokes in other areas. However, due to various limitations, which were explained in **??**, the renderer is not able to render single brushstrokes in a 1MP frame. Similar approaches only perform on relatively small canvas sizes, likely due to this issue .

add ref

Dissecting the target image into many small patches and then running the optimization procedure on these individual patches is a workaround for this. After the patches are rendered, they are then fused along their edges to give the full image. A good comparison is a grid of renders where each grid cell is the center of many renderings at the same time . The major issue of such an approach is the adjacent edges where the grid cells are joined. Also, a grid structure will almost always differ from the inherent distribution of brushstrokes in an image; grid edges will, more often than not, separate brushstrokes between two grid cells. A simple solution to this problem is transitioning from a stacked grid structure to an overlapping grid.

add figure to this

graphics!!!!!

By overlapping the grid cell, obvious edges between render windows are hidden as every edge lies within the frame of a different render window. Choosing a lattice vector size smaller than the render window's dimensions such a grid can easily be realized.

Still, this kind of initialization requires a very even distribution of brushstrokes, ideally with a brushstroke at the center of each cell. As this is not the case, and stroke densities will vary locally, the grid layout is prone to erroneously enforce a grid-like layout of strokes where there is none. This is due to the inability of the grid to account for local changes in density and the following propagation of error. Starting with a single region of high-density strokes in the vicinity of one grid cell, this cell would ideally render a narrow stroke to achieve high accuracy. Neighboring cells then have to shift their strokes towards the center of that grid cell to account for the free space that is not covered by the narrow brushstrokes. This shift must then be accounted for by the next neighboring cells and so on, which will cause all strokes in a row or column to shift towards this one spot with a high stroke density.

Now, a painting usually has many such high-density regions, which possibly cancel the shift that is caused by other areas. As a result, the renderings are likely to not shift at all, as shifting mostly cancels out over the whole image. Subsequently, an area of high stroke density will not have enough strokes available in its local region and thus will be covered up by a single broad stroke as this minimizes the L2 loss.

The core of the problem is the previously imposed lattice structure that propagates local density shifts along its principal axes.

One possible solution is getting rid of the lattice structure and replacing it with a more random structure that also covers the image sufficiently. This can be accomplished by using **superpixels** [**superpixels**]. Superpixels were popular in the pre-neural network era of computer vision and were often used in image segmentation tasks ([**img segmentation with SP**]). Nevertheless, superpixels are also a popular starting point for brushstroke extraction algorithms [**brushs stroke extraction**].

Basically, superpixels are pairwise disjoint groups of pixels in an image that would usually join pixels with similar colors in a local region. Straight away, it is obvious why this is interesting for brushstroke extraction. The distribution of superpixels will not follow a grid-layout as the previous approach, and the location of superpixels should relate to the given color distribution in the image. It is easy to imagine that the location of the superpixel centers would be a good prior for locations of render windows as well. Also, as the colors of pixels inside a superpixel should be similar, one can use the mean color of a superpixel as an initialization for the color of the brushstroke.

Ultimately, a superpixel segmentation will be used to infer positions for render windows as well as the color initialization of each stroke.

Rendering Order

Another problem that will come up during the optimization procedure is the order in which strokes are rendered (already mentioned in Section **??**). Real-world brushstrokes are also subject to the same issue as the current brushstroke will always be placed on top of brushstrokes painted earlier, (see section 6.3).

As the optimization-based approach relies on parallel optimization of brushstrokes, it must decide which strokes are in the foreground and which are in the background. Otherwise, as this would randomly change, edges in the image might be obstructed, and optimization could oscillate between solutions where different strokes lie in the foreground. It could also prevent brushstrokes from overlapping such that they cover disjoint areas. All of these outcomes would be unfavorable as it tends to produce worse results in the end.

The solution which is presented in this thesis is an additional parameter that describes a brushstroke's accuracy. The accuracy is defined as the L2 distance of each stroke's pixel to the corresponding pixel in the target image multiplied by this each pixel's alpha value. This removes any pixels which are of no interest from the loss and focusses only on the rendered pixels.

$$\text{accuracy} = 1 - \frac{1}{N} \sum_{p \in \text{pixels}} \|p - p'\|_2^2 \text{ with } \mathbf{p} = \mathbf{p'} \qquad (6.6)$$

improve this, especially the p=p' part

The resulting value describes how well the pixels of the rendered stroke match their respective pixels in the target image. Consequently, any brushstroke with higher accuracy will be more faithful to the target image than strokes with lower accuracy. Placing these brushstrokes in the foreground should thus result in a smaller L2 loss than the other way around. Vice versa, brushstrokes that connect two same colored areas will aggregate a lower accuracy as the brushstroke is compared at the intersection as well. A rendered brushstroke that fits the foreground brushstroke will not be affected by this, thus keeping its high accuracy and laying on top of the other brush stroke.

Notably, the accuracy should not be included in the brushstroke's loss, as this would prohibit background strokes from covering larger areas and result in behavior that is similar to the non-overlapping issue previously described. Thus, the accuracy of each stroke will be calculated as it is rendered.

Initilization

The following section will focus on initialization details for all parameters of a brushstroke, their position, and the confidence value.

Besides the original ten parameters of each brushstroke, which were explained in section 6.2, the previous two sections introduced an accuracy parameter for ordering and two translation parameters that define the position of the render window along each axis. All of these parameters must be initialized before the optimization procedure starts. Ideally, the initialization should not introduce any bias to the optimization process. At the same time, an initialization should facilitate training and accelerate convergence in the early stages of optimization.

Unfortunately, the placement of the render window will surely enforce a bias on the optimization (see section 6.3). Thus, a superpixel initialization was motivated for the translation parameters as well as the color of the brushstrokes. Subsequently, the translation parameter for each render window will be equal to the position of the weighted mass center of its respective superpixel.

The initial color will be taken from the mean color value of the superpixel.

The brush size will be initialized with the minimum possible value. This will let brushstrokes not overlap at the beginning of optimization. Only when the brushstrokes already roughly fit their local region, they shall intersect and be ordered by their accuracy. Therefore, the initial accuracy will be 0 everywhere, as the accuracy is recalculated after every optimization step.

Other patch parameters, notably **s**, **e** and **c**, will be initialised using a narrow Gaussian distribution with $\sigma = .1$ and values clipped to $[-1, 1]$ because there is no prior information available on how the brushstrokes are oriented. Instead, this approach relies on the optimization procedure to be minimally biased by this initialization of the path variables.

Partial Updates

When building an optimizer based on the information provided until now, GPU memory limitations become a guaranteed issue. Even as it might not be evident at first, the optimization procedure imposes a considerable requirement for memory on the graphics card, due to two parts of the training:

First, the number of brushstrokes can easily become very high with large images as input. With a render window size of 64x64, the brushstrokes are relatively small, and paintings can consist of a few thousand brushstrokes. This would equate to a batch size of a few thousand for the brushstroke renderer. Tests have shown that on the latest hardware with 12GB memory, the maximum number of brushstrokes rendered in parallel is ≈ 256. Obviously, this is one to two orders of magnitude smaller than what would be needed to optimize all strokes in the painting in parallel. Still, there is a way around this bottleneck, by optimizing the image not as a whole but as smaller patches consisting of 256 brushstrokes at a time, giving a partial update routine.

Each patch comprises the 256 nearest render windows to a randomly sampled location on the canvas. These 256 brushstrokes are then rendered from their parameters in order to obtain a gradient later on. Then strokes are ordered according to their accuracy, placed on canvas ('padded') and blended ('stitched'). At last, the loss is calculated and back-propagated to update the parameters of the patch.

When restricting the number of brush strokes, bordercases do become an issue. Brushstrokes that lie at the perimeter of the patch are not fully surrounded by other patches. The result is a discrepancy of what the gradient to the brushstroke would look like if the brushstroke had laid in the middle of the patch. A solution to this is laying a ring of already rendered brushstrokes around the patch. The ring guarantees that all brush strokes that brushstrokes at the border of the patch are surrounded by neighboring brushstrokes the same way, as if they were in the middle of the patch. Notably, the brushstrokes that belong to the ring around the patch, serve as 'dummy' brushstrokes and are optimized. Without any need for optimization, these brushstrokes do not need to be rendered in this optimization step. Instead, they could have been rendered earlier and the rendered image is just reused. For this reason rendered brushstrokes for all parameters are saved in a **render image catalog**. Equally, the collection of brushstrokes that shall later compose the image shall be called a **parameter catalog**. The parameters as well as the images in these catalogs are updated whenever the respective brushstroke parameter has been updated. With the render image catalog at hand, it is possible to use the previously rendered brushstrokes to stitch the image as a whole with the newly rendered brushstrokes of the image patch embedded.

As this will cause the border strokes to be surrounded by other brushstrokes to all sides (even if not all of them are freshly rendered), the effect of the partial update routine vanishes.

The other problem for huge input images is that the stitching of brushstrokes takes up a considerable amount of memory. In detail, each brushstroke must be placed on the virtual canvas individually, where the canvas' size is that of the input image. This would equate to a couple of thousand 1MP images being stored in memory before they are stitched to a single 1MP images. As a single 1MP image carries roughly $1000 \times 1000 \times 4\text{channels} \times 8Bit = 32 \times 10^6 Bit = 4 \times 10^6 Byte = 4MB$ of information, a few thousand of these will easily exceed the memory of most graphics cards.

Images atypically saved as unsigned 8-Bit integers, thus 8-Bit per channel.

Luckily, the previous workaround for optimizing only 256 brushstrokes will work as well without rendering the full image at every step. Since most brushstrokes are not re-rendered and thus will not be supplied with a gradient, their main task is to regulate losses for edge strokes of the image patch. As this does not need far away strokes, but only those close to the optimized strokes, a ring of pre-rendered strokes around the re-rendered patch will suffice.

This reduces the number of involved brushstrokes per optimization step from a few thousand down to a couple hundred. Besides allowing for the partial update routine to be performed at all, it should also increase performance significantly compared to an approach that involves all brushstrokes.

approximate the number of brushstrokes needed to surround the patch

Image Placing & Blending

As the update and optimization procedure has now been explained thoroughly, it is now time to explain the process of placing and blending a rendered brushstroke a bit further.

After each stroke has been rendered, it needs to be placed according to the translation parameters (see 6.3). This requires dynamically placing each brushstroke inside a zero-filled tensor.

By calculating each pixel's global positioning in the rendered image individually, it is possible to scatter the pixels of the original rendered image into the larger zero-filled tensor and obtain a globally placed brushstroke.

The task of blending the resulting canvas-sized rendered brushstrokes together, or stitching them, is more complicated, though. Due to the canvas' alpha channel, it is possible to blend only relevant information while the rest of the image will be ignored.

As far as conventional alpha blending goes, two images are blended by multiplying each pixel value with the alpha value of the top-layer image while the background image is multiplied with the complement to the alpha value:

$$p_{x,y} = p_{x,y}^{\text{top}} \times \alpha_{x,y}^{top} + p_{x,y}^{\text{bottom}} \times (1 - \alpha_{x,y}^{top}) \forall (x, y) \in \mathcal{D}(\text{image}) \tag{6.7}$$

correct this equation and make it nicer

For multiple layers, this process can be repeated in various fashions, after the strokes are ordered according to their accuracy. Either one could start from the bottom and blend the two back-most strokes, followed by the next third last strokes and so one, or one could start this process from the front with the two strokes in the very front being blended at first, then the stroke with the third-highest accuracy *etc.*.

Both of the approaches would yield the same result but differ only in the order in which they were blended. Subsequently, both methods will have $(n-1)$ blending operations to compute per pixel.

Blending brushstrokes in a position-aware manner can reduce this number. The majority of pixels for each padded brushstroke is non-informative, as the alpha value is zero (see Figure **??**). This opens the possibility of go from blending *all* pixels of *all* brush strokes to blending just those pixels with non-zero alpha values. As before there were many layers that represented one single brush stroke each, few layers that do not represent brush strokes can achieve the same result. Instead of representing brush strokes, these few layers represent the order in which pixels should be blended. This means going from a layer-focussed approach to a rather pixel-focussed approach to alpha-blending.

The easiest way to accomplish this, is to first find the maximum blending-depth over all pixels, where the blending-depth k is the number of layers where the alpha value is not zero.

$$k = \underset{p \in \text{pixels}}{\arg\max} \sum_{i \in \#\text{layers}} \mathbb{1}(\alpha_i > 0) \tag{6.8}$$

Then the top k layer indices for each pixel are picked, which reduces the number of blending operations from $(n-1)$ to $(k-1)$. Importantly, the top k indices should not be ordered by their alpha values but remain in the order that was imposed by sorting according to the accuracy value. Otherwise, the order will most certainly be mixed up, and the pixel with the highest alpha value will always lie on top instead of the pixel that belongs to the most accurate brushstroke. Especially, as brushstroke renderings fade out towards their edges, this makes a significant difference.

Another way of accelerating the process of alpha-blending is **vectorizing**. Instead of iteratively applying the computations, a vectorized operation can perform these computations in parallel. Vectorizing makes it necessary to construct a tensor with the following properties:

For $\mathbf{I} \in [0,1]^{H\times W\times 4}$ the image target, the shape will be defined as $\mathcal{S}(\mathbf{I}) = (H, W, 4)$. Each alpha channel will have the values $\alpha^{hw} \in [0,1]$ for $h = 0, ..., H$ and $w = 0, ..., W$.

The set of rendered and padded brushstrokes $\mathbf{J}$ will have the shape $(N, H, W, 4)$ with N depicting the number of brushstrokes that ought to be stitched simultaneously.

Now, looking at each individual pixel in $\mathbf{J}$, which is described by $(z_n^{hw}, \alpha_n^{hw})$ for $n = 1, ..., N$ and $z_n^{hw} \in [0,1]^3$, z^{hw} describes the RGB values and α^{hw} the alpha-channel for a pixel at (h, w).

A blending operation can then be defined by

$$z'^{hw} = \tilde{\alpha}^{hw} \cdot z^{hw} \tag{6.9}$$

$$\text{or} \tag{6.10}$$

$$z'^{hw} = \sum_{n=1}^{N} \tilde{\alpha}_n^{hw} z_n^{hw} \tag{6.11}$$

$$\tag{6.12}$$

with z'^{hw} the resulting RGB values of the blended pixel and $\tilde{\alpha}^{hw}$ a vector that holds the merged alpha values for each pixel:

$$\tilde{\alpha}^{hw} = \begin{pmatrix} \alpha_1^{hw} & & \\ \alpha_2^{hw} & (1-\alpha_1^{hw}) & \\ \alpha_3^{hw} & (1-\alpha_2^{hw}) & (1-\alpha_1^{hw}) \\ \vdots & & \end{pmatrix} \tag{6.13}$$

$$= \alpha^{hw} \odot \begin{pmatrix} 1 & \\ (1-\alpha_1^{hw}) & \\ (1-\alpha_2^{hw}) & (1-\alpha_1^{hw}) \\ \vdots & \end{pmatrix} \tag{6.14}$$

$$\rightarrow \tilde{\alpha}_n^{hw} = \alpha_n^{hw} \prod_{i=1}^{n-1} (1-\alpha_i^{hw}) \tag{6.15}$$

with $\odot$ the element-wise product

What is left, is to find a way to construct $\tilde{\alpha}^{hw}$ from α^{hw}.

For this an auxiliary matrix β^{hw} is constructed:

$$\beta^{hw} = \alpha^{hw} \times \mathbb{1}_{1\times N} = \begin{pmatrix} \alpha_1^{hw} & \alpha_2^{hw} & \dots & \alpha_n^{hw} \\ \alpha_1^{hw} & \alpha_2^{hw} & \dots & \alpha_n^{hw} \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_1^{hw} & \alpha_2^{hw} & \dots & \alpha_n^{hw} \end{pmatrix} \tag{6.16}$$

with

$$\mathbb{1}_{1\times N} = \begin{pmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{pmatrix}^T$$

Then β^{hw} is strictly triangulated such that:

$$\gamma^{hw} = \beta^{hw} \odot \begin{pmatrix} 0 & 0 & 0 & \dots & 0 \\ 1 & 0 & 0 & \dots & 0 \\ 1 & 1 & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 1 & 1 & \dots & 1 & 0 \end{pmatrix} \tag{6.17}$$

$$= \begin{pmatrix} 0 & 0 & 0 & \dots & 0 \\ \alpha_1^{hw} & 0 & 0 & \dots & 0 \\ \alpha_1^{hw} & \alpha_2^{hw} & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ \alpha_1^{hw} & \alpha_2^{hw} & \dots & \alpha_{n-1}^{hw} & 0 \end{pmatrix} \tag{6.18}$$

$$\rightarrow \delta^{hw} = 1 - \gamma^{hw} = \begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ (1-\alpha_1^{hw}) & 1 & 1 & \dots & 1 \\ (1-\alpha_1^{hw}) & (1-\alpha_2^{hw}) & 1 & \dots & 1 \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ (1-\alpha_1^{hw}) & (1-\alpha_2^{hw}) & \dots & (1-\alpha_{n-1}^{hw}) & 1 \end{pmatrix} \tag{6.19}$$

By multiplying the elements along each row in δ^{hw} one gets:

$$\epsilon_i^{hw} = \prod_{j=1}^{N} \delta_{ij}^{hw} = \begin{pmatrix} 1 \\ (1-\alpha_1^{hw}) \\ \vdots \\ \prod_{j=1}^{N-1}(1-\alpha_j^{hw}) \end{pmatrix} \tag{6.20}$$

$$\rightarrow \tilde{\alpha}^{hw} = \epsilon^{hw} \odot \alpha^{hw} = \begin{pmatrix} \alpha_1^{hw} & & \\ \alpha_2^{hw} & (1-\alpha_1^{hw}) & \\ \alpha_3^{hw} & (1-\alpha_2^{hw}) & (1-\alpha_1^{hw}) \\ & \vdots & \\ \alpha_N^{hw} & \prod_{j=1}^{N-1}(1-\alpha_j^{hw}) & \end{pmatrix} \tag{6.21}$$

This vectorized version of alpha blending will introduce a new possible bottleneck as it is, since β^{hw} will be a tensor of shape (N, N, H, W), which will equate to

$$256 \times 256 \times 256 \times 256 \times 4\text{B} = 2^{36}\text{B} = 64\text{GiB}$$

alone.

This is where the previous position-aware alpha blending tricks becomes useful. By computing β^{hw} only through the top k values of α^{hw} instead of the full tensor α^{hw}, the size will be reduced to

$$k \times k \times 256 \times 256 \times 4\text{B} = k^2 2^{22}\text{B} = k^2 \times 4\text{MiB}$$

as the shape is reduced to (k, k, H, W).

Ultimately, this accelerates optimization by a factor of $2-3$, as it will be shown in **??**.

It must be mentioned that the upper boundary for computational complexity in using this kind of alpha-blending is $\mathcal{O}(N \log k)$, since the top k search is bound by this complexity.

calcualte this

Losses

In this section, the different kinds of losses for the optimization procedure will be discussed.

First off, the L2 loss or **mean squared error** is an obvious choice for this task.

$$L_{\mathrm{MSE}} = \frac{1}{HW} \sum_{p \in \mathrm{pixels}} \left\| z(p) - z'(p) \right\|_2^2 \tag{6.22}$$

Since the MSE loss focuses on minimizing the pixel-wise error between the target image and the fully rendered approximation by brushstrokes, it will cause the rendered image to match the target image mainly in color. At the same time, MSE loss is prone to blurring, which results in washed out edges in the rendered image. Thus, this loss is must be accompanied by additional losses to make up for the shortcoming of MSE loss.

A popular choice for preserving the content in an image, which is associated with preserving edges, is **perceptual loss**. Perceptual loss is based on a VGG Network [**VGG**] that is pre-trained on ImageNet [**ImageNet**]. To compute the loss, the activations of deep layers (usually the fourth convolutional block) of the pre-trained VGG network are inferred and then compared using MSE loss.

$$L_{\mathrm{perceptual}} = \frac{1}{H_f, W_f} \sum_{p \in \mathrm{pixels}} \left\| f(p) - f'(p) \right\|_2^2 \tag{6.23}$$

The resulting distance is meant to capture how well edges between the two input images are preserved which should be equal to whether the content in both images is the same. Together with MSE loss, a perceptual loss is often used to get better reconstructions than with MSE loss alone, as edges of objects in the image are better preserved, prohibiting blurriness that would occur otherwise.

As an extension to perceptual loss, **lpips** introduced perceptual similarity or **LPIPS loss**, which weighs the different layers of the pre-trained VGG-network differently in order to increase the effectiveness of perceptual distance between two images [**lpips**]. LPIPS loss is meant to preserve edges even better than perceptual loss does with a similar computational overhead.

Besides losses that operate in pixel space, it is also necessary to restrict the action space for each brushstroke. As explained in section **??**, the renderer has been trained on a limited data set, which puts constraints on how curved brushstrokes may be and how the ratio between length and width ought to look like. These constraints must be enforced in the optimizing process as well. Because the renderer has not been trained on data outside of the generated data set, the renderer will likely break if the input parameters lie too far outside the training space. The results would then be renderings with no output, distorted brushstrokes, or just noisy output, as seen in **??**.

Thus, one must think of an additional loss to confine the parameter space to the same space as the generated brushstroke data during optimization. There are two ways of achieving this:

- Discriminators
- Explicitly coded losses

Discriminators are a popular choice in this context because even if the data distribution is not known beforehand, a discriminator is still able to learn the distribution from data and thus point out wrong parameter combinations in this case. Still, a discriminator comes with a few compromises, as the target distribution will never be entirely learned rather than well approximated by it. This leaves room for weaknesses as well as local minima in the discriminator's prediction, which would result in worse quality for this task. Usually, these weaknesses are made up for during adversarial training as if the generator overfits to such weaknesses, and the discriminator will quickly penalize such a solution. In the optimization routine, which is employed for this problem, it is not possible to train the discriminator online as the limited amount of brushstrokes will allow the discriminator to overfit the problem easily. Thus only a pre-trained discriminator with its said weaknesses can be used in this case.

Handpicked losses As the data distribution for the generated data set is actually known in this case (see 6.2), it is also possible to manually define losses that confine the brushstrokes. The width constraint – as a first example – can easily be enforced by penalizing whenever the brushstroke's width $w(x)$ is more than half the length $l(x)$ between the start point $\mathbf{s}(x)$ and the end point $\mathbf{e}(x)$ of the brushstroke

x:

$$L_{\mathrm{bs}} = \frac{1}{|X|}\sum_{x\in X}\max(0, 2w(x) - l(x)) \tag{6.24}$$

$$= \frac{1}{|X|}\sum_{x\in X}\max(0, 2w(x) - \|\mathbf{s}(\mathbf{x}) - \mathbf{e}(\mathbf{x})\|_2) \tag{6.25}$$

This is a bit more complicated regarding the limitation that is introduced to the control point $\mathbf{c}(x)$. As $\mathbf{c}(x)$ was sampled from a multivariate Gaussian with fixed parameters in the data set, it should now follow a similar distribution in relation to the direction $\mathbf{s}(x) - \mathbf{e}(x)$ of each stroke.

This can be achieved by first defining two orthonormal basis vectors which are either parallel $\mathbf{n}_{se}^{\|}(x)$ or orthogonal $\mathbf{n}_{se}^{\perp}(x)$ to the directional vector $\mathbf{s}(x) - \mathbf{e}(x)$:

$$\mathbf{n}_{se}^{\|}(x) = \frac{\mathbf{s}(x) - \mathbf{e}(x)}{\|\mathbf{s}(x) - \mathbf{e}(x)\|_2} \tag{6.26}$$

$$\mathbf{n}_{se}^{\perp}(x) = R_{\pi/2}\frac{\mathbf{s}(x) - \mathbf{e}(x)}{\|\mathbf{s}(x) - \mathbf{e}(x)\|_2} \tag{6.27}$$

$$\text{with } R_{\pi/2} = \begin{pmatrix} \cos\pi/2 & -\sin\pi/2 \\ \sin\pi/2 & \cos\pi/2 \end{pmatrix} \tag{6.28}$$

Then $\mathbf{c}(x)$ can be projected into the coordinate system spanned by $\mathbf{n}_{se}^{\|}(x)$ and $\mathbf{n}_{se}^{\perp}(x)$:

$$c^{\|}(x) = (\mathbf{c}(x) - \mathbf{a}(x)) \cdot \mathbf{n}_{se}^{\|}(x) \tag{6.29}$$

$$c^{\perp}(x) = (\mathbf{c}(x) - \mathbf{a}(x)) \cdot \mathbf{n}_{se}^{\perp}(x) \tag{6.30}$$

$$\mathbf{a}(x) = \frac{\mathbf{s}(x) + \mathbf{e}(x)}{2} \tag{6.31}$$

Now, the axes of the original multivariate distribution co-align with $\mathbf{c}^{\|}(x)$ and $\mathbf{c}^{\perp}(x)$. By calculating the mean and standard deviation along these projections, they can be compared to the parameters of the original data distribution.

$$\mu = \frac{1}{|X|}\sum_{x\in X}\begin{pmatrix} c^{\|}(x) \\ c^{\perp}(x) \end{pmatrix} \tag{6.32}$$

$$\mathbf{\Sigma} = (\frac{1}{|X|}\sum_{x\in X}(\begin{pmatrix} c^{\|}(x) \\ c^{\perp}(x) \end{pmatrix} - \mu)(\begin{pmatrix} c^{\|}(x) \\ c^{\perp}(x) \end{pmatrix} - \mu)^T)^{\frac{1}{2}} \tag{6.33}$$

$$\tag{6.34}$$

Using the Kullback-Leibler divergence for multivariate Gaussian distributions, the compliance with the data sets distribution can be checked:

$$\mathcal{L}_{\mathrm{KL}} = \frac{1}{2}\left[\log\frac{|\Sigma|}{|\tilde{\Sigma}|} - d + \mathrm{tr}(\Sigma^{-1}\tilde{\Sigma}) + (\mu-\tilde{\mu})^T\Sigma^{-1}(\mu-\tilde{\mu})\right] \tag{6.35}$$

with

$$\tilde{\mu} = \begin{pmatrix}0\\0\end{pmatrix} \tag{6.36}$$

$$\tilde{\Sigma} = \begin{pmatrix}\frac{1}{200} & 0\\ 0 & \frac{1}{25}\end{pmatrix} \text{ see (6.4)} \tag{6.37}$$

since the origin of the projection is at $\mathbf{a}(x) = \frac{\mathbf{s}(x)+\mathbf{e}(x)}{2}$ and coincides with the center of the data distribution.

In theory, L_{KL} and L_{bs} should be able to capture any deviation from the source data and ensure that parameters stay within the training space of the renderer. One problem that obviously could arise with this formulation is for patches with very similar brushstrokes that show a mean other than $\tilde{\mu}$ as well as a very low values inside the covariance matrix and non-zero off-diagonal values. Thus it will be favorable to include as many strokes as possible when calculating this loss, ideally all strokes in the global parameter catalog.

Style Transfer

One question that arises when discussing the optimization procedure is whether style transfer could be performed with this approach. Especially, since the optimization procedure has explicitly been compared to the approach by Gatys *et al.* [3], it is natural to assume that such an approach could also be applicable in this case.

This mainly requires to introduce a style loss as it has been done by Gatys *et al.* [3] since a content loss is already in place (see (6.23)). A style loss can then be implemented similarly by aggregating the activations of more layers and then calculating the gram matrices:

$$\mathcal{L}_{\text{style}} = \sum_{l=0}^{L} \frac{w_l}{4N_l^2 M_l^2} \sum_{ij} (G_{ij}^l - A_{ij}^l)^2 G_{ij}^l \quad = \sum_k F_{ik}^l F_{jk}^l \tag{6.38}$$

The problem, which arises when trying to apply style transfer with this particular approach is the partial update routine. Since the gram-matrices are meant to catch global second-order statistics of the image, a small patch would be object to a wrongful assumption that the patch represents the whole image. The only chance of dealing with a local patch in a global context in image space is a cached version of the whole current image in which the patch is embedded.

Optimization Details

As it has been explained in Section **??**, it is not possible to optimize brushstrokes for the whole image in parallel. This is partly due to the neural renderer's memory requirements, which scales with the number of rendered patches. The memory requirements of placing and blending, on the other hand, scales with the number of rendered brushstrokes as well as the number of surrounding fixed brushstrokes and the patch window size. Experiments have shown that a combination of 256 rendered brushstrokes with 128 surrounding brushstrokes paired with a patch window size of 320x320 pixels occupies around 9.5GiB of memory which leaves enough space for more advanced losses and tweaks to the network architecture.

The learning rate for the optimization procedure can be significantly larger than for training the neural renderer. With a learning rate of 0.01, the optimization procedure will converge significantly faster to a solution without any instability issues.

One choice, which has to be made individually per target image is how many brushstrokes will cover the image. As larger images obviously require more brushstrokes than smaller images, what should remain the same is the **brushstrokes density**. The brushstroke density will decide how many pixels on average should be covered by each brushstroke and thus be used during initialization. A number of 100pixels/brushstroke has produced the best results during the experiments.

Another important choice that goes along the choice of how dense the brushstrokes should be distributed is that of how many optimization steps each brushstroke will be object to. As for too few steps, the training will not have converged, and for too many, optimization will take an unnecessary amount of time. As this can vary between images, since some images require more time to converge, about 1500steps/brushstroke have been empirically chosen. There is no direct enforcement that each brushstroke is updated exactly this often, but it can be expected that due to the uniform sampling of render patches, there will be no major deviation for some brushstrokes. Since 256 brushstrokes will be optimized in every step, the total number of optimization steps can be calculated together with the brushstroke density and the image's size.

Another minor detail is the fact that for each render patch, five consecutive optimization steps are performed as this safes memory bandwidth since data must be written and read from memory each time a different render patch is optimized.

Results

Results can be seen in Figure **??** for Starry Night as the standard reference image of this thesis. It took approximately 15,000 optimization steps, which corresponds to about 2h. The rendering consists of roughly 10,000 brushstrokes.

Figure **??** shows the result for a natural photo as target image which took also 2h to compute with 8.000 brushstrokes and 12.000 optimization steps.

Evaluations & Ablation Experiments 7

In this chapter, the ablation experiments for to the approaches presented in Chapter **??** shall be presented. This is meant to show some weaknesses of each method as well as give a visual understanding of the effects of some losses and parameters. Together with an interpretation it should help to get a better understanding of previous motivations.

7.1 Neural Renderer

The neural renderer will be evaluated with respect to a few aspects. First, the performance for larger resolution images will be evaluated. Then, the effect of the discriminator will shown by training the renderer without the discriminator loss. At last, the weakness of the renderer, which is data points outside the training space, will be analyzed.

7.2 Optimization Procedure

The optimization procedure gives many starting points for ablation experiments. At first, the effects of each loss will be visually compared. Then, different initializations will be shown as well as various choices for training parameters.

Conclusion

Discussion 8

Outlook 9

APPENDIX

Appendix A

Bibliography

Here are the references in citation order.

[1] Aaron Hertzmann *et al.* 'Image analogies'. In: (2001), pp. 327–340. DOI: 10.1145/383259.383295 (cited on page 23).

[2] Hochang Lee *et al.* 'Directional texture transfer'. In: (2010), pp. 43–48. DOI: 10.1145/1809939.1809945 (cited on page 24).

[3] Leon A Gatys, Alexander S Ecker, and Matthias Bethge. 'A Neural Algorithm of Artistic Style'. In: (2015) (cited on pages 24–27, 29, 30, 38, 61, 75).

[4] Yanghao Li *et al.* 'Demystifying Neural Style Transfer'. In: arXiv (2017) (cited on pages 27, 28).

[5] Justin Johnson, Alexandre Alahi, and Li Fei-Fei. 'Perceptual Losses for Real-Time Style Transfer and Super-Resolution'. In: (2016) (cited on pages 29, 30).

[6] Xun Huang and Serge Belongie. 'Arbitrary Style Transfer in Real-time with Adaptive Instance Normalization'. In: (2017) (cited on pages 29, 30).

[7] Yijun Li *et al.* 'Universal Style Transfer via Feature Transforms'. In: (2017) (cited on page 30).

[8] Jun-Yan Zhu *et al.* 'Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks'. In: (2017) (cited on page 31).

[9] Artsiom Sanakoyeu *et al.* 'A Style-Aware Content Loss for Real-time HD Style Transfer'. In: (2018) (cited on pages 31, 32).

[10] Paul Haeberli. 'Paint by numbers: abstract image representations'. In: ACM SIGGRAPH Computer G 24.4 (1990), pp. 207–214. DOI: 10.1145/97879.97902 (cited on pages 34–36).

[11] A. Hertzmann. 'A survey of stroke-based rendering'. In: IEEE Computer Graphics and Applications 23.4 (2003), pp. 70–81. DOI: 10.1109/mcg.2003.1210867 (cited on page 34).

[12] Siddharth Hegde, Christos Gatzidis, and Feng Tian. 'Painterly rendering techniques: a state-of-the-art review of current approaches'. In: Computer Animation and Virtual Worlds 24.1 (2013), pp. 43–64. DOI: 10.1002/cav.1435 (cited on page 34).

[13] Peter Litwinowicz. 'Processing images and video for an impressionist effect'. In: (1997), pp. 407–414. DOI: 10.1145/258734.258893 (cited on pages 34, 35).

List of Figures

List of Tables