

Making Offline Handwriting Editable

Master's Thesis in Computer Science

submitted
by

Martin Hubert Stumpf

born September 27, 1990 in Naila

Written at

Lehrstuhl für Mustererkennung (Informatik 5)
Department Informatik
Friedrich-Alexander-Universität Erlangen-Nürnberg.

Advisor: Dr.-Ing. V. Christlein, Prof. Dr.-Ing. habil. A. Maier

Started: March 1, 2019

Finished: September 1, 2019

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Die Richtlinien des Lehrstuhls für Studien- und Diplomarbeiten habe ich gelesen und anerkannt, insbesondere die Regelung des Nutzungsrechts.

Erlangen, den 30. August 2019

Abstract

In recent years, multiple approaches for analyzing and recreating handwritten text emerged and were often coupled with important insights into the nature of recurrent networks. But all of them were based on online handwriting data, and extending those approaches to offline handwriting should be a major challenge. Nevertheless, there is reason to believe that it is possible, and all the necessary building elements already exist.

Based on existing techniques, we explore ways to combine **Convolutional Neural Networks (CNNs)**, **Recurrent Neural Networks (RNNs)** and classic techniques to create a pipeline that achieves offline to offline handwriting synthesis and style transfer. First, we demonstrate that the [pix2pix](#) network can be used for robust handwriting skeletonization. We then use traditional techniques to further refine the skeletons and introduce a method to generate a pseudo-online representation of the handwriting. For the generation of synthetic handwriting, we show that the pseudo-online handwriting is a sufficient input for existing online-to-online solutions. To further achieve realistic looking text, we propose an extension of [pix2pix](#) to perform a conditional style transfer. Finally, we attempt to incorporate the background of the image in the style transfer and analyze the problems we encounter.

Übersicht

In den vergangenen Jahren gab es mehrere Ansätze, handschriftliche Texte zu analysieren und zu generieren. Alle diese Ansätze basieren jedoch auf einer temporalen Representation der Handschrift, und eine Methode zu entwickeln, die sich von der tempralen Komponente distanziert, verspricht eine Herausforderung zu sein. Dennoch sind alle Bausteine, die dazu benötigt werden, bereits vorhanden und es gibt keinen Grund zu glauben, dass dieses Problem unmöglich wäre.

Unter mithilfe existierender Technologien und Algorithmen vereinen wir **CNNs**, **RNNs** und andere Algorithmen, um eine Pipeline zu erstellen, die Bilder von Handschrift analysiert und reproduziert. Zuerst zeigen wir, dass man das [pix2pix](#) als robustes Tool zur Handschrift-Skeletonisierung benutzen kann. Anschließend verfeinern wir die resultierenden Skelette mithilfe traditioneller Algorithmen und generieren eine künstliche zeitliche Komponente. Wir benutzen diese nun als Eingabe für existierende, temporal basierte Algorithmen, um inhaltlich veränderte Handschrift zu erzeugen. Weiterhin erweitern wir das [pix2pix](#) Netzwerk, um realistischen Text zu erhalten, der dem ursprünglichen Text visuell gleicht. Abschließend versuchen wir, auch den Hintergrund des Ursprungstextes mit einzubeziehen, und analysieren die Probleme, auf die wir dabei stoßen.

Acknowledgments

First, I would like to thank Prof. Andreas Maier and Dr.-Ing. Vincent Christlein for advising me on the thesis. Their advice and criticism in this project was of great value to me.

Next up, I would like to thank the entire Chair of Pattern Recognition at the Friedrich-Alexander-University Erlangen-Nuremberg and especially the Computer Vision Group, whose hardware resources were invaluable for this project. I had some very interesting, insightful and sometimes heated scientific discussions in this group which guided me along the way and I would like to specifically mention Anguelos Nicolaou and Mathias Seuret for their contribution. Likewise, I want to mention Philipp Klumpp, whose assistance and cooperation concerning the group's hardware resources is much appreciated.

Furthermore, I would like to thank Adrian Lemoine, Joe Breuer and Margareta Büche for proof-reading this thesis and supplying me with valuable feedback of both scientific and linguistic nature.

I would like to thank my entire family, whose love and support I could not be more grateful for. Last but not least, I would like to especially thank my partner Ina Rupprecht, not only for correcting my thesis, but also for her everlasting support and her patience during the time of writing it. Thank you!

Contents

1	Introduction	1
2	Background	3
2.1	Neural Networks	3
2.2	Convolutional Neural Networks	5
2.3	Generative Adversarial Networks	6
2.4	Recurrent Neural Networks	8
3	Pipeline Overview	11
4	Skeletonization	15
4.1	Introduction	15
4.2	Methodology	16
4.2.1	pix2pix	16
4.2.2	CycleGAN	18
4.2.3	Iterative knowledge transfer from naive algorithms	19
4.3	Evaluation	21
4.3.1	Datasets	21
4.3.2	Implementation Details	21
4.3.3	Experiments	25
4.4	Results	29
5	Approximation of an online representation	33
5.1	Introduction	33
5.2	Methodology	34
5.2.1	Conversion to strokes	34
5.2.2	Resampling	37

5.2.3	Ordering	40
5.3	Results	42
5.3.1	Typical Failure Modes	42
6	Writer Style Transfer	45
6.1	Introduction	45
6.2	Methodology	45
6.2.1	Graves' Handwriting Generation	46
6.2.2	DeepWriting	49
6.3	Evaluation	49
6.3.1	Implementation Details	49
6.3.2	Datasets	50
6.3.3	Experiments	51
6.4	Results	54
6.4.1	Typical Failure Modes	54
7	Pen Style Transfer	57
7.1	Introduction	57
7.2	Methodology	57
7.2.1	Modifying the pix2pix network	58
7.2.2	Modifying the pix2pix loss function	59
7.3	Evaluation	60
7.3.1	Dataset	60
7.3.2	Implementation Details	60
7.4	Results	61
8	Background Style Transfer	63
8.1	Dataset	63
8.2	Conditional Pix2Pix	64
8.2.1	Modifications to the network	65
8.3	SPADE	65
8.3.1	Results	66
8.4	Multi Step Approach	66
8.4.1	Results	68
9	Future Work	71

9.1	Skeletonization and Sampling	71
9.2	Background Style Transfer	72
9.3	Direct offline-to-offline style transfer	72
10	Conclusion	73
	Glossary	75
	List of Figures	77
	List of Tables	79
	Bibliography	81

Chapter 1

Introduction

In the last few years, the utilization of large-scale neural networks revolutionized machine learning. Its development vastly improved the results in many areas of research, from object detection, speech recognition, artificial intelligence and medical imaging all the way to speech synthesis, image generation, extra- and interpolation of data and computer graphics, just to name a few. While the theoretical background of neural networks has been known for decades, the power of the computing hardware just recently crossed a boundary where this theoretical knowledge became practical. Especially with the introduction of high-performance general-purpose GPUs the concept of deep networks, which contain millions of parameters over dozens of layers, became viable and deep learning became the tool of choice for most machine learning tasks.

Nonetheless, the subject of deep learning is still young and a lot of it is yet to be discovered. For this reason, many companies utilize networks for tasks that are not directly beneficial, but give us new insights and a deeper understanding of neural networks. One excellent example of this is Google’s AlphaStar network [Tea], which is trained to play the computer game Starcraft II and proved to be a great success, as it defeated multiple world-class human players. While this doesn’t seem very beneficial at first, it gives deep insight into the capacity, possibilities and problems that arise when working with neural networks at that scale. This knowledge might prove invaluable and directly transferable to the task of autonomous driving.

For this reason, in this thesis, we will investigate the topic of style transfer on handwritten text. While there are certain use cases where this might be useful, like automatically generated handwritten letters, its main purpose is to get new insights into the capabilities and challenges of neural networks.

We will build upon the existing work of Alex Graves [[Gra13](#)] and Aksan, Pece and Hilliges [[APH18](#)] to convert their approach of online handwriting style transfer to an offline version. We define online handwriting to be a set of data points with positional and temporal data, that represent the actual movement of a pen. Offline handwriting, on the other hand, represents simple images of written text.

Extending their work to offline handwriting will provide a major challenge and hopefully teach us a lot about both the robustness of their approach as well as the capacities and possibilities of CNNs, Generative Adversarial Networks ([GANs](#)) and specifically the [pix2pix](#) network. [[IZZ⁺16](#)]

The main contributions of this thesis are as follows:

- Creating a full pipeline for the synthesis of artificial handwriting, recreating both the visual and the writer-specific style ([Chapter 3](#))
- Showing how [pix2pix](#) can be used for handwriting skeletonization ([Chapter 4](#)) and visual style transfer ([Chapter 7](#))
- Demonstrating that a heuristic conversion method to approximate online handwriting from offline data can be sufficient to utilize the existing online-based style transfer methods ([Chapter 5](#), [Chapter 6](#))
- Analyzing further approaches on how to include the background in the style transfer ([Chapter 8](#))
- Gaining new insights into the inner workings of neural networks along the way.

The entire source code of this thesis is publicly available [[Stud](#)].

Chapter 2

Background

Most of the steps in this thesis utilize neural networks. Therefore, we briefly discuss what a neural network is and describe the types of neural networks that are relevant for this thesis, which include Convolutional Neural Networks (CNNs), Generative Adversarial Networks (GANs) and Recurrent Neural Networks (RNNs).

2.1 Neural Networks

With its original name being the *multi-layer perceptron*, a neural network is a type of pattern recognizer that is somewhat similar to the human brain structure. Its main components are neurons, which are usually connected in a specific pattern to create a learnable mapping from an input vector to an output vector. Those vectors can be almost anything, like images, text, measurements or class labels.

In its original configuration, the neurons were organized in layers, with the output of one layer being the input of the next layer. The *depth* of a network is then said to be the number of successive layers between the input and the output of the network.

The two biggest inventions that made neural networks successful were the inclusion of *nonlinearities* and the *backpropagation algorithm*.

Nonlinearities. In its first version, the output of a neuron was simply a linear combination of all of its inputs plus a constant bias. Or, if you combine all neurons of a layer into a single

mathematical operation, the output of the layer was simply the input of the layer multiplied with a weight matrix plus a bias vector. The problem that arises is the fact that having multiple layers stacked on top of each other doesn't change the fact that the entire network can still be reduced down to a single linear operation on the input data. It therefore isn't possible to learn any function that contains a non-linearity, which includes almost any practical function.

The solution was to apply a non-linear activation function to each neuron output value. In the beginning, this function used to be the *sigmoid* or *tanh* function, but later studies revealed that simpler functions yield comparable or even improved results and have large performance benefits. Most modern networks use the *relu* function or its variants, like the *leaky relu*.

A layer of a neural network might therefore look like this:

$$\text{layer}(\mathbf{x}) = \text{relu}(\mathbf{Ax} + \mathbf{b}) \quad (2.1)$$

$$\text{relu}(\mathbf{x}) = \begin{cases} x_i, & \text{for all } x_i > 0 \\ 0, & \text{for all others} \end{cases} \quad (2.2)$$

The backpropagation algorithm. Most networks are trained by defining a loss function that compares the network output to the desired output, followed by minimizing the result of the loss function by modifying the network parameters. While there were already several algorithms that perform this minimum search, like the *gradient descent* algorithm, most of them require the *gradient* of the loss value with respect to the network weights. A network that is multiple layers deep and contains an adequate amount of neurons per layer quickly becomes very resource-intensive to train, and computing every single gradient by solving the network equation over and over again becomes impractical. This is where the backpropagation algorithm comes into play. Its main idea is to compute the gradients layer by layer, starting at the loss value. It achieves that by utilizing the decomposition of a gradient into layer-wise partial derivatives, as shown in the following example. Here, we compute the gradient of the loss E for the weights w_1 through the network layer outputs x_1 and x_2 :

$$\frac{\delta E}{\delta w_1} = \frac{\delta E}{\delta x_2} \frac{\delta x_2}{\delta x_1} \frac{\delta x_1}{\delta w_1} \quad (2.3)$$

Network variants. The configuration where all neurons of one layer are being connected to all neurons of the next layer is nowadays known as a *fully connected* network, as shown in Figure 2.1.

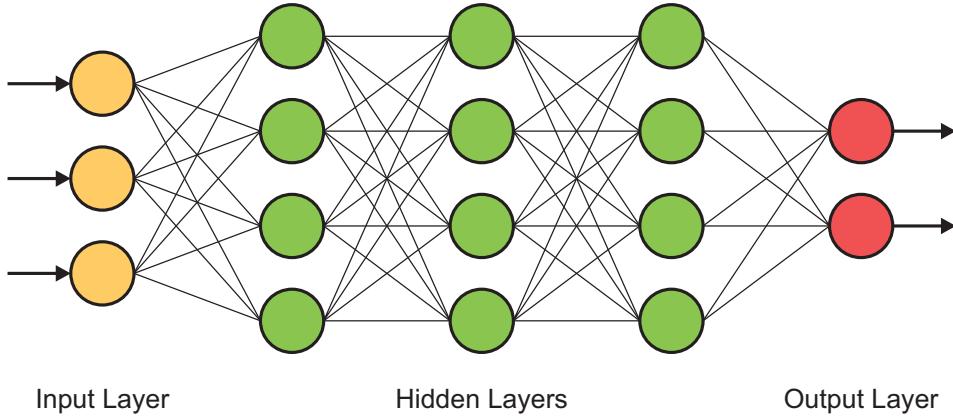


Figure 2.1: Schematic of a fully connected network. Each neuron is connected to all neurons of the previous layer.

It has the advantage that it can represent almost any function, but it also has major drawbacks. Most data types have invariants that this type of network does not have built-in, and it therefore needs a lot of training data to learn them explicitly. This is especially true if we deal with image data, as images are very high dimensional data with an inherent translation invariance for most recognition tasks. This lead to the development of Convolutional Neural Networks (CNNs).

2.2 Convolutional Neural Networks

This type of network builds on the layer-wise architecture of the fully connected network, but introduces some crucial improvements for image data. Specifically, it enforces translation invariance by tying weights of different spacial locations together.

Its first step is to push the one-dimensional vector representation into a higher dimensionality. For two dimensional images, this is usually a matrix of size $(width \times height \times features)$, keeping the spacial dimensions encoded separately, and having one feature vector per spacial coordinate.

The network layer can still be represented in a matrix multiplication like [Equation \(2.1\)](#), but the matrix itself now represents a convolution operation with a specific kernel. That kernel connects each output coordinate to a number of neighboring input coordinates. This means that each feature point is no longer connected to all feature points of the previous layer, but only to those that are spacially relevant for it. Further, every feature point of one layer is computed with the same kernel, only the position of the kernel is adjusted for each point.

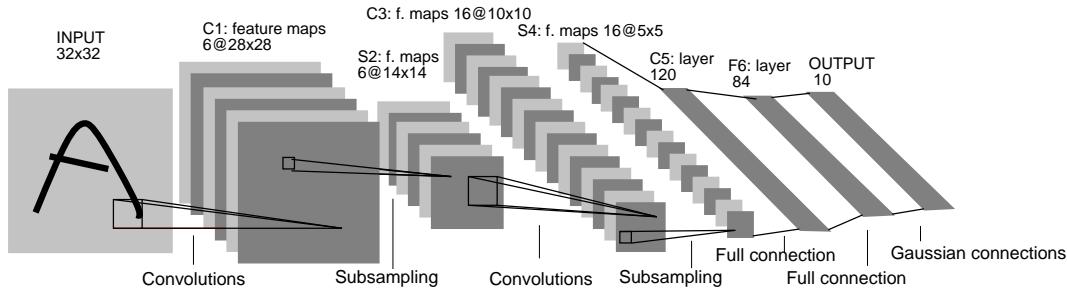


Figure 2.2: Example of a **CNN** architecture. This network, called *LeNet-5*, is a **CNN** with two convolution layers with subsampling, followed by fully connected layers. Its purpose is to detect digits. Source: [LBB⁺98]

Combining multiple convolution layers has the effect of combining low-level features to higher level, more complex features. To visualize that, the first layer might detect line segments of different directions, the second layer might combine those to patterns like curves, checkerboard patterns or gradients, the third layer might combine those to more complex shapes like eyelashes or teeth, the fourth one might combine those to eyes, mouth or ears, and the fifth layer could then be a face detector. An example for such a **CNN** can be seen in Figure 2.2.

In most cases, convolution layers are followed by a subsampling, which causes the information to become more and more global from layer to layer, so that the final layer usually only consists of a single feature vector with no spacial information. This loss of locality is mitigated by giving deeper layers increasingly more feature depth, effectively shifting the information from a spacial to a global representation. In some cases, the convolutional network is followed by a fully connected network for further processing.

Tying features together in a shared kernel reduces the dimensionality of the network drastically while preserving the ability to robustly detect and process almost any humanly distinguishable image content. Nonetheless, training **CNNs** is still a demanding task, which caused the development to stall until the computing performance finally caught up in recent years.

2.3 Generative Adversarial Networks

Contrary to **CNNs**, **GANs** don't actually describe a specific network layout, but instead refer to a new training method for existing networks. This mainly consists of replacing or extending the existing loss function with another network that learns the loss during training. Especially in

image generation, most loss functions used to be based on the Mean Square Error (MSE) between generated and real training images. This technique has major drawbacks, as a simple thought experiment shows. Imagine a task where the goal is to produce red and green squares: A network trained with an MSE loss would always produce yellow squares, because on average, this is always the best guess. However, if we replace the loss with another network that tries to find the differences between real and generated images, it will immediately pick up on the fact that yellow squares strongly indicate generated images. This in turn forces the generating network to produce either red or green squares. Those two networks are called *generator* and *discriminator*. The *generator* network (G) generates the output data and the *discriminator* network (D) tries to find the difference between the real and the generated data.

The entire training process can then be expressed by a min-max game between the generator and the discriminator, where d_{real} is the training data, consisting of pairs of real inputs and outputs. The goal is that the network learns a mapping function between them. The game would then look like this:

$$\min_D \max_G \sum_{\mathbf{x}, \mathbf{y} \in d_{real}} \|1 - D(\mathbf{y})\| + \|D(G(\mathbf{x}))\| \quad (2.4)$$

This means that the discriminator gets rewarded for outputting a value close to 1 for real images and a value close to 0 for generated images, in the attempt to minimize this entire term. The generator, however, gets rewarded for preventing the discriminator from doing so, by generating images that are indistinguishable for the discriminator. This game reaches its equilibrium point when there is no noticeable difference between the generated and real images any more, allowing for a very high-quality output of the network.

While Equation (2.4) is not the original mathematical representation, it is the most commonly used version, as it cannot saturate easily.

A very good demonstration of the effectiveness of this approach is the generation of human faces, as shown in Figure 2.3.



Figure 2.3: Photorealistic human faces generated by a GAN. Source: [KLA18]

2.4 Recurrent Neural Networks

So far, all neural networks we talked about were uni-directional, meaning they had one defined input and one defined output, and all connections were directed towards that output. This is insufficient for many problems that do not have a fixed input or output size, but rather a variably sized stream of data, like video or audio samples.

The state-of-the-art solution for those problems are *Recurrent Neural Networks* (RNNs). Recurrent networks also have an input and output vector, but they also have an additional connection to the next and previous timestep. This enables them to *memorize* information and therefore to process data with long term temporal dependencies.

It was quickly discovered that passing information to the next time step through a simple side connection is not sufficient, as backpropagation through a big number of time steps causes the gradient to vanish, effectively preventing the network from learning long term dependencies. This problem was overcome by adding **Long Short-Term Memory (LSTM)** cells, which give the network the ability to decide by itself how long information should be stored. This is achieved by adding *gates* to the memory connections with which the network itself can decide whether information should be kept or overwritten. This enables the gradients during the learning phase to effectively skip time steps, preventing them from vanishing.

At the time of writing, **LSTM** networks are state-of-the-art for most problems that require RNNs.

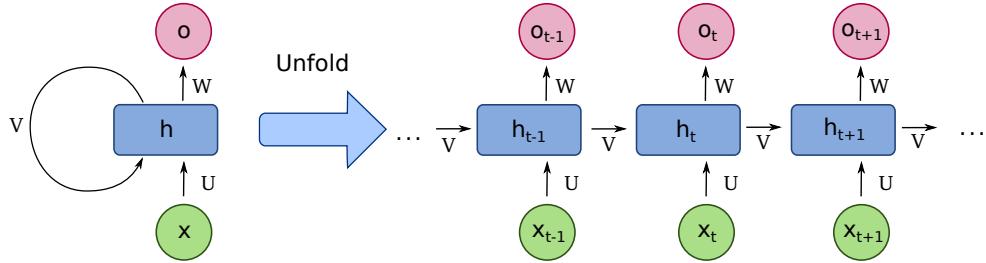


Figure 2.4: Schematic of an **RNN**. The left side shows a compressed version, the right side shows an unfolded version. x is the network input, h is the hidden state and o is the network output. U , V and W are the weights of the network, in many cases themselves being fully connected neural networks. Note that V is the recurrent connection, which feeds data from one time step to the next. Source: [Del]

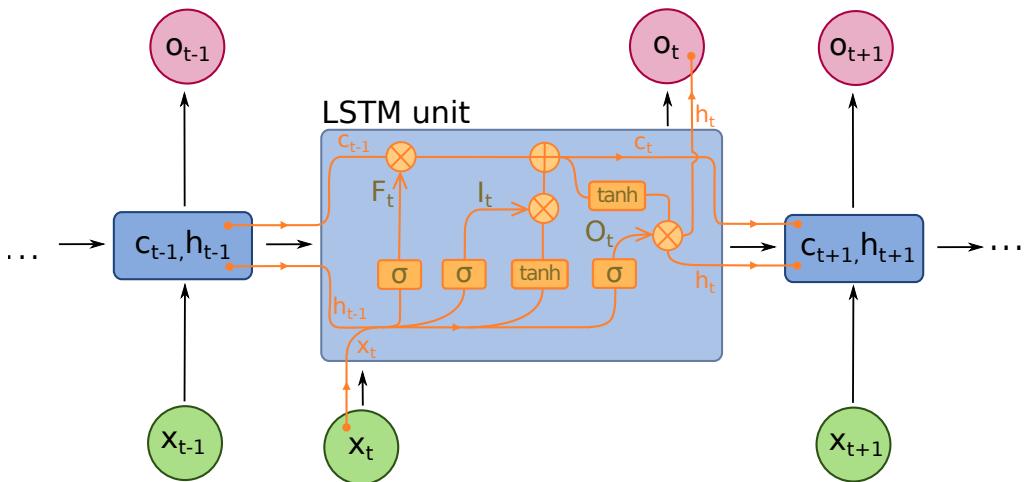


Figure 2.5: Schematic of an **LSTM** cell. Note the \otimes symbols, which represent the gates that decide whether the cell passes on its stored information to the next timestep or replaces it with new information. Source: [Del]

Chapter 3

Pipeline Overview

There are several reasons why the task of handwriting style transfer should be split into multiple steps instead of trying to create a neural network end-to-end solution. Neural networks are very good at fulfilling a single task, but as soon as this single task consists of multiple sub-steps, it becomes increasingly difficult for a neural network to figure out all of the steps at once, resulting in a network training without convergence. For handwriting style transfer, this complexity is quite obvious: The network would first have to learn the difference between the text and the meaningless background. It then has to grasp the concept of a stroke opposed to the color and type of the pen, and then further separate that into writer style and text content. Additionally, it has to remember the writer style, the color and type of the pen and the original background to produce a result with a style that matches the input image. To learn all of that at once without any guidance is almost impossible with the current type of networks. This is even more obvious when looking at the type of the content: While background, pen and writer style are static problems that could be solved with [CNNs](#), the text content has a sequential component and therefore requires some type of [RNN](#).

Another reason for splitting the problem into multiple stages is that most of those sub-problems are very well trainable on their own, allowing human prior knowledge to both guide the process as also to evaluate the single steps separately. Additionally, not all of those tasks require a neural network, some might even benefit from having an algorithmic solution. We therefore decided to break the problem down into a multi-staged pipeline.

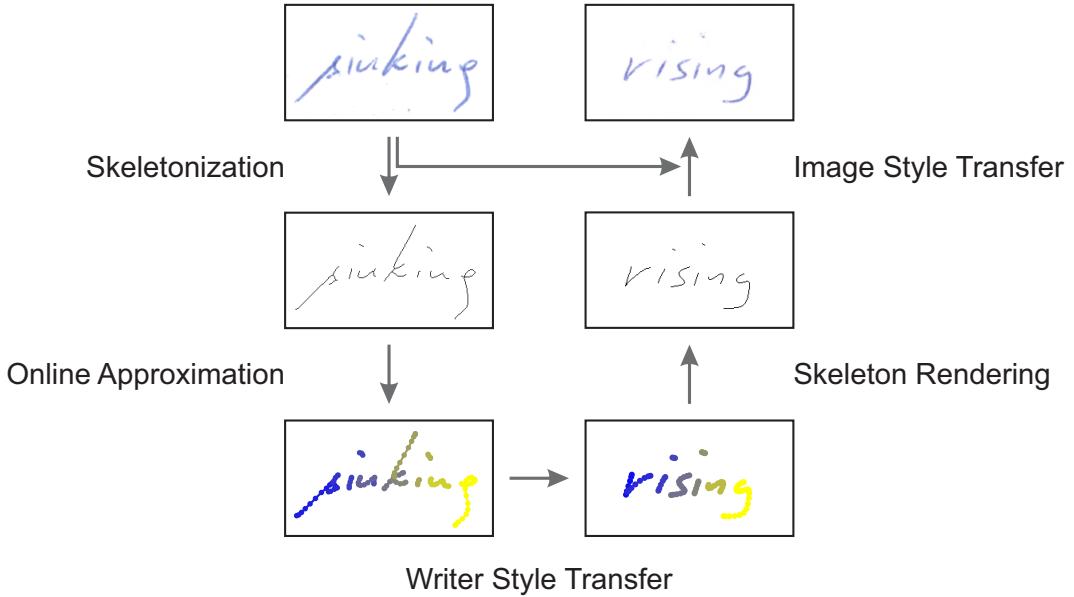


Figure 3.1: Overview of the style transfer pipeline

The proposed pipeline consists of the following five steps, as also seen in Figure 3.1:

1. Skeletonization
2. Approximation of an online representation
3. Writer Style Transfer
4. Rendering to skeletons
5. Image Style Transfer

Skeletonization. This stage takes pictures of handwritten text and converts them to binary skeleton images. While this problem isn't something new, our use case requires a high degree of robustness. More specifically, the output skeleton images cannot contain salt and pepper noise and all lines must be connected continuously. This renders most traditional techniques insufficient and calls for new methods, which in our case was the utilization of a neural network. Provided with the correct training data, neural networks can become quite robust in understanding the actual content of an image opposed to simple thresholding or grouping.

Approximation of an online representation. The core of this thesis is the utilization of existing online-to-online writer style transfer networks, which require an online representation of the handwriting. We therefore heuristically transform the offline data into an online format that the

writer style transfer step can understand. As this step involves manually crafted heuristics, we did not train a neural network for it but instead hand-crafted an algorithmic solution.

Writer Style Transfer. This stage takes the synthetic online representation, extracts the writer style and synthesizes new text in the given style. This step mostly utilizes the existing work of Graves. [Gra13]

Rendering to skeletons. The next stage renders the online representation of the synthesized handwriting to offline skeletons. As this is simply a matter of drawing lines on a bitmap, it will not be discussed any further.

Image Style Transfer. The purpose of this step is to take the generated skeletons and transform them into a realistic-looking image. At the moment of writing, lots of CNN based solutions to similar problems produce excellent results, so we also utilized a CNN based algorithm to produce those images. We tackled this step twice, first with colored text on a white background, and the second time including more realistic backgrounds. We achieve quite believable results without the backgrounds, but encounter problems as soon as we include them. Nonetheless, we analyze the problems and propose multiple possible solutions.

Chapter 4

Skeletonization

4.1 Introduction

The purpose of this pipeline stage is to take images of real handwriting as inputs and output their corresponding skeletons, as shown in Figure 4.1.

The biggest problem to overcome is that while individual datasets exist of both real handwriting pictures and skeletons, at the time of writing, there is no dataset that annotates a mapping between those two. We will try two possible solutions to overcome that problem.

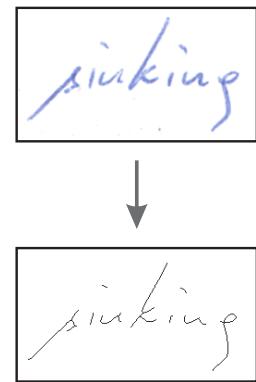


Figure 4.1

First, we will describe an approach based on CycleGAN and examine why this approach did not work. We then introduce an iterative algorithm based on pix2pix and show its effectiveness. Further, we discuss the issues we faced and their solutions.

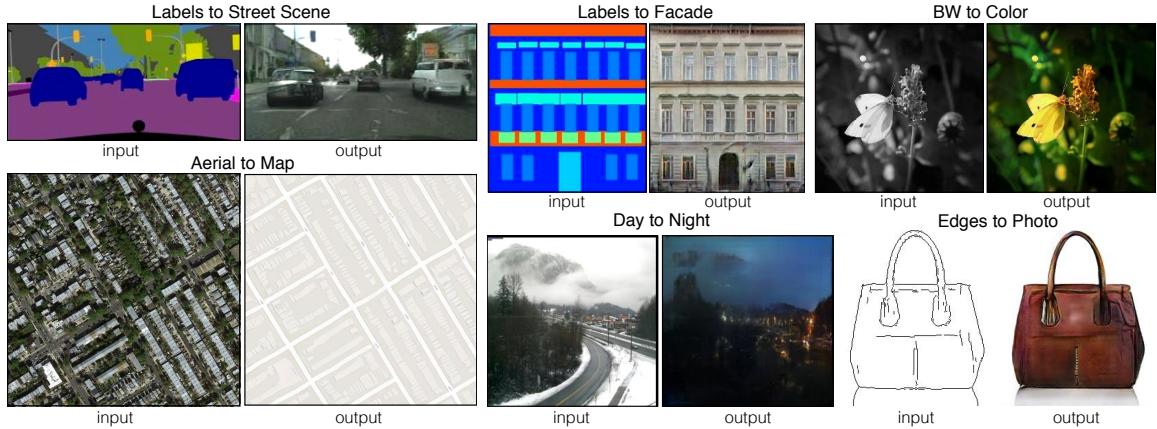


Figure 4.2: Pix2pix examples. Source: [IZZ⁺16]

4.2 Methodology

4.2.1 pix2pix

The [pix2pix](#) [IZZ⁺16] network is a Convolutional Adversarial Network proposed in 2016 by Isola et al. Its goal was to create a method to map an input image to an output image of the same size, with both images showing the same content, but in different representations. Examples of this can be seen in [Figure 4.2](#).

Their approach turned out to be both robust and easy to adapt, and after the publication of [pix2pix](#)'s source code it gained a lot of media attention. Numerous projects based on their code followed, among the most famous being the edges2cat [Hes] and the PokémonTM [Aly] generator.

To achieve the quality of [pix2pix](#)'s mappings, they utilized two crucial elements: The *U-Net*, which enables a mapping with rich detail and resolution, and their *loss function*, which is very robust to various image contents.

U-Net. In the early days of [CNNs](#), they mostly consisted of convolutions and subsampling and were used for object recognition [LBB⁺98]. But during the following years, it became apparent that they could do a lot more than that. In 2014, Long et al. [LSD14] introduced the idea of an inverse convolution, which enabled networks to *learn* a parameterized upsampling and made the construction of combined down- and upsampling networks possible, also called encoder-decoder

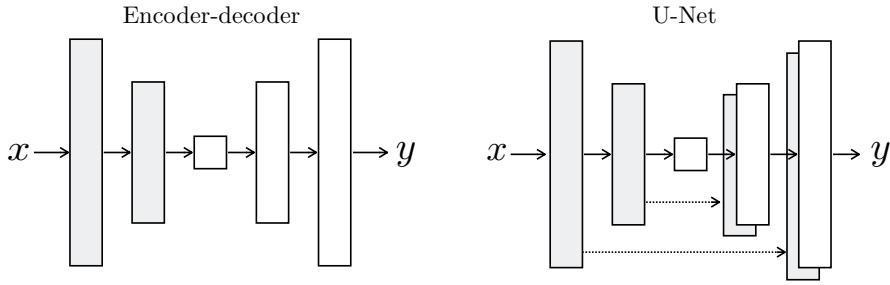


Figure 4.3: The U-Net network layout, compared to classic encoder-decoder network.
Source: [IZZ⁺16]

networks. This can be used to map two image representations to each other, which in Long’s case was medical images to a segmentation map.

However, the results lacked clarity and detail, and the paper was quickly followed up by Ronneberger et al [RFB15]. They proposed to add *skip-connections* between the respective down- and upsampling layers, as seen in Figure 4.3, and called the resulting network *U-Net*. The skip-connections helped to guide the upsampling layers, allowing them to produce a much more detailed output that matched the geometry of the input with pixel accuracy.

Still, the use case for this network was to create a segmentation map, and outputs with more photo-realism weren’t possible yet, due to the lack of a better loss function. Such a loss function would be introduced by the *pix2pix* network.

Loss Function. Most networks would have the capacity of producing highly realistic outputs if their weights would be trained correctly. The loss function is one major influence that determines the quality of the resulting network output. Sadly, hardcoded loss functions like the **MSE** loss produce ‘washed out’ images with little to no detail. This is intuitive because if there are multiple possible outputs for a network, the optimal solution in the sense of **MSE** is the weighted mean of all possibilities, which results in the loss of details.

The biggest change that *pix2pix* introduced was the adaptation of the **GAN** loss to image mappings. **GANs** replace the traditional, hard-coded loss with a second network, called the *discriminator network*, that tries to find the differences between a real and a generated image. The *generator* and the *discriminator* networks then compete against each other, until the discriminator is unable to

recognize the difference between a generated and a real image. This has the potential to produce very realistic images in high resolution.

Further, the paper states that the **GAN** loss doesn't have to span across the entire output image. Experimental results showed that comparing only patches of the generated with the real image is sufficient, but increases the performance a lot, while also rendering the loss independent of the image size.

Nonetheless, the **GAN** loss alone tends to focus too much on details, causing artifacts and discrepancies in the lower frequencies. Therefore they paired the **GAN** loss with an L1-loss, which combined produced excellent results and turned out to be very robust and adaptable to a lot of different types of images.

Applicability

The **pix2pix** network has already demonstrated in various cases that it is capable of mapping between realistic images and abstract representations, like segmentation maps or outlines. Therefore it should be well suited for the skeletonization of handwritten text.

4.2.2 CycleGAN

CycleGAN [ZPI⁺17] is an extension to **pix2pix**. While **pix2pix** requires annotated pairs of training data, **CycleGAN** tries to overcome that restriction. This is especially relevant in our case, as, at the time of writing, no pairwise annotated skeletonization dataset exists.

CycleGAN is a **GAN** that further enforces *cycle consistency*. A traditional **GAN** would fail to map images without pairwise annotations, as no mechanism enforces the network to retain a meaningful connection between the input and output domain. It is free to use the inputs as *random noise* for the generation of **GAN**-consistent output images. Cycle consistency mitigates this problem, as it enforces the network to store enough information in the mapped image to recreate the original image. To achieve that, **CycleGAN** utilizes two loss functions, the **GAN** loss and the cycle consistency loss, as seen in [Figure 4.4](#).

The cycle consistency also requires the network to always train both a forward and an inverse mapping simultaneously. Combined with two discriminator networks, there is a total of four

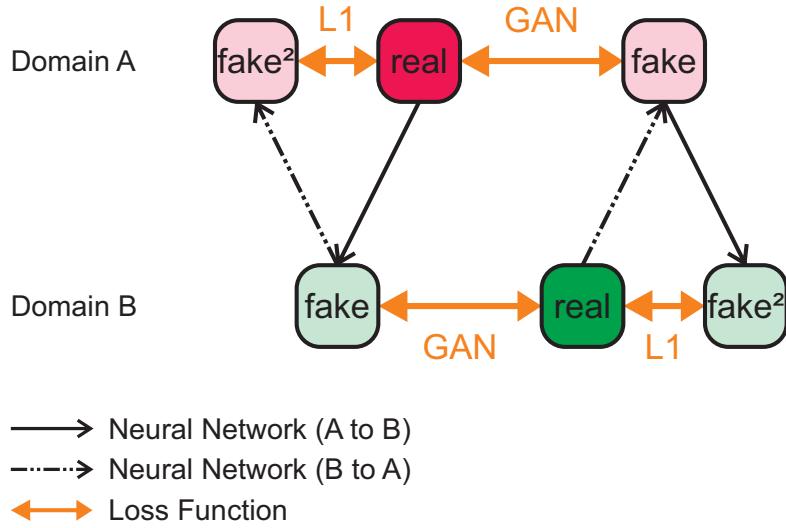


Figure 4.4: CycleGAN loss functions. Note the L1 losses, which enforce cycle consistency.

simultaneously trained networks. This makes CycleGAN one of the more memory-intensive networks.

4.2.3 Iterative knowledge transfer from naive algorithms

Training an unsupervised mapping with an algorithm like CycleGAN has one major restriction: The algorithm has to *guess* the transfer function. This could lead to several problems because it is not guaranteed that the resulting mapping will still keep spacial consistency. It could freely move lines, as long as the result is GAN-consistent and contains enough information to perform an inverse mapping. For that reason, we also looked into alternatives to CycleGAN.

Primitive algorithms that solve the problem of skeletonization already exist, but are not very robust. Nonetheless, they could be used to *guide* the training, to incorporate prior knowledge about the mapping. Therefore we developed a method to extract the knowledge of one of those algorithms to transfer it to a neural network.

The proposed method is not limited to this specific use case. It is rather a general method to transfer knowledge from an existing mapping function to a neural network while improving and generalizing it along the way.

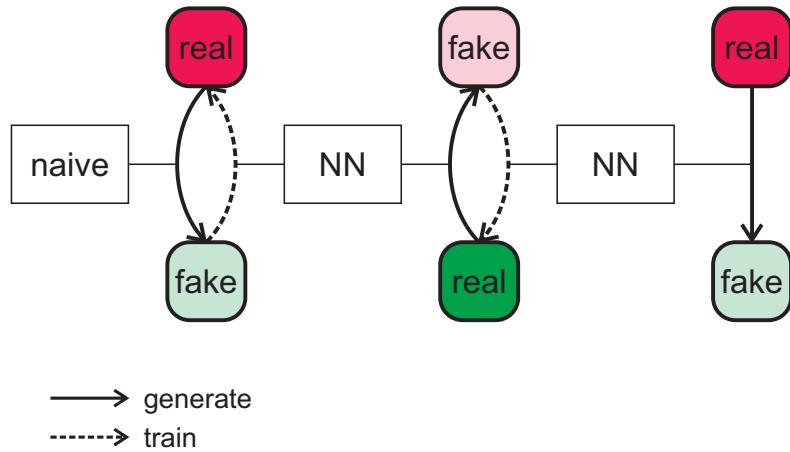


Figure 4.5: Illustration of our method to transfer knowledge from a naive mapping algorithm to a neural network. In our case, this is used to train the handwriting skeletonizer. This generalizes to all types of mapping problems for which a naive mapping function exists.

The method requires a naive mapping function and two non-paired datasets that we want to map between, the source dataset and the target dataset.

It consists of the following steps, as illustrated in [Figure 4.5](#):

1. Generation of a synthetic dataset from the real source dataset using the naive mapping
2. Training of an inverse mapping based on the synthetic dataset
3. Generation of a synthetic inverse dataset from the real destination dataset using the trained mapping
4. Training of final mapping using the synthetic inverse dataset

The first step takes the knowledge of the naive mapping algorithm and encodes it in a dataset. This dataset is expected to be erroneous. The second step then trains a network to do an inverse mapping, enforcing generalization. The network capacity needs to be sized correctly, so that generalization happens without picking up on the errors of the naive implementation. Also note that the output training data of the network is real data, so the network learns to map erroneous data to real data, further improving its quality.

The third step then takes the trained network and feeds it with the real destination dataset. The network was only ever trained to output real data, so the expectation at this step is that the switch

from erroneous input data to real input data will not produce additional artifacts, but real-like output data that matches the input data. This will finally create a dataset that maps real destination data to real-like source data, and we can now take this dataset to train our final network.

Both network training steps benefit heavily from augmentations like added noise, resizing, cropping and color jitter.

For our problem, we decided to use [pix2pix](#) as the mapping network, as it is the current state-of-the-art solution for mappings between images of identical size.

4.3 Evaluation

4.3.1 Datasets

The biggest problem for every network training is the acquisition of an appropriate dataset. In our case, we need a dataset that contains pictures of real handwriting and their respective skeletonized versions. As already mentioned, such a dataset does not exist, and annotating one by hand would exceed the scope of this thesis. There are multiple datasets of both non-skeletonized handwriting and online handwriting. Specifically, we use CVL [[KFD⁺13](#)] as the source for real offline data and IAM-Online [[LB05](#)] for real skeleton data.

4.3.2 Implementation Details

The Sharpness Problem

The *skeleton* of a binary mask traditionally consists of only background and one-pixel diameter line segments. After several attempts, it became apparent that generating a one-pixel line is really hard for neural networks, for multiple reasons. For one, it is generally hard for [CNNs](#) to deal with high frequencies in images. But maybe even more important, the loss function loses its continuity property. Combined, those problems result in the appearance of major artifacts that the network seems to be unable to resolve.

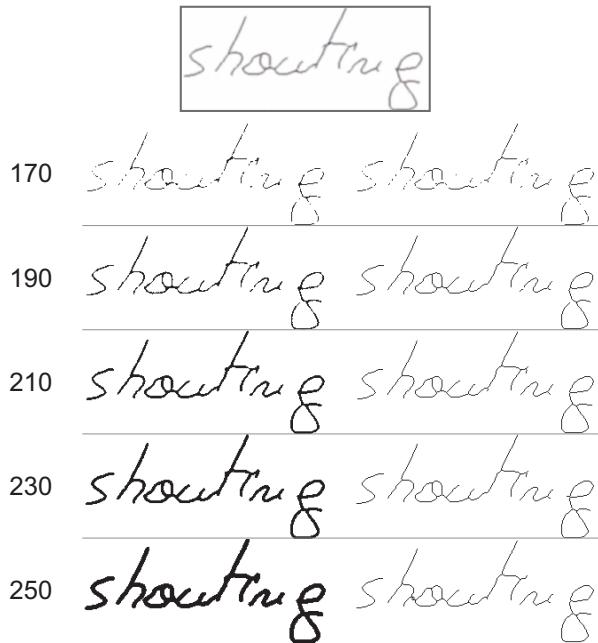


Figure 4.6: Finding the optimal blur reversal thresholding value. On the top is the original, blurred skeleton. The left column displays the image after thresholding, with different threshold values, the right column is with additional thinning, which represents the final result of the skeletonization. If the threshold value is too small, lines start to break apart. If it is too large, adjacent lines get connected, which can be seen at 250 between the *h* and the *o*.

Our solution was to apply a gaussian filter with $\sigma^2 = 1$ to the skeleton mask. This removes the necessity for the network to generate high frequencies and gives it the chance to recover gracefully from off-by-one problems.

Blur Reversal

It is very easy to generate a dataset that maps blurred skeletons to sharp skeletons. Therefore, it was natural that the first approach to achieve a de-blurring was to train yet another pix2pix network. This idea was quickly discarded for a multitude of reasons. For one, as already expected, the network didn't converge well and produced a lot of artifacts. But what was even more important, it didn't create perfect skeletons but instead generated a lot of interruptions and line duplications.

For a skeleton to be useful for handwriting analysis, it must consist of uninterrupted lines that represent the real movement of the pen. Interruptions or redundant lines would create a major

problem later on, as they would completely change the perceived writing style of the handwriting generator.

The best working solution turned out to be simple thresholding ($\text{foreground} < 215$) followed by morphological thinning [GH89]. The threshold value was determined empirically. If it is too low, line segments become interrupted, and if it is too high, distinct line segments ‘meld’ together, as seen in [Figure 4.6](#).

Nonetheless, the number can be backed up numerically: The goal of a de-blurring is to reconstruct a real skeleton without losing detail, which means that single pixels in the original skeleton should be conserved, but adjacent lines should not get merged. Therefore, we want the threshold to be as low as possible without losing single pixels. If we filter a single black pixel ($= 0$) on a white background ($= 255$) with the same gaussian filter we applied to the skeleton masks, the black pixel gets the value 213, which means a thresholding of 215 is about as low as possible for still preserving single pixels.

Scaling to variable-sized images

`Pix2pix` is intended to map a 256×256 image to a 256×256 image. It was never designed to handle variable input sizes, and the authors [[IZZ⁺16](#)] never mentioned any research efforts regarding this issue. Nonetheless, `pix2pix` is a completely convolutional encoder-decoder network and none of its layers has a hardcoded size, therefore the `pix2pix` network can theoretically handle all sizes that are multiples of 256.

The only exception is the innermost layer, which is expected to have a fixed size of 1×1 . This is the layer where all the global information gets gathered and made available to the entire output image. By increasing the image size beyond 256 pixels, this global communication channel gets lost, and it is not possible to propagate information from one border of the input image to the opposite border of the output image anymore, as demonstrated in [Figure 4.7](#). There are multiple ways to account for that. For example, an additional convolutional or fully connected layer could be added, but that would only change the output size statically and would require retraining for every single image size. A more practical approach would be to gather the information of the innermost layer via a maximum or mean operation, and then distribute that data back to the necessary shape, re-enabling that global information flow.

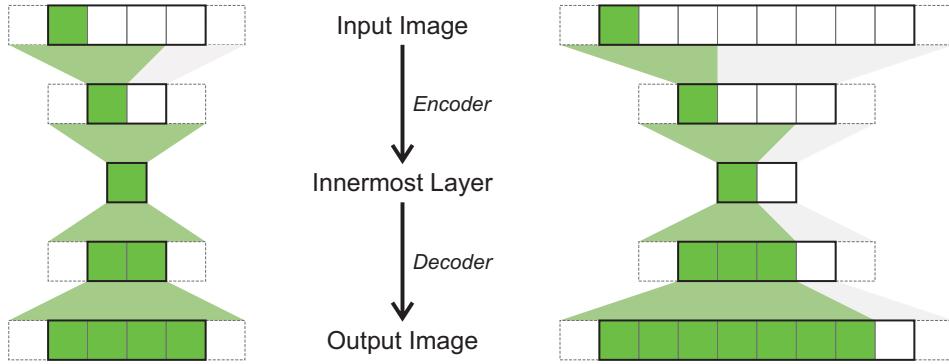
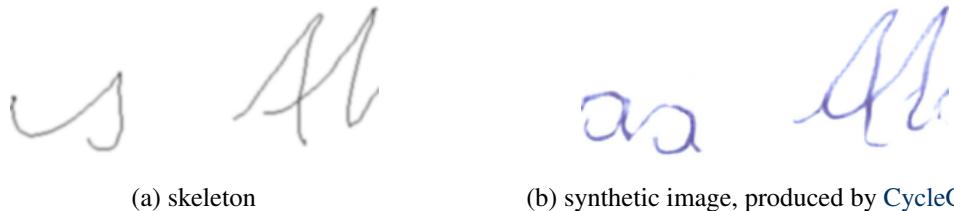


Figure 4.7: Information flow in a [pix2pix](#) network. *Left:* input image with correct size; *Right:* oversized input image. Without loss of generality, this depiction shows a simplified [pix2pix](#) network with a depth of two layers and only one feature per layer, operating on one-dimensional images, without skip connections. Every highlighted cell depends on the leftmost pixel of the input image. On the right side, the oversized output image contains pixels that don't depend on the leftmost input pixel, which shows that no global information flow exists.

Unexpectedly, in practice it looks like skeletonization with [pix2pix](#) doesn't require any global information exchange, and an unmodified [pix2pix](#) network turned out to be absolutely sufficient for the task. Several reasons would support those findings. Skeletons don't contain any non-positional meta information, like style, color or similar, and therefore the encoder strives to strip that information from the input image. Therefore, after the encoder step, all information that is left is positional data, and global information exchange would not be beneficial in any way.

The only difference between a trained 256×256 network and a size-agnostic network is that former network expects to find zero-paddings on all sides of the filters, and might produce artifacts if it doesn't. Therefore it is necessary to re-train the network with images that are definitely bigger than the perceptual range of the output pixels. In our case, we chose an image size of 2048×256 . The reasoning behind only increasing the image width is that we decided to make the images only variable in the horizontal direction and keep the vertical direction fixed to 256 pixels, as we don't scale our inputs and limit the skeletonization to a single line of text.

Nevertheless, we later discovered that the skeletonization network turned out to be size agnostic in both directions anyway.



(a) skeleton

(b) synthetic image, produced by CycleGAN

Figure 4.8: Dominant failure mode of the CycleGAN based skeletonization. While certain similarities still exist, the network completely disregarded some lines and replaced them with others.

4.3.3 Experiments

CycleGAN

As we did not have annotated data, our first attempt to train a skeletonizer was by utilizing CycleGAN.

Sadly, the CycleGAN failed for a pretty simple and, in hindsight, obvious reason: Nobody told it to only change the line- and background style of the image, but not the content. The training datasets were CVL [KFD⁺13] for real offline data and a gaussian filtered rendering of the IAM-Online [LB05] dataset for skeleton data. The writers of those datasets were different people, and once the GAN loss picked up on that fact, CycleGAN started to modify the writing style of the text and to creatively add and move lines. This can be clearly seen in Figure 4.8 and renders this approach futile.

It was apparent that the network solved its task, but the task wasn't specified correctly. It missed some crucial information, some *guidance* that showed the network what we actually wanted instead of letting it guess based on non-annotated data. For this reason, we utilized the multi-step knowledge transfer method.

Multi-step approach using pix2pix

The idea of cyclic consistency is still applicable, but it is apparent that the training needs additional guidance.

To achieve that, we apply the iterative knowledge transfer method as already described in [Section 4.2.3](#), consisting of the following steps:

1. Skeletonization of CVL using traditional techniques
2. Training of an inverse mapping with [pix2pix](#)
3. Applying the inverse mapping to IAM-Online
4. Training of a [pix2pix](#) mapping of generated ‘fake’ CVL-like data to real IAM-Online skeletons

It should be noted that all [pix2pix](#) networks are trained on artificial input data and real output data. This should lead to network outputs being as close to real data as possible.

Skeletonization of CVL using traditional techniques

The first step is to create a rough, error-prone skeleton dataset from CVL, to act as a guideline for the next steps. While having an error-prone dataset is generally not desirable, it is tolerable in this case as the errors are expected to be hidden in the generalization process of the following training steps.

The skeletonization of CVL was done in the following manner:

1. Thresholding (foreground if any color < 240)
2. Opening with cross shape of radius 1
3. Closing with cross shape of radius 1
4. Skeletonization with Zhang’s algorithm [[ZS84](#)]

The resulting skeletons are erroneous, as expected. Typical failure modes are, as seen in [Figure 4.9](#):

- Merging of adjacent lines
- Breaking apart of continuous lines at intersections
- Removal of details
- Cutting corners

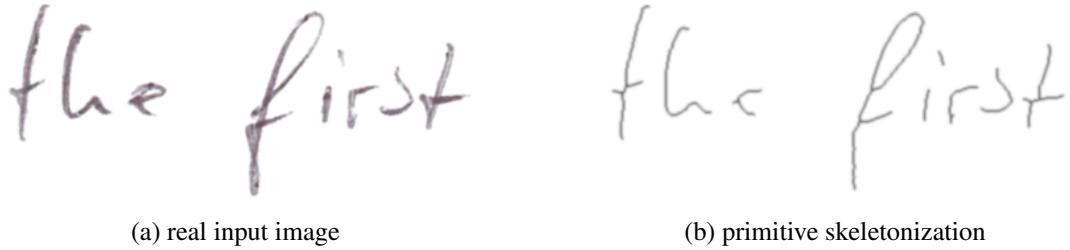


Figure 4.9: Typical failure modes of the primitive skeletonization. Note the broken lines at the *t*, the merged lines at the *e* and the removed detail and cut corner at the *f*.

Training of an inverse mapping with pix2pix

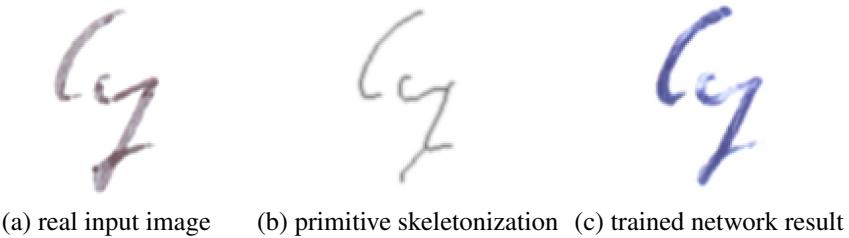
With this first annotated dataset, we can attempt to force generalization by training the inverse mapping. The idea is to train a network to produce realistic CVL-like images. Even with erroneous skeleton data, it will keep producing realistic CVL-like images if we feed it with real skeleton data.

As already discussed, the network of choice is [pix2pix](#), to be more precise, its PyTorch implementation [[Zhu](#)]. At the time of writing, there were multiple incompatibilities of this implementation with our version of Python(3.6.8) and PyTorch(1.1.0). Therefore, we created a fork and fixed the issues. This is the implementation of [pix2pix](#) that we will use in all our experiments [[Stua](#)].

The training process itself was straightforward and without further complications. Random flips were disabled because handwritten text is directional. To further improve results, random scaling was disabled. For one, generalizing to arbitrary scales caused the output to degrade, but more importantly, skeletons do not carry information about line thickness, and thus having a constant scale is important for the mapping process to succeed. Input images were of size 2048×256 .

It is to be noted, though, that the output images are not connected to the style of the training images by any means, therefore the style of the output images is randomly determined by the dropout of the network, as described in the [pix2pix](#) paper [[IZZ⁺16](#)]. This is not a problem for now as we don't need to reproduce any style to generate a skeletonization dataset, but it will become relevant in [Chapter 7](#).

The training process converged without any issues, and results of the trained network can be seen in [Figure 4.10](#). The network even learned to mitigate the errors of the primitive skeletonization, as we intended.



(a) real input image (b) primitive skeletonization (c) trained network result

Figure 4.10: Results of the inverse training of the skeletonization. The mismatched style is intended, as the network does not contain a direct style transfer. Note how the primitive skeletonization broke the bottom part of the y, and the network learned to correct that mistake.

Applying the inverse mapping to IAM-Online

We now have a network that is trained to map from less-than-perfect skeletons to realistic CVL-like images. We can apply that network to real skeleton data that we generated from the IAM-Online dataset. There was the possibility that the network might create undesired artifacts on real skeleton data, as it only saw erroneous skeletons so far, but this turned out not to be the case.

On the contrary, the generated CVL-like data matched the input skeletons closely, which allowed us to create a dataset that mapped real skeletons to their respective generated CVL-like representation. This dataset will be the basis for the final training step.

Training of the actual skeletonization network

Armed with a dataset that maps CVL-like data to real skeletons, we were finally able to train a network to do the actual skeletonization.

As before, we trained the network on images of size 2048×256 .

As the training inputs of this network are generated, we took the opportunity to add noise and variation to the training input to make the resulting network more robust. In our case, we augmented the training input samples with random resize and cropping, additive gaussian noise with a random standard deviation of 0 to 0.3, random grayscale transformation and random variations in brightness (0.5), contrast (0.5), saturation (0.3) and hue (0.2).

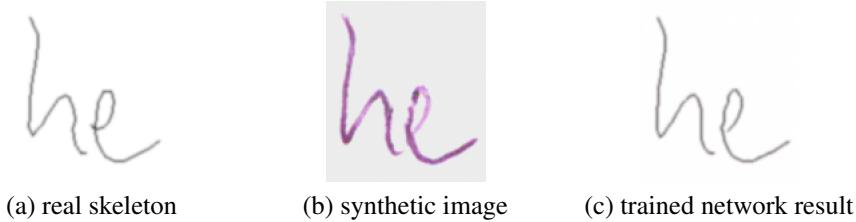


Figure 4.11: Error correction by generalization. (a) is a real skeleton image, taken from IAM-Online. (b) is a CVL-like image, generated from the real skeleton using the previously trained inverse network, followed by augmentation. (c) is the skeletonized version of (b), as produced by the trained skeletonization network. Notice the details of the *e*, which the inverse network failed to reproduce correctly. Nonetheless, the skeletonization network generalized well enough to match (b) without overfitting on its training image (a).

The network generalized so well that it was able to correct most of the mistakes of the inverse network, as seen in Figure 4.11. This resulted in a pix2pix network with the ability to robustly detect strokes of handwriting in various image conditions.

4.4 Results

As there are no real metrics to compare skeletonization results, we rely on visual perception to judge the result of our method. As the human vision is easily fooled, it is important to analyze both positive and negative aspects of the results.

To start off with the positive aspects: The knowledge transfer process definitely improved the quality of the skeletonization against the primitive approach, as seen in Figure 4.12.

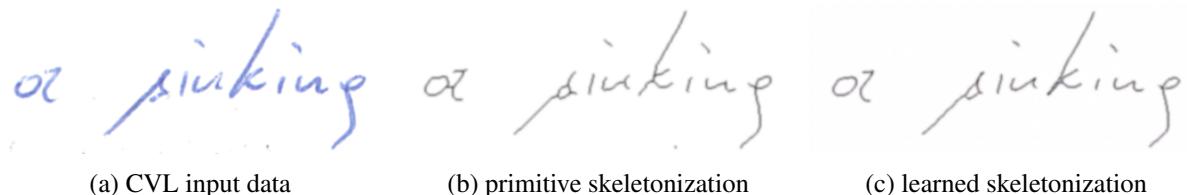


Figure 4.12: Qualitative comparison between primitive and learned skeletonization

The network learned to mitigate most of the errors of the primitive skeletonization. The superfluous line in the left word is gone and the long line of the *k* is straight. The artifacts in the *s* and *n* are

mostly corrected. Also notice the small dot below the text, which must be a stain on the paper. Skeletonization based on thresholding has major problems with paper stains, but the network correctly identified it as not part of the text and removed it.

Furthermore, because we trained the network with color and noise augmentation, it turned out to be quite robust towards noise in the image. This is especially useful in our use case, as small salt-and-pepper noise in the skeleton image would be a major problem for the writer style transfer in the following chapters.

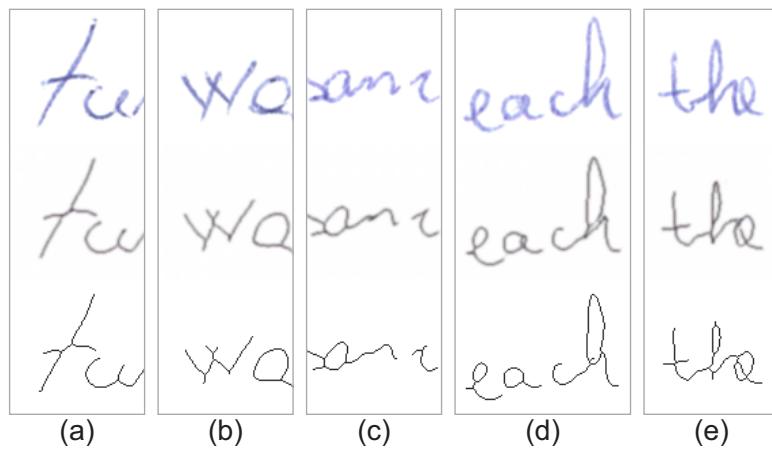


Figure 4.13: Typical failure modes of the final skeletonization network. From top to bottom: input image, skeletonization network output, final skeleton after deblurring. (a) and (b) show that crossing lines tend to be broken apart. (c) shows the loss of detail by over-smoothing. (d) shows another example where crossing lines get broken apart at the *e* and merged lines at the *h*, caused by the deblurring step. (e) shows a combination of them. The *h* gets distorted by both the network and the deblurring step, and the *e* ends up touching the *h*.

Nonetheless, the network is not perfect. It still makes mistakes, and it seems that some problems carried over from the naive implementation. One of the problems was the merging of lines, which becomes especially noticeable when two lines cross at a shallow angle. This problem partially carried over from the naive skeletonization, which shows that the generalization was not perfect yet. The other mistakes are mostly based on the mixing of adjacent lines or over-smoothing, which are typical problems of CNNs in general. Examples of those errors can be seen in Figure 4.13.

This brings us to the last part of the skeletonization: the de-blurring. As neural network deblurring did not achieve satisfactory results, we had to resort back to traditional algorithms. As discussed, the method we ended up using was thresholding followed by thinning.

Sadly, this reintroduced some of the artifacts we have seen earlier. While it is better and a lot more robust than the direct thresholding without the neural network, the thresholding mitigates a lot of the quality that we get from the network. Those artifacts directly influence the quality of the next step, the writer style transfer. Therefore the skeletonization step remains one of the biggest potential improvements for future work.

Chapter 5

Approximation of an online representation

5.1 Introduction

The purpose of this pipeline stage is to take skeleton images of real handwriting as inputs and convert them to an approximate online representation, as seen in [Figure 5.1](#).

As skeletons do not contain temporal annotations, this step will require some heuristical approach and will not perfectly match real online data.

This stage consists of two steps:

1. Conversion of the bitmap representation to strokes
2. Temporal resampling and ordering

This section will not contain any neural networks, instead, we will solve the problems analytically. In some cases, a mixed pipeline of traditional and neural network approaches can be beneficial. Neural networks are really good for all tasks that require ‘intuition’. But whenever an analytical solution to the problem exists, it will be superior to what a neural network could approximate. After all, neural networks are always just approximations.

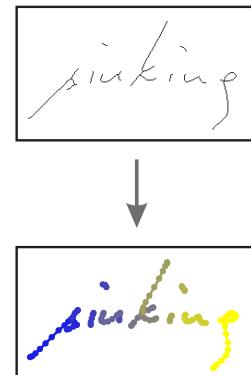


Figure 5.1

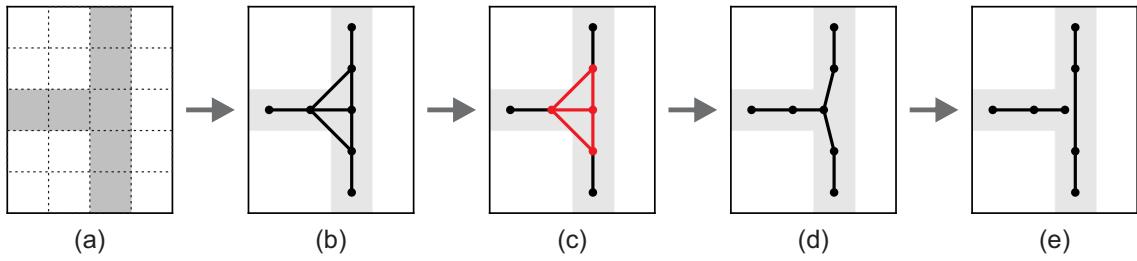


Figure 5.2: Conversion of a skeleton image to strokes. a) The original skeleton image. b) The primitively generated graph by connecting neighboring pixel. c) The detected cluster. d) The graph after the cluster got replaced by a mean node. e) The final strokes after resolving intersections.

5.2 Methodology

5.2.1 Conversion to strokes

The goal of this step is to convert a skeleton image to a set of strokes. Those strokes will not contain cycles any more, but do not yet have to be sorted or oriented.

The concept that multiple pixels belong together and form a line or curve is something that comes very natural to the human mind but is far from obvious for machines. While the introduction of neural networks gave us the ability to teach machines the concept of intuitive thinking, we could not find a neural network layout that was able to solve the problem of converting pixels to curves. Therefore, a classic algorithmic approach seemed the best fit.

An overview of the algorithm steps can be seen in [Figure 5.2](#).

Graph creation

The first step is to build a rough graph by defining all skeleton pixels as graph nodes, followed by connecting all directly or diagonally touching pixels.

This is where it becomes important that the lines of our skeleton are complete and do not contain gaps. The graph would otherwise be disconnected and would later result in two separate strokes. At this step, we assume continuous lines are in fact connected and that the decision of whether two lines should be connected was already implicitly made in [Chapter 4](#).

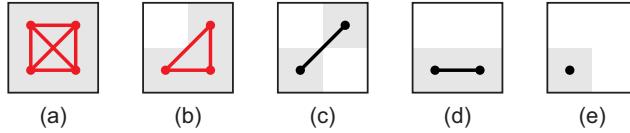


Figure 5.3: All possible triangle group configurations in a 2×2 square. Keep in mind that all patterns except (a) have rotated versions of them. This visualizes that each square can contain either four(a), one(b) or no(c,d,e) triangles. Further, a square cannot contain triangles of two different groups, as all triangles in (a) must be in the same group, as they are touching.

Dealing with clusters

The created graph still contains pixel clusters, as can be seen in [Figure 5.2](#). This is problematic because the strokes we strive to compute should not contain cycles.

It is important to note that we require all of those clusters to consist of triangles. Graph cycles that do not consist of triangles are not considered to be a ‘cluster’ artifact, but rather intentional and should be preserved. An example of that would be if people write the dot on the ‘i’ as a little circle. We do not consider the entire circle as such a cluster, but instead keep it as a circle, as it is part of the writer’s style.

We will refer to those clusters as *triangle groups*. We define two triangles to be in the same triangle group if they have at least one common edge. Once we found a triangle group we can then replace all the nodes of the group, that are not connected to outside nodes, with a new node at the mean position of the group.

The implementation of this algorithm turned out to be more difficult than expected, as performance is somewhat critical due to the large datasets we were working with.

The most important insight that enabled us to speed up performance is that we do not need to consider every single triangle. Instead, it is sufficient to focus on 2×2 pixel squares.

There are three possible triangle configurations in a square: zero, one, or four triangles. This is simple to understand, as a square only consists of up to four nodes. Two or fewer nodes cannot form a triangle, three nodes form exactly one triangle and four nodes form four triangles, as seen in [Figure 5.3](#). This also means that every square can only be part of one triangle group because if it has four triangles, they are all from the same group.

We can therefore look at all squares that contain triangles and then group them with a simple recursive fill. Working with squares is a lot faster because squares cannot overlap, have exactly four neighbors and can easily be represented, identified and accessed with 2D coordinates.

Removing intersections and cycles

While merging triangle groups removed a lot of cycles, the goal of having only line segments isn't quite achieved yet. An implicit requirement of having only line segments is that every graph node has either one or two neighbors, depending on whether it is inside or at the ends of a line segment. Further, it requires that the graph does not contain any cycles.

We define an intersection to be a node with more than two neighbors. There are two types of intersections: crossing intersections (meaning even numbers of neighbor) and joining intersections (meaning odd number of neighbors).

Crossing intersections are easy to resolve by simply connecting neighbors of opposite sides and removing the crossover node.

Joining intersections require a little more math to become useful. Every neighbor has two crossover candidates that are on the opposite side of the crossover. We compute all the angles between all candidates. Then, we iteratively connect the neighbors with the largest angle, disconnecting them from the crossover, until only one neighbor is left. We keep that one neighbor connected to the original crossover, which is now the end of a curve.

To prevent the creation of new cycles, we further check whether each line join will introduce a new cycle. If joining two nodes would create a new cycle, it simply gets skipped.

Resolving intersections removes all nodes with more than two neighbors and replaces them with nodes with either two or one neighbor. This means that we fulfilled the first requirement.

Removing cycles. Having two or fewer neighbors for all nodes does not prevent all cycles. Therefore, we follow the intersection resolution with an explicit cycle check. As all nodes only have one or two neighbors, the graph is already divided into subgraphs, with each representing a curve. So a cycle check is simply a matter of stepping through each subgraph until we either reach a node with only one neighbor or the starting node. Once we found a cycle, we split it on the upmost node; mostly because we strive for consistent behavior, and humans also tend to split circles at the upmost position.

5.2.2 Resampling

While we have lines now, we still lack a temporal component. There are multiple ways to encode time data, and the simplest and most desirable for our next stage is an encoding with constant time differences, where the timestamp of successive nodes is increasing in identical intervals. This also means that we don't need to store any temporal information explicitly because it is implicit.

However, we cannot just take the line graphs created in [Section 5.2.1](#), as all our nodes are either 1 (vertical and horizontal) or $\sqrt{2}$ (diagonal) apart from each other, which is very unrealistic and hard to deal with for a predicting neural network. We therefore tried two resampling approaches to make the node spacing more realistic: constant velocity resampling and maximum acceleration resampling.

Constant velocity resampling

The goal of this resampling method is to create a curve consisting of line segments of identical length, and therefore constant velocity. This requires the target velocity as a hyperparameter, which we determine empirically. If the velocity is too large, we lose a lot of detail, but if it is too small, we get a lot of undesired noise since our starting graph only consists of horizontal, vertical or diagonal segments. Further, setting it too small might require large amounts of memory during writer style transfer.

The resampling itself is quite straight forward: We start at the first node and then keep adding line segments until we overshoot our desired node distance. We then cut the last line segment in two sub-segments, to match the exact desired distance. We compute the desired node distance from the target velocity and round it to hit the last node without overshoot.

Maximum acceleration resampling

Real human beings, however, would not write with constant pen velocity. They would much rather start with zero velocity, then accelerate on straight parts and decelerate on curved parts, until they reach the end of the line. While not quite realistic, for simplicity, we assume that the end of the line is again reached with zero velocity.

Resampling in such a way is not quite as straight forward as constant velocity resampling, it requires an optimum search.

This algorithm requires the maximum acceleration as a hyperparameter. Like previously, this value was determined empirically by comparing it with real human online handwriting. It ended up being 0.025 times the height of the text.

We propose the following algorithm:

1. Resampling of the curve to sufficiently small intervals
2. Creating a reachability graph between nodes, to prevent cutting corners
3. Analysing acceleratability, meaning the acceleration required between two nodes. This will push the problem into 4D space: (x, y, v_x, v_y) .
4. Searching shortest path using directed Dijkstra

Constant pre-sampling. The Dijkstra search does not include the possibility to cut lines into pieces. It can only select a set of optimal nodes from the existing nodes. Consequently, we need to make sure that the existing nodes are spaced appropriately by resampling them in small constant intervals. The distance between the nodes was determined empirically. Smaller distances yield better results at the cost of more computational power. We found a sufficiently small value to be 1/3 of the maximum acceleration value.

Reachability graph. We have to explicitly enforce the algorithm to not cut corners, so that it is required to slow down during curved sections. We therefore create a graph, encoded in a boolean matrix of size $N \times N$ (where N is the number of nodes in the presampled graph), that stores the pairwise reachability between all nodes.

We define points \mathbf{p}_i and \mathbf{p}_{i+n} to be reachable, when

$$\max_{j=i..i+n} d(\mathbf{p}_j) < t \quad (5.1)$$

with $d(\mathbf{p}_j)$ being the distance of the point \mathbf{p}_j to the line between \mathbf{p}_i and \mathbf{p}_{i+n} and t being a given threshold hyperparameter.

We found the threshold parameter to be optimal at 3 times the node distance of the presampling step.

Acceleratability. So far, our nodes are two dimensional: $\mathbf{p} = (x, y)$. We now split them into four dimensions by adding the incoming velocities $\mathbf{v} = (v_x, v_y)$. As the incoming velocity \mathbf{v}_i of node

\mathbf{p}_i from node \mathbf{p}_{i-n} is computed by

$$\mathbf{v}_i = \mathbf{p}_i - \mathbf{p}_{i-n} \quad (5.2)$$

only one 4D node exists for each preceding 2D node, with multiple possible edges from different velocities of that preceding node.

We now create a 4D accelerability graph based on the 2D reachability graph that connects all 4D nodes $(\mathbf{p}_i, \mathbf{v}_i)$ and $(\mathbf{p}_j, \mathbf{v}_j)$ that fulfill the following ‘accelerability’ criterium:

$$\mathbf{v}_j = \mathbf{p}_j - \mathbf{p}_i \quad \wedge \quad \|\mathbf{v}_j - \mathbf{v}_i\| < a \quad (5.3)$$

with a being the maximum acceleration hyperparameter. As we will never go back to a previous point of the curve, we make all the connections in the graph directed. This gives us a graph that contains all possible pen trajectories that create the given curve. It is now just a matter of finding the shortest path.

Shortest path search. The set of possible paths is quite large, therefore it is important to utilize an efficient algorithm. The basic idea is a Dijkstra shortest path search, but with some optimizations specific to the 4D case.

The first thing we can make use of is the fact that the graph is directed. We will always start at one end of the stroke and move towards the other, never backwards. We can therefore step through the curve, 2D node by 2D node, and compute all optimal paths *to* that node for all possible velocities *at* that node. The number of possible velocities is quite limited and is equivalent to the number of incoming edges to that node.

Computing an optimal path to a given position \mathbf{p} and velocity \mathbf{v} is actually quite easy, if the optimal paths to the previous nodes are already known. First, we need to find the position of the previous node:

$$\mathbf{p}_{prev} = \mathbf{p} - \mathbf{v} \quad (5.4)$$

We now take all the possible velocities \mathbf{v}_{prev} at position \mathbf{p}_{prev} that can reach \mathbf{p} , based on the accelerability criterium 5.3:

$$\|\mathbf{v} - \mathbf{v}_{prev}\| < a \quad (5.5)$$

The shortest path l to (\mathbf{p}, \mathbf{v}) is then:

$$l(\mathbf{p}, \mathbf{v}) = \min_{\mathbf{v}_{prev}} l(\mathbf{p}_{prev}, \mathbf{v}_{prev}) + 1 \quad (5.6)$$

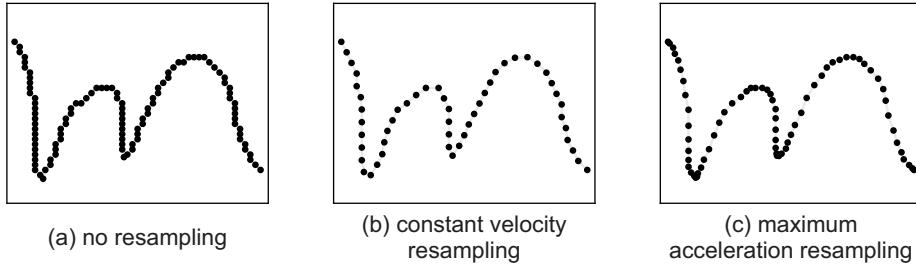


Figure 5.4: Comparison between resampling methods

We start the entire algorithm at one end of the curve, which we define as the starting point. Further, we set $\mathbf{p} = \mathbf{p}_{start}$, for which the only valid velocity is $\mathbf{v}_{start} = 0$ and $l(\mathbf{p}_{start}, \mathbf{v}_{start}) = 0$. We then iterate through the entire curve until we reach \mathbf{p}_{end} . As we defined both start and end velocities to be zero, we then look at $l(\mathbf{p}_{end}, \mathbf{v}_{end})$ with $\mathbf{v}_{end} = 0$ and backtrack to get the optimal path.

The maximum acceleration resampling was quite computation heavy, so we had to re-implement it in C. The source code can be found at [\[Stud\]](#).

The difference between the resampling methods can be seen in Figure 5.4.

5.2.3 Ordering

The only difference between the curves we have now and real human handwriting is the temporal ordering. Human western handwriting usually starts at the left and progresses towards the right. This is true for both words and letters, but sadly not always for single strokes. Luckily, we do not need to recreate the perfect original ordering to train a writer style transfer network on it. We rather need to create an artificial, but consistent ordering.

The requirements for the writer style transfer network are:

- Somewhat grouped by letters
- Consistent stroke ordering, writing the same letter multiple times should have a similar stroke order
- Letter-wise ordered from left to right

The first thing we did to increase consistency is to set the direction of every single stroke so that its start node is further left than its end node.

Sorting Key	Skeleton Data		Color Images	
	Training Loss	Validation loss	Training Loss	Validation Loss
Mean	-2.569	-2.351	-2.480	-2.274
Start	-2.530	-2.321	-2.479	-2.247
End	-2.515	-2.353	-2.463	-2.251
Leftmost	-2.524	-2.332	-2.470	-2.253
Rightmost	-2.557	-2.349	-2.461	-2.245

Table 5.1: Comparison between different sorting methods. The network trained to generate those results was a re-implementation of the Graves' network. [Stub] The results are very close, but in general, sorting by mean seems to work the best.

Then, we tried multiple heuristics to sort strokes:

- by leftmost node of the stroke
- by rightmost node of the stroke
- by start node
- by end node
- by the mean of all nodes in the stroke

None of them succeeded in reproducing the exact stroke order as a human would generate it. The most common mistakes were that the horizontal line of a *t* or *f* or the dot of the *i* was drawn before the rest of the letter. Luckily, we don't strive to reproduce the exact behavior of a human, but instead to behave predictable enough to train a writer style transfer network with it. Therefore, to get a qualitative comparison for our specific use case, we actually trained said network with all of the sorting methods, to get a meaningful numerical comparison. To be specific, we trained Graves' network, but we will talk about that in more detail in the next chapter.

To get a better understanding of how the skeletonization stage influences our result, we trained the network twice for every sorting method. The first training was with skeletons that are directly rendered from the IAM-Online dataset [LB05]. For the second one, we created a CVL-like dataset from the IAM-Online skeletons, using the network we trained during the iterative knowledge transfer (Section 4.2.3). To get the final training set, we then skeletonized those CVL-like images again. This is a lot closer to the use case of the final pipeline and is therefore hopefully also more meaningful.

The results can be seen in Table 5.1. Two things are noteworthy. For once, the skeletonization definitely had a negative impact on the results, which is expected due to the artifacts it generates,

as already mentioned earlier. Moreover, none of the sorting methods really stand out and all of the results are very close, which means that the writer style transfer is quite robust to the actual stroke order. Nonetheless, as the *mean* sorting still comes out a little bit ahead of the other methods, it is our method of choice.

5.3 Results

As already mentioned, it is really hard to evaluate the results numerically, as we have neither a metric nor another algorithm to compare it against. If we had a metric, we could generate skeletons from real online data and then compare our artificial online results to the real online data, but we were not able to come up with a meaningful metric. We, therefore, decided to empirically look at the results and search for inconsistencies manually. Additionally, we used the convergence of the writer style transfer network from the next pipeline stage as guidance, which will be further discussed in the next chapter.

5.3.1 Typical Failure Modes

There are some situations where the results clearly differ from the original strokes, as seen in Figure 5.5. The most common errors are:

- Intersecting lines fail to get connected or get connected incorrectly
- Lines that are directly on top of each other are always represented as a single line
- Lines with sharp edges get broken up

Nonetheless, those errors are not crucial for our pipeline, as we do not strive to recreate human behavior. Instead, consistency is much more important for the rest of the pipeline, and to that end, our results are absolutely satisfactory.

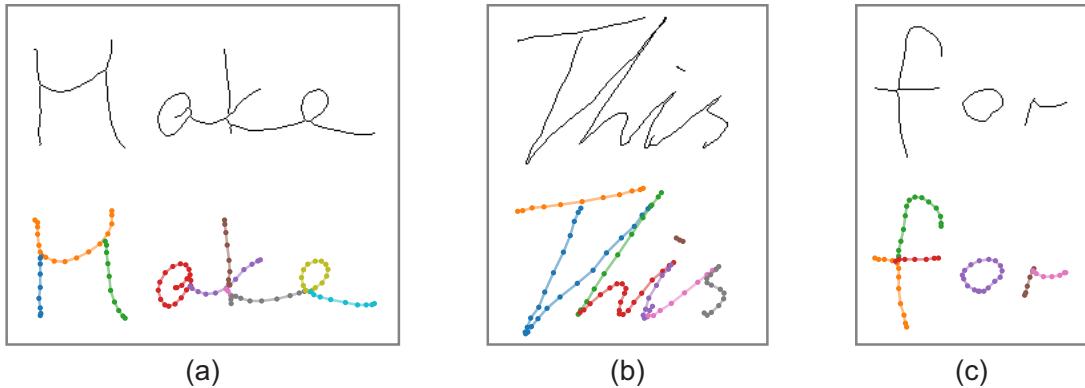


Figure 5.5: Typical failure modes of the online approximation. (a) The *M* demonstrates the inability of the algorithm to understand lines that are drawn on top of each other, causing the continuous line to be split into three segments. Further, the *k* and the *e* show cases where the algorithm fails to connect crossing lines properly. (b) This demonstrates that the algorithm has problems with sharp edges. The entire word is mostly one continuous line, but the algorithm split it into lots of segments. (c) Here, the intersecting lines of the *f* got connected incorrectly.

Chapter 6

Writer Style Transfer

6.1 Introduction

The purpose of this stage is to take handwriting in an online representation, analyze the handwriting style and then generate new handwriting with a given text content, as seen in [Figure 6.1](#).

As this step mostly consists of existing work, we will not describe our methods at length.

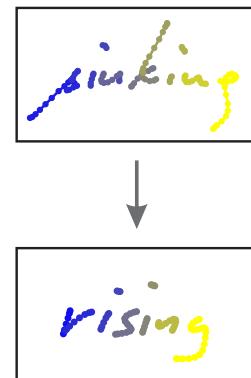


Figure 6.1

6.2 Methodology

Producing realistic handwriting is a challenging task for neural networks. Online handwriting is a very good example of a problem that consists of both content and style, in a strongly interwoven way. The network therefore has to understand the difference between style and content while not overfitting on them. To produce convincing handwriting, it needs to reproduce the given content exactly, while keeping the style consistent, but not constant. Real human handwriting will repeat the same content almost identically, but still with some variance. This requires a solid long term memory, combined with some guidance from the required content.

*Somebody once told me the world is gonna roll me
I ain't the sharpest tool in the shed*

Figure 6.2: Synthetic handwriting generated by Graves' network. Source: [Vas]

While there are multiple ways to create neural networks that could deal with that kind of data, the most obvious choice is to utilize an **RNN**. In particular, we looked into two **RNNs**, one written by Alex Graves [Gra13], which is the ‘original’ network that demonstrated that generating sequences with complex long term dependencies is actually possible. The other network is **DeepWriting** by Aksan et al [APH18].

6.2.1 Graves’ Handwriting Generation

Graves’ goal with his paper [Gra13] was not primarily handwriting generation, but instead to show that **LSTM** cells are capable of generating complex structures with long-term contextual dependencies. He used handwriting to demonstrate his technique, with remarkable success, as seen in Figure 6.2.

He first started with pure pen position predictions, completely disconnected from the actual text content. He, therefore, defined a network that predicts the next pen position and pen-up events, based on all previous pen positions. The network itself is a fully connected network with multiple layers, where the input of every layer consists of the output of the parent layer (if a parent layer exists) and a skip connection from the raw input of the network. Further, every layer consists of **LSTM** cells and therefore also depends on its previous states, which enables the network to make predictions in context of the previous pen positions. The output of the network is fully connected to all intermediate layers. The skip connections from the input and to the output enable the network to decide by itself how shallow or deep the datapath should go, giving it full control over the functionality of the **LSTM** cells. A rough outline of the network can be seen in Figure 6.3.

The output of the network consists of *mixture densities*. This enables the network to express uncertainty, which is important in the case of handwriting, and enables variance, as the next sample can be generated by drawing from the mixtures densities.

This configuration already learned how to predict handwriting quite well. Nonetheless, its prediction was completely decoupled from the content as it had to predict both the content and the

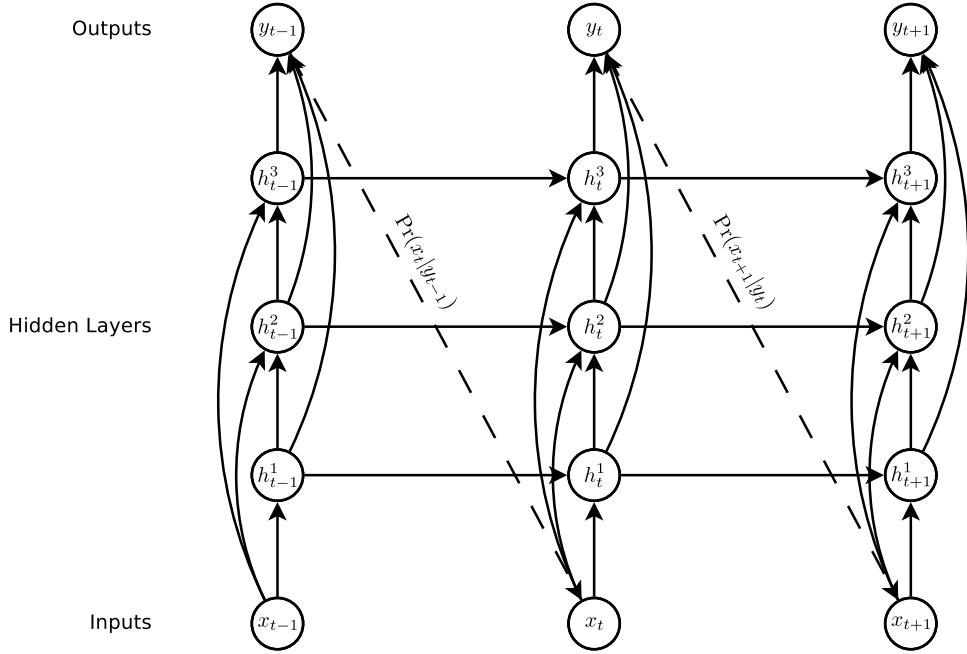


Figure 6.3: Graves' prediction network architecture. The circles represent network layers, the solid lines represent weighted connections and the dashed lines represent predictions. Source: [Gra13]

style simultaneously. Graves therefore refined his approach by adding the content of the text as a side input to one of the intermediate layers. He did not let the network see the entire content sequence at once, instead, he let the network itself decide what it wants to look at. To achieve this, he used another mixture density output from intermediate layers that would decide which part of the content gets delivered to the network. This would nowadays be called an *attention* mechanism. The adjusted network can be seen in Figure 6.4.

Giving the network access to the content improved the predictions substantially, which was especially visible in the network output, as the variance of the mixture components reduced drastically. Especially between letters, the network became a lot more confident, as it now knows which letter follows instead of having to guess. Actually, the quality of the output got so much better that the network was now able to generate entire sequences on its own, instead of just predicting single steps. Generating sequences worked by feeding the prediction of the network back into it instead of using real pen positions.

This had the pleasant side effect that the network could now be *primed* with a specific sequence, which is then continued by the network based on a given content. During the priming, the network

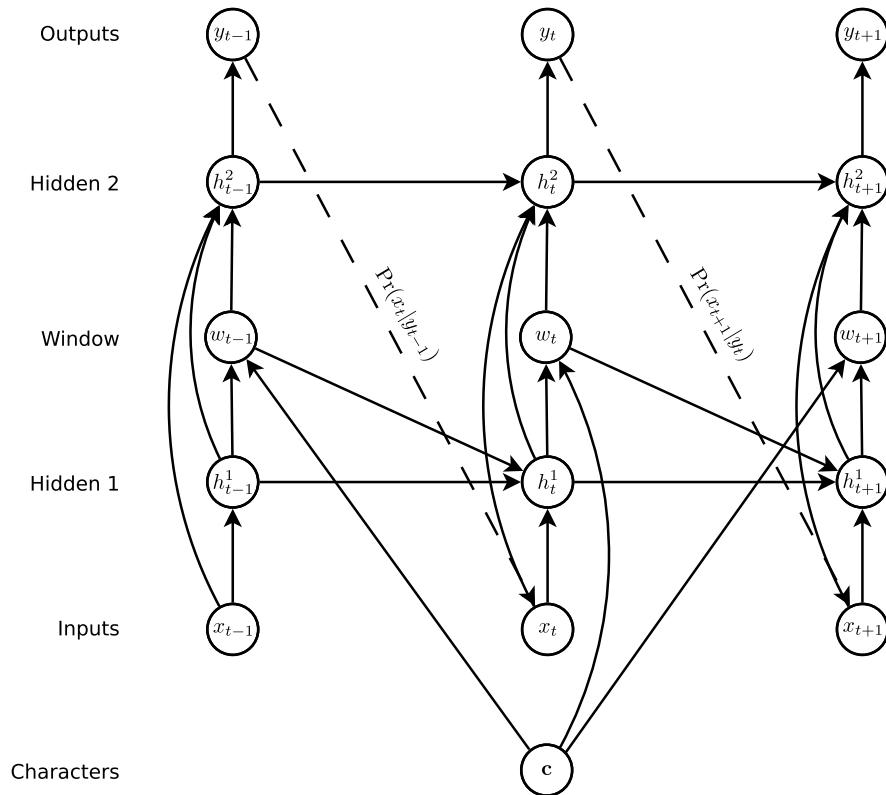


Figure 6.4: Graves' synthesis network architecture. The difference to Figure 6.3 is the extra input from the character sequence \mathbf{c} , which gets fed to the network through a window layer.
Source: [Gra13]

implicitly stores the style of the priming sequence in its memory cells, and when asked to continue, it keeps writing in the same style, effectively doing a writer style transfer.

6.2.2 DeepWriting

DeepWriting built on Graves' idea of predicting single pen positions, but instead of relying on the internal memory of the network to store the style, their goal was to explicitly extract style and content from the input. They achieved that by utilizing a [Conditional Variational Recurrent Neural Network \(CVRNN\)](#) to split the input into two separate latent random variables, representing style and content.

Their approach is quite complex, and we will not detail it here. There are two big differences between Graves' network and DeepWriting: The *input dataset annotations* and the *handling of the text annotations*.

Input dataset annotations. Deepwriting needs the input strokes to be split into words and characters. Consequently, it requires the input data to contain [beginning-of-word \(bow\)](#) and [end-of-character \(eoc\)](#) labels. This is a big disadvantage of DeepWriting, as those labels have to be manually annotated and cannot be automatically determined from offline handwriting.

Handling of text annotations. While Graves' network implemented an attention mechanism to give it the freedom to figure out at what time which letter is important, DeepWriting uses the generated [eoc](#) labels to switch letters. This removes some predictive capabilities of the network, as it can only see which letter follows as soon as the next letter is already about to be written. This is a clear disadvantage, as experiments in Graves' paper clearly show that the network decides to look at multiple letters at once and uses that information for smooth transitions between letters ([Figure 6.5](#)).

6.3 Evaluation

6.3.1 Implementation Details

DeepWriting published its source code, so we used it without further modification. [APH]

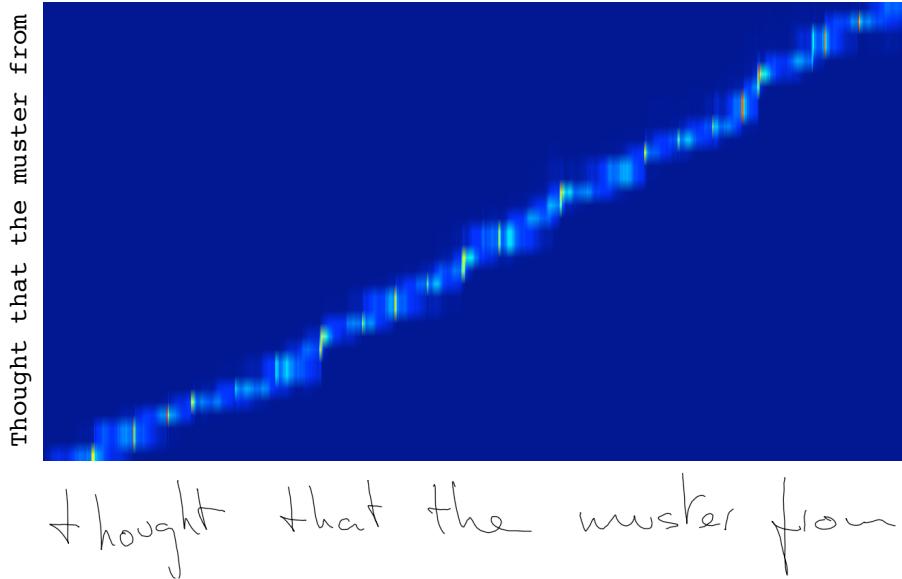


Figure 6.5: The position of Graves' attention windows. The horizontal axis represents the handwriting position, the vertical axis the accompanying text content position. The bright line is the position of the window function. The important detail to note is that the window doesn't only look at the current letter, but also to the next and previous letters, which helps the network to make smoother transitions between letters. Source: [Gra13]

Graves' source, however, was quite outdated, so we took a modern, TensorFlow based implementation [Vas] and augmented it to our needs [Stub].

6.3.2 Datasets

For our initial testing, we used the handwriting dataset provided by the DeepWriting paper. This was necessary as it is the only available dataset with `bow` and `eoc` labels, as required by the DeepWriting network.

Later, we discarded the idea of utilizing the DeepWriting network and therefore also lost the necessity to use their dataset. We therefore switched to the IAM-Online dataset [LB05] and, for full offline handwriting testing, the CVL dataset. [KFD⁺13]

6.3.3 Experiments

As both networks already proved that they can handle synthesis and style transfer of online handwriting data, the real question is how the networks respond to replacing the real online input style with increasingly artificial variants. We conducted our testing in two steps. In both steps, we used synthetic online data as training samples, but first, we created the data from skeletonized real online date, and later, we created it using the entire previous pipeline on CVL-like data.

Skeletonized online data

The first step was to create artificial online data from skeletonized real online data. We used this step to compare the different resampling methods described in [Section 5.2.2](#).

For our first experiments, we utilized the [DeepWriting](#) network, until we encountered too many problems with it and decided to switch to Graves' network.

No resampling. Using the generated online data without resampling didn't work for a multitude of reasons. For one, the data was too large, and the 16 GB RAM of the Tesla V100's we used were insufficient to actually train on the artificial online data without resampling it. But further, it is quite obvious why it would not have worked, even if we had the storage capacity: The network can only predict 8 possible outcomes: Horizontal, vertical and diagonal steps of one single pixel. As previously discussed, neural networks can easily deal with continuous, smooth data, but have big problems with sharp, high frequencies and their respective aliasing. This would have most likely prevented the network from converging.

Constant velocity resampling. With constant velocity resampling, we were now actually able to train the [DeepWriting](#) network. It converged, and learned to produce [eoc](#) labels and strokes of roughly the correct length, but didn't actually grasp the concept of entire letters. This is supported by the fact that not all elements of the loss function converged.

In our search for the difference between the real online date (on which the network converged properly) and the generated online data, we found that the main problem was in fact the constant line segments, which removed an entire degree of freedom from the prediction capability of the network. It forced the output of the network to also consist of line segments of constant length, and the only thing that the network could still modify was the direction of those lines. This led to the implementation of maximum acceleration resampling, as described previously.



Figure 6.6: Comparison between handwriting style transfer methods. *Top left:* DeepWriting with constant velocity resampling. *Top right:* DeepWriting with maximum acceleration resampling. *Bottom left:* Graves' network with maximum acceleration resampling. *Bottom right:* The real skeleton that was used for style input.

Maximum acceleration resampling. With maximum acceleration resampling, the network finally converged with all of its loss functions. While the output is still quite unreadable, first patterns of letters and even words showed up. While this is a big step in the right direction, it still is not good enough to be useful. Sadly, this is a strong indication that other problems with our generated online data still exist or that DeepWriting itself just isn't robust enough to deal with artificial data.

Graves' network. As the difference between generated online data and real online data became increasingly difficult to tell, we decided to try the network of Graves as a reference, to see if the problem is actually our data or DeepWriting itself. To our surprise, Graves' network converged well and the output was readable and satisfactory. This made us discard the DeepWriting network along with its dataset.

The results of the experiments can be seen in Figure 6.6.

CVL data

Sadly, the CVL dataset does not contain line-wise text annotations. Therefore we cannot train Graves' network with CVL directly. Luckily, we can use the IAM-Online [LB05] dataset and the skeleton-to-cvl converter of Chapter 4 to generate a CVL-like dataset with text annotations. We can then convert those images back to artificial online data to get samples that are very close to the ones we will get in the final pipeline.

Training Data	Style Input	Result
synthetic	synthetic	I am a synthetic sample
synthetic	real	I am a synthetic sample
real	synthetic	I am a synthetic sample
real	real	I am a synthetic sample
Reference Style		The Prime Minister looked puzzled,

Table 6.1: Performance evaluation of real vs synthesized training data. We tested four times, with real and synthetic data both for the training dataset as also for the style input at evaluation time. Note that the network that was trained on real data produces a lot less artifacts, even if the style input is from synthetic data.

An interesting question we explored was whether to train the handwriting generator with real or synthetic skeletons. The style input of the full pipeline will come from synthetic skeletons, therefore, it will be easier for the handwriting generator to extract the style if it was also trained on synthetic skeletons. On the other hand, synthetic skeletons always contain artifacts, which the generator will then also produce. Using real skeletons as training input for the generator could mitigate those artifacts, but during inference, it would increase the chance of failing to parse the style input entirely.

To answer this question, we trained the generator twice, once with real skeletons and once with the generated CVL-like skeletons. Then, we used the trained networks to synthesize new handwriting, once with real data as style input and once with synthetic data as style input, as seen in [Table 6.1](#).

The results were clear: The network trained on real data produced far fewer artifacts than the one trained with synthetic data. This was already expected, as the real training data also contains fewer artifacts than the synthetic training data, and after all, the network tries to recreate what it observes during training. Surprisingly, in this specific example, the network trained on real data seemed to perform better on synthetic style inputs than on real ones. This was consistent with other tests we ran, but we are not certain why this is the case.

Nonetheless, these results demonstrated that it is better to train the writer style transfer network on real skeleton data, even if it will receive synthetic skeletons as style inputs later.

Style Input	Output
spineless!" "You think you could	I am a synthetic sample
Michael Caxton The father	I am a synthetic sample
they get' em. We've been alert	I am a synthetic sample
one for me?" Jones did so, and	I am a synthetic sample
any better? You know it was! He	I am a synthetic sample
and change the money	I am a synthetic sample

Table 6.2: Qualitative evaluation of the writer style transfer. The style inputs are from the test set, the network has not seen them in the training process.

6.4 Results

The results we achieved are quite satisfying. We successfully demonstrated that Graves' network is robust enough to adapt to our synthetic online data.

We demonstrate the ability of the trained network to extract and mimic unknown styles by generating texts from style samples of our test set, which the network has not seen during the training phase. The results can be seen in Table 6.2.

6.4.1 Typical Failure Modes

However, the network still does make mistakes that clearly identify some of the results as artificially generated, as seen in Figure 6.7.

The typical failure modes consist of:

- Addition of superfluous lines
- Incorrect splitting of lines

- Incorrect positioning of lines
- Detail removal, like *t*-, *f*-lines or *i*-dots
- Skipping letters at the start of the message, which is an artifact that happens when the network was unable to correctly parse the input style

Those failure cases clearly show that there is still room for improvement in future work. Nonetheless, the results are quite satisfactory for now.

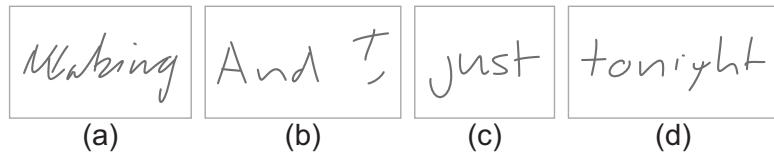


Figure 6.7: Typical failure modes of the writer style transfer. (a) A redundant line was generated into the *M*. (b) The vertical line of the *I* is too short and placed incorrectly. (c) The *j* is missing its dot. (d) The *g* is broken into two parts that aren't properly connected.

Chapter 7

Pen Style Transfer

7.1 Introduction

The purpose of this pipeline stage is to take skeleton images of handwriting and convert them to realistic images. In this stage, the generated background will always be white, similar to the data of the CVL dataset. [KFD⁺13]

While we already trained a network that accomplishes such a task as part of the iterative knowledge transfer of Section 4.2.3, that network is incapable of generating images in a specific pen style, but instead randomly chooses one.

Therefore, the challenge in this stage is to create a similar network to what we already have, but with an additional style transfer functionality built-in.

7.2 Methodology

As we had really good results with the [pix2pix](#) network so far, we decided to use it as the basis for the conditional image style transfer. To achieve the desired functionality, we had to augment both the network itself and its loss function.

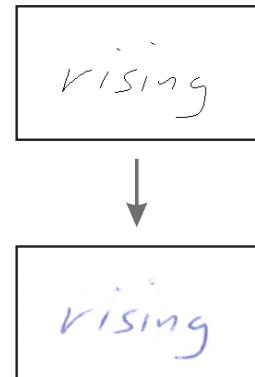


Figure 7.1

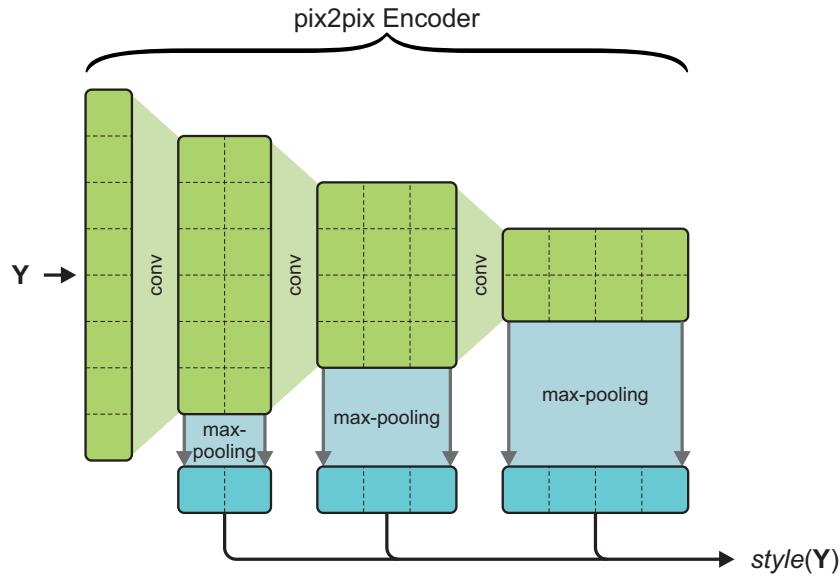


Figure 7.2: Extracting style information from an image. This is a simplified schematic and not to scale. The green network is a reused encoder of the pix2pix generator network. The max-pooling extracts the style information after every activation function layer.

7.2.1 Modifying the pix2pix network

The pix2pix network consists of an encoder and a decoder network, with additional skip connections in between. Both the encoder and decoder are **CNNs**, with the encoder being a downsampling **CNN**, and the decoder being an upsampling **CNN**. The innermost layer of pix2pix is supposed to consist of global information that does not contain positional data anymore. All the positional data gets passed through the network via the skip connections.

The first step in creating a style transfer network is to extract the style information from a given image. For that, we utilized the encoder part of the **pix2pix** network. We used the style image as input for the encoder and simply took the outputs of all activation function layers as style information. To strip away the spacial dimensions, we applied max-pooling. This process is as shown in Figure 7.2.

We then injected that style information into the **pix2pix** generator network by concatenating it with the innermost layer of the network, as seen in Figure 7.3. To keep the size-agnostic property of the network, we repeated the style information along the two spatial axes to match the size of the innermost layer.

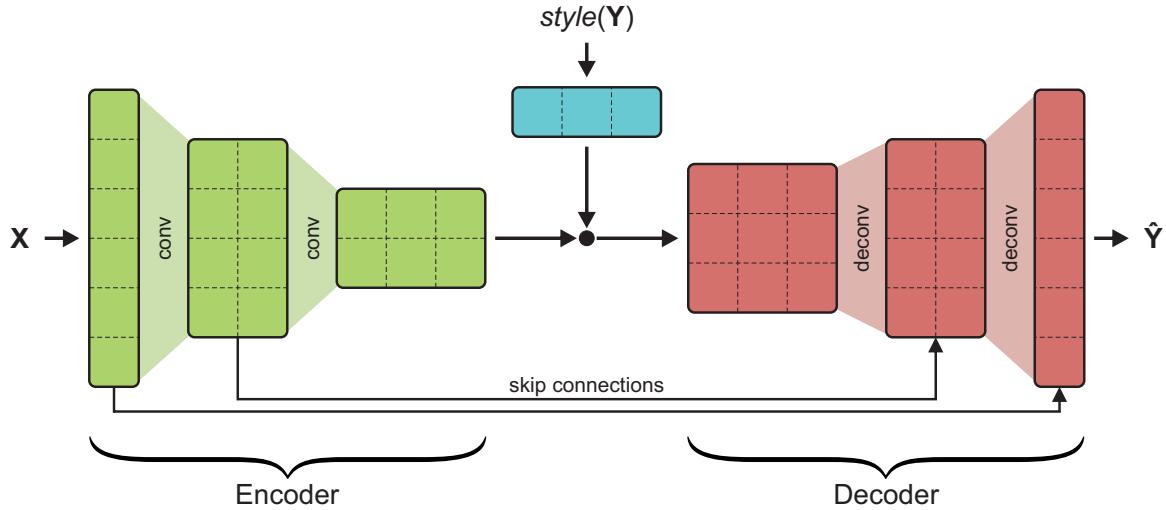


Figure 7.3: Modified pix2pix generator network for conditional style transfer. This is a simplified schematic and not to scale. The blue style information is the output of Figure 7.2. The style information gets concatenated with the output of the encoder, so that the decoder can use it to generate the output image.

7.2.2 Modifying the pix2pix loss function

With the current loss function setup, the network has no motivation to actually learn the style information, as it doesn't get penalized for not doing so. We therefore need a way to include that in the training.

The current loss function consists of the input and output image, concatenated, followed by a CNN with a single output. As this is a GAN loss, part of the loss function is the discriminator, which we train to label images as real or fake. We simultaneously punish the generator network for producing outputs that the discriminator can distinguish from real images

To include the style extraction network in the training process, we simply add the output of the style extraction network to the input of the discriminator, as shown in Figure 7.4. Feeding this information to the discriminator has the effect that the discriminator can now differentiate between real and generated images by comparing their style output. This, in turn, forces both the generator to generate images that include the style, as well as the style extraction network to produce meaningful styles so that the generator can use them. At this point, it is especially important that the style only contains global information, as the discriminator could otherwise

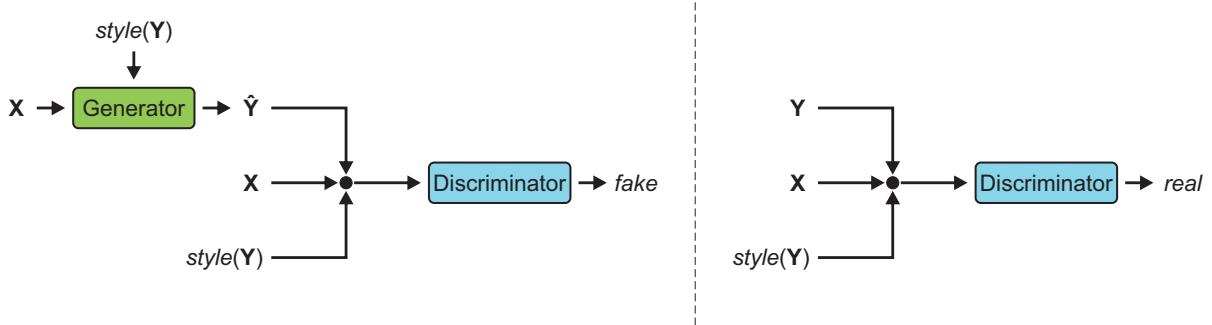


Figure 7.4: Training process of pix2pix with added style information. GAN discriminators are always trained on both real and synthetic images, to learn how to distinguish between them. X : input skeleton, Y : real output image, \hat{Y} : synthetic output image. *Left*: The training step for the generator and for the discriminator on synthetic images. *Right*: The training step for the discriminator on real images.

start to discriminate by content instead. We ensure that this is, in fact, the case by utilizing the previously mentioned max pooling.

7.3 Evaluation

7.3.1 Dataset

As we wanted the output of our network to be as realistic as possible, we used the CVL dataset [KFD⁺13], which contains images of real handwriting, as our training output. For training input, we converted the CVL dataset to skeletons using the skeletonization network from Chapter 4. Further, we also used the CVL dataset for the style input, effectively converting the network to an autoencoder during the training phase.

7.3.2 Implementation Details

In our first version, we took the same layer and feature configuration as the original pix2pix network for both the style extraction and image generation. However, we quickly realized that this is not suitable for skeleton-to-image transfer. The original configuration of the network is 8 layers deep, with its layers having a feature size of 64 (first layer) to 512 (deepest layer). These are way too many features for a simple encoding of skeletonized handwriting, leaving the network

with a lot of extra capacity, which it used to overfit on our training data. The resulting network was able to reproduce the original images exactly, but completely bypassed the style transfer step, and instead *remembered* which skeleton belongs to which style. Obviously, the network didn't work on the evaluation dataset.

We overcame this problem by re-implementing the U-Net blocks of the network with an asymmetric version. This means we can now change the encoder and decoder feature size separately. We then shortened the network down to 4 layers and reduced the number of encoder features. The configuration that ended up working the best utilized the following feature sizes: [16, 32, 64, 64] for the encoder, [32, 64, 128, 128] for the style extractor and [192, 256, 128, 64] for the decoder. This prevented the network from overfitting and we were quite happy with the results.

The obvious concern when reducing the depth of a network is whether its perceptual range is still sufficient, but we did not find any indication that it was detrimentally diminished.

The modified network is available under [\[Stua\]](#).

7.4 Results

The trained pen style transfer works really well, and a short demonstration of its output quality can be seen in [Table 7.1](#).

It is noteworthy, as already mentioned earlier, that the network learned to correct for small mistakes in the skeletons, as seen in [Figure 7.5](#). This is excellent, as it might mitigate some of the artifacts generated in the writer style transfer step.

As we couldn't find any obvious failure modes, were very satisfied with the results of this pipeline stage.

Style Input	Output
closed suddenly	pinking
Lines, Thangles	pinking
the deep and dark	pinking
much in Foseln	pinking
a rebel's whore	pinking
Skeleton Input	
	pinking

Table 7.1: Qualitative evaluation of the pen style transfer. The style inputs are from the evaluation dataset, the network has not seen them during the training phase. Note that the network does not only adjust the color, but also the width of the lines as well as geometric details and imperfections.



(a) skeleton input

(b) network output

Figure 7.5: Error correction properties of the pen style transfer. Note that the horizontal line of the *t* is broken in the skeleton image, but the network managed to correct that problem. Further, the network realized that the additional line at the *t* is an artifact of the skeletonization or writer style transfer step, and reduced its strength to a more realistic level.

Chapter 8

Background Style Transfer

In this chapter, we explore the possibilities of extending the pen style transfer to include the background image. The reason why we excluded the background so far is mainly that the background requires a lot more global information than the pen strokes, which mostly depend on their local surroundings. This makes the pen strokes easily synthesizable with a shallow network, as demonstrated in the previous chapter. The background, however, requires global consistency of not just the style, but also of the generated shapes themselves.

During our research, we attempted to transfer the background style using three different methods. First, we tried to extend the network of [Chapter 7](#) to increase its capacity for the additional background information. Secondly, we utilized [SPADE](#), a network specifically built for conditional image style transfer. Finally, we tried to break the problem down into multiple steps in an attempt to solve each challenge separately.

8.1 Dataset

To create a dataset that includes background images, we merged the images of the CVL dataset with stock images of paper. To get more variety, we added random shifts in brightness, contrast, hue and saturation of both the CVL and the background images. Then, we applied a random scaling to the background, followed by a random transformation and cropping of both the foreground and the background. This resulted in a dataset that should cover most use cases and encourage generalization.



Figure 8.1: Results of the pix2pix background style transfer. While the network managed to capture both the pen and background color, its outputs are dull and lack detail.

8.2 Conditional Pix2Pix

The network of [Chapter 7](#) was tuned to converge on CVL Images and therefore wasn't meant to have the required depth or feature width to include the background image. Reducing its capacity was necessary to prevent overfitting but should create obvious overgeneralization problems as soon as the background is included.

Nonetheless, we decided to train it anyway, to have a baseline to compare against. Surprisingly, the results were not as bad as expected and can be seen in [Figure 8.1](#). It successfully learned to reproduce the pen style, although with a lot less detail than when it was trained on white backgrounds. This makes sense as the style features now had to be used to represent both the foreground and the background. Additionally, as the dataset included color transformations, it had a lot more variety than the original CVL dataset. This encouraged a more aggressive generalization during the network training process, which might have contributed to the loss of detail.

While the network did learn to reproduce the color of the background, almost all of its details were lost. Similar to the pen strokes, this is most likely based on the fact that the number of style features of the network is too small.

As an implementation detail, we found that the network only converged when being trained with a fairly large batch size. To be specific, while we were able to train the network with 16 samples per batch, training attempts with a batch size of one or two were dominated by artifacts. The same is to be said about the image synthesis, which created large artifacts if run with other batch sizes than the network was trained on. This is most likely caused by the batch normalization stages, though further studies are required to gain a deeper understanding of that phenomenon.

Style Input	
Output	

Figure 8.2: Results of the **SPADE** network. While the network was able to synthesize quite realistic backgrounds, it never learned to match the original text color.

8.2.1 Modifications to the network

In an attempt to improve the results, we decided to reconfigure the network to be more suited for the current task. As the background requires a lot more global information than the pen strokes, we increased the depth and the feature count of both the decoder and style extractor. The encoder still only encounters skeleton images, and therefore we left the encoder at 4 layers, with skip connections to the 4 uppermost layers of the decoder.

To give the network more control over the style input, we decided to merge the style information into all layers of the decoder network, not just the innermost layer. This should give the network the power to decide at what depth it wants to use which style information.

Sadly, we weren't able to successfully train the network, as we didn't find a converging configuration. All our attempts were dominated by major artifacts without any sign of improvement. As the reconfiguration increased the size of the network, we were forced to reduce the number of samples per batch again, which might be partially responsible for our results. Nonetheless, further research might solve this problem, which makes this approach a potential candidate for future work.

8.3 SPADE

At the time of writing, NVidia just released the **SPADE** network [PLW⁺19], which is built to generate photorealistic images from a style reference photo and a segmentation map. Contrary to our attempt, the **SPADE** network does not add the style information by concatenation with the feature vectors but instead uses it to control the normalization parameters of the network, which

seems to give the network the ability to learn the styles of the different objects separately and with more detail.

As the network internally downsamples the segmentation map multiple times, our skeleton images are incompatible with it, as all strokes with a diameter of a single pixel would disappear during downsampling. To solve that problem, we applied morphologic dilation to the skeleton images until the lines were thick enough to be picked up by the network. To be specific, we used a circular dilation with a radius of 6 pixels.

8.3.1 Results

[SPADE](#) was able to understand and recreate the backgrounds of the images better than our modified [pix2pix](#), as seen in [Figure 8.2](#). It managed to include some detail, especially if the background included some degree of noise. Nonetheless, it wasn't able to reproduce high-frequency details, like thin lines, which are quite common on paper. This is similar to the problems we encountered earlier and is a more general problem with [CNNs](#) than with this specific network.

Sadly, we never got it to converge on the color of the pen strokes. This might be based on the fact that the pen strokes only take up a very small portion of the image compared to the background, reducing the priority of the pen strokes far enough that they don't get included in the style information. This could possibly be mitigated by modifying the loss function and should be researched further in future work.

8.4 Multi Step Approach

As the previous attempts failed, we decided to split the task into sub-problems that are either already solved, or relatively simple. In theory, if we managed to extract the foreground from the background, we could apply the pen style transfer algorithm of [Chapter 7](#) to the foreground and then merge it back into the background.

We therefore propose a pipeline with the following steps, as seen in [Figure 8.3](#):

1. Extracting the foreground, the actual handwriting, from the input image
2. Creating a foreground mask from the foreground image by thresholding and dilation

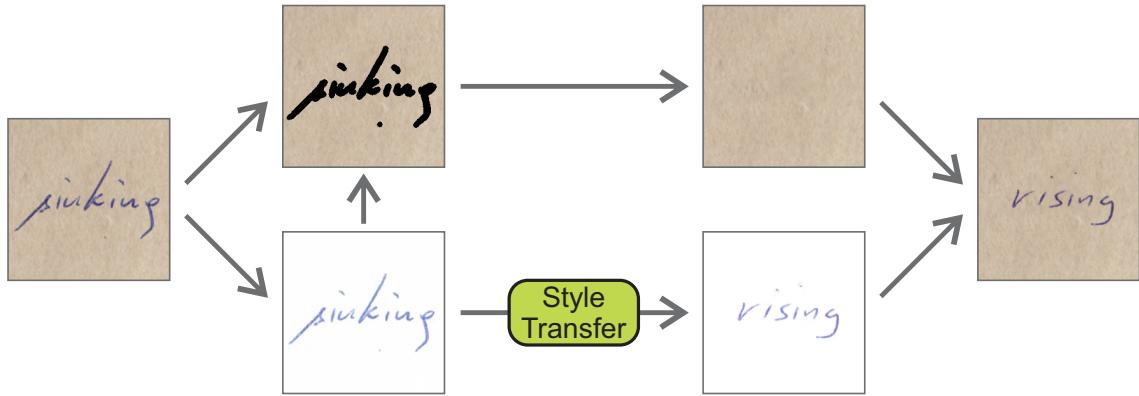


Figure 8.3: The multi-step approach for the background style transfer. First, the foreground of the input image is extracted. The extracted foreground image is then removed from the background, leaving holes behind. Next, the background is reconstructed with a hole inpainting algorithm. In the meantime, the extracted foreground is passed into the style transfer pipeline of the previous chapters to create the synthetic text image. Finally, the foreground and the background are merged back together.

3. Removing the masked regions from the background image, followed by an inpainting of the remaining holes
4. Running the existing style transfer pipeline on the foreground image
5. Merging the result of the style transfer pipeline back into the background image

Foreground extraction. Training a foreground extractor was straight forward, as this problem is similar to the skeletonization problem described in [Chapter 4](#), for which we achieved satisfying results by utilizing the [pix2pix](#) network. The difference was that we needed a new training dataset, which we were able to generate by storing both the foregrounds, backgrounds and merged images separately during the generation of the dataset of [Section 8.1](#).

Hole filling. With the foreground extracted, we were able to create a binary mask and remove it from the background, leaving holes. Filling the holes back in was a non-trivial task, and we decided to utilize NVidia's hole filling network [[LRS⁺18](#)], which is based on partial convolutions, a concept also developed by NVidia. [[LSW⁺18](#)] As they did not publish the code of their image inpainting demo, we decided to implement it ourselves [[Stuc](#)]. This gave us a background that is not just style transferred from the original image, but mostly identical.

Merging. Merging the result of the network back into the images was also a non-trivial step. To get a result that looks identical to the original image, it was important that the merging was

equivalent to undoing the foreground extraction step. As the foreground extraction was trained on a synthetic dataset, the merging operation had to be identical to the one used during dataset generation. While most certainly more sophisticated algorithms exist, in our case we merged with a simple inverted addition:

$$c_{out} = 255 - ((255 - c_{foreground}) + (255 - c_{background})) \quad (8.1)$$

$$c_{out} = c_{foreground} + c_{background} - 255 \quad (8.2)$$

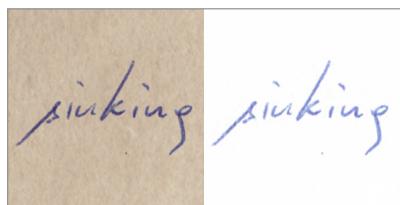
As the output is supposed to be a color value between 0 and 255, we then clipped the result into the desired range.

8.4.1 Results

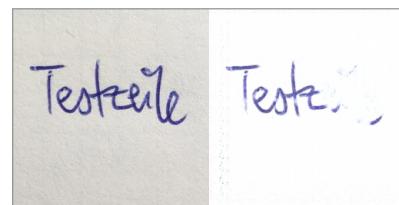
While our approach is promising, we cannot draw a definite conclusion at this point. Multiple issues need to be resolved before a real evaluation can take place.

For one, the style transfer pipeline of the previous chapters was only trained on CVL-like images so far and is therefore not capable of generating all the possible styles we have in the background transfer dataset. This is caused by the random color transformations we used during dataset generation and would require a re-training of the pen style transfer stage.

Further, the foreground extraction we trained did fairly well on our synthetic dataset, but performed poorly on real-world data, as seen in Figure 8.4. As the network itself is most certainly capable of solving this task properly, the reason for the poor behavior is most likely the dataset itself, and finding better alternatives for the dataset generation would be one of the first improvements for future work.



(a) Foreground extraction from synthetic data



(b) Foreground extraction from real data

Figure 8.4: Problems with the foreground extraction on real data. While the network performs flawlessly on the test dataset, it seems to struggle with real world data. This indicates that some crucial detail of our dataset is different from real world data.

Chapter 9

Future Work

9.1 Skeletonization and Sampling

The skeletonization and sampling steps are the most important ones that determine the quality of the writer style transfer. Yet, they are currently the weakest links, as they produce a number of artifacts.

Multiple possible approaches might further improve their quality. For one, the thresholding of the blurred skeletons is still one of the biggest flaws of the entire pipeline and should at some point be replaced with something more sophisticated, like an iterative solver or a more refined neural network.

Furthermore, instead of separating skeletonization and sampling into two separate steps, we could use a modified Graves' network trained on real online handwriting to do the skeletonization and sampling simultaneously, as it already has the prior knowledge of how letters should look like and in what order the strokes are most likely to be written. This might require adding a [CNN](#) as a side-input to Graves' network and might prove challenging, but if it works, it could yield higher quality results and would be one step closer to an end-to-end network.

9.2 Background Style Transfer

The results in [Section 8.2](#) and [Section 8.3](#) clearly show that networks are capable of reproducing both the background- and the pen-style, however, we never succeeded in completing both tasks at the same time. Nonetheless, we are confident that it is possible and will certainly be achieved in future studies.

Nonetheless, there are alternatives that might be just as viable. Instead of creating a network that does the style transfer in one pass, it might prove a lot easier to find an iterative solution, like the artistic style transfer algorithm of Gatys et al. [[GEB15](#)].

All in all, the background style transfer is a problem with a lot of potential for future research.

9.3 Direct offline-to-offline style transfer

The central goal of this work was to augment an online-to-online style transfer network to operate with offline data. While we accomplished this goal, the results could still be improved. It might be worth investigating whether skipping the entire online part of the pipeline could improve the results further.

To achieve that, we could combine a conditional [RNN](#) with [CNNs](#) at both the input and output side and train it to directly transfer the style of offline images to new offline images. It would most likely still make sense to skeletonize the offline images first, as this step proved quite robust and removed a lot of input and output variation. This would skip the thresholding, the generation of synthetic online data and the online style transfer, which are the three steps responsible for the majority of artifacts in our output images.

However, the question of whether such an approach would even converge, let alone produce viable results, still remains an open question and requires further research.

Chapter 10

Conclusion

In this thesis, we investigated the possibilities of extending existing online-to-online handwriting style transfer algorithms to offline handwriting. We achieved that by breaking the problem down into several sub-problems which together formed a multi-stage pipeline.

After a short overview of the theoretical background of this thesis in [Chapter 2](#), we gave a brief overview of the proposed pipeline in [Chapter 3](#), outlining the tasks and challenges of each stage. We then continued by describing the stages in more detail.

In [Chapter 4](#), we looked into the possibilities to extract the strokes from images of handwritten text into skeleton images. To accomplish that, we introduced a novel algorithm for transferring existing traditional algorithms to neural networks and used it to convert a threshold-based naive skeletonization algorithm to the `pix2pix` [[IZZ⁺16](#)] network, greatly improving the quality and robustness of the skeletonization.

In [Chapter 5](#), we tackled the problem of converting a skeleton image into a meaningful online representation. We did not utilize neural networks for that step and hand-crafted an algorithmic solution instead. First, we converted the skeleton image to a graph by simply connecting neighboring pixels. This was followed by several graph cleanup steps, to remove clusters, intersections and cycles, until all sub-graphs represented disjoined directed acyclic strokes. We then investigated several ways to resample those strokes to both increase their predictability and reduce the required computation power in further processing. We found the best solution to be maximum acceleration resampling, a complex resampling method that we introduced, based on the physical properties of real handwriting.

In Chapter 6, we evaluated the generated online data on multiple existing style transfer networks and found that Graves' network [Gra13] produced the best results on our data.

In Chapter 7, we converted the online data produced by Graves' network back into skeletons, followed by a visual style transfer from the input image. In this chapter, we reduced the problem down to images of colored pens on a white background. Again, we used pix2pix as the base network and extended it to include a style input. This turned out to be a success, yielding realistic images with great detail.

Finally, in Chapter 8, we attempted to extend the image style transfer to include a more realistic background. We proposed several solutions, but none of them were completely satisfactory. Nonetheless, the methods showed various degrees of success and should serve as a good basis for future research.

Overall, we consider the results of this thesis a success. We gained valuable insights into how neural networks perceive human handwriting and what the challenges and possibilities are. Along the way, we developed novel solutions to the overcome the problems we faced. Some of those solutions, like the method that allowed us to transfer knowledge from naive algorithms to neural networks, are not limited to the scope of this thesis but contribute to other research areas as well.

While many questions still remain open and inspire more research, our results demonstrate the potential of offline to online handwriting conversion and should serve as a solid base for future work.

Glossary

bow beginning-of-word 49, 50

CNN Convolutional Neural Network v, 2, 3, 5, 6, 11, 13, 16, 21, 30, 58, 59, 66, 71, 72

CVRNN Conditional Variational Recurrent Neural Network 49

CycleGAN The ‘CycleGAN’ Convolutional Adversarial Network [ZPI⁺17] 15, 18, 19, 25

DeepWriting The ‘DeepWriting’ Recurrent Neural Network [APH18] 46, 49–52

eoc end-of-character 49–51

GAN Generative Adversarial Network 2, 3, 6, 8, 17–19, 25, 59, 60

LSTM Long Short-Term Memory 8, 9, 46

MSE Mean Square Error 7, 17

pix2pix The ‘pix2pix’ Convolutional Adversarial Network [IZZ⁺16] v, 2, 15–18, 20, 23, 24, 26, 27, 29, 57–60, 64, 66, 67, 73, 74

RNN Recurrent Neural Network v, 3, 8, 9, 11, 46, 72

SPADE The ‘Spatially-Adaptive Denormalization Network’ [PLW⁺19] 63, 65, 66

List of Figures

2.1	Schematic of a fully connected network	5
2.2	Example of a CNN architecture	6
2.3	Photorealistic human faces generated by a GAN	8
2.4	Schematic of an RNN	9
2.5	Schematic of an LSTM cell	9
3.1	Overview of the style transfer pipeline	12
4.1	Skeletonization	15
4.2	Pix2pix examples	16
4.3	U-Net network layout	17
4.4	CycleGAN loss functions	19
4.5	Knowledge transfer from a naive mapping algorithm to a neural network	20
4.6	Finding the optimal blur reversal thresholding value	22
4.7	Information flow in a pix2pix network	24
4.8	Dominant failure mode of the CycleGAN based skeletonization	25
4.9	Typical failure modes of the primitive skeletonization	27
4.10	Results of the inverse training of the skeletonization	28
4.11	Error correction by generalization	29
4.12	Qualitative comparison between primitive and learned skeletonization	29
4.13	Typical failure modes of the final skeletonization network	30
5.1	Approximation of an online representation	33
5.2	Conversion of a skeleton image to strokes	34
5.3	All possible triangle group configurations in a 2×2 square	35
5.4	Comparison between resampling methods	40
5.5	Typical failure modes of the online approximation	43

6.1	Writer style transfer	45
6.2	Synthetic handwriting generated by Graves' network	46
6.3	Graves' prediction network architecture	47
6.4	Graves' synthesis network architecture	48
6.5	The position of Graves' attention windows	50
6.6	Comparison between handwriting style transfer methods	52
6.7	Typical failure modes of the writer style transfer	55
7.1	Pen style transfer	57
7.2	Extracting style information from an image	58
7.3	Modified <code>pix2pix</code> generator network for conditional style transfer	59
7.4	Training process of <code>pix2pix</code> with added style information	60
7.5	Error correction properties of the pen style transfer	62
8.1	Results of the <code>pix2pix</code> based background style transfer	64
8.2	Results of the <code>SPADE</code> network	65
8.3	The multi-step approach for the background style transfer	67
8.4	Problems with the foreground extraction on real data	69

List of Tables

5.1	Comparison between different sorting methods	41
6.1	Performance evaluation of real vs synthesized training data	53
6.2	Qualitative evaluation of the writer style transfer	54
7.1	Qualitative evaluation of the pen style transfer	62

Bibliography

- [Aly] Zaid Alyafeai. Pix2pix. URL: <https://zaidalyafeai.github.io/pix2pix/pokemon.html> (visited on 07/2019) (cited on page 16).
- [APH] Emre Aksan, Fabrizio Pece, and Otmar Hilliges. DeepWriting. URL: <https://github.com/emreaksan/deepwriting> (visited on 07/2019) (cited on page 49).
- [APH18] Emre Aksan, Fabrizio Pece, and Otmar Hilliges. DeepWriting: Making Digital Ink Editable via Deep Generative Modeling. In *SIGCHI Conference on Human Factors in Computing Systems*, CHI '18, Montréal, Canada. ACM, 2018 (cited on pages 2, 46, 75).
- [Del] François Deloche. Recurrent neural network - Wikipedia. URL: https://en.wikipedia.org/wiki/Recurrent_neural_network (visited on 08/2019) (cited on page 9).
- [GEB15] Leon A. Gatys, Alexander S. Ecker, and Matthias Bethge. A neural algorithm of artistic style. *CoRR*, abs/1508.06576, 2015. arXiv: [1508.06576](https://arxiv.org/abs/1508.06576) (cited on page 72).
- [GH89] Zicheng Guo and Richard W. Hall. Parallel Thinning with Two-subiteration Algorithms. *Commun. ACM*, 32(3):359–373, March 1989. ISSN: 0001-0782. DOI: [10.1145/62065.62074](https://doi.org/10.1145/62065.62074) (cited on page 23).
- [Gra13] Alex Graves. Generating Sequences With Recurrent Neural Networks. *CoRR*, abs/1308.0850, 2013. arXiv: [1308.0850](https://arxiv.org/abs/1308.0850) (cited on pages 2, 13, 46–48, 50, 74).
- [Hes] Christopher Hesse. Image-to-Image Demo. URL: <https://affinelayer.com/pixsrv/> (visited on 07/2019) (cited on page 16).
- [IZZ⁺16] Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, and Alexei A Efros. Image-to-Image Translation with Conditional Adversarial Networks. *arxiv*, 2016 (cited on pages 2, 16, 17, 23, 27, 73, 75).

- [KFD⁺13] Florian Kleber, Stefan Fiel, Markus Diem, and Robert Sablatnig. CVL-DataBase: An Off-Line Database for Writer Retrieval, Writer Identification and Word Spotting. In *Proceedings of the 2013 12th International Conference on Document Analysis and Recognition*, ICDAR '13, pages 560–564, Washington, DC, USA. IEEE Computer Society, 2013. ISBN: 978-0-7695-4999-6. DOI: [10.1109/ICDAR.2013.117](https://doi.org/10.1109/ICDAR.2013.117) (cited on pages 21, 25, 50, 57, 60).
- [KLA18] Tero Karras, Samuli Laine, and Timo Aila. A style-based generator architecture for generative adversarial networks. *CoRR*, abs/1812.04948, 2018. arXiv: [1812.04948](https://arxiv.org/abs/1812.04948) (cited on page 8).
- [LB05] Marcus Liwicki and Horst Bunke. IAM-OnDB - an On-Line English Sentence Database Acquired from Handwritten Text on a Whiteboard. In *8th Intl. Conf. on Document Analysis and Recognition*, volume 2, pages 956–961, 2005 (cited on pages 21, 25, 41, 50, 52).
- [LBB⁺98] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, November 1998. ISSN: 0018-9219. DOI: [10.1109/5.726791](https://doi.org/10.1109/5.726791) (cited on pages 6, 16).
- [LRS⁺18] Guilin Liu, Fitsum A. Reda, Kevin J. Shih, Ting-Chun Wang, Andrew Tao, and Bryan Catanzaro. Image Inpainting for Irregular Holes Using Partial Convolutions. In *The European Conference on Computer Vision (ECCV)*, 2018 (cited on page 67).
- [LSD14] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully Convolutional Networks for Semantic Segmentation. *CoRR*, abs/1411.4038, 2014. arXiv: [1411.4038](https://arxiv.org/abs/1411.4038) (cited on page 16).
- [LSW⁺18] Guilin Liu, Kevin J. Shih, Ting-Chun Wang, Fitsum A. Reda, Karan Sapra, Zhiding Yu, Andrew Tao, and Bryan Catanzaro. Partial Convolution based Padding. In *arXiv preprint arXiv:1811.11718*, 2018 (cited on page 67).
- [PLW⁺19] Taesung Park, Ming-Yu Liu, Ting-Chun Wang, and Jun-Yan Zhu. Semantic Image Synthesis with Spatially-Adaptive Normalization. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2019 (cited on pages 65, 75).
- [RFB15] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-Net: Convolutional Networks for Biomedical Image Segmentation. *CoRR*, abs/1505.04597, 2015. arXiv: [1505.04597](https://arxiv.org/abs/1505.04597) (cited on page 17).

- [Stua] Martin Stumpf. CycleGAN and pix2pix in PyTorch. URL: <https://github.com/Finomnis/pytorch-CycleGAN-and-pix2pix> (visited on 07/2019) (cited on pages 27, 61).
- [Stub] Martin Stumpf. Handwriting Synthesis with RNNs. URL: <https://github.com/finomnis/handwriting-synthesis> (visited on 07/2019) (cited on pages 41, 50).
- [Stuc] Martin Stumpf. inpaint-pconv. URL: <https://github.com/Finomnis/inpaint-pconv> (visited on 08/2019) (cited on page 67).
- [Stud] Martin Stumpf. MasterThesis-FakeHandwriting. URL: <https://bitbucket.org/Finomnis/masterthesis-fakehandwriting/> (visited on 08/2019) (cited on pages 2, 40).
- [Tea] The AlphaStar Team. AlphaStar: Mastering the Real-Time Strategy Game StarCraft II. URL: <https://deepmind.com/blog/article/alphastar-mastering-real-time-strategy-game-starcraft-ii> (visited on 08/2019) (cited on page 1).
- [Vas] Sean Vasquez. Handwriting Synthesis with RNNs. URL: <https://github.com/sjvasquez/handwriting-synthesis> (visited on 07/2019) (cited on pages 46, 50).
- [Zhu] Jun-Yan Zhu. CycleGAN and pix2pix in PyTorch. URL: <https://github.com/junyanz/pytorch-CycleGAN-and-pix2pix> (visited on 07/2019) (cited on page 27).
- [ZPI⁺17] Jun-Yan Zhu, Taesung Park, Phillip Isola, and Alexei A Efros. Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks. In *Computer Vision (ICCV), 2017 IEEE International Conference on*, 2017 (cited on pages 18, 75).
- [ZS84] T. Y. Zhang and C. Y. Suen. A Fast Parallel Algorithm for Thinning Digital Patterns. *Commun. ACM*, 27(3):236–239, March 1984. ISSN: 0001-0782. DOI: [10.1145/357994.358023](https://doi.org/10.1145/357994.358023) (cited on page 26).