ingenieur
wissenschaften
htw saar

Hochschule für
Technik und Wirtschaft
des Saarlandes
University of
Applied Sciences

STL SYSTEMTECHNIKLABOR

Reverse Engineering The Cauzin Softstrip

Michael Reimsbach

Technical Report – STL-TR-2018-03 – ISSN 2364-7167

The Softstrip System Reader
Enter a New World of Information: Print that Your Computer Can Read!
FREE! ACCESSORY KIT
StripWare
PORTFOLIO EVALUATION
THE BASIC APPLE
STAR GAZING
THE FINANCIAL ADVISOR
CLASSIC GAMES
SECOND GIANT BOOK OF COMPUTER GAMES
MacART FOR BUSINESS

stl.htwsaar.de

Technische Berichte des Systemtechniklabors (STL) der htw saar
Technical Reports of the System Technology Lab (STL) at htw saar
ISSN 2364-7167

Michael Reimsbach: Reverse Engineering The Cauzin Softstrip
Technical report id: STL-TR-2018-03

First published: October 2018
Last revision: October 2018
Internal review: John Aycock, André Miede

For the most recent version of this report see: https://stl.htwsaar.de/

Title image source: John Aycock

htw saar – Hochschule für Technik und Wirtschaft des Saarlandes (University of Applied Sciences)
Fakultät für Ingenieurwissenschaften (School of Engineering)
STL – Systemtechniklabor (System Technology Lab)
Prof. Dr.-Ing. André Miede (andre.miede@htwsaar.de)
Goebenstraße 40
66117 Saarbrücken, Germany
https://stl.htwsaar.de

UNIVERSITY OF CALGARY

Hochschule für Technik und Wirtschaft des Saarlandes
University of Applied Sciences

Master's Thesis

A thesis submitted in partial fulfillment of the requirements for the degree of
Master of Science (M. Sc.)
at Hochschule für Technik und Wirtschaft des Saarlandes
in Applied Informatics

Reverse Engineering The Cauzin Softstrip

submitted by
Michael Reimsbach

supervised by
Prof. Dr.-Ing. André Miede
Prof. Dr. John Aycock

University of Calgary, Alberta (Canada), 2018-10-31

Declaration

I declare that this thesis has been composed solely by myself and that it has not been submitted, in whole or in part, in any previous application for a degree. Except where states otherwise by reference or acknowledgement, the work presented is entirely my own.

University of Calgary, Alberta (Canada), 2018-10-31

Michael Reimsbach

Abstract

The Cauzin Softstrip is one of the first two-dimensional bar codes and was released in 1985 by Cauzin Systems. It could be used for encoding all kind of digital data such as software, text, or graphics. The bar codes were printed on paper and served as an inexpensive distribution medium compared to floppy disks. Unfortunately, the Softstrip was not as successful as anticipated and disappeared with the Cauzin Softstrip Reader around 1987. Softstrips can be found in old computer magazines, computer science books or as scanned collections on the internet. However, they are useless without access to a Cauzin Softstrip Reader. Obtaining a working Softstrip Reader is already extremely difficult and will most likely be impossible in the following years. In order to preserve the encoded data, this work examines the bar code format and creates a digital Softstrip reader, making Softstrips accessible for people without an optical reader. Furthermore, it highlights the challenges while dealing with such an old bar code and shows which results can be expected, which might be helpful for similar projects in the future.

A decoding strategy with the use of a Convolutional Neural Network is presented and evaluated on a corpus of various Softstrips, including digital generated Sofstrips and scanned Softstrips from books, magazines and StripWares. The presented decoding strategy is able to decode 91% of the 1230 Softstrips in the corpus.

Zusammenfassung

Der Cauzin Softstrip ist einer der allerersten zwei-dimensionalen Barcodes. Er wurde von Cauzin Systems entwickelt und im Jahr 1985 veröffentlicht. Er konnte dazu genutzt werden um jegliche Art von digitalen Daten wie z.B. Software, Texte oder Grafiken zu enkodieren. Anschließend wurden die Barcodes auf Papier gedruckt und dienten, im Gegensatz zur Diskette, als kostengünstiges Austauschmedium. Bedauerlicherweise war der Cauzin Softstrip nicht so erfolgreich wie erhofft und verschwand bereits nach kurzer Zeit wieder. Man kann Softstrips noch immer in alten Computer-Magazinen, Informatik-Büchern oder als Sammlung im Internet finden. Allerdings sind diese nutzlos ohne einen Cauzin Softstrip Reader. Es ist bereits jetzt schwierig einen funktionierenden, gebrauchten Softstrip Reader zu finden und es wird in den kommenden Jahren nicht leichter werden, wenn nicht sogar unmöglich. Damit die enkodierten Daten nicht in Vergessenheit geraten, untersucht diese Thesis das Cauzin Softstrip Format und erstellt einen digitalen Softstrip Reader. Darüber hinaus werden die Herausforderungen aufgezeigt, die beim Umgang mit solch einem alten Barcode entstehen können. Außerdem wird gezeigt, welche Resultate erwartet werden können. Diese Erfahrungen können für ähnliche, zukünftige Projekte nützlich sein.

Es wird eine Dekodierungs-Strategie präsentiert, die auf einem Convolutional Neural Network aufbaut. Anschließend wird diese auf einer Softstrip-Sammlung evaluiert, die gescannte Softstrips und digital generierte Softstrips enthält. Die Scans stammen aus Büchern, Magazinen und StripWares. Die vorgestellte Dekodierungs-Strategie ist in der Lage 91% der 1230 zur Verfügung stehendenen Softstrips zu dekodieren.

You call this archaeology?

— Indiana Jones and the Last Crusade

Acknowledgment

I would like to express my gratitude to my supervisor Prof. Dr.-Ing. André Miede who encouraged me to conduct my research at the University of Calgary and who also arranged this collaboration.

I would also like to thank Prof. Dr. John Aycock who welcomed me in Calgary and introduced me to a world of almost forgotten computer relics. It was a pleasure to work with him and I have greatly benefited from his feedback during our meetings.

Contents

1 Introduction

1.1 Motivation

We are living in a society where computers have become ubiquitous. Any information is just a click away and communication around the globe is easier than ever before. There are different ways to encode information; one of them is by using bar codes. Two of the most successful bar codes today are the Universal Product Code (UPC) for identifying products and the Quick Response (QR) code which is used in many different areas such as manufacturing, health care and marketing [1].

This thesis focuses on an almost forgotten relic, the Cauzin Softstrip, a two-dimensional bar code format released in 1985 by Cauzin Systems. All kind of digital data, such as graphics, software or text files can be encoded as a Cauzin Softstrip and then be printed on paper (see figure 1.1). In order to understand the reasons behind the Softstrip, it is helpful to be familiar with the home computer revolution in the late seventies and early eighties.

The first microcomputer was arguably the Altair 8800 released by MITS in 1974 [2]. However, it was far away from what we call a Personal Computer (PC) today. First of all, it could be purchased as a build-it-yourself kit for under $400 and had to be assembled before it could be used. There was neither a keyboard nor a monitor. User input was entered by using switches on the front panel and the output was shown by multiple Light Emitting Diodes (LEDs). Obviously, the early home computers were designed for computer enthusiasts instead of the mass market. This changed, however, with the introduction of the 6502 8-bit microprocessor, developed by MOS Technologies, in 1975. Before, microprocessors were expensive: the Intel 8008, for example, was sold for $360 in 1974 [3]. In contrast, the 6502 was considerably less expensive and was sold for only $25 [4]. Soon many new home computers using the 6502 microprocessor (or variants of the 6502) were released in North America, such as the Apple II in 1977 [5], the Commodore PET in 1977 [6] and even the home video gaming console Nintendo Entertainment System in 1983 [7].

Even though PCs were becoming more affordable and targeting the mass market instead of computer enthusiasts, they still required a lot of technological knowledge. For instance, PC handbooks often included a section about BASIC programming, teaching users how to program their own software [5]. It was also common for computer magazines to release software in the form of source code for readers. In order to use the programs, people had to not only type in the code but also compile it in some cases. One can imagine that typing in the source code is a tedious task, where even a single mistake can result in an error. Finding the error can be even more time-consuming, especially if the source code was in machine language [8].

To make things even worse, storage devices were usually not part of a PC and had to be purchased separately by the user. This means typing in the source code had to be repeated every time a user wanted to use the program, unless they had access to a hard drive, cassette tape deck or floppy disk drive to save the program. However, external storage devices were expensive. For instance, the DISK II, a floppy disk system for the Apple II, was introduced in 1978 and could be pre-ordered for $495 [9]. The regular price after its launch was $595 [10]. Of course floppy disks were needed as well. The price for a box of 10 diskettes (5 1/4" and 8") ranged from $29 to $57 in 1980 [10]. That is when some companies came up with a new solution for storing digital data. They used bar codes.

One of the earliest bar codes for encoding software was the Paperbyte in 1977 [11]. The Paperbyte bar codes were printed in Paperbyte books and Byte magazines, where they encoded assembly programs. A few years later, in 1983, the Optical Scanning Reader (OSCAR) bar code was released for encoding software in bar code format in the Databar magazine (see section 4.1 for more details). Unfortunately, only a single issue of the Databar magazine was ever published [12].

The most successful bar code for encoding software from this era is probably the Cauzin Softstrip. The Softstrip encoding software, the Stripper, and the optical reader, the Cauzin Softstrip Reader, were available for about $220. It was compatible with the IBM-PC, Apple II and MacIntosh. The System was promoted with multiple demo Softstrips, among them John Romero's[1] game Bongo's Bash [13]. After its release, stores started selling computer programs as Softstrips. Besides that, Softstrips were advertised for several use cases [14]. For instance, they could be printed next to human readable data on paper, allowing people to scan the bar code instead of typing in the data into their computer. This not only saves time — scanning a single Softstrips takes about 30 seconds [15] — but also reduces the number of errors that could occur while typing the data in. For these reasons, computer magazines started adopting the Cauzin Softstrip and printed it next to their source code. Some of the magazines which published the Cauzin Softstrip are Byte [16], II Computing [17] and the inCider [18]. In addition, they can be found in computer science books, such as Animated Algorithms: A Self-teaching Course in Data Structures and Fundamental Algorithms [19].

The Softstrip was also interesting for publishers because it was an inexpensive medium for distributing data. Instead of buying expensive floppy disks and saving the data on them, it could simply be printed and then published. Another useful feature of the Cauzin Softstrip is that it enables users to transfer data between different operating systems. For example, a Softstrip could be created on an IBM PC and then be read again on an Apple II.

> *Fascinating, isn't it? Anything you can do with disks can be done with the Softstrip data system - faster, easier and at lower cost - ON PAPER*
> — Softstrip System: The new advantage in business computing [14]

Although the MacUser magazine awarded the Cauzin Softstrip as the most innovative concept in 1986 [20] and Cauzin Systems had plans to use the Cauzin Softstrip on cards such as credit or calling cards [21], the technology was not as successful as anticipated and eventually disappeared a few years after its release in 1985. One reason for the failure might be that floppy disks and modems became more affordable, therefore the need for bar codes as a distribution medium decreased. Additionally, the Softstrip is impractical for larger data. A single Softstrip can store up to 5500 bytes. If the data was larger, it had to be divided into multiple Softstrips. This requires not only the user to adjust the Softstrip

[1]John Romero is a programmer, game designer and co-founder of id Software. He worked on many revolutionary games such as Doom, Quake and Wolfenstein.

Reader again for each Softstrip, but also increases the scanning time [15]. Lastly, the Cauzin Softstrip certainly suffered a chicken-and-egg problem [22]. On the one hand, the limited number of available Softstrips did not attract enough customers to buy a Cauzin Softstrip Reader, whereas on the other hand magazines stopped publishing Softstrips because the target group was not large enough. As a result, Softstrips disappeared and Cauzin Systems stopped producing the Cauzin Softstrip Reader long ago, before the company finally disappeared as well.

1.2 Thesis Objectives

At the time of writing it is already difficult to find a working Cauzin Softstrip Reader. The only option to decode the Softstrips without access to a Cauzin Softstrip Reader is to do it by hand which is not only a time-consuming but also an error-prone task. A Softstrip had to be decoded manually for this thesis to locate a decoding error. This process took about eight hours until the checksum finally confirmed that the decoding was successful. In order to preserve the Softstrips and the encoded content, this thesis reverse engineers the Cauzin Softstrip and creates a digital Softstrip reader. During the development of the system, the challenges while dealing with scans of old bar codes are highlighted and appropriate strategies to overcome them are presented. The following questions must be answered at the end:

1. Is it possible to decode the Cauzin Softstrip with a digital reader?
2. What are the main difficulties while decoding the Cauzin Softstrip?
3. What are the limitations of the digital reader?

1.3 Overview

This thesis is divided into following chapters:

Chapter 2 explains the fundamental concepts of image processing and Convolutional neural networks (CNNs) which are necessary to understand the following chapters.

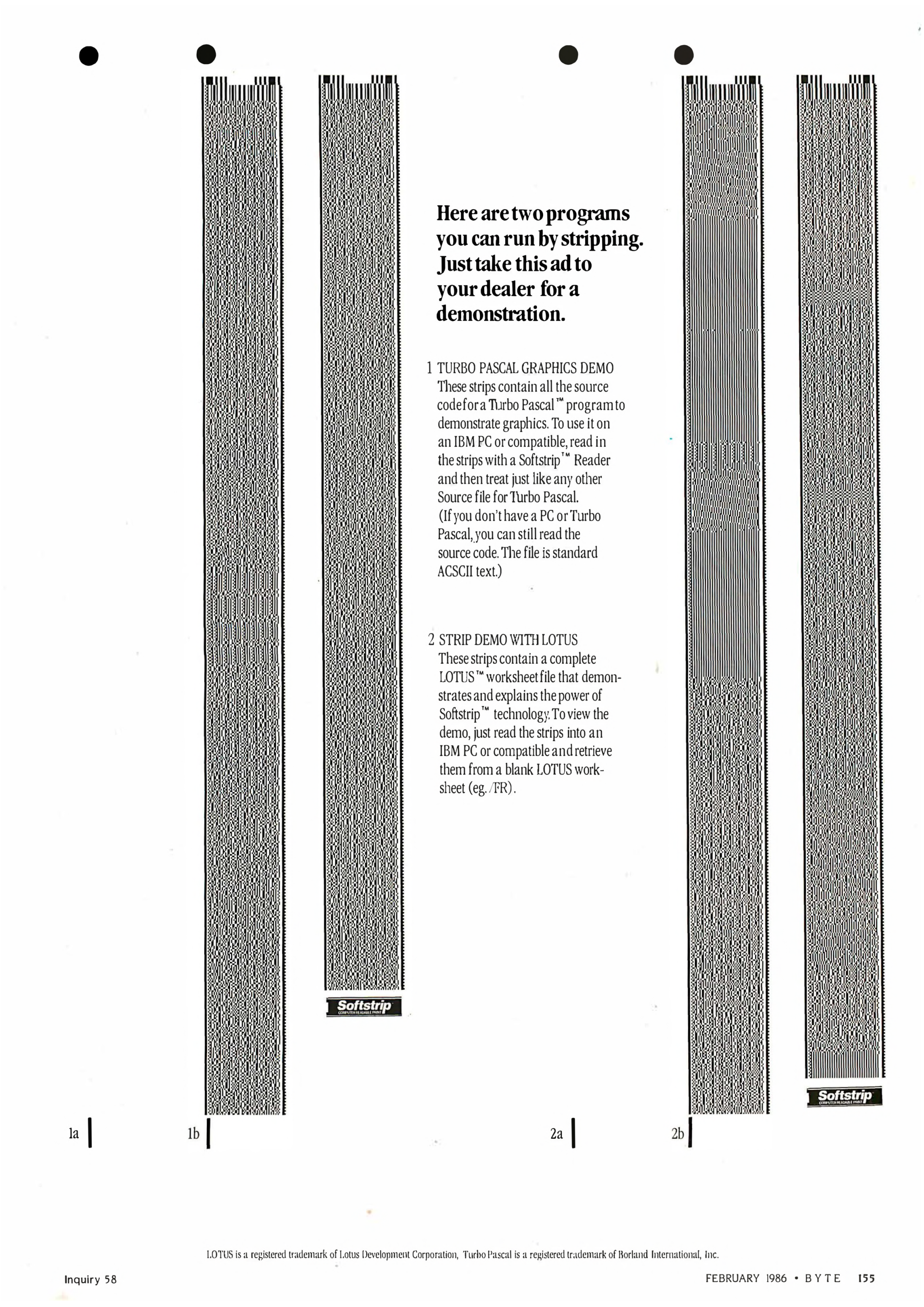
Chapter 3 gives an introduction to bar codes in general and explains the decoding process of the PDF-417 bar code. After that, the Cauzin Softstrip and the available Softstrip corpus is analyzed.

Chapter 4 presents the digital decoder for the OSCAR bar code and the CauzCoin challenge.

Chapter 5 presents the design of the digital Softstrip decoder. A general architecture is presented and a solution for each component is presented.

Chapter 6 shows that the system presented in the previous chapter can be implemented. The used technologies are presented and more implementation details are provided for each component.

Chapter 7 evaluates the implemented solution. The different decoding methods are compared with each other and their performance on the different datasets is evaluated. In addition, the runtime performance is evaluated and the limitations of the decoder are highlighted.



Here are two programs you can run by stripping. Just take this ad to your dealer for a demonstration.

1 TURBO PASCAL GRAPHICS DEMO
These strips contain all the source code for a Turbo Pascal™ program to demonstrate graphics. To use it on an IBM PC or compatible, read in the strips with a Softstrip™ Reader and then treat just like any other Source file for Turbo Pascal.
(If you don't have a PC or Turbo Pascal, you can still read the source code. The file is standard ACSCII text.)

2 STRIP DEMO WITH LOTUS
These strips contain a complete LOTUS™ worksheet file that demonstrates and explains the power of Softstrip™ technology. To view the demo, just read the strips into an IBM PC or compatible and retrieve them from a blank LOTUS worksheet (eg. /FR).

1a 1b 2a 2b

Softstrip

LOTUS is a registered trademark of Lotus Development Corporation, Turbo Pascal is a registered trademark of Borland International, Inc.

Inquiry 58

FEBRUARY 1986 • BYTE 155

Figure 1.1: Cauzin Softstrip Advertisement
Cauzin Softstrip Advertisement [23]

Chapter 8 explains what happened with the Cauzin Softstrip after is disappeared. It also discusses potential future work and finally draws a conclusion.

2 Fundamentals

This chapter covers the fundamental concepts of image processing (section 2.2) and CNNs (section 2.1) which are necessary to understand the following chapters in this thesis. Readers who are familiar with those topics might skip this chapter.

2.1 Convolutional Neural Networks

CNNs have become the state of the art method for solving computer vision tasks [24]. Although CNNs were already used in 1998 by LeCun et al. for recognizing handwritten digits, where they achieved remarkable results [25], the hype about CNNs started after Krizhevsky et al. won the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) with a CNN in 2012 [26]. The dataset in this challenge contained 10,000,000 labeled images with more than 10,000 object categories. Krizhevsky et al. classified the images with a top-5 test error rate of 15.3%, whereas the second best top-5 error rate achieved 26.2% [26]. Since then CNNs have been used for many different tasks such as learning how to play Atari games [27] or learning how to cook by watching YouTube videos [28].

One reason for their success is how they handle feature engineering. Feature engineering is a crucial step for the success of a classification technique where characteristic properties are selected to predict a class. For instance, a useful feature to recognize a Simpsons character might be the number of yellow pixels. However, in many cases it turns out to be difficult to find useful features which might result in a poor performing classifier. By contrast, CNNs are extremely good at learning useful features by themselves [29]. Additionally, they are able to learn a hierarchical representations of these features [30] and once the CNN has learned the different patterns, it is able to detect them at any location. For example, the first layer could scan the image for straight lines. The second layer could combine the previous lines to scan the image for shapes like rectangles and the following layer could use those shapes to detect a house.

Due to the aforementioned reasons, CNNs are used in this work for classifying dibits and extracting the rows from the bar code. This section explains first how feedforward networks work and then highlights the characteristics of CNNs.

2.1.1 Feedforward Networks

The name feedforward network comes from the fact that information can only flow in one direction, i.e., forward [24]. By contrast, a node's output can be fed into itself again in a Recurrent neural network (RNN) [24]. A simple feedforward network is the Single Layer Perceptron (SLP). Its structure is shown in figure 2.1. Each input variable x_i has a corresponding weight w_i. The variable x_0 is always set to 1 therefore w_0 functions as a constant term — the bias. The weighted sum is calculated with equation 2.1 and is the input for the activation function f (see equation 2.2). Activation functions are covered in

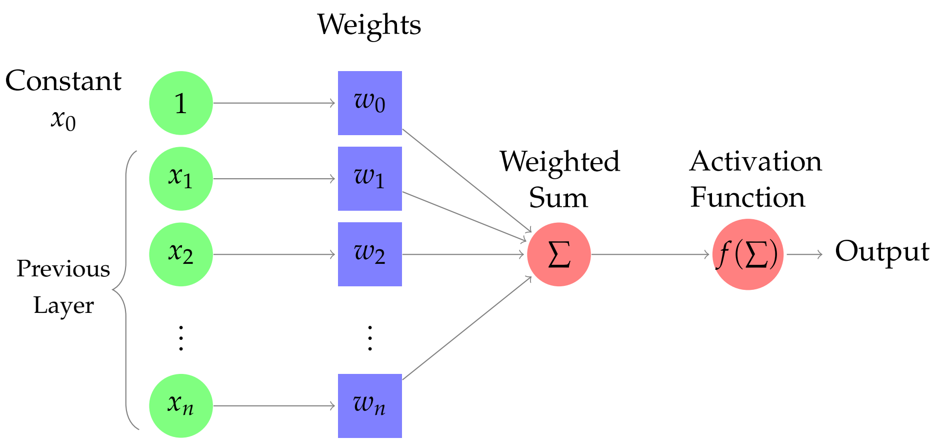


Figure 2.1: Single Layer Perceptron (SLP)

detail in section 2.1.2. Figure 2.3 shows two example activation functions.

$$z = \sum_{i=0}^{n} w_i x_i \tag{2.1}$$

$$y = f(z) \tag{2.2}$$

A typical Multilayer perceptron (MLP) is shown in figure 2.2. The first layer is the input layer which feeds the data into the network. Then, one ore more hidden layers follow. The number of hidden layers, also called *depth*, depends on the complexity of the task and is usually determined experimentally [29]. MLPs with many hidden layers are often called deep neural networks or deep learning models [24]. The final layer outputs the computed result for the given data.

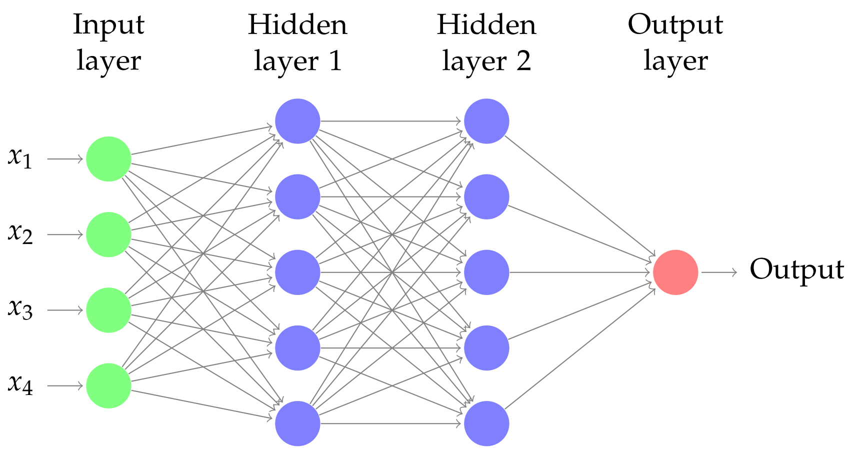


Figure 2.2: Fully connected neural network with two hidden layers.

2.1.2 Activation Function

Activation functions are usually non-linear functions used in single perceptrons to approximate complex functions [29]. A perceptron with a linear activation function is only able to perform linear transformations (see equation 2.1). As a result, it is not able to learn more complex models than a linear regression. Even a network with multiple linear perceptrons is not more powerful since it can be replaced by a single linear perceptron

[29]. The most popular activation function is the Rectified Linear Unit (ReLU) function (see figure 2.3). The reason for this is that it allows faster learning [31]. The last layer in a MLP outputs the final result, so the selected activation function in this layer directly affects the result. The selection depends on the task, for instance, if the task is to perform a binary classification, a sigmoid function is useful, whereas a softmax function is more appropriate for multiclass classification [29].

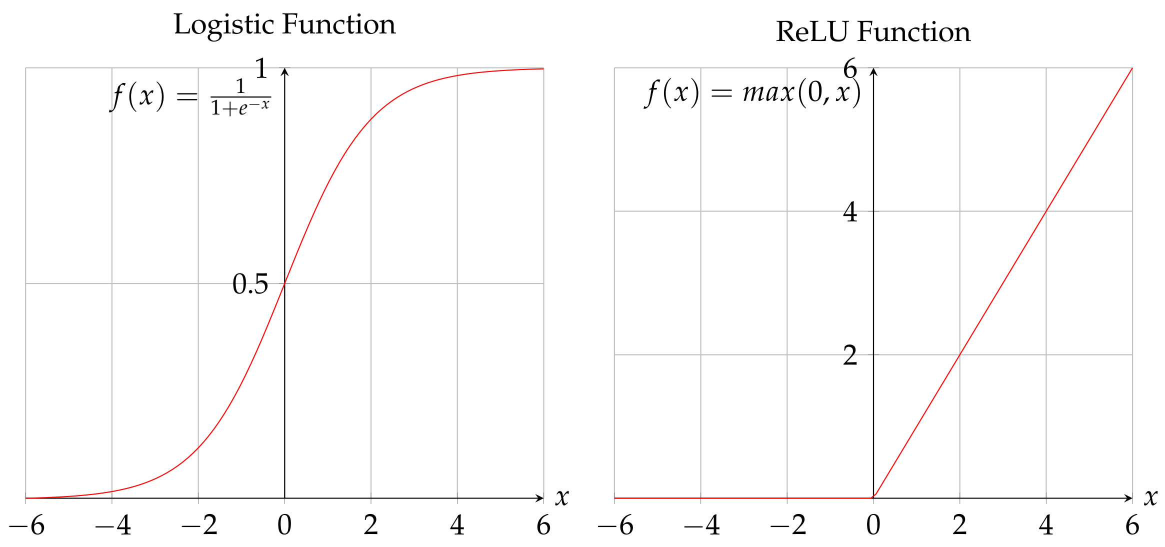


Figure 2.3: Activation Functions

2.1.3 Loss Function and Optimizer

Loss functions, also called error or cost functions, are used to measure the difference between the predicted value and the true value [24]. A popular loss function is the mean squared error (see equation 2.3), where Y_i is the predicted and $\hat{Y}_i$ the true value. Other loss functions exist and are chosen according to the task. The smaller the loss the better the network's performance, so the goal is to minimize the loss function. One way to achieve this is by using the optimization algorithm gradient descent which was originally developed by Augustin-Louis Cauchy in 1847 [32].

$$MSE(x) = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2 \tag{2.3}$$

This optimization process is also known as learning or the training. The gradient descent algorithm updates the parameters (weights) in each iteration to the direction of the negative gradient with a learning rate η which determines the stepping size to this direction. This is repeated until the algorithm converges to a (local) minimum [33]. The learning rate η has to be chosen carefully; if η is too large it can miss a (local) minimum. On the other hand, if η is too small, it requires a lot of iterations until a (local) minimum is reached. Other optimization algorithms such as Adam [34] and RMSProp [35] exist.

MLPs use the backpropagation algorithm to compute the gradients efficiently [36]. In the first step, a forward propagation is applied for computing the predicted values. The predictions are then compared to the true values and the error is calculated. This error is then back propagated in order to determine how much every perceptron contributed to the error. Therefore, the local gradients are calculated with equation 2.4 with l as the layer, k the perceptron on this layer, $w_{k,j}$ the weight from j to k, η the learning parameter and a the output of the activation function. Subsequently, the weights are updated with equation 2.5. The whole forward-backward process is repeated until either a defined number of iterations is reached or a desired error.

$$\delta_j^{(l-1)} = \sum_k w_{k,j}^{(l-1)} \delta_k^{(l)} a_j^{(l-1)} (1 - a_j^{(l-1)}) \tag{2.4}$$

$$w_{i,j}^{(l)} = w_{i,j}^{(l)} - \eta a_j^{(l)} \delta_i^{(l+1)} \tag{2.5}$$

2.1.4 The Convolution Operation

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n) K(i-m, j-n) \tag{2.6}$$

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n) K(i+m, j+n) \tag{2.7}$$

The discrete convolution operation is shown in equation 2.6. The inputs are an image I with dimensions i, j and a kernel K, also called filter, with dimensions m and n [24]. The result of the convolution operation is often called a feature map [24]. Equation 2.7 shows a very similar function called cross-correlation which is used in many machine learning libraries but also called convolution [24]. The main difference between the convolution and cross-correlation operation is that the kernel is flipped in the convolution operation (see figure 2.4).

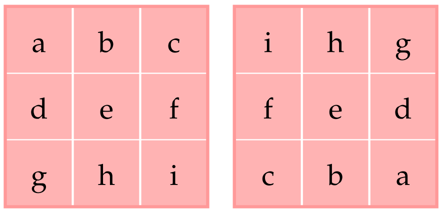


Figure 2.4: Two $3x3$ kernels. The right kernel is a flipped representation of the left kernel.

Figure 2.5 shows how the convolution operation without kernel flipping is applied. The blue matrix represents a $3x3$ image (in this example pixel values are either 1 or 0) and the red matrix represents a $2x2$ kernel. The kernel is then moved across the image with a defined stride. In this example the stride is one, which means the kernel is moved by one pixel in every step, and the kernel is always inside the image. However, it is also possible that parts of the kernel are outside of the image. In this case the image needs to be padded. Every time the kernel is moved, the dot product between the kernel and the overlapping image is calculated and inserted into the corresponding position in the green matrix. Figure 2.6 shows the result of a filter for detecting edges. Convolution is the key operation in CNNs and is used to learn the different features in an image [29].

2.1.5 Feature Map

Multiple kernels (or filters) are used to detect certain features. Each application of the convolution operation creates a different feature map from the same input image, which shows where the defined features were detected [29]. The kernel weights for detecting features are determined during the training process.

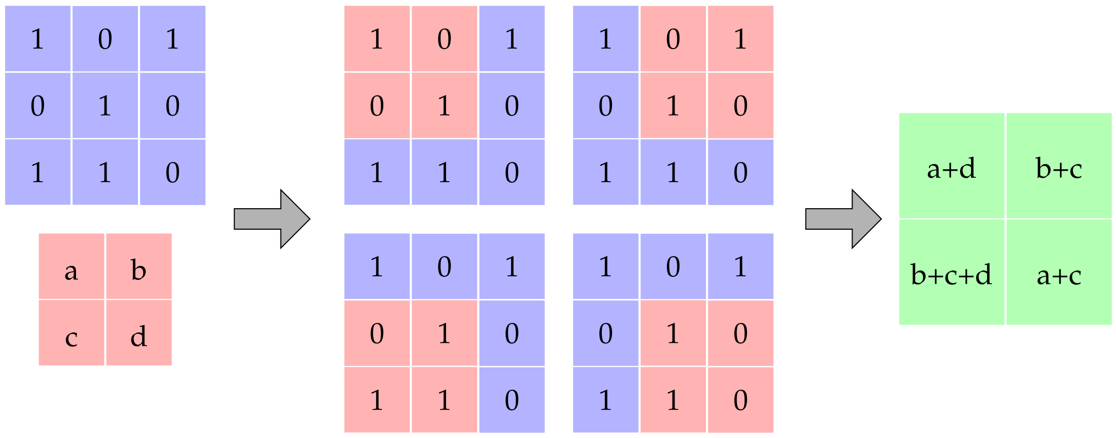


Figure 2.5: Example of the convolution operation.

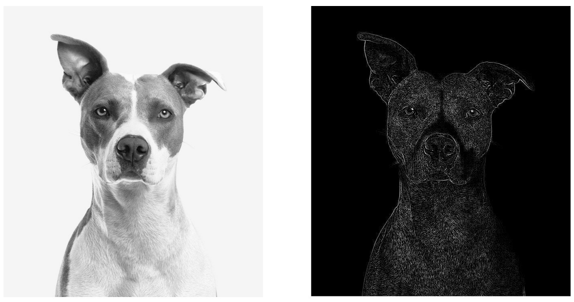


Figure 2.6: Convolution for Edge Detection

2.1.6 Pooling

A pooling layer aggregates the information of nearby values [29]. Therefore, a pooling window is slid with a defined stride across a two-dimensional feature map. If the stride is smaller than the window, overlapping pooling is applied. At every location, the values inside the window are combined. This can be done in various ways, for instance by selecting the maximum value or computing the average value [37]. The result is a shrunken two-dimensional representation which still contains the relevant feature information but not the exact spatial position. The reason for applying pooling is to reduce the number of parameters and therefore reduce the computational cost [29]. Furthermore, it is a more general representation of the image and therefore helps for preventing overfitting [29].

2.1.7 Overfitting

In general, the data is split into three different sets: training, test and validation [29]. The training set is used to train the neural network, i.e., determining the weights. In the next step the unseen validation set is used to tune hyperparameters like the number of hidden layers or the number of perceptrons in those layers. Finally, the overall performance is evaluated on the unseen test set.

K-fold Cross-Validation is a different method for evaluating the model's performance [29]. In this case the data is separated into k partitions with equal size. The model is then trained with $k - 1$ partitions and tested on 1 partition. This process is repeated k times and each subset is used once. At the end, the loss of all iterations is averaged.

A model is overfitted if it is not able to generalize and only performs well on the training data but poorly on unseen data [29]. This can happen if the training data set is too small and not representative of the overall data. For example, if the task is to distinguish bananas from apples and the training data set only includes ripe yellow bananas and green apples, the model might not be able to recognize an unripe green banana because the model associates the color green only with apples. Overfitting can also be the result of too much training. In this case, the model starts adjusting the parameters until it is also able to predict noisy data in the training data. Therefore, it is beneficial having more training data and stop training as soon as the model's performance on the test data set declines.

A complex architecture is also very likely to overfit. Instead of learning the right parameters, the model simply memorizes which data point corresponds to which class. One way for keeping the architecture simple is by using a penalty for more complex models. This is called regularization. Two popular methods are LASSO (see equation 2.9) and Ridge Regression (see equation 2.8). In both cases a term proportional to the weight parameters is added to the loss function. The λ parameter determines the penalty influence of the weights. The key difference between the two methods is that LASSO uses only a subset of the parameters since many parameters become 0 [24].

$$Ridge = Loss + \lambda \sum_{j=1}^{m} w_j^2 \tag{2.8}$$

$$LASSO = Loss + \lambda \sum_{j=1}^{m} |w_j| \tag{2.9}$$

Another way for avoiding overfitting in neural networks is to use Dropout [38]. In each iteration a number of perceptrons are randomly selected and removed (including their ingoing and outgoing edges).

2.2 Image Processing

The Cauzin Softstrip collection consists of digital images. In order to process the encoded information, a variety of image manipulation techniques have to be applied. This section explains the most relevant concepts which are necessary to understand this work.

2.2.1 Image Representation

A digital image can be mathematically described as a two-dimensional function $f(x, y)$ with the *discrete* and *finite* spatial coordinates x and y [39]. Each point, called a pixel, (x, y) is mapped to a discrete intensity value — the color. The intensity dimension depends on the color mode. Black & white (binary) and grayscale images both use one-dimensional intensity values while color models such as RGB, CMYK and HSV use three dimensional intensity values.

The pixels in a binary image are either black (0) or white (1) [40]. Grayscale images consist only of different gray levels. In the case of 8-bit grayscale images, each pixel is mapped to a value between 0 (black) and 255 (white) [40]. The conversion from a grayscale image to a binary image is explained in section 2.2.4.

Many different color models exist, the most common ones are RGB, CMYK and HSV. Every color in the RGB model is a combination of the colors red, green and blue [41]. Each color channel has a total of 256 possible intensity values (for 8-bit RGB). RGB uses so-called additive color mixing, i.e., new colors are produced by overlapping at least two of the three colors red, blue and green [41]. The color white is created by overlapping all three colors with full intensity (255, 255, 255). The pixels from a RGB image can be simply converted to a grayscale image with equation 2.10 [42][43].

$$GRAY = 0.299 * R + 0.587 * G + 0.114 * B \tag{2.10}$$

Two additive colors can be combined to a subtractive color [41]. The combination of blue and green produces the subtractive color cyan; the combination of red and green produces the subtractive color yellow and the combination of blue and red produces the subtractive color magenta. These colors are used in the CMYK color model and if combined again produce the additive colors from the RGB color model [41]. The CMYK model uses subtractive mixing, i.e., the colors cyan, magenta and yellow block wavelengths to create new colors [41]. For example, cyan blocks the colors blue and green. The combination of cyan, magenta and yellow creates the color black. The CMYK color model is usually used for printing. In order to save printing costs, a separate black ink was added which is the K in CMYK [41].

A color model which is more similar to the human visual perception is the Hue-Saturation-Value (HSV) model [41]. The hue defines the pure color. The value determines the brightness, and the saturation defines how much the color differs from a neutral gray [41].

Histograms are used to illustrate the intensity distributions in images [44]. A histogram consists of two dimensions: the x-axis for the intensity values and the y-axis for the number of occurrences. The location of the intensity values in the image are not relevant for the distribution. If the image uses a color model, a histogram for each color channel can be generated [45].

2.2.2 Dilation and Erosion

Dilation and erosion belong to the class of morphological image processing techniques and are useful for noise reduction [46]. They can operate on binary images as well as on grayscale images [47]. However, this section focuses only on the binary operations. The dilation operation is denoted by the operator $\oplus$ (see equation 2.11) and adds new pixels to object boundaries, i.e., lets them grow [39]. In contrast, the erosion operation is denoted by the operator $\ominus$ (see equation 2.11) and removes pixels from object boundaries, i.e., shrinks them [39].

Both operations use a combination of convolution and thresholding [48]. The kernel in the convolution operation is called the structuring element and can be of any shape. One cell in the structuring element is defined as origin, or anchor point, and is moved along the image during the convolution operation. [48] defines the operations as:

$$f \oplus s = \theta(c, 1) \tag{2.11}$$

$$f \ominus s = \theta(c, S) \tag{2.12}$$

With c as the result of the convolution operation:

$$c = f \otimes s \tag{2.13}$$

Where f is the image, s the structuring element and S its size. The thresholding operation is defined as:

$$\theta(f, t) = \begin{cases} 1 & \text{if } f \geq t \\ 0 & \text{otherwise} \end{cases} \tag{2.14}$$

Figure 2.7 illustrates the erosion and dilation operation on a Cauzin Softstrip segment. It is important to note that the objects are the white and *not* the black areas. As one can see, the eroded operation removed many of the white dots inside the black areas whereas the dilated operation increased the white dots. Dilation and erosion can be combined to create two new operations: opening (see equation 2.15) and closing (see equation 2.16) [39].

$$f \circ s = (f \ominus s) \oplus s \tag{2.15}$$

$$f \bullet s = (f \oplus s) \ominus s \tag{2.16}$$

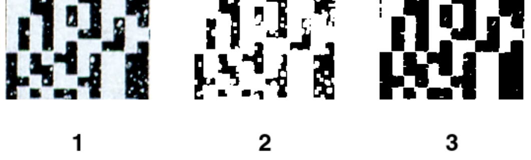


Figure 2.7: Dilation and Erosion
1: original Softstrip; 2: dilated Softstrip; 3: eroded Softstrip

2.2.3 Convex Hull

The convex hull (see figure 2.8) for a set of points is the smallest convex polygon where each point is either inside the polygon or on its boundary [49]. A polygon is convex if [49]:

1. It is a simple polygon, i.e., it is not self-intersecting.
2. All possible edges between two vertices are either inside the polygon or on its boundary.

Algorithms for computing the convex hull are sometimes referred to as "gift wrapping" algorithms [50] which is an intuitive description of the convex hull. Two popular algorithms for computing the convex hull are Graham's scan [51] and Jarvis march [52]. With n being the number of points and h the number of points on the convex hull, Graham's scan runs in $\mathcal{O}(n \log n)$, whereas Jarvis march runs in $\mathcal{O}(nh)$ which, in the worst case, can fall to $\mathcal{O}(n^2)$. A more performant alternative which runs in $\mathcal{O}(n)$ is Sklansky's algorithm [53], an extended version of his first algorithm published in 1972 [54]. Although it was proven that both algorithms by Sklansky do not always work [55][56], they are implemented in the image processing library OpenCV [57], which is used in this work.

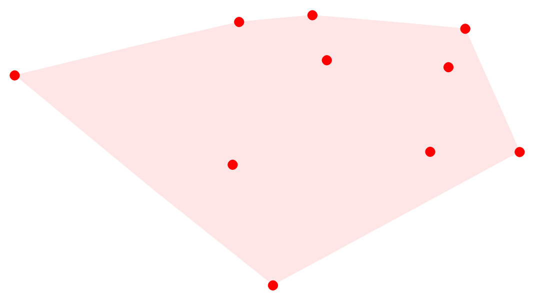


Figure 2.8: Convex Hull

2.2.4 Thresholding

Thresholding is a method for converting grayscale images to binary images. The simplest form, global thresholding, uses a single (global) threshold T and compares it with each intensity value in the image [39]. Figure 2.9 shows a thresholding example [58]. First, a histogram for the intensity distribution is created. Here, the histogram reveals a bimodal distribution, i.e., two peaks are clearly visible. In such a clear bimodal distribution, determining a single threshold (indicated by a red line in figure 2.9) is straightforward and can be determined automatically [39]:

1. Start with an initial random threshold T
2. Divide the image into two groups g_1 and g_2 using T
3. Calculate the average gray level values μ_1 and μ_2 for each group
4. Update the threshold value: $T = \frac{1}{2}(\mu_1 + \mu_2)$
5. Repeat previous steps until $T_i - T_{(i+1)} < \epsilon$ with ϵ as a predefined parameter

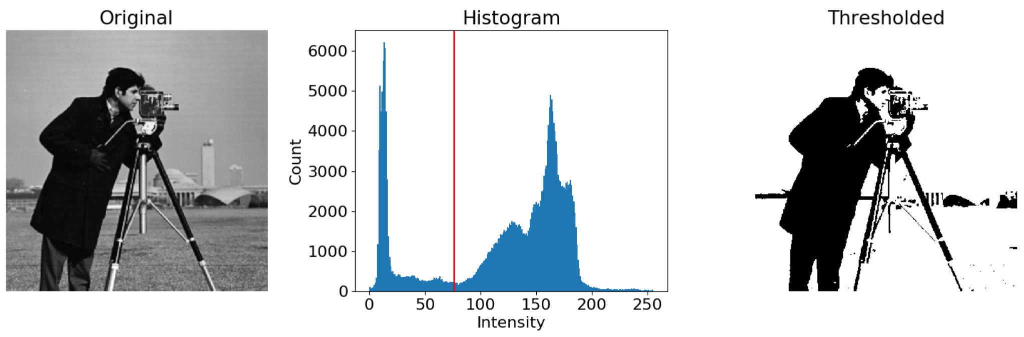


Figure 2.9: Thresholding Example

A different way for finding the optimal global threshold value is Otsu's method [59]. Otsu approaches the task as a binary classification problem. Each gray value is either classified as black or white. The overall goal is to minimize the misclassification error. Thus, the *within-class variance* σ_w^2 (see equation 2.17) is minimized. It is the sum of the weighted class variances σ_1^2 and σ_2^2. Equation 2.18 shows how the weights are calculated, with t as the current threshold value, $P(i)$ the probability of the intensity i, and I the total number of intensity values.

$$\sigma_w^2(t) = \omega_1(t)\sigma_1^2(t) + \omega_2(t)\sigma_2^2(t) \tag{2.17}$$

$$\begin{aligned} \omega_1(t) &= \sum_{i=1}^{t} P(i) \\ \omega_2(t) &= \sum_{i=t+1}^{I} P(i) \end{aligned} \tag{2.18}$$

For all possible values of t ({0, ..., 255} for 8-bit grayscale images), the algorithm computes the *within-class variance* σ_w^2 and then chooses the t with the smallest *within-class variance* [60].

Unfortunately, not every image has a bimodal intensity distribution. Uneven distributed illumination across the image can lead to some very bright and some very dark areas which cannot be separated well by a global threshold [39]. The key idea to threshold those images is to use local thresholding, i.e., dividing the image into several subimages and finding a separate threshold value T for each of the subimages [39].

2.2.5 Edge Detection

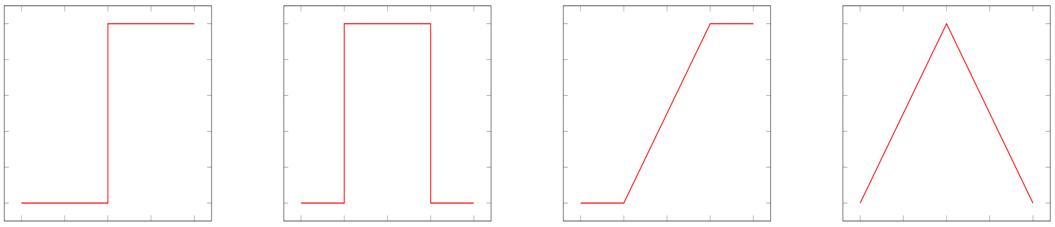
Edges are defined as regions with a significant change of intensity. They can be classified into four categories [61] (see also figure 2.10):

Step: The change of intensity happens rapidly. For example, a white and black pixel next to each other.

Ramp: In contrast to step edges, the change of intensity happens gradually. For example, four pixels next to each other with respective intensities of {100, 125, 150, 175}.

Line: Two step edges next to each other. For example, a white pixel in between two black pixels.

Roof: The combination of two ramp edges next to each other, e.g., a blurry line.



(a) Step Edge (b) Line Edge (c) Ramp Edge (d) Roof Edge

Figure 2.10: Edges

Most images contain a certain degree of noise. As a result, step and line edges are rather rare. Their smoother variants, ramp and roof edges, are more common [61]. This section introduces two edge detection methods, the Sobel operator [62] and the Canny Edge Detection algorithm [63].

2.2.5.1 Sobel Operator

The Sobel operator uses two $3x3$ convolution kernels (see figure 2.11) to compute the gradients G_x and G_y in grayscale images: one for detecting horizontal and one for detecting vertical edges, which is simply a rotated version of the previous kernel [62]. The kernels compare the region around the potential edge by subtracting both sides from each other [62]. If both sides are identical, the result will be 0. The more the intensity values differ, the larger the absolute difference. The kernels are applied separately and the total magnitude of the gradient is computed with equation 2.19 [62].

1	2	1
0	0	0
-1	-2	-1

-1	0	1
-2	0	2
-1	0	1

Figure 2.11: Sobel Kernel
1: Kernel for computing G_y; 2: Kernel for computing G_x

$$|G| = \sqrt{G_x^2 + G_y^2} \tag{2.19}$$

The edge's angle can be computed with equation 2.20 [62].

$$\alpha = arctan\left(\frac{G_y}{G_x}\right) \tag{2.20}$$

2.2.5.2 Canny Edge Detection

Canny defines the criteria for an optimal edge detection algorithm in his work as follows [63]:

Good Detection: Minimizing the false-positive and false-negative error rates, i.e., present edges should be detected and detected edges should be present.

Good Localization: The location of the detected edge should be as close as possible to the center of the actual edge.

Single Response: The width of a true edge should be minimized.

The algorithm consists of five steps [64]:

1. Remove noise with a Gaussian filter
2. Compute the gradient with the Sobel operation
3. Find edge direction
4. Non-maximum supression
5. Hysteresis Thresholding

For smoothing the image with a Gaussian filter, a kernel needs to be calculated first. Equation 2.21 shows how this can be done, where σ is the standard deviation of the Gaussian distribution, x the distance of the horizontal axis from the kernel's origin and y the distance from the vertical axis from the kernel's origin [44].

$$g(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{2.21}$$

The Sobel operation (see 2.2.5.1) can be used to perform step two and three. In most cases, edges are several pixels wide. For this reason, step four thins all edges by analyzing the gradients and setting all pixels except for the local maximums to 0 [64].

The last step performs thresholding to remove noisy edges. Two threshold values T_1 and T_2 are determined and each pixel $p \geq T_1$ is considered as a final edge [64]. If a pixel lies between both threshold values ($T_2 \leq p < T_1$) and is connected to a pixel which is greater than T_1, it is also considered as a final edge. The remaining pixels $p < T_2$ are removed [64].

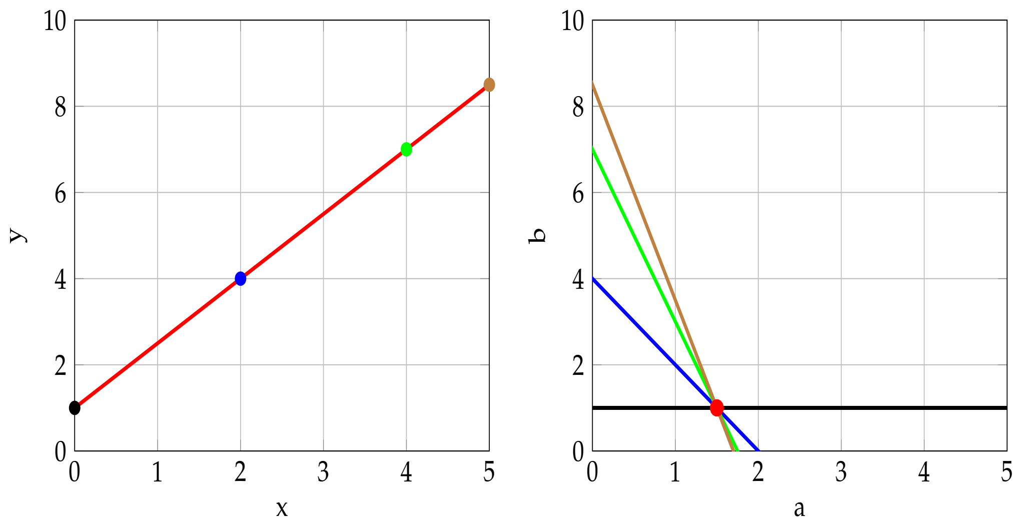
2.2.6 Hough Transform

Before the Hough transform [65][39] can be performed, the edges in the image must be detected first. The result of the edge detection step is an image with a black background and white pixels on it — the edges. Each edge consists of several pixels which may be aligned in a particular form, e.g., a line. The Hough transform scans the image for those pixels and replaces them with actual lines [39].

Let's assume there are n points in an image and a subset of them form a straight line. The general equation for a line is shown in 2.22 with a as the slope and b as the intercept. For a single point (x_i, y_i), there is an infinite number of lines that pass through this point, all with different values for a and b.

$$y = ax + b \tag{2.22}$$

Equation 2.22 can be written as shown in equation 2.23. A point in the x, y space is a line in the a, b (parameter space) space and vice versa (see figure 2.12) [39]. Consider a second point x_j, y_j which lies on the same line as x_i, y_i in the x, y space. This second point can also be represented in the parameter space as a line and it will intersect with the line



(a) XY Space (b) Parameter Space

Figure 2.12: XY and Parameter Space
Each point in the xy space is a line in the parameter space and each point in the parameter space is a line in the xy space.

of x_i, y_i at a', b'. In fact, all points on the line in the x, y space will intersect at the point a', b' in the parameter space [39].

$$b = -xa + y \tag{2.23}$$

The parameter space is now divided into cells, the accumulator cells, and all cells are initialized with 0. Every point x_i, y_i from the x, y space is then transformed to a line in the parameter space and the value in the corresponding accumulator cell for point (a_i, b_i) is incremented by one. If a cell (i, j) has a high value, this means many lines pass through this cell, i.e., the slope and intercept for the line in the x, y space are a_i, b_i.

There is still one problem: What happens if the line in the x, y space is a vertical line? In this case, the slope $a = \infty$. This can be avoided by using the normal representation, shown in equation 2.24 [39]. ρ is the distance of the line from the origin and bounded between $\pm\sqrt{2D}$ where D is the diagonal of the image. θ is the angle between the x-axis and ρ with a range from $-90°$ to $+90°$. The accumulator cells are determined in the same way as before.

$$x \cos\theta + y \sin\theta = \rho \tag{2.24}$$

2.3 Summary

The first part 2.1 explained the fundamental concepts of neural networks and in particular of CNNs. First, section 2.1.1 introduced feedforward networks which CNNs belong to. Feedforward networks are one category of neural networks where information can only flow in one direction, i.e., forward. Next, the perceptron was explained and how they can be combined to build a MLP. The following section 2.1.2 covered one component of a perceptron in detail — the activation function. It was highlighted that non-linear activation functions are required to model complex problems. Section 2.1.3 explained first how a loss function can be used to measure the neural network's performance and then how the performance can be optimized with gradient descent and backpropagation. The convolution operation was explained in section 2.1.4. It is the key layer in a CNN and extracts the features of an image. The following section 2.1.5 explained feature maps which are the result of the convolution operations. The pooling operation was explained in section 2.1.6 and is helpful for reducing the dimensions of the feature maps. The last section 2.1.7 explained the problematic of overfitting and showed some techniques such as regularization and dropout for reducing overfitting.

The second part 2.2 focused on image manipulation techniques. It first explained the basic representation of an image in 2.2.1 and then the different color models such as black & white, grayscale, RGB, CMYK and HSV. Furthermore, it was shown how histograms can be used to analyze the intensity distribution. Section 2.2.2 explained the two morphological operations dilation and erosion. They can be used to manipulate the object's boundary and are useful to remove noise. The next section 2.2.3 illustrated the concept of the convex hull. After that, section 2.2.5 introduced two thresholding algorithms, i.e., two algorithms to convert grayscale images to binary images. Section 2.2.5 explained not only the four edge types step, line, ramp and roof but also how they can be detected with the Sobel operator or Canny's algorithm. The last section 2.2.6 explained how *lines* can be detected after an edge detection step.

3 Analysis

This chapter starts first with a general introduction to bar codes. Before the Cauzin Softstrip symbology is explained, a very similar bar code is presented — the PDF-417. An algorithm for recognizing PDF-417 bar codes is explained as well as one algorithm for decoding the PDF-417 symbology. The next sections analyze the Cauzin Softstrip in detail. They include the Cauzin Softstrip's symbology and a detailed analysis of the error detection methods. The last section presents the Softstrip dataset and highlights the characteristics of each subset.

3.1 Barcodes

Note: The information provided in this section is based on The Bar Code Book [66] unless stated otherwise.

In general, bar codes encode human readable information into a pattern of different symbols. The *symbology* describes how information is translated into the different symbols. The actual data is referred to as *code*.

Bar codes can be divided into two broad categories: one-dimensional and two-dimensional bar codes. One-dimensional bar codes, also called linear bar codes [1], use a parallel pattern of spaces and bars to encode information. They are omnipresent and most known for encoding product information in stores. Two examples are shown in figure 3.1. Those bar codes use *width-modulated* symbols, i.e., different bar widths are used to encode information, and the height only provides redundant information which might be useful if the bar code is damaged. Bar codes with *height-modulated* symbols, i.e., bar codes which use varying bar heights to encode information, exist as well. POSTNET is such a bar code and was used by the United States Postal Service [67]. In order to store more information

1 23456 78900 5

(a) UPC A

1 234567 890005

(b) EAN 13

Figure 3.1: One-Dimensional Bar Codes

and use the area more efficiently, two-dimensional bar codes were created. They can be divided into two categories: stacked and matrix symbologies. Figure 3.2 shows a few example bar codes for each category. Stacked bar codes are composed of multiple rows where each row looks like a one-dimensional linear bar code. Matrix bar codes use the horizontal and vertical area to encode information and allow more sophisticated symbols.

[1]There exist also circular symbologies such as the ShotCode.

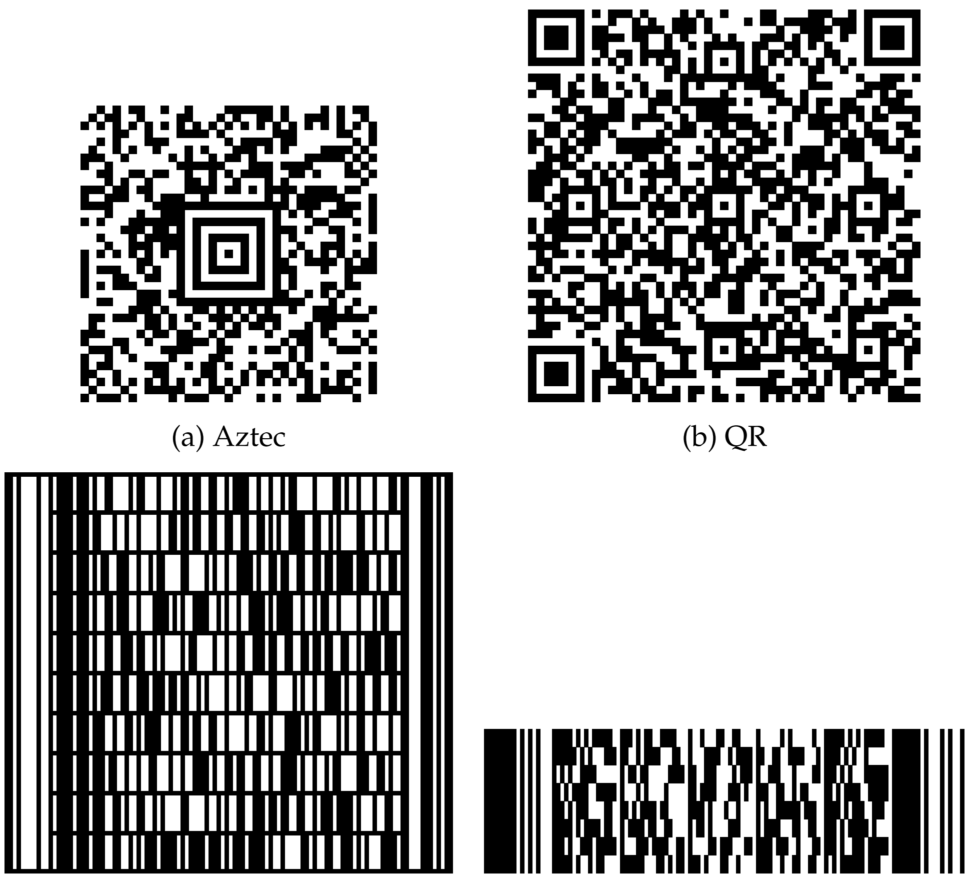


Figure 3.2: Two-Dimensional Bar Codes

The Cauzin Softstrip belongs to the category of two-dimensional stacked bar codes and its symbology is explained in section 3.3. At first sight, the PDF-417 bar code symbology looks similar to the Cauzin Softstrip. Therefore, section 3.2 explains its symbology first and then presents one algorithm for recognizing and one for decoding PDF-417 bar codes.

3.2 PDF-417

Note: The information about the PDF-417 symbology is based on The Bar Code Book [66] and the PDF-417 patent [68].

The PDF-417 is a two dimensional bar code which was developed by Ynjiun Wang [68]. The bar code is used in many areas such as on airline boarding passes [69] or on North American driver's licenses [70]. It belongs to the category of stacked bar codes and looks similar to the Cauzin Softstrip (see figure 3.3). Therefore, this section examines how the bar code is processed.

3.2.1 Symbology

Figure 3.3 shows a PDF-417 bar code. The start and end of the bar code are indicated by a start pattern on the left side and a stop pattern on the right side. PDF-417 consists of multiple data rows. Each row has a left and right row indicator which provide some meta information such as the current row number or the total number of rows. The actual data is encoded in the data section between the left and right row indicator. A value, which can be a character or number, is encoded as a *codeword*, a pattern of four black bars and four white spaces. Each codeword consists of 17 *units*. A unit is the smallest width for a single bar or space. The combination of 4 codewords and 17 units is the reason for the name PDF-417. There are 929 codewords in total and three different clusters. Each cluster encodes all 929 codewords uniquely, i.e., the first cluster encodes the value 0 in a different way than the second or third cluster. A row can only use codewords from one cluster. The used cluster changes with each row, so adjacent rows use always different codewords and can be distinguished from each other.

The bar code uses Reed-Solomon error correction with different correction levels. A bar code with the highest correction level can even be decoded if 50% of it is missing [71].

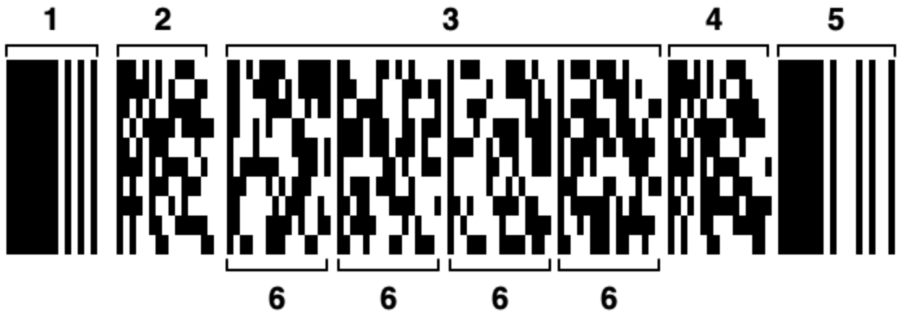


Figure 3.3: PDF-417
The beginning of the bar code is indicated by (1) and the ending by (5). The left (2) and right (4) row indicator contain meta information about the row. The data section (3) consists of multiple codewords (6).

3.2.2 Recognition

The task of localizing the bar code in an image is called bar code recognition. A recognition process for the PDF-417 bar code is presented in [72]. The algorithm consists of four steps:

1. Grayscale conversion
2. Region of Interest (ROI) detection
3. Rectify the bar code
4. Decode the data

In the first step, the color image is converted to a grayscale image. This does not only reduce the storage space but is also required for operations like edge detection [72]. The idea behind the ROI detection is to detect the boundary of the bar code. Therefore, the area inside the bar code is first set to one color by applying erosion. Afterwards, a Hough transform is applied which is able to detect shapes (see figure 3.4). From there, the corner points can easily be detected and a perspective transformation can be applied to correct the perspective.

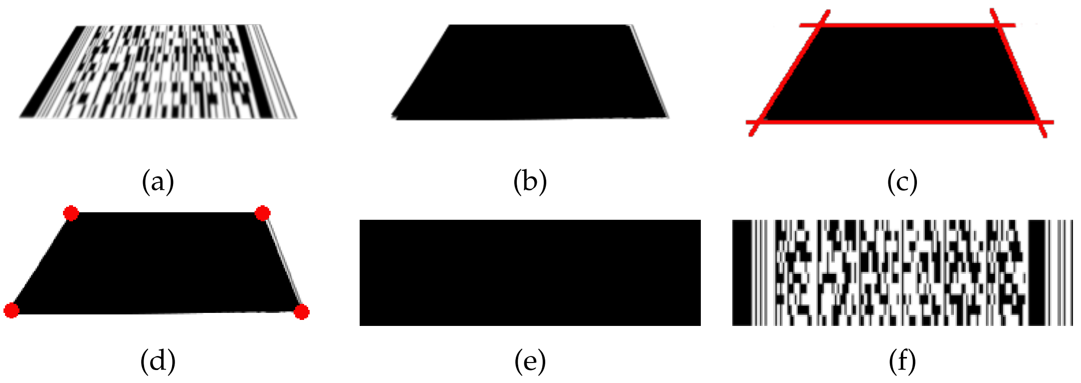


Figure 3.4: PDF-417 Recognition
The original bar code is shown in the first image. The second image shows the bar code after the erosion step. The Hough transformation is applied in the third image. Image four shows the corner points that can be calculated from the third image. The corner points can be used to correct the perspective as shown in image five. The last image shows the corrected bar code after applying dilation.

3.2.3 Decoding

After the bar code has been located and rectified, the decoding algorithm starts. Many decoding strategies exist, one algorithm is described in [73]. The key idea is to place a grid over the bar code, where one cell has the size of the smallest bar. Then, the average pixel value in each cell is computed. Finally, a binarization step is applied and the codewords can be decoded.

In order to create a grid, the number of rows and columns must be known. As in subsection 3.2.1 explained, consecutive rows use codewords from different clusters. This means, the codewords in one column differ in at least one module [73]. Therefore, the number of rows can be determined by calculating these transitions. The property that each codeword starts with a black and ends with a white module is used for determining the number of columns. Consequently, there is always a white to black transition between two codewords. Therefore, the number of columns can be determined by locating and counting the white to black transitions between the codewords.

3.3 The Softstrip Symbology

This section explains the structure of the Cauzin Softstrip and how digital data is encoded. The information presented in this section is based on the Cauzin Systems patents [74][75][76].

3.3.1 Basics

Figure 1.1 on page 4 shows four typical Softstrips in a computer magazine. A strip is usually up to 255 mm long and up to 16 mm wide. Each Softstrip has two positioning marks for the Softstrip Reader, a circle in the upper left and a rectangle in the lower left. Each Softstrip is divided into three sections (see figure 3.5). The first part is the horizontal synchronization section, the second part is the vertical synchronization section and the last part is the information part. The horizontal and vertical synchronization sections are referred to as the header and contain encoded meta information about the Softstrip itself.

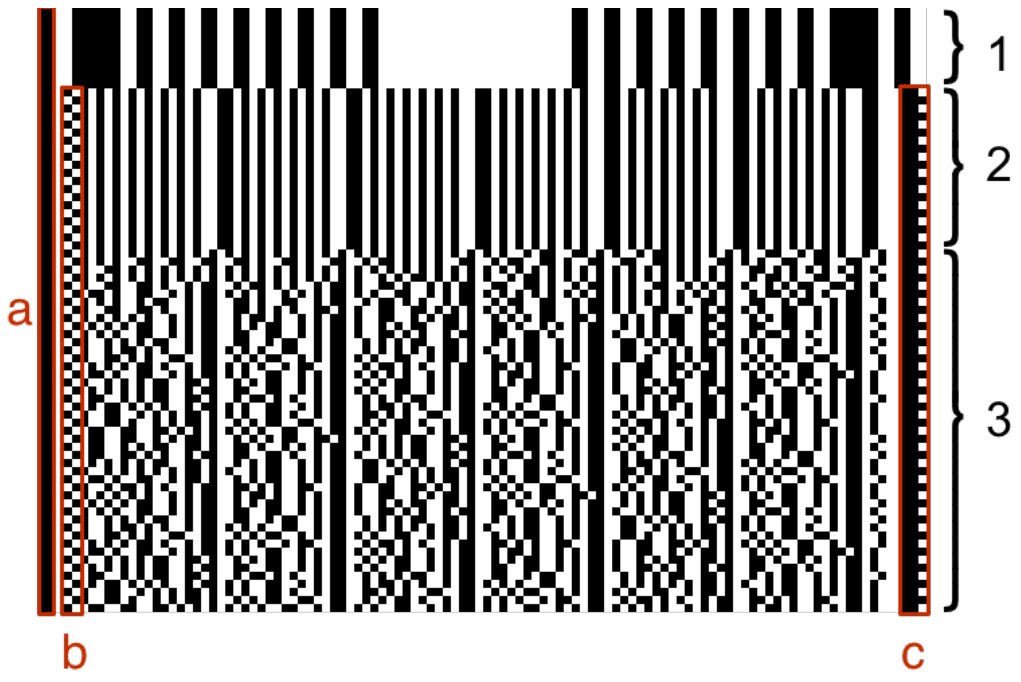


Figure 3.5: Softstrip Structure
The Softstrip is divided into three parts: the horizontal synchronization section (1), vertical synchronization section (2) and the information part (3). A black bar marks the start of the Softstrip (a). The checkerboard (b) and the rack (c) are used to identify the start and end of a row.

Each part is explained in detail in the following sections.

The Cauzin Softstrip uses two bits, called dibit, for encoding a single information bit. All possible dibit values are shown in figure 3.6. A zero data bit is encoded by a black square followed by a white square, whereas a one data bit is encoded by a white square followed by a black square. A dibit with two white or two black squares is invalid.

(a) Zero
(b) One
(c) Invalid
(d) Invalid

Figure 3.6: Possible Dibit Values

The start of the Softstrip is indicated by a one dibit wide black bar on the left side. Both the checkerboard and rack (see figure 3.5) change in each line and are used to determine the start and end of a row. If the checkerboard dibit is zero (black-white), then the rack consists of two black squares and one white square. The next row must have a checkerboard with a one dibit (white-black) and a rack with three black squares. A typical row in the vertical synchronization and information section is shown in figure 3.7.
The row begins with the start bar which consists of two black squares and is followed by one white square. The next two squares are the checkerboard pattern, which is either black-white or white-black. After that comes the first parity dibit for detecting errors in the following data dibits. The number of data dibits is defined in the horizontal synchronization section. A second parity dibit is added after the data dibits. The left parity dibit checks the odd data dibit positions, whereas the right parity checks the even data dibit

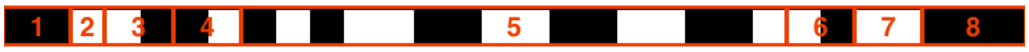


Figure 3.7: Row Example
The row consists of a one dibit wide start bar (1), one half dibit wide white space (2), checkerboard (3), left parity dibit (4), multiple data dibits (5), right parity dibit (6), one dibit wide white space (7) and the rack (8).

positions (see equation 3.1). The row ends with one or two white squares after the second parity dibit and either two or three black squares for the rack.

$$\text{left parity} = \sum_{0 \le i \le n, \text{i odd}} dibit_i \qquad \text{right parity} = \sum_{0 \le i \le n, \text{i even}} dibit_i \tag{3.1}$$

3.3.2 Horizontal Synchronization

The first part of each Softstrip is the horizontal synchronization section. It includes the number of nibbles per row and is used to align the optical reader for scanning the strip. The reader determines the contrast between paper and ink color by measuring the reflectance of the broad bars and compares it with the reflectance of the non-printed area above the Softstrip. However, the contrast value is not required for this work.

The horizontal synchronization section has an even number of symmetrically aligned bars. For instance, a horizontal synchronization section with 10 bars has 5 bars on the left and 5 bars on the right side. A Softstrip must have at least four bars. The two broader bars are three dibits and the other bars are one dibit wide. The space between the bars is one dibit wide as well, except for the center, which separates the left from the right side.

The number of data bits per row is encoded as nibbles. A nibble is a half byte and has consequently 4 bits. The data bits are all bits between the left and the right parity dibit. In order to obtain the number of nibbles, one has to count the white to black transitions in the horizontal synchronization section. It should be noted that the white to black transition at the start bar counts as well. The number of nibbles per line can then be calculated with equation 3.2.

$$Nibbles = \frac{transitions + 4}{2} \tag{3.2}$$

3.3.3 Vertical Synchronization

The horizontal synchronization section is followed by the vertical synchronization section. The height of the dibits is encoded as a 8-bit number and is repeated multiple times per row. The number itself is split into nibbles and the height is measured in 1/16 scan steps. The upper nibble is the number of full scans, while the lower nibble is the number of $\frac{1}{16}$ scans. For instance, the dibit height 0x65 requires 6 and 5 sixteenths scans with a scan step distance of 0.0635 mm.

3.3.4 Data Information

The actual data is encoded in this section. It is separated from the vertical synchronization section by three 0 bytes. A file header with meta information about the file is prepended. A detailed overview about the meta information can be found in table 3.1.

Field	**Name**	**Length in Bytes**	**Description**
1	Length	2	First byte is LSB and second byte is MSB. Length is the total number of bytes of all following fields.
2	Checksum	1	Two's complement of the one byte binary addition with carry bit of all following bytes.
3	Strip Id	6	Multi strips should have the same id.
4	Sequence Number	1	Is a 7-bit binary number.
5	Strip Type	1	Types are listed in table A.1
6	Software Expansion	2	Flag byte for future features.
7	Operating System Type	1	Operating systems are listed in table A.2
8	Number of Files	1	
9	Cauzin Type	1	Cauzin types are listed in table A.3
10	Operating System Filetype	1	Operating system filetypes are listed in table A.4
11	File Length	3	First byte is LSB and third byte is MSB.
12	Filename	Variable length	
13	Terminator	1	Indicates the end of the filename. Either 0x00 for terminating or 0xFF for signaling to run the program after reading.
14	Block Expand	Variable length	For future features. Default is 0x00.
15	Data	Variable length	
16	CRC	2	Not implemented

Table 3.1: Data fields [77]

3.4 Error Detection

The Cauzin Softstrip uses a combination of parity bits and a checksum to detect errors during the decoding process. This section analyzes both error detection methods and highlights their flaws.

3.4.1 Parity Check

Each data row in the Cauzin Softstrip includes two parity (di)bits to detect errors in the data (di)bits. First of all, the parity bits only check the data in between both parity bits. Therefore, they are not able to detect if one of the parity bits themselves is corrupted. As a consequence, the data in a row could be classified as invalid although it is correct. If the parity bits are correct and they detect an error, it is not possible to locate the error. The only information they provide is whether the corrupted bit is at an even or uneven position. It is also not possible to detect the actual number of errors. In fact, it is even possible that errors cancel each other out so that they will not be detected at all. Let's assume a Softstrip row was decoded into the following bits:

$$Row = [1, 1, 0, 0, 0, 0, 0, 1, 0, 1, 1, 1] \tag{3.3}$$

The left parity bit ($Row[0]$) is 1 and the right parity bit ($Row[11]$) is 1. The parity check confirms that the data bits are correct:

$$\begin{aligned} Left &= (0 + 0 + 0 + 0 + 1) \, mod \, 2 = 1 \\ Right &= (1 + 0 + 0 + 1 + 1) \, mod \, 2 = 1 \end{aligned} \tag{3.4}$$

Now, let's assume the same row is damaged so that the decoded bits are:

$$Row = [1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1] \tag{3.5}$$

Although two bits are altered the parity check confirms again that this row is correct:

$$\begin{aligned} Left &= (0 + 0 + 0 + 0 + 1) \, mod \, 2 = 1 \\ Right &= (1 + 1 + 1 + 1 + 1) \, mod \, 2 = 1 \end{aligned} \tag{3.6}$$

A single parity bit is only able to detect an error in a set if the number of errors in this set is uneven.

3.4.2 Checksum

Every Softstrip contains a one-byte checksum which is calculated in two steps:

1. One-byte add with carry bit of all bytes after the checksum field
2. Apply the two's complement

As the previous section highlighted, it is possible that wrong data passes the parity check. An error which passes the parity check can be detected by the checksum check. However, this is not always true. Assume there is an undetected error in a row and this row contains the checksum field. Consequently, the undetected error corrupts the checksum itself and it does not match the checksum of the correct data anymore, i.e., correctly decoded data will not be accepted. If, however, additional errors occur during the decoding process, the checksum field might match the checksum of the decoded data again and wrong data will be accepted. A user would notice this later when they try to use the exported program and it does not work as intended or even crashes.

Another problem is that the checksum does not consider the position of the data bytes. For example, let's assume the decoded bytes are $[0, 4, 5, 8]$. The checksum for those bytes is 239. Let's consider the row was decoded again. However, this time the parity check missed some errors, so the decoded bytes are $[5, 4, 0, 8]$ which results also in a checksum of 239. As one can see, wrong data did not only pass the parity check but also the checksum check.

This section showed that the parity bits and the checksum are not able to detect all errors. However, they are the only error detection methods used by the Cauzin Softstrip. Consequently, there is no reliable method to verify that the decoded data is correct.

3.5 Dataset

The Softstrips used in this work are from different sources, including digital generated Softstrips, scans from the book *Animated Algorithms* [19], scans from various StripWares, a collection of scanned Softstrips from computer magazines [78], and a collection of scanned Softstrips from Cauzin Softstrip Application Notes and Marketing Material [79]. The following subsections highlight the characteristics of each dataset.

3.5.1 Digital Generated Softstrips

The Softstrip creation tool by Chris Osborn [80] was used to generate 870 Softstrips in total. An icon collection served as input data for those Softstrips. Each icon was encoded into a single Softstrip with a typical image resolution of 390×2500 pixels. The Softstrips in this dataset are of the highest quality, i.e., each dibit has the exact same size, only black and white pixels exist, and there is neither noise, distortions nor rotations. A few typical rows are shown in figure 3.8. The purpose of this large dataset is to develop a base decoder which is able to decode high quality Softstrips. Once the decoder is able to decode this dataset, it can be optimized for the other datasets.

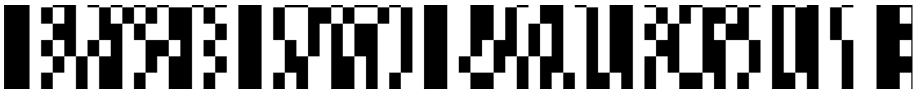


Figure 3.8: Generated Rows

3.5.2 Scanning Distortions

The Cauzin Softstrip has the shape of a rectangle, so it is naturally to assume that this is also true for a scanned Softstrip. However, if a Softstrip was not scanned carefully enough, the Softstrip's shape is slightly curved. Figure 3.9 illustrates this. This makes it more difficult to scan fixed areas on the image for certain properties. Those distortions occur on all following datasets.

3.5.3 StripWares

Computer programs in form of Cauzin Softstrips were sold in computer stores as booklets called StripWares. Usually, several programs of the same category, for instance, financing were sold in one StripWare. A few StripWares in very good condition, some were still sealed, were available for this thesis. Those are:

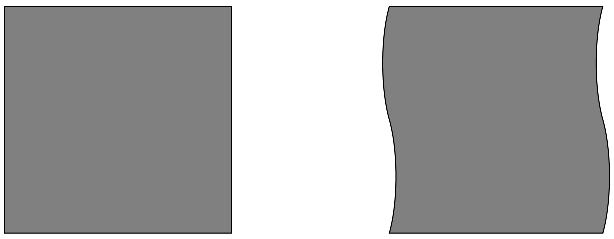


Figure 3.9: Scanning Distortions

- Classic Games for the Apple II
- The Financial Advisor for the MacIntosh
- MacArt for Business for the MacIntosh
- Portfolio Evaluation for the IBM PC
- Star Gazing for the IBM PC
- The Basic Apple for the Apple II
- Second Giant Book of Computer Games for the IBM PC
- Cauzin Softstrip Sampler

103 Softstrips in total were scanned with 1200 Dots per inch (DPI). The image resolutions ranges from about 750 $\times$ 6000 pixels up to 800 $\times$ 9600 pixels. The collection includes both low and high density Softstrips (see figure 3.10). Furthermore, it includes programs consisting of a single and programs consisting of multiple Softstrips. Although the StripWare collection is in a very good condition there are several visual artifacts on the Softstrips as shown in figure 3.11.

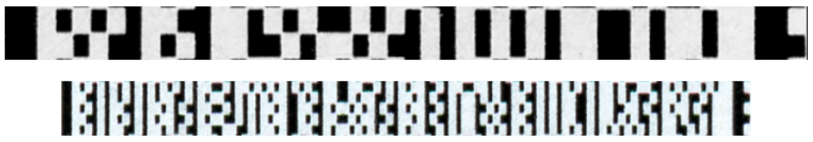


Figure 3.10: Density Example
The upper figure shows two rows with a low data density. The lower figure shows five rows with a high data density.

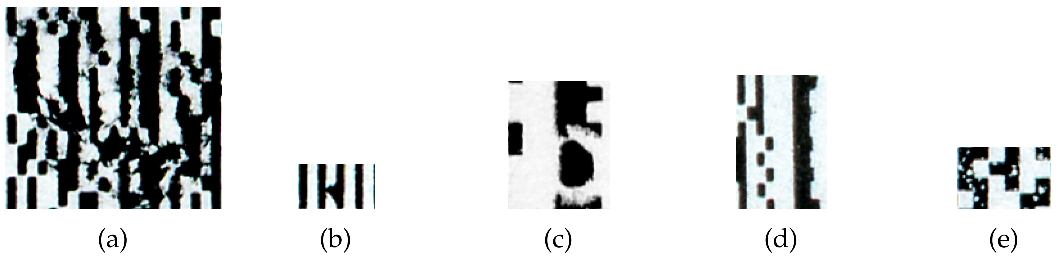


Figure 3.11: StripWare Artifacts
(a) and (b) show areas with smeared printer ink. (c) and (d) show strips with a damaged rack. (e) shows a strip with a lot of white noise on black areas.

3.5.4 Softstrips Animated Algorithms

The book Animated Algorithms [19] is one of the few books which uses the Cauzin Softstrip. It includes 40 Softstrips to run the presented algorithms. All programs consist of multiple Softstrips and are printed in a medium density compared to the Softstrps from the StripWares. They were scanned with 300 DPI and have an image resolution of about 360 × 4200 pixels. The Softstrips have also some visual artifacts as shown in figure 3.12.

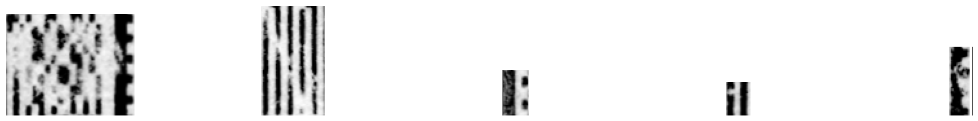

(a) (b) (c) (d) (e)

Figure 3.12: Animated Algorithms Artifacts
(a) shows an area with damaged dibits. (b) and (d) show an area with missing dibit parts. (c) shows a start bar with a lot of white noise. (e) shows a damaged rack.

3.5.5 Cauzin Softstrip Application Notes and Marketing Material Collection

This collection was found on the internet archive [79] and consists of 58 Softstrips for converting programs from one operating system to another. There are no information about the DPI for these scans. The Softstrips are printed in a low and medium density. Most programs contain only of a single Softstrip. The images from the PDF were extracted with a high resolution of about 1500 × 15000 pixels. In contrast to the other datasets, many Softstrips in this collection contain some blurred areas as shown in figure 3.13.

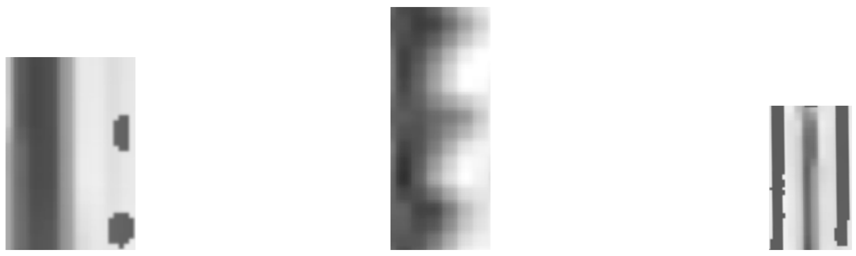

(a) (b) (c)

Figure 3.13: Cauzin Softstrip Application Notes and Marketing Material Artifacts
(a) shows a blurred start bar. (b) shows a blurred rack. (c) shows blurred dibits.

3.5.6 Magazine Collection

This collection consists of scans from various computer magazines and contains 159 Softstrips in total (without duplicates). Similar to the previous collection, this collection does not include information about the scanned DPI either. The Softstrips are printed in various densities. The images from the PDF were extracted with the same resolution as the ones from the previous collection. This collection contains by far the worst quality Softstrips with some Softstrips where even humans would have trouble to decode them. A few examples are shown in figure 3.14.
The following abbreviations will be used in the next chapters to refer to the respective dataset:

DS1: Generated Softstrips

Figure 3.14: Magazine Artifacts
(a) shows an area with missing dibit parts. (b) shows an area without a checkerboard. (c) shows a damaged rack. (d) shows an area with faded black areas.

DS2: Softstrips from the book *Animated Algorithms*

DS3: Cauzin Softstrip Application Notes and Marketing Material Collection

DS4: Magazine Collection

DS5: StripWare Collection

3.6 Summary

This chapter started with a general bar code introduction. One two-dimensional bar code, the PDF-417, was analyzed in more detail in section 3.2. An algorithm for recognizing PDF-417 bar codes was presented and one algorithm for decoding its symbology.

Section 3.3 explained the Softstrip symbology. The Softstrip is divided into three sections. The first section, the horizontal synchronization section, encodes the number of bytes stored in each row. The second section, the vertical synchronization section, encodes the row height in *mm*. The last section, the information section, encodes the actual data in form of dibits.

Section 3.4 analyzed the Softstrip's error detection methods. The parity bits and the checksum. It was shown that they are not able to detect all kind of errors and in some cases even verify wrong data as correct.

The last section, section 3.5, analyzed the different Softstrip datasets. In total, five different datasets are available for this work. The first one consists of digital generated Softstrips. The second dataset contains scanned Softstrips from [19]. Dataset three and four are scanned Softstrip collection found on the internet. The last dataset contains scans from various StripWares.

4 Related Work

This chapter presents relevant and related work for this thesis, highlights their limitations and shows why this work is important. Section 4.1 presents a digital reader for the OSCAR bar code and section 4.2 the Softstrip generation program with a simple Softstrip decoder.

4.1 OSCAR

OSCAR is a bar code developed by the Databar Corporation and was released two years before the Cauzin Softstrip in 1983. An example bar code is shown in figure 4.1. A typical program was four pages long and took about three minutes to scan [81]. Ads for the OSCAR bar code highlighted how easy it allegedly was to use the system and claimed even children could scan programs in a couple of minutes [82]. However, the optical scanner had to be carefully adjusted and sometimes required multiple tries until a row was successfully scanned [12]. The optical reader was available for $79.95 and the bar codes were published in the Databar magazine. Additionally, the same bar codes could be purchased in stores. The Databar reader was compatible with Atari (400, 600, 800, 1200XL, 1400XL and 1450XL), Commodore (PET, VIC-20 and 64), Timex Sinclair (1000, 1500 and 2000), TRS Color Computer and Texas Instruments TI 99/4A [83]. One could join the Databar Club for 12 or 24 months for an additional $120 or $240 and would receive a monthly issue of the Databar magazine including eight programs. Unfortunately, only one issue was published [12]. In 2016, an OSCAR bar code was published in [84] as a puzzle. In order to decode it, Philippe Teuwen created a digital OSCAR reader in Python and published it on GitHub [85].

The number of published OSCAR bar codes is limited since only one issue of the Databar magazine was published. On the contrary, Softstrips were published in plenty of magazines as well as purchasable programs. This means the digital Softstrip reader has to deal with different visual artifacts coming from different resources and needs a more general decoding strategy than the OSCAR decoder.

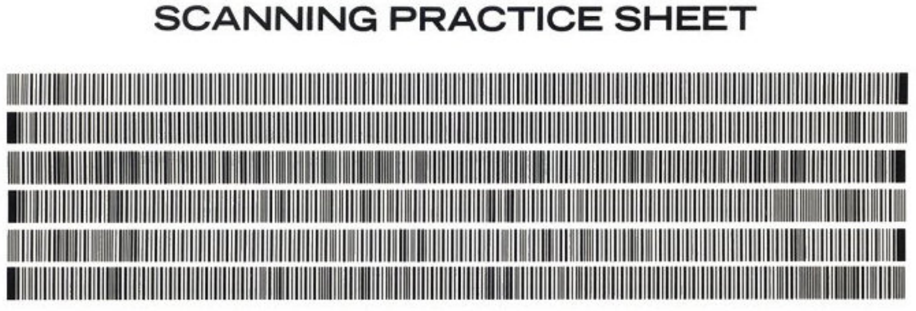


Figure 4.1: Databar Practice Sheet
Databar Practice Sheet [86]

4.2 The CauzCoin Challenge

Chris Osborn is the owner of a working Cauzin Softstrip Reader and published an article about the Cauzin Softstrip on his blog Insentricity [87]. He decided to create his own Softstrips and found a disk image with the Stripper Software online [88]. However, the software comes with a few limitations. First, it has no option to save a Softstrip as an image. They can only be printed on an Epson FX-80 or an Apple ImageWriter. Second, the software is only able to print low density Softstrips. Last, the software works only on Apple DOS 3.3. For this reasons, he created his own software to generate Softstrips [80]. Although the current version does not support splitting files into multiple Softstrips, the generated Softstrips are still useful for this work, since the only difference between decoding the first Softstrip in a set and the other Softstrips is additional meta information on the first strip.

After Chris Osborn created his Softstrip generation program, he used it to create a new Retro Challenge, which he published on his Blog [89] and on the Retro Battlestations subreddit[1]. The challenge was to decode a Softstrip, which contained the private key to a Bitcoin wallet. The first person, who was able to decode the Softstrip could claim the Bitcoins, which were then worth about $15. The winner was Will Sowerbutts, who published his decoding process on his website [90]. His approach is analyzed in the next section.

4.2.1 Window Decoding

Sowerbutts' decoder [90] uses a sliding window approach for decoding the Cauzin Softstrip. The window has the height of a single dibit and half of its width. Sowerbutts does determine the window size manually. However, the number of bits in each row is encoded in the horizontal synchronization section. Therefore, the window width can simply be determined with equation 4.1.

$$width = \frac{\text{bar code width}}{\text{bits per row} * 2} \tag{4.1}$$

As in section 3.3.3 mentioned, the row height is encoded in *mm* in the vertical synchronization section. However, the image resolution of a scanned Softstrip can vary. The same Softstrip could be scanned once in a low resolution and once in a high resolution. In both cases the encoded row height in *mm* is the same but the height in pixels is different. This means, that one *mm* cannot be mapped to a default number of pixels. The Softstrip does not include the total number of rows so an alternative solution is required to obtain the total number of rows. The rack or the checkerboard could be used to count the number of rows since they change with each row. Once the total number of rows is known, the Sofstrip can be divided by this number to obtain the height of a single row.

Once the window size is known, it is slid across the bar code and the center pixel is located at every position to determine if the cell is black or white. Figure 4.2 illustrates this approach. This approach works great if all dibits have exactly the same size. Figure 4.3 shows the result on a scanned Softstrip with varying dibit width and height. As one can see, the green dots do not match the cell center. In some cases the located center has a different color than the cell.

[1]`https://www.reddit.com/r/retrobattlestations/comments/50hoyw/another_retro_challenge_has_arrived_cauzcoin/d75b65t/`

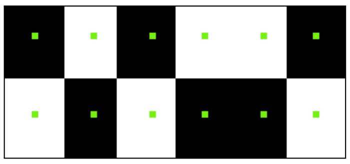


Figure 4.2: Windowing Example

Although this approach works great on the digital generated Softstrips it is error prone on scanned Softstrips. For this reason, a different decoding strategy is presented in chapter 5.

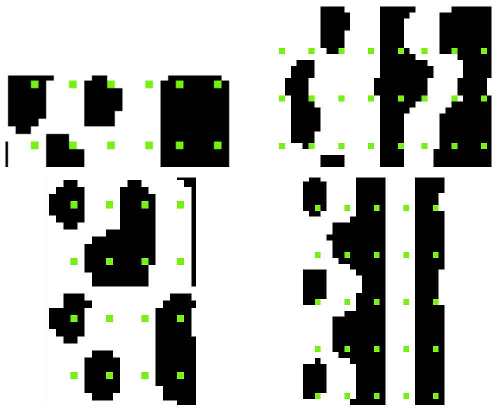


Figure 4.3: Windowing on the Quicksort Softstrip

4.2.1.1 Limitations

First, Sowerbutts' decoder focuses only on decoding this particular Softstrip. For this reason, it can make some assumptions which do not hold for other Softstrips. First of all, it assumes that all squares and rows have the exact same size, which is true for the generated Softstrip but does not hold for the scanned bar code images. Secondly, since it only needs to decode a single Softstrip, the implementation does not support decoding data which consists of multiple Softstrips. Lastly, it requires the user to count both the number of squares per row and number of rows in total. By contrast, the digital reader of this work does work with a variety of Softstrips, including both generated Softstrips and scanned Softstrips from magazines or books. It does support decoding data divided into multiple Softstrips and does not require the user to count the number of rows or number of squares per row.

5 System Design

Bar codes and in particular the Cauzin Softstrip were analyzed in chapter 3. Its symbology was explained and potential challenges were highlighted. This chapter presents a solution to overcome those challenges and shows how to decode the Cauzin Softstrip. Each section in this chapter focuses on a single component of the digital decoder and explains the general idea behind the solution. More details of the implementation are presented in chapter 6.

5.1 Softstrip Recognition

Although Softstrip recognition is not part of this thesis, the general steps to perform this task are presented in this section so it can be implemented in future work. The steps are based on the PDF-417 recognition framework [72], explained in section 3.2.2:

1. Preprocessing
2. Bar code detection
3. Bar code extraction
4. Perspective transformation

The first step is required for the following steps. It includes grayscale conversion and thresholding to obtain a binary image. The bar code boundary can then be detected with contour detection which is based on edge detection. Once the bar code boundary is known, the corner points can be located to extract the bar code. This could be done with the Hough transformation. In the last step, a four point perspective transformation could be applied with the corner points to correct the perspective.

5.2 User Interface

As already mentioned in section 5.1, an automatic bar code recognition for the Softstrip is beyond the scope of this work. A Graphical User Interface (GUI) is provided as an alternative, which includes some basic functions to allow the user to extract the Softstrip from a scanned image. These functions are:

- Image Rotation
- Image Scaling
- Softstrip Selection

Data can be divided into multiple Softstrips. Therefore, a user has the option to select multiple Softstrips from an image and to order them. If a user is not satisfied with a selected Softstrip, they can delete the selected Softstrip and repeat the step again.

5.3 Horizontal Synchronization Section

The horizontal synchronization section is the first part of a Softstrip and contains the number of bytes per row. In order to decode the number of bytes per row, the white to black transitions need to be counted. This is an easy task for a human but more complicated for a computer. An horizontal synchronization section with a lot of noise could have more white to black transitions than actually exist (see figure 5.1). If the digital decoder does

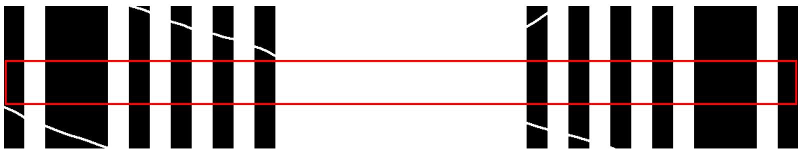


Figure 5.1: Damaged Horizontal Synchronization Section

not recognize the noise, the whole decoding process already fails with the first step. For this purpose, the very first step is noise reduction. This can be done with:

- Gaussian Filter
- Erosion and Dilation

There is no guarantee that the noise reduction step removes *all* of the noise. Therefore, the decoder needs to locate a part of the horizontal synchronization section which very likely does not contain any noise. The challenge is to find this part. The idea is to consider each *pixel* line and compare it with its surrounding lines. If they are all similar, and the number of surrounding lines is large enough, one can be confident that this part does not contain noise. The red rectangle in figure 5.1 highlights such a part. A common similarity measurement is the Euclidean distance (see equation 5.1).

$$d(p,q) = \sqrt{\sum_{i=1}^{k} (p_i - q_i)^2} \tag{5.1}$$

It measures the distance between the k features of the data points p and q. Here, the features of two pixel lines are used to measure the distance. These features are:

- Average bar width
- Maximum bar width
- Minimum bar width
- Number of bars

A bar refers to both black and white bars. A pixel line can now be compared to its neighbours and if it is similar to, for instance, 10 neighbours, it is a strong indicator that there is no noise in this area. The actual number of neighbours which is considered needs to be determined experimentally. Here, 5 proved to be a good choice and the Euclidean distance must be less than 0.1.

After an area with no noise is extracted, the decoder needs to count the white to black transitions, which is straightforward. There is one minor pitfall left: the decoder does not know the whole original image but only the extracted Softstrip from this image. However, a printed Softstrip is surrounded by a *quiet zone*, i.e., a white area which simplifies the bar code location. This means, there is a white to black transition the decoder cannot see. Figure 5.2 illustrates this transition. The outer red rectangle includes the quiet zone and the inner red rectangle highlights what the digital decoder sees. Once the horizontal syn-

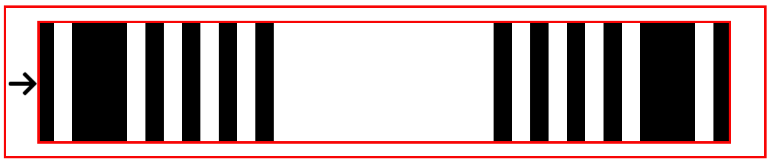


Figure 5.2: Quiet Zone Transition

chronization section is processed, it is removed from the bar code. The structure changes drastically from the horizontal to the vertical synchronization section. This change can also be seen in the previously determined properties. Therefore, the Euclidean distance is used again to locate the boundary between both sections. The number of bars is used to spot the transition from the horizontal to the vertical synchronization section. If the bar count differs at least by 10 and the Euclidean distance of the next 3 lines is less than 0.1, the vertical synchronization starts.

5.4 Row Extraction

As already mentioned in section 4.2.1 on page 36, the encoded row height in the vertical section is not useful for the digital decoder. Here, two approaches for extracting the rows are presented: one with the help of a CNN and one without.

5.4.1 Algorithmic Approach

This approach determines the checkerboard and rack pattern for each *pixel* line and groups subsequent lines with the same pattern together. There are five categories a line can fall into:

- No checkerboard found (invalid)
- "White-Black" checkerboard dibit and the last square of the rack is black (valid)
- "Black-White" checkerboard dibit and the last square of the rack is white (valid)
- "White-Black" checkerboard dibit and the last square of the rack is white (invalid)
- "Black-White" checkerboard dibit and the last square of the rack is black (invalid)

As soon as a pixel line changes from one valid pattern to a different valid pattern, a new row starts. It is possible that, due to noise, patterns changes more often than they are supposed to do. If so, a minimum amount of consecutive lines with the same pattern must occur to be considered as a row.

Section 3.5 highlighted that not every Softstrip is a straight bar but rather a slightly curved bar. This is also illustrated by figure 5.3. This shifting means the rack and checkerboard cannot be simply located by scanning a fixed area. For detecting the checkerboard start,the position of the first black pixel after the start bar is located. Given this pixel is not noise, it can either be the start of the first square in a black-white checkerboard or the start of the second square in a white-black checkerboard. A similar procedure is done for the rack.

The position of the last black pixel in a line is located, which, given it is not noise, can either be the end of the second or the end of the third rack square. These information are collected for all pixel lines. The Softstrip is then divided into multiple sections with the same size and the outlier positions in each section, i.e., positions which occur only once or twice, are removed. Then, the minimum and maximum position for the first black pixel position after the start bar and the last black pixel are determined for each section. The area between the minimum and maximum position for the first black pixel can be used to locate the checkerboard and the area between the minimum and maximum position for the last black pixel can be used to locate the last square of the rack.

(a) Left Shifted Rack (b) Right Shifted Rack

Figure 5.3: Rack
Both rack patterns are from the same Softstrip. Due to some distortions, the distance between the rack and the image boundary can vary.

5.4.2 CNN Approach

The row extraction task can also be interpreted as a classification task. Let's assume a few pixel lines are selected, e.g., 10 lines, which contain more or less two rows. If the task is to split those 10 pixel lines into two rows then there exist 10 different splitting points. Those splitting points can be interpreted as different classes so there are 10 possible classes. A CNN can be trained to perform this task. The checkerboard, rack, or both can be used to train the CNN.

5.5 Row Decoding

Once the rows are extracted, the dibits need to be extracted and decoded. Again, two approaches are presented: one with a CNN and one without.

5.5.1 Algorithmic Approach

A row can be divided by the number of dibits per row to obtain all dibits. However, as figure 5.5 highlights, extracting a dibit with this method is not precise. Such an extracted dibit usually contains some information from its neighbours which makes classifying the dibit more difficult. An alternative approach is to detect the different color areas. Figure 5.4 illustrates this idea. Let's refer to a single square of a dibit as *unit*, i.e., a dibit consists of two *units*. Once those areas are detected, they need to be classified into two categories: areas which are one unit wide and areas which are two units wide. If the right parity dibit is 0 (black-white), then a three unit wide color area is also present (see figure 3.7 on page 26). The missing units can be calculated with equation 5.2:

$$\text{Missing Units} = (14 + \text{nibbles per row} * 4 * 2) - \text{Number of Units} \tag{5.2}$$

The 14 units in equation 5.2 are the ones beyond the data area:

- 2 units for the left start bar
- 1 unit for the space after the start bar
- 2 units for the checkerboard
- 2 units for the left parity dibit
- 2 units for the right parity dibit
- 2 units for the space after the right parity dibit
- 3 units for the rack

The color area sizes can then be sorted and the n widest units (calculated with equation 5.2 are two units wide. A parameter ϵ can be defined to decide whether the widest color area is three units wide or not.

In the last step, the dibits are created by combining two units. Since the unit colors are known, the classification of the dibit is straightforward.

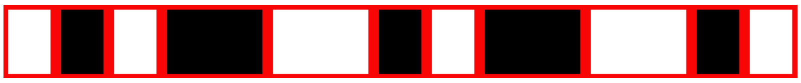


Figure 5.4: Color Area Detection

5.5.2 CNN Approach

First, the row is divided by the number of dibits contained in a row. This method will result in dibits which contain most likely parts of adjacent dibits as shown in figure 5.5. Decoding a single dibit is a binary classification task; a dibit can either be a 0 or a 1. Therefore, a CNN can be trained to classify those dibits.

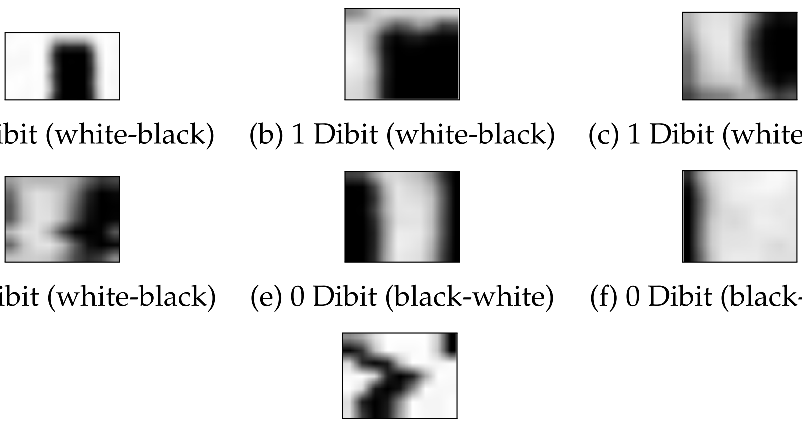
5.6 Data Export

Once the rows are extracted and all dibits are classified, the encoded data can be processed. First, the data section needs to be separated from the vertical synchronization section. As section 3.3.4 on page 27 highlights, they are separated by three 0 bytes. Then, the file header can be processed by using the information from table 3.1 on page 28. In the last step, a binary file is created with the remaining bytes.

5.7 Software Architecture

A pipe and filter architecture is chosen for the implementation. This architecture pattern is most known from the command line where the result of one command can be fed as input into the next command [91].

The overall goal of a user using the digital reader is to extract the encoded data. If the digital reader fails at a single step, the whole decoding process is unsuccessful. For example, the second step extracts the number of bytes in each row. Assume there are 3 bytes in a


(a) 1 Dibit (white-black) (b) 1 Dibit (white-black) (c) 1 Dibit (white-black)

(d) 1 Dibit (white-black) (e) 0 Dibit (black-white) (f) 0 Dibit (black-white)

(g) 1 Dibit (black-white)

Figure 5.5: Dibits

row but due to noise on the bar code image the digital reader thinks there are 4 bytes in each row. Consequently, step five (Decode Rows) is going to fail because the digital reader tries to extract 32 dibits although only 24 dibits exist. As a result, the user is not able to access the encoded data. Their only option at this point is to decode the whole Softstrip by themself. A more user friendly alternative is to tell them which step fails and offer them the choice to perform this step by hand. In the previous example they only need to count the white to black transitions in the horizontal synchronization section and insert it into equation 3.2 on page 26. This only takes a few seconds and is therefore better than decoding the whole Softstrip by hand. However, it requires that the user is familiar with the Softstrip symbology and also willing to perform some steps.

5.8 Summary

This chapter used the knowledge from chapter 3 to build a digital Cauzin Softstrip decoder.

Section 5.1 presented a general approach to recognize Cauzin Softstrips and extract them from an image. This approach will not be implemented in this work but might be useful for future work.

Section 5.2 explained the functionality of the GUI which will be used as an alternative to the automatic bar code recognition. It provides basic image manipulation operations such as rotation, rescaling and Softstrip selection.

Section 5.3 explained how the number of bytes (or nibbles) can be extracted from the horizontal synchronization section. First, a noise reduction step is applied, then a segment is located where each pixel line in this segment is similar to all the other pixel lines in this segment. The Euclidean distance is used to measure the similarity between two lines based on features such as average bar length, minimum bar length, maximum bar length and number of bars.

Section 5.4 presented two approaches to extract the rows from the Cauzin Softstrip. The

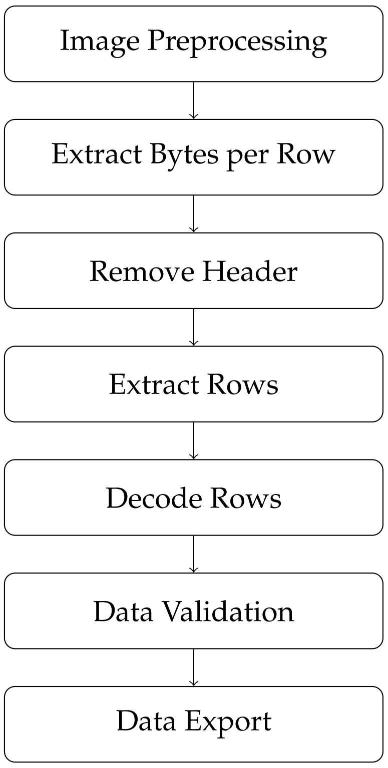


Figure 5.6: Pipe and Filter Architecture

first one is an algorithmic approach and uses the checkerboard and rack to decide whether a pixel line belongs to the current row or is the start of a new row. The second approach uses a CNN to extract rows. A segment with a fixed size is fed into the CNN and it will output the splitting point.

Section 5.5 covered the row decoding. The first approach is an algorithmic approach and analyzes the color areas in a row for a better dibit extraction. The challenge is to decide whether a color area is one, two or three units wide. The second approach uses a CNN. The row is first divided by the number of dibits to extract the dibits. Each dibit is then fed into the CNN to classify it.

Section 5.6 highlighted that the vertical synchronization section is separated by three 0 bytes from the data section. Table 3.1 shows how the data fields can be parsed.

Section 5.7 presented the chosen software architecture. It is a pipe and filter architecture. The reason behind this choice is that, if a step in this architecture fails, it can be performed by the user and fed again into the pipeline which might avoid a decoding failure.

6 Implementation

Chapter 5 presented the general system design to decode the Cauzin Softstrip. This chapter provides more implementation details. First, the used technologies are presented in section 6.1. The following sections cover the different components which were introduced in chapter 5. Section 6.2 presents more details about the implementation of the GUI. The next section, section 6.3, explains the implemented algorithm for the row extraction and the used CNN architecture including the used training data. The last section, section 6.4, provides more details about the row decoding algorithm and the CNN used to classify the dibits.

6.1 Technologies

This section introduces the used technologies to develop the digital Softstrip decoder and explains why those technologies were chosen. The following questions need to be answered:

- What programming language should be used?
- What image processing library should be used?
- What machine learning/deep learning library should be used?
- What GUI library should be used?

The overall goal is to demonstrate a working prototype, not a sophisticated, production-ready software product. Therefore, the following requirements should be kept in mind:

- Runtime is not a priority since the corpus of scanned Softstrips is relatively small.
- The technology should allow fast programming.
- The technology should be well documented.

In order to process the Softstrips, a number of image manipulation techniques have to be applied. Therefore, an image processing library is preferred which simplifies this step. A very popular framework for this task is OpenCV[1]. It is an open source project and supports the programming languages: C++, Python and Java. The OpenCV project provides an excellent documentation[2], including many examples and additional explanations about the computer vision and image processing concepts. Furthermore, many books exist for learning how to use OpenCV[3]. An alternative to OpenCV is the open source project Scikit-Image[4]. Similar to OpenCV, Scikit-Image provides an excellent documentation with many practical examples[5]. In contrast to OpenCV, it only supports the programming language Python.

[1]`https://opencv.org/`
[2]`https://docs.opencv.org/4.0.0-beta/`
[3]`https://opencv.org/books.html`
[4]`https://scikit-image.org/`
[5]`http://scikit-image.org/docs/stable/auto_examples/index.html`

Next, a machine learning library is required which supports CNNs. A huge number of machine learning frameworks exist such as Theano[6], Caffee[7], Tensorflow[8], MXNet[9] and many more. Analyzing each one of them is beyond the scope of this thesis. The most popular machine learning framework is arguably Tensorflow. It was originally developed by Google's Machine Intelligence Research organization and is now open source. Tensorflow is used by many companies such as Google, Intel and Twitter [92] and supports several programming languages such as C/C++, Python, Java, JavaScript, Go and Swift. It can be used together with the high-level deep learning library Keras[10].

Keras is a Python library and optimized for easy and fast development [93]. Only a few lines of code are required to develop a CNN with Keras. Its user friendly design is the main reason why it was chosen for this work. It can be used with different backend engines such as Tensorflow, Theano, or CNTK. However, Keras recommends in their installation guide to use Tensorflow[11]. For this reason, Tensorflow is used as a backend engine for Keras. Since Keras can only be used with Python, Python is used for the implementation. Both image processing libraries fulfill the requirements and also support Python. However, due to personal preferences OpenCV was chosen as image processing library.

Now, a GUI framework for Python needs to be chosen. Once again, there are many possibilities such as Tkinter[12], Pyforms[13], PyQt[14], Kivy[15] and more. Due to the fact that Qt proved to be easy to use in the past, it is used here again. It should be noted that Qt requires a license for commercial usage. However, this is not an issue since this implementation will be published as an open source project.

6.2 Graphical User Interface

This section presents more details about the GUI implementation. The features: rescaling and rotation are implemented by using the *imutils* library[16] which provides basic image processing functions on top of OpenCV. A bug was found in the imutils rotation function during the development phase. An integer division was used to calculate the image center which reduced the image quality after multiple rotations. The bug could simply be prevented by using a floating point division to determine the image center. A bug fix was submitted and accepted[17]. In order to improve the user experience further, the Softstrip selection use case has been optimized. A user only needs to click on the four corners of a Softstrip and it will be extracted automatically. The Softstrip extraction algorithm is shown in figure 6.1.

After a user clicked on the four Softstrip corners, a perspective transformation is applied.

[6]`http://deeplearning.net/software/theano/`
[7]`http://caffe.berkeleyvision.org/`
[8]`https://www.tensorflow.org/`
[9]`https://mxnet.apache.org/`
[10]`https://keras.io`
[11]`https://keras.io/#installation`
[12]`https://docs.python.org/3/library/tk.html`
[13]`https://github.com/UmSenhorQualquer/pyforms`
[14]`https://www.qt.io/`
[15]`http://kivy.org/`
[16]`https://github.com/jrosebr1/imutils`
[17]`https://github.com/jrosebr1/imutils/pull/62`

This is done by using the imutils library. Internally, the imutils library computes the dimensions of the transformed image[18] and then calls the OpenCV API to actually perform the transformation[19]. The next step first converts the color image to a grayscale image and then applies Otsu's thresholding[20] to obtain a binary image.

Usually, a user does not click exactly on the corner positions. In fact, it is recommended to leave a few *mm* distance to the corners, so that a little bit of the quiet zone will be included. This will guarantee that no important information of the Softstrip itself will be removed during the extraction. The Softstrip's boundary is then detected with a contour detection[21]. The contour detection function will find the boundaries of all white objects. Since the left start bar and the rack are black, the colors needs to be inverted so that the whole Softstrip boundary can be detected. Usually, the contour detection will find not only one but many contours, e.g., a contour for the left start bar, contours for each bar in the horizontal synchronization section and many more. For this reason, a Gaussian filter is applied until only one contour is found — the Softstrip contour.

If the detected contour is now used to extract the Softstrip, some of the black areas will be removed, e.g., the middle section in the horizontal synchronization section. Therefore, the convex hull[22] is computed. Once the complex hull is known, it can be used to extract the Softstrip and the Softstrip is ready for further processing.

6.3 Row Extraction

Two methods for extracting the rows from the Cauzin Softstrip were presented in section 5.4 on page 41. More implementation details are presented in this section. First, the algorithmic approach is presented in section 6.3.1 and then the CNN approach is presented in section 6.3.2.

6.3.1 Algorithmic Approach

The steps in the algorithmic row extraction approach are shown in figure 6.3. First, the whole Softstrip is divided into multiple segments where each section consists of 70 pixel lines. The boundaries of the black checkerboard and the last rack square are determined for each segment. The black checkerboard square is located by looking for a white-black-black pixel pattern. This pixel can either be the start of the first or the second checkerboard square (see figure 6.2). The rack boundary is located by looking for the first black pixel in a right-to-left scan (see figure 6.2). The locations are determined for each pixel line in the segment. After all pixel lines are scanned, the collected locations are filtered. Every checkerboard location must be within the *checkerboard window*. This window marks roughly the checkerboard area and is calculated with equation 6.1. In addition, every checkerboard and rack location must occur at least 6 times in the segment. Once the boundaries are known, the checkerboard and rack pattern can be determined for each pixel line. Therefore, the color of the center pixel of the left checkerboard and last rack

[18] https://github.com/jrosebr1/imutils/blob/master/imutils/perspective.py
[19] https://docs.opencv.org/3.4.3/da/d54/group__imgproc__transform.html#gaf73673a7e8e18ec6963e3774e6a94b87
[20] https://docs.opencv.org/3.4.0/d7/d1b/group__imgproc__misc.html#gae8a4a146d1ca78c626a53577199e9c57
[21] https://docs.opencv.org/3.3.1/d3/dc0/group__imgproc__shape.html#ga17ed9f5d79ae97bd4c7cf18403e1689a
[22] https://docs.opencv.org/3.3.1/d3/dc0/group__imgproc__shape.html#ga014b28e56cb8854c0de4a211cb2be656

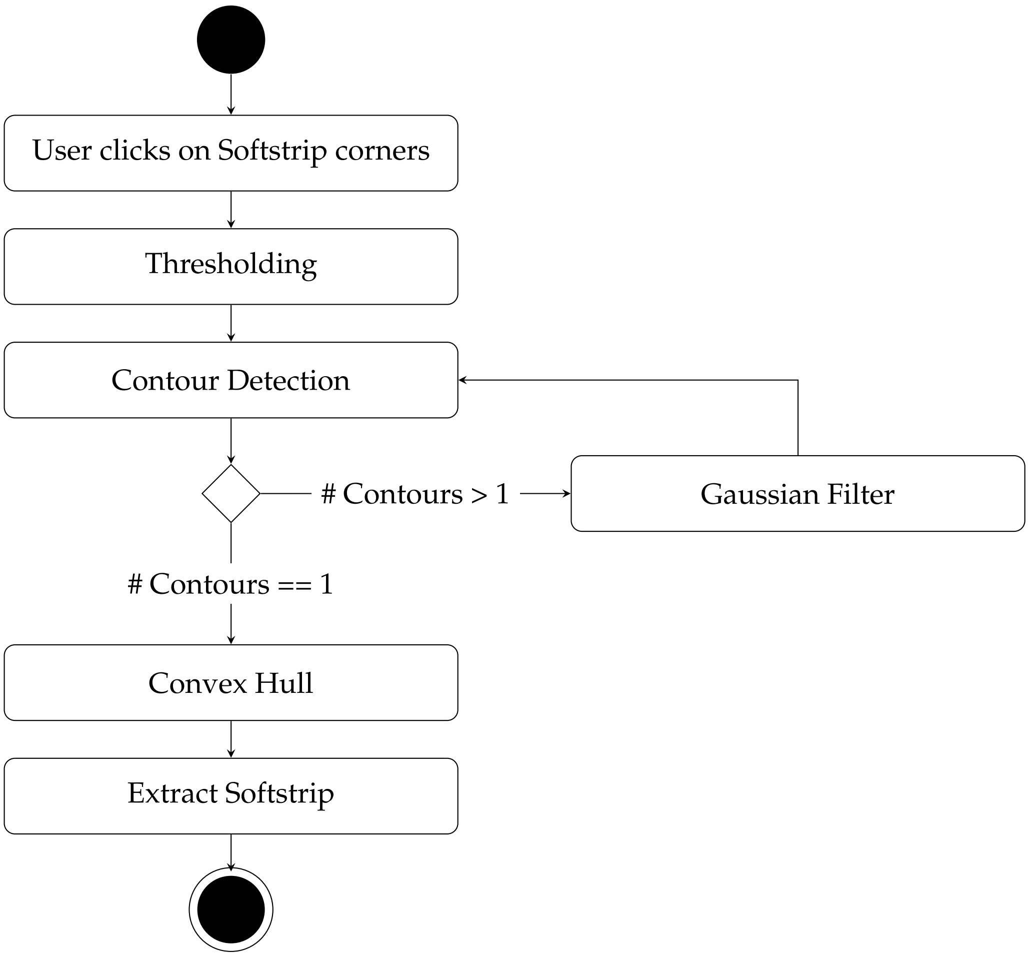
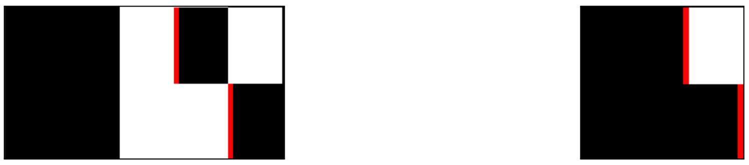


Figure 6.1: Image Preprocessing

square is checked. After all segments are processed, adjacent pixel lines with the same valid pattern, i.e., white-black checkerboard and a rack with a last black square or a black-white checkerboard and a rack with a last white square, are grouped together. Other checkerboard-rack combinations are ignored. There is no minimum amount of pixel lines required to form a row. The reason for this decision is that in some cases only a single pixel line can be found in a row.

$$\begin{aligned} \text{checkerboard start} &= \text{square width} * 2.25 \\ \text{checkerboard end} &= \text{square width} * 4.25 \end{aligned} \tag{6.1}$$



(a) First Checkerboard Square Boundary (b) Last Rack Square Boundary

Figure 6.2: Boundary Detection

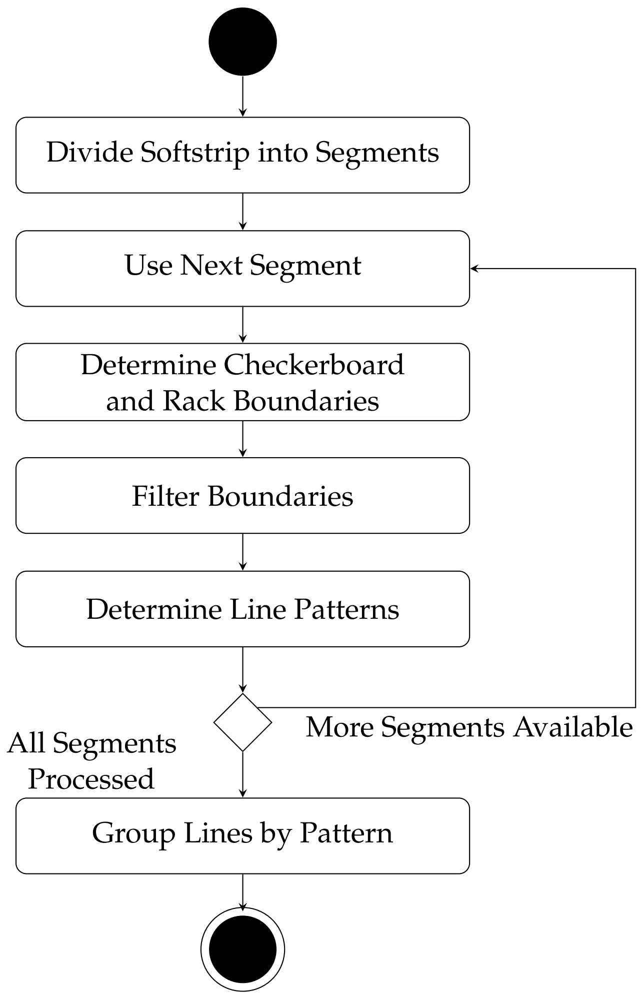


Figure 6.3: Algorithmic Row Extraction

6.3.2 CNN Approach

After the CNN row decoding approach proved to be successful, a CNN for extracting rows was tested as well. First, the training dataset is presented and then the CNN architecture.

6.3.2.1 Training Data

Instead of using the extracted rows by the algorithmic approach, the rows used to train the CNN were extracted by hand. The main reason for this decision is that the algorithmic approach is sometimes only able to extract a single pixel line from a row. The CNN should not learn the mistakes from the previous approach but rather learn how humans would perform this task. Extracting the rows by hand is a tedious and time-consuming task. As a consequence, only a small dataset was created. Furthermore, Softstrips usually consist of only a few hundred rows so in order to reach a dataset size similar to the dibit dataset, a significant amount of the available Softstrips need to be used to train the CNN. Alternatively, synthetic training data could be generated, which might be interesting for future work. Four Softstrips were selected in total to create a training dataset. Those four Softstrips are:

- Quiksort 1
- Valpost 3

- Muldirec 6
- Fspath 8

All Softstrips are from DS2 and were chosen for a particular reason: the algorithmic approach had some trouble to extract the rows exactle because they contain visual artifacts which make the row extraction more difficult. Including those Softstrips into the training dataset should make the CNN more robust to this kind of visual artifacts. There are several options for the input data: the checkerboard and the rack, only the checkerboard, only the rack, or the whole row. Using the whole row does not necessarily provide more information since the data bits could be the same in both rows. Only the rack and the checkerboard change with every row and serve therefore as an orientation for the start and end of a row. During the row extraction process, it was noted that the rack and checkerboard do not always start in the same pixel line. For instance, the checkerboard could start in pixel line 30 and end in pixel line 40 whereas the rack could start in pixel line 35 and end in pixel line 45. So where does the row begin? If pixel line 30 is chosen as the start, the pixel line will most likely contain the rack from the previous row. If pixel line 45 is chosen, however, pixel line 45 will most likely contain the checkerboard from the next row. For this reason, only the checkerboard is chosen as input. Table 6.1 shows the class distribution in the training dataset.

Class (Splitting Point)	**# Samples**
0	1
4	1
5	8
6	141
7	364
8	980
9	976
10	136
Total	2607

Table 6.1: Row Extraction Class Distribution

Those numbers were created by defining a row window which was slid across the Softstrip. The window is 20 pixel lines high. In the first step, the window is placed at pixel line 0 and the manually extracted rows are used to locate the splitting point. The window is then moved to this splitting point and the steps are repeated until the end of the Softstrip is reached. The window height of 20 pixel lines was chosen because it roughly includes two rows in the before mentioned Softstrips. This is also confirmed by the class distribution in table 6.1. Most splitting points are at pixel line 8 and 9. A few outliers are also included, namely line 0, 4 and 5. After reviewing those outliers it was decided to remove them from the training dataset. The remaining classes are still imbalanced. One technique to tackle this problem is to use *oversampling* [94], where samples from the minority classes are randomly selected and replicated. Table 6.2 shows the training dataset after the preprocessing step.

Class (Splitting Point)	# Samples
6	980
7	980
8	980
9	980
10	980
Total	4900

Table 6.2: Row Extraction Class Distribution After Preprocessing

6.3.2.2 CNN Architecture

The CNN architecture is shown in listing 6.1. The input for the CNN is a 20 $\times$ 10 pixel grayscale image — the window. The first convolution layer uses a 5 $\times$ 5 kernel and outputs 64 filters. The ReLU activation function is used. The next layer applies maxpooling to reduce the dimensionality. A 2 $\times$ 2 kernel is therefore used with a stride of 2. Then follows a combination of convolution and maxpooling again. This time, the convolution layer outputs 128 filters. The flatten layer transforms the data to one dimension so it can be used by the following fully connected layer. The dense layer consists of 500 units (nodes) and uses the ReLU activation function. The second fully connected layer uses the same activation function but only 250 units. The last layer consists of 5 units, one for each class, and uses the softmax activation function which is recommended for multi class classification [29]. The model was compiled with the recommended parameters [29]:

Loss Function: Categorical Crossentropy

Optimizer: Adam

Metric: Accuracy

```
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 20, 10, 64)        1664
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 10, 5, 64)         0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 10, 5, 128)        204928
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 5, 2, 128)         0
_________________________________________________________________
flatten_1 (Flatten)          (None, 1280)              0
_________________________________________________________________
dense_1 (Dense)              (None, 500)               640500
_________________________________________________________________
dense_2 (Dense)              (None, 250)               125250
_________________________________________________________________
dense_3 (Dense)              (None, 5)                 1255
=================================================================
Total params: 973,597
Trainable params: 973,597
Non-trainable params: 0
```

Listing 6.1: Row Extraction CNN Architecture

The training results are shown in figure 6.4. The loss on both the training and validation set falls dramatically during the first few epochs. One can see multiple peaks after 20 epochs where the loss on the validation set rises significantly. The training can already be stopped after around 7 epochs with an accuracy of about 95% on the validation set since the validation loss starts fluctuating from there on and only the training loss decreases further.

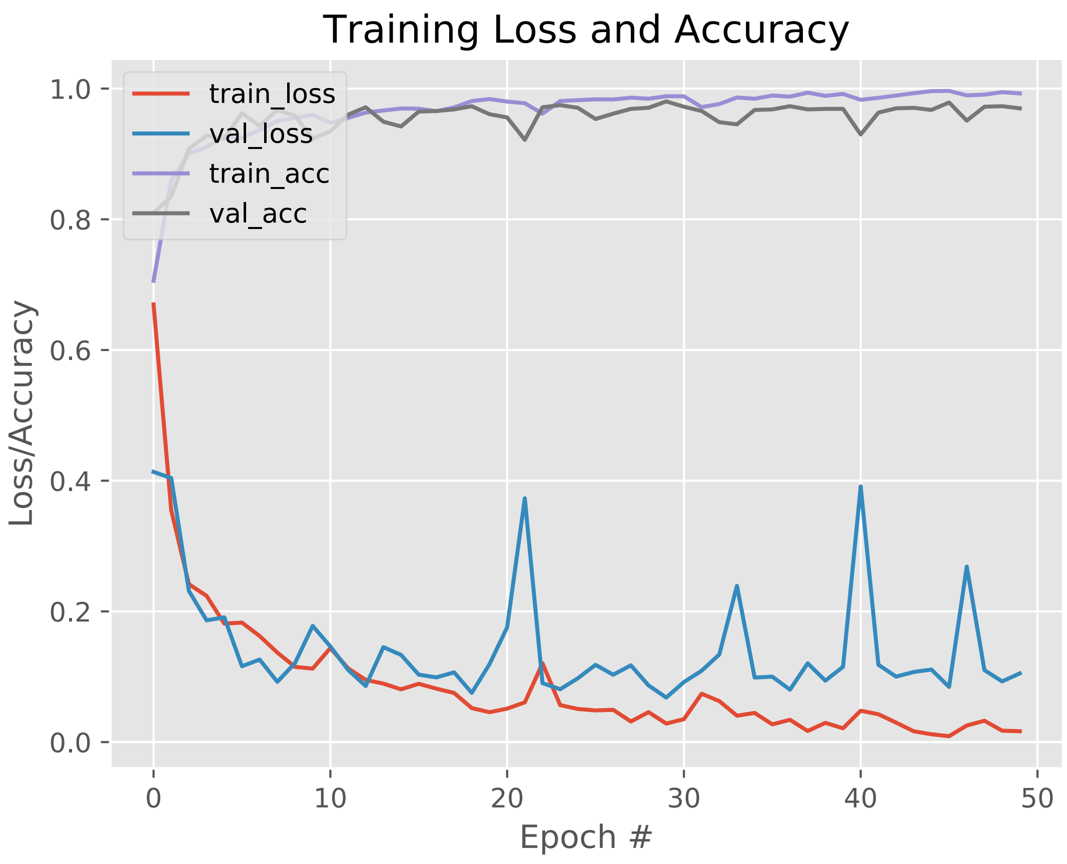


Figure 6.4: Row Extraction Training

6.4 Row Decoding

A few additional techniques are used to improve the row decoding in general. These techniques are:

- Row Splitting
- Row Shifting
- Change dibits with low confidence (only for CNN approach)

Assume the dibits in figure 6.5 are found in a row. The dibits are only damaged in the upper half but could still lead to a wrong decoding. The lower half of the row, however, is completely undamaged. If the row is split into two rows and each half is decoded separately, the upper half will most likely fail whereas the lower half will most likely succeed. In some cases, a single row split might not be enough. For example, if only the upper half

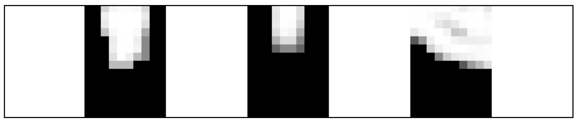


Figure 6.5: Damaged Dibits

in the lower half is undamaged. In the worst case, only a single pixel line in the entire row will lead to a correct result. Therefore, if the row cannot be decoded successfully, a breadth-first-search is applied to split the row into multiple segments. If two row segments are decoded successfully it does not necessarily mean that the decoded dibits are equal in both segments. For this reason, all successfully decoded rows need to be stored and considered later during the checksum test.

All white pixels that are located left to the start bar are removed during the preprocessing. If a Softstrip has a damaged start bar, e.g., it is only half a dibit wide, the whole row is shifted to the left. Figure 6.6 illustrates this. The start bar in the upper row is one dibit

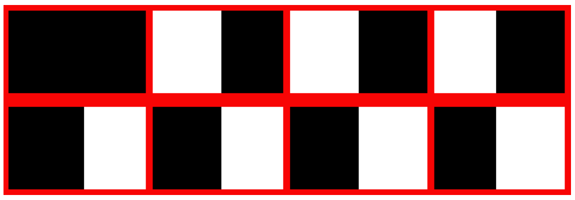


Figure 6.6: Shifted Row

wide. The lower row is identical to the upper row except for the start bar which is only half as wide as in the previous row. As a result, all other dibits are shifted to the left by half a dibit. Both rows contain the same amount of data dibits. However, the data dibits are completely different due to the damaged start bar. This can simply be fixed by shifting the dibits to the right. It is also possible that all data dibits are shifted to the right. This can happen if the start bar is wider than one dibit. For this reason, if the decoding of a row fails, the row can be shifted in both directions and the decoding can be repeated.

Every time the CNN classifies a dibit, it also outputs a confidence value. This confidence value describes how confident the CNN is that its classification is correct. The higher the confidence value the better. As figure 5.5 on page 44 highlighted, it can be difficult to classify the correct class to a dibit. For example, the CNN could be 55% confident that a dibit is a 0, which is a low value. If the row decoding fails, this dibit might be the reason for the failure. So it might be reasonable to flip this dibit and try the decoding again.

6.4.1 Algorithmic Approach

Figure 6.7 shows the flow in the algorithmic row decoding approach. First, the size of each color area is determined. Therefore, each column in the row is scanned and if, for example, the majority of the pixels in a column are black, this column is considered as black. After the color of all columns is known, the sizes of the different color areas can be counted. If the smallest color area is only one column wide, the dilation operation with a 3×3 kernel is applied and the color area sizes are determined again. In the next step, the color area sizes are classified, i.e., whether it is one, two, or three dibit squares wide. The described method in section 5.5 is used to determine the areas which are one dibit wide. If the rack consists of three black squares, it can be checked if the color area before the rack has the same size. If so, it is consequently three squares wide. If the rack consists only of two black squares, the color area before the rack is three squares wide if it is at least two pixels wider than the second widest color area. The final step creates a reduced row where each color square is represented by a single value.

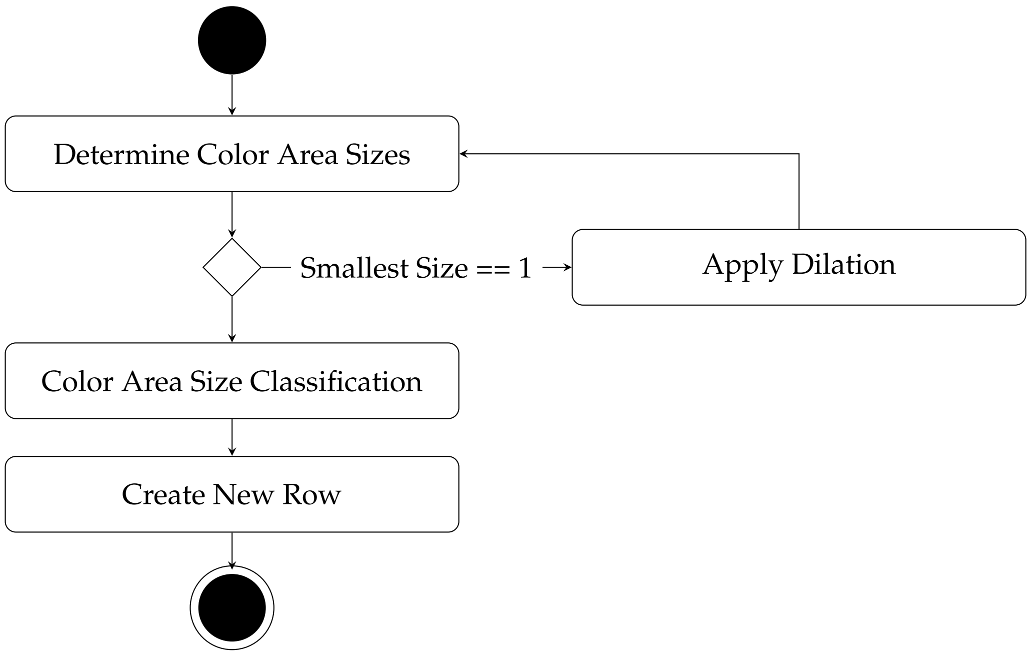


Figure 6.7: Algorithmic Row Decoding

6.4.2 CNN Approach

The algorithmic row decoding approach was designed and implemented first. At that time, DS2 was the only available dataset with real Softstrips and the algorithmic row decoding approach was able to decode about 50% of the Softstrips successfully (the approach could be improved further later). It was decided to try a different approach with a CNN. The successfully extracted and decoded dibits by the algorithmic approach could be used to train the CNN. For this reason, only Softstrips from DS2 were used to train the CNN. Later, the CNN was tested on other datasets as well and proved to be very successful. Those results are presented in chapter 7.

6.4.2.1 Training Data

The very first CNN was trained with data from a single Softstrip — the first Softstrip of the Quicksort program. Although this CNN already performed better than the algorithmic approach it was decided to use more training data. The next CNN was trained with data from five Softstrips:

- First Softstrip of the Quiksort program
- First Softstrip of the Valpost program
- First Softstrip of the Linklist program
- First Softstrip of the Fspath program
- First Softstrip of the Muldirec program

There is no particular reason why these Softstrips were chosen except for the fact that they are the first ones of each program. Again, the data was extracted with the help of the algorithmic row extraction and row decoding method. Table 6.3 shows the class distribution in the dataset.

Total # Samples	# Samples Class 1	# Samples Class 2
43690	17094	26596

Table 6.3: Row Decoding Training Samples

6.4.2.2 CNN Architecture

Two different CNN architecture are presented in this section: a simple one and a more complex one. A detailed evaluation of both architectures on all datasets can be found in chapter 7.

Listing 6.2 shows the simple CNN architecture.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 20, 20, 1)         10
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 10, 10, 1)         0
_________________________________________________________________
dropout_1 (Dropout)          (None, 10, 10, 1)         0
_________________________________________________________________
batch_normalization_1 (Batch (None, 10, 10, 1)         4
_________________________________________________________________
flatten_1 (Flatten)          (None, 100)               0
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 101
=================================================================
Total params: 115
Trainable params: 113
Non-trainable params: 2
```

Listing 6.2: Simple CNN Architecture

The input for the CNN is a 20×20 pixel grayscale image of a dibit. The convolution layer (conv2d_1) uses a 3×3 kernel and outputs one filter. The ReLU activation function is used for this layer. The next layer applies maxpooling to reduce the dimensions. A 3×3 kernel is used with a stride of 2. The dropout layer randomly selects 50% of the nodes and deactivates them. The batch normalization layer normalizes the activations of the convolution layer after each batch; here a batch size of 32 is used. The next layer (flatten_1) transforms the data to one dimension so it can be processed in the fully connected layer (dense_1). The last layer uses the sigmoid activation function and outputs a single value. The model was compiled with the following parameters:

Loss Function: Binary Crossentropy

Optimizer: Adam

Metric: Accuracy

Listing 6.3 shows the architecture of the more complex model:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 20, 20, 20)        520
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 10, 10, 20)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 10, 10, 50)        25050
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 5, 5, 50)          0
_________________________________________________________________
flatten_1 (Flatten)          (None, 1250)              0
_________________________________________________________________
dense_1 (Dense)              (None, 500)               625500
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 501
=================================================================
Total params: 651,571
Trainable params: 651,571
Non-trainable params: 0
```

Listing 6.3: Complex CNN Architecutre

The input is the same as for the previous CNN; a 20×20 pixel grayscale image. This time 20 filters are used to detect features instead of a single one. A 5×5 kernel is used in the first convolution layer (conv2d_1) and the ReLU activation function. As before, the dimensions are reduced with a maxpooling layer which uses a 2×2 kernel and a stride of 2. This time, a second convolution layer is used with a 5×5 kernel which outpus 50 filters. The activation function is again the ReLU function. After the second convolution layer follows a maxpooling layer to reduce the dimensions again. It uses the same parameters as the first maxpooling layer. The flatten layer transforms the data to one dimension and is followed by a fully connected layer (dense_1) with 500 nodes and a ReLU activation function. The last layer (dense_2) uses the sigmoid activation function and outputs a single value. The model was compiled with the same parameters as the first architecture.

Both models achieve almost 100% accuracy on the validation set after 1 epoch. A more detailed evaluation is presented in chapter 7.

6.5 Summary

This chapter provided more details about the concrete implementation. First, the used technologies were presented in section 6.1. The following technologies are used:

Programming Language: Python

Image Processing: OpenCV

Deep Learning: Keras with Tensorflow as backend engine

GUI: PyQt

Section 6.2 focused on the GUI implementation. The imutils library was used to implement the functions: rotation and rescaling. An algorithm was presented to extract the

Softstrip from an image by clicking on the four Softstrip corners.

Section 6.3 first explained the algorithmic row extraction approach in detail and then the CNN approach. The training dataset for the CNN was manually created from four Softstrips. Oversampling was used to create a balanced training dataset. The CNN architecture was presented and the training results were shown. It was shown that the model achieves about 95% accuracy on the validation dataset after about 7 epochs.

Section 6.4 first introduced three methods to improve the row decoding in general: splitting rows, shifting rows and flip dibits with a low confidence value. The last technique can only be applied if the CNN approach is used. Next, the algorithmic row decoding approach was explained in detail. Lastly, the CNN row decoding approach was presented. Two different CNN architecture are used: a simple one and a more complex one. The dibits of five Softstrips were used to train the CNNs.

7 Evaluation

This section evaluates the digital reader's performance. First, an evaluation method needs to be defined. In the case of decoding a Softstrip, the decoding can either be successful or unsuccessful. The optical reader uses the combination of parity bits and the checksum to decide whether the decoding was successful or not. As already pointed out in section 3.4 on page 29, this is not a reliable evaluation criteria.

During the development process one particular Softstrip was decoded which contained a text file. The decoding failed due to an invalid checksum, so the decoded text file was inspected to locate the error. Listing 7.1 shows the content of the decoded text file.

```
RED2BLUE.COM converts Apple II and Mac TEXT or DIF files from
strips for use on a PC. This program adds a Linefeed (LF) after
every Carriage Return (CR). PC's need LF's but Apple does not
supply them. RED2BLUE also turns off high bits to convert Apple
Dos 3.3 text to PC DOS text.

HOW TO USE:
- RED2BLUE.COM is a PC program.
- From DOS type RED2BLUE and press ENTER
- Enter Source filename and Destination filename.
- That's it.

This is a Public Domain utility. Use it freely. You are responsible
for its proper use. There is no$warranty.
```

Listing 7.1: Decoded Text File With Invalid Checksum

As one can see, a single byte is wrongly decoded, i.e., the $ sign between no and warranty is supposed to be a whitespace. Once the error was known, a few parameters were changed in the source code and the Softstrip was decoded again. The decoded text file was then inspected again to make sure that everything was correctly decoded. The content of the decoded file is shown in listing 7.2.

```
RED2BLUE.COM converts Apple II and Mac TEXT or DIF files from
strips for use on a PC. This program adds a Linefeed (LF) after
every Carriage Return (CR). PC's need LF's but Apple does not
supply them. RED2BLUE also turns off high bits to convert Apple
Dos 3.3 text to PC DOS text.

HOW TO USE:
- RED2BLUE.COM is a PC program.
- From DOS type RED2BLUE and press ENTER
- Enter Source filename and Destination filename.
- That's it.

This is a Public Domain utility. Use it freely. You are responsible
for id?dproper use. There ij$warranty. ?%
```

Listing 7.2: Decoded Text File With Valid Checksum

As one can see in listing 7.2, the text file is still not correct. Instead of fixing the whitespace error, even more errors were added which resulted in a correct checksum again.

The only way to be absolutely confident that the Softstrip was decoded successfully, is to inspect the decoded file. If the decoded file is a short text file with text in English, this might take only a few seconds. However, the Softstrip was mainly used to encode software so usually the decoded file contains a few hundred lines of source code. Finding a subtle error in a source code one is not familiar with can take a while. Since inspecting each file is not feasible, this section refers to a Softstrip as successfully decoded if both the checksum and all parity checks are correct.

7.1 Accuracy

First of all, four different methods are evaluated in this chapter. They are shown in table 7.1.

Method	Row Extraction	Row Decoding
1	Algorithmic	Algorithmic
2	Algorithmic	CNN (Simple)
3	Algorithmic	CNN (Complex)
4	CNN	CNN (Simple)

Table 7.1: Decoding Methods

Table 7.2 shows the number of successfully decoded Softstrips for each decoding method. It should be noted that the image quality of the Softstrips in DS1, DS2 and DS5 could be controlled, i.e., the size of the generated Softstrips could be controlled, the resolution of the scanned Softstrips could be controlled and most important a Softstrip could be rescanned if the first scan was distorted. As a result, almost all of the Softstrips from those datasets are decodable. This is even more impressive if one considers some of the visual artifacts shown in section 3.5 on page 30. The image quality of the Softstrips in the datasets DS3 and DS4 could not be controlled but some methods are still able to decode about 50% of them successfully.

Method	DS1 (870)	DS2 (40)	DS3 (58)	DS4 (159)	DS5 (103)	Total (1230)
1	870	31	-	-	85	986
2	870	38	31	86	99	1124
3	870	38	35	77	99	1119
4	870	39	0	0	1	910

Table 7.2: Decodable Softstrips

DS1 can be decoded successfully by all methods which is no surprise since it only includes digitally generated Softstrips without any noise. Most of the Softstrips in DS2 can also be decoded by all methods. Method 4 achieves the best result on DS2 since both the row extraction and row decoding CNN were trained with data from DS2. The results of the different methods on DS3 and DS4 vary drastically. The results of method 1 are not included at all for those datasets. The reason behind this decision is explained in section 7.2. The fourth method is not able to decode a single Softstrip from DS3 and DS4. The decoding fails because the CNN row extraction method is not able to extract the rows properly. As a consequence, the CNN row decoding method cannot decode the dibits and the whole decoding fails. One reason for this is that the row extraction CNN was only trained with a few training samples. This was clearly not sufficient to learn the relevant features. However, as already mentioned in section 6.3 creating a large training dataset manually is a time consuming task and was therefore not feasible for this thesis. Method 2 and 3 are still able to decode almost 50% of the Softstrips in DS3 and DS4. Considering the bad quality of the Softstrips in those datasets, this is still a very good result. All methods except the last one perform well on DS5. Again, the small training dataset is the main reason for the poor performance of method four.

A closer look to method 2 and 3 shows that their performance is almost identical although the CNN in method 3 uses a far more complex architecture. This clearly shows that the simple architecture which was chosen in method 2 is sufficient to detect the relevant features in a dibit. This is also confirmed by the overall result: method 2 achieves the best result and is able to decode 1124 of the 1230 Softstrips successfully (about 91%).

7.2 Multiple Rows

Section 6.4 on page 54 introduced three strategies that can be applied if the row decoding fails. Those strategies are: row splitting, row shifting, and changing dibits with a low confidence value. Usually, all three strategies are applied. The problem is that they might return different successfully decoded rows. At this point, there is no option to decide which one is the right solution so they need to be tested during the checksum test. If multiple rows have more than one solution, all possible combinations need to be tested. The combinations can be determined with the cartesian product and the total number of combinations can be determined with equation 7.1, where R_i is the row at position i.

$$|Combinations| = |R_0| \times |R_1| \times \cdots \times |R_n| \tag{7.1}$$

The Softstrip rows in DS3 and DS4 could not be decoded with method 1 on the first try, i.e., the aforementioned strategies had to be applied. Since they had to be applied for almost every row and returned multiple possible solutions for each row, the total number of combinations was extremely high. Even if the checksum was correct in those cases, it was sometimes just pure coincidence. This was also confirmed by the file meta information such as filename or strip id which did not make any sense in the successfully decoded cases. For this reason, the results of method 1 for DS3 and DS4 are not included in table 7.2.

This highlights also a second problem. The first combination which passes the checksum test, will be accepted. However, there is no guarantee that this combination is the right one. A different approach is to only use the most common solution from the different decoding strategies instead of using them all. Table 7.3 compares both approaches.

Strategy	Method	DS1	DS2	DS3	DS4	DS5	Total
Cartesian Product	**3**	870	38	35	77	99	1119
Most Common	**3**	870	34	34	31	77	1050

Table 7.3: Comparison of Cartesian Product and Most Common Strategy

As one can see, the cartesian product strategy performs better than the most common strategy. Most notable is the difference for DS4. However, it should be noted that the cartesian product strategy most likely creates some false positives.

7.3 Runtime Performance

Although the runtime performance has a low priority this section provides some runtime statistics. The digital decoder was tested on the following system:

Operating System: Arch Linux

CPU: Intel i7-7700K (8) with 4.500GHz

Memory: 16 GB

Python: Version 3.6.6

The performance of the different methods on DS1 is shown in table 7.4. The Softstrips in this dataset can be decoded extremely fast. Method 1 is by far the fastest method for this dataset, with an average decoding time of only 3 seconds. The slowest method is method 4, with an average decoding time of 27 seconds. As one can see, the methods with a CNN are significantly slower than method 1.

Method	Minimum	Maximum	Average
1	00:02	00:04	00:03
2	00:07	00:37	00:18
3	00:11	00:26	00:18
4	00:07	00:55	00:27

Table 7.4: DS1 Runtime Performance
Format: MM:SS

Table 7.5 shows the performance of the different methods on DS2. Again, the fastest decoding was performed by method 1. However, one can see that the maximum runtime is extremely high. Some rows could not be decoded successfully on the first try. Therefore, the different row decoding strategies had to be applied. As a result, all combinations of the valid rows had to be tested during the checksum check which increases the runtime dramatically. It should be noted that the decoding was stopped after 21 minutes. Since the average time is still low, only a few Softstrips reached the time limit. Method 3 is the only one that does not reach the time limit on this dataset and has therefore the lowest average runtime.

Method	Minimum	Maximum	Average
1	00:04	21:00	01:14
2	00:12	21:00	00:55
3	00:15	00:30	00:17
4	00:14	21:00	01:22

Table 7.5: DS2 Runtime Performance
Format: MM:SS

Table 7.6 shows the performance of the different methods on DS3. Since the decoding results of method 1 for DS3 and DS4 were not included in table 7.2, they are left out here as well. At first sight, method 4 seems to be the fastest method on DS3. However, table 7.2 showed that method 3 is not able to decode a single Softstrip from DS3. In fact, the row extraction fails early so the whole decoding process stops early. As a result, the average runtime is low. One can see that method 3 and 4 reached the maximum time limit again. Compared to table 7.5, the average runtime for both methods increased by a few minutes. Due to the lower image quality, more rows have multiple possible solutions which increase the number of combinations during the checksum test and therefore the overall runtime.

Method	**Minimum**	**Maximum**	**Average**
1	-	-	-
2	00:17	21:00	06:28
3	00:15	21:00	05:07
4	00:21	03:23	01:22

Table 7.6: DS3 Runtime Performance
Format: MM:SS

Table 7.7 shows the performance of the different methods on DS4. The results of method 1 are again not included for the same reason as before. The results look similar to the ones in table 7.6 which is no surprise since the Softstrip image quality is also similar.

Method	**Minimum**	**Maximum**	**Average**
1	-	-	-
2	00:29	21:00	06:55
3	00:29	21:00	06:39
4	00:32	21:00	03:53

Table 7.7: DS4 Runtime Performance
Format: MM:SS

Table 7.8 shows the performance of the different methods on DS5. Although the maximum time limit was reached by all methods the average runtime is relatively low, except for the average time of method 4. Method 2 and 3 can decode the same amount of Softstrips from DS5 but the average time of the method 2 is almost twice as high as the average time of method 3. Apparently, the simple CNN architecture used in method 2 does make more decoding errors than the complex CNN architecture used in method 3, which is why the other decoding strategies have to be applied and consequently increase the runtime.

Method	**Minimum**	**Maximum**	**Average**
1	00:07	21:00	02:39
2	00:13	21:00	01:44
3	00:08	21:00	00:51
4	00:11	21:00	05:24

Table 7.8: DS5 Runtime Performance
Format: MM:SS

To conclude this section, the better the image quality the faster the decoding. The different row decoding strategies are useful to increase the number of successfully decoded Softstrips but they can increase the runtime significantly. Especially, if the image resolution is high. In this case, a row consists of more pixel lines which means there are more splitting possibilities if a row cannot be coded on the first try.

7.4 Limitations

So far the digital Softstrip decoder was only tested with high resolution images. This section evaluates the performance of the digital decoder on lower resolution images. In order to reduce the overall runtime, only a subset of DS5 is chosen for this task. The dataset contains 42 Softstrips in total. In each iteration, the image resolution is reduced by additional 10%, i.e., the first iteration uses the original image with 100% of the image resolution, the second iteration reduces the original image resolution by 10%, the third iteration reduces the original image resolution by 20% and so on. The Softstrips in DS5 were scanned with 1200 DPI and have an image resolution of about 800 × 8000 pixels. The results are shown in table 7.9.

Image Resolution	**Method 1**	**Method 2**
100%	24	42
90%	28	40
80%	32	40
70%	28	38
60%	17	37
50%	27	33
40%	29	Error
30%	8	Error
20%	0	Error
10%	0	Error

Table 7.9: Image Resolution Test

Table 7.9 shows that the lower the image resolution the less Softstrips can be successfully decoded with method 2. The results for method 1 are fluctuating. It is interesting that method 1 performs better with slightly lower image resolutions. Even after the image resolution was reduced by 70% it could still decode 8 Softstrips successfully. The lowest image resolution that could be decoded by method 2 was about 380 × 4800. Method 1 could even decode Softstrips with an image resolution of about 220 × 2800.

7.5 Summary

Different aspects of the digital Softstrip reader were evaluated in this chapter. Table 7.2 showed that method 2 is the most successful method that can decode 1124 of the 1230 successfully (about 91%). The worst performing method is still able to decode 910 of the 1230 Softstrips. Section 7.3 evaluated the runtime performance of the different methods and showed that the fastest decoding was done in only 2 seconds. However, the decoding increased dramatically as soon as the image quality became worse. In some cases the decoding had to be stopped after 21 minutes. Section 7.4 showed that the digital Softstrip decoder is also able to deal with different image resolutions. However, method 2 works better with high resolution images.

In general, there are two reasons for a decoding failure: either the row extraction or the row decoding fails. However, it is more likely that the dibit extraction fails than the dibit classification during the row decoding. If the input image for the CNN is of good quality, the classification will most likely be correct. In many cases, however, the classification is even for a human difficult. In order to improve the row decoding, the dibit extraction needs to be improved. The most reliable part of the decoding pipeline is the header extraction. It can determine the correct number of bytes in each row for all Softstrips in the corpus.

8 Discussion & Future Work

First, more information are provided about what happened after the Cauzin Softstrip disappeared. Then, a conclusion is drawn before the last section presents improvements and research questions for future work.

8.1 Datastrip

It should be noted that the information presented in this section are based on various website snapshots of the Datastrip website from the wayback machine of the internet archive [1]. Many of the statements could not be verified with other sources. However, the statements are informative and therefore included in this section.

Unfortunately, the Cauzin Softstrip could not establish itself as a software distribution alternative. However, this does not mean that the Softstrip vanished forever. In 1993, Jack Walker acquired the Softstrip patent rights and formed Datastrip Holdings [95]. The Cauzin Softstrip was rebranded as Datastrip and its new field of application was identification cards. It is interesting that Cauzin Systems was the original assignee of the patent for this particular use case [21], but for unknown reasons they did not pursue this idea. From then on the Datastrip was used for driver's licenses, id cards, social security cards and more.

A new series of bar code readers were released together with the Datastrip [96]:

Wand Reader: Two variants of wand readers were available: one with an integrated LCD display and one which required a host for displaying information. It could be used like a typical handheld scanner and was able to read Datastrips on a variety of documents.

Swipe Reader: Similar to the Wand Reader, the Swipe Reader was available in two variants — with an integrated LCD and without. In order to decode Datastrips, the card or document just needed to be swiped.

LCD Reader: The LCD Reader was the premium model. It was a motorised device and could read standard ISO/IEC 7810 cards in approximately one second. The scanned information was displayed on the integrated LCD display.

A few years later, the Datastrip was used to store biometric data such as fingerprints. For this purpose, a new version of the Datastrip was developed — the 2DSuperscript. The main difference between both bar codes was [97]:

Datastrip: The usage of the Datastrip is intended for the public domain.

2DSuperscript: The 2DSuperscript can be used with proprietary modules such as compression or fingerprinting. These modules require additional licenses.

[1] `https://archive.org/`

In order to check if the finger print of a person matches with the stored one in the Datastrip, the bar code reader needed to be extended with a fingerprint scanner. For this reason, new bar code readers were produced [98]:

DSVerify2D:	A portable bar code reader for 2DSuperscript, Datastrip2D, PDF-417 and OCR-B. It also includes a fingerprint sensor for biometric verification.
PCRead2D:	Reads bar codes on identification cards and requires a connection to a PC.

The transition from distributing software to storing personal information on identification cards proved to be very successful. A few examples of its application are:

- Continental Trust Bank in Nigeria introduced the Datastrip for their customer saving accounts [99].
- Airport Aviation Services Sri Lanka introduced the Datastrip on ID cards for their employees [100].
- Integration of the Datastrip into Synercard's Asure ID photo identification software [101].
- Datastrip on national ID cards in Cambodia [102].
- Diamond Bank in Nigeria on banking cards [103].
- Asia Pacific Economic Cooperation introduces the Datastrip on business travel cards [104].
- Datastrip on national ID cards in Yemen [105].
- Datastrip on social security ID cards in Zimbabwe [106].
- Datastrip on student id cards [107].
- Liberian International Ship and Corporate Registry introduces the Datastrip on seafarer identity cards [108].
- San Andrés Island in Colombia introduces the Datastrip on new resident ID cards [109].

Both bar codes are improved versions of the original Cauzin Softstrip [110]. An example of a Datastrip is shown in figure 8.1. Most notably, the Datastrip replaced the rack by a black bar (185), similar to the start bar on the left side (125). The checkerboard is removed as well. Instead, a 4-bit gray code row address pattern, one on the left side (130) and one on the right side (190), is added. The row address pattern is incremented in every row. Incrementing a digit in a gray code always changes only a single bit [111]. Table 8.1 shows the gray code for a 4-bit digit [111].

Apparently, the Datastrip does not use dibits anymore. The patent states:

> *The fundamental unit for encoding information is called a "bit area", which may be printed or blank ... information in a preferred embodiment is encoded in the code using a direct binary encoding method wherein a bit area in the printed code may represent a bit of user data. This achieves a significant improvement in information density over the dibit encoding methodology used*

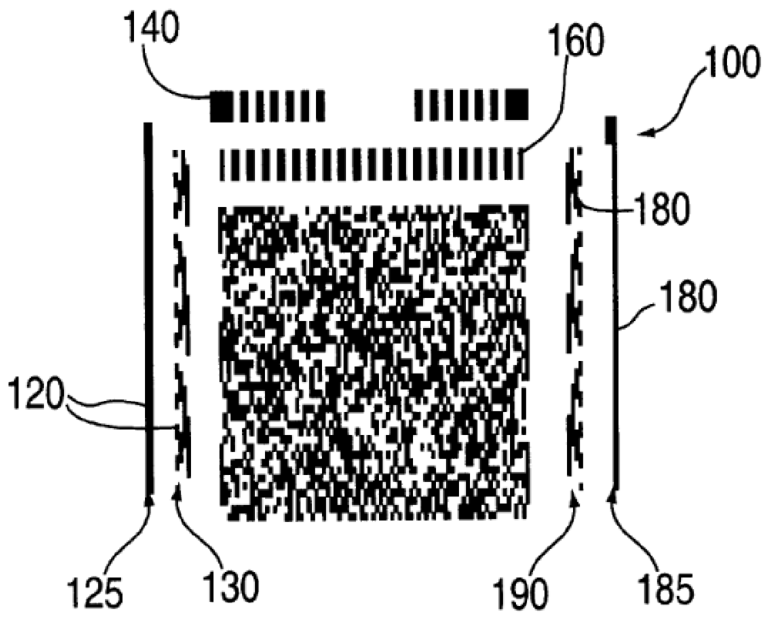


Figure 8.1: Datastrip
This figure is originally from [110].

Decimal	Gray Code	Decimal	Gray Code
0	0000	8	1100
1	0001	9	1101
2	0011	10	1111
3	0010	11	1110
4	0110	12	1010
5	0111	13	1011
6	0101	14	1001
7	0100	15	1000

Table 8.1: Gray Code

in U.S. Pat. No. 4,782,221.
— Two-Dimensional Printed Code For Storing Biometric Information And Integrated Off-Line Apparatus For Reading Same [110]

This is also confirmed by the row address pattern. Both row address patterns consist of four bit areas, one bit area for one bit of the gray code. If the Datastrip were using dibits, it would require two bit areas for one gray code bit. Additionally, if one looks closely at the Datastrip in figure 8.1 they can see four or even more black bit areas side by side. These areas would be invalid in a Cauzin Softstrip. However, the book *Bar Codes: technology and implementation* devotes one section to the Datastrip and describes it as follows:

A Datastrip Code consists of a matrix pattern, comprising very small, rectangular black and white areas (or Dibits).
— Datastrip Code in Bar Codes: technology and implementation [112]

This gives the impression that the Datastrip still uses dibits. However, the information in the patent strongly indicates that dibits were replaced with single bit areas.

Instead of parity and checksum checks, the Datastrip uses the Reed-Solomon error correction algorithm. Different error correction levels can be selected in the Datastrip bar code creation software, which allow a restoration of the Datastrip even if 50% of its content is damaged [113].

Eventually, Datastrip Inc. disappeared and the Datastrip rights were aquired by Smart Media Innovations in 2011 [114].

8.2 Conclusion

Three research questions were asked in the beginning which can be answered now:

Is it possible to decode the Cauzin Softstrip with a digital reader?

This thesis created a digital Softstrip and presented different methods for extracting and decoding the Softstrip rows. All of them are able to decode most of the Sofstrips in the corpus. The best method is able to decode about 91% of the 1230 Softstrips so this question can be answered with a clear yes.

What are the main difficulties while decoding the Cauzin Softstrip?

The first difficulty is to extract the bar code rows exactly. Although the Cauzin Softstrip uses two alignment guides, the checkerboard and the rack, distortions or noise can make the row extraction more complicated. Once the row extraction is finished, the next difficulty is to extract and to classify the dibits in a row. The included parity bits are not sufficient to detect all decoding errors. The same is true for the checksum. A better error detection or even error correction method would have made the decoding a lot easier.

What are the limitations of the digital reader?

Chapter 7 showed that the decoding is more successful on high resolution images. This is not an issue if one can control the scanning quality. However, it makes decoding Softstrips where one cannot control the image quality more difficult. In addition, the decoding time increases dramatically if the image quality becomes worse.

One of the main challenges during the development of the digital Softstrip decoder was the limited amount of available Softstrips. In the beginning of this work, only digital generated Softstrips and scanned Softstrips from the book *Animated Algorithms* were available. Therefore, the digital decoder was optimized to decode those Softstrips. Later, a StripWare collection was found on ebay which could be obtained. The new Softstrips introduced many new challenges to the digital decoder. This happened again when the scanned Softstrip collections from the internet were found. Due to the small dataset in the beginning, a more specialized decoder was created first which needed to be more generalized every time a new dataset was introduced. Nevertheless, the main goal of this thesis has been fulfilled: Giving people access to the encoded data in the Cauzin Softstrip again without the need of an optical Softstrip reader.

8.3 Future Work

8.3.1 Softstrip Recognition

In the beginning of this work, a GUI was designed and implemented which allows user to extract multiple Softstrips from an image. During the implementation of the digital decoder itself, it was noticed that one has to be extremely carefully while extracting the Softstrips. Even minor rotations of 0.01° might lead to a decoding failure. For this reason, an image manipulation program such as Gimp was used to extract the Softstrips from an image. However, it required sometimes several tries until the Softstrip was extracted correctly and could be successfully decoded by the digital reader. Even though the Softstrip was extracted while using a 500% zoom. Therefore, an automatic Softstrip recognition and extraction system might be implemented in future work.

8.3.2 Optical vs Digital

Chapter 7 evaluated the performance of the digital Softstrip reader. Although the results are remarkable it would be interesting to see how the digital reader compares to an optical Softstrip reader. Unfortunately, a working optical reader could not be obtained for this work so this comparison needs to be done in a future work.

The digital reader has some minor advantages compared to the optical reader:

1. Alignment marks are not required.
2. It is easier to fix a damaged Softstrip digitally.
3. The Softstrip size is not restricted.

Point one is just a minor bonus that might be useful if a Softstrip is missing the alignment marks. However, the alignment marks of all the Softstrips used in this thesis were intact. Only the digital generated ones do not have alignment marks but they were not created by the official Softstrip software.

The next point might be more useful. The Cauzin Softstrip Reader uses infrared light for scanning the bar code. This has the huge advantage that, if, for example, coffee is spilled over the bar code, it is still decodable [115]. One can even use a pen to write on the bar code and the optical reader is still able to decode the Softstrip[115]. This can be used as a copy protection. If such a Softstrip is copied, the pen becomes printer ink and will be detected by the optical reader and will cause a decoding error. Although users benefit from this feature, it makes repairing a Softstrip more complicated. It is not possible to take a pen and draw, for example, a missing rack. A special pen is required whose ink matches the reflectance property of printer ink. On the other hand, repairing a damaged Softstrip for the digital reader only requires an image manipulation program such as Gimp[2].

Lastly, it is not required to restrict the Softstrip's size if the digital reader is used. Instead of splitting large files into multiple Softstrips, a single large Softstrip can be generated. This removes the overhead of the additional file header on the other Softstrips. However, this feature is only nice to have since no new Softstrips are generated anymore.

Scanning and decoding a single Softstrip with the optical reader takes about 30 seconds

[2]`https://www.gimp.org/`

[15]. If a file consists of multiple Softstrips, the optical reader needs to be adjusted again for each Softstrip. This increases the total decoding time. If the optical reader is not adjusted carefully enough, a decoding error occurs and a rescan is required, increasing the total decoding time further. It would be interesting to know how common this is.

Advertisements for the Cauzin Softstrip Reader often highlighted its accuracy:

> *In fact, strip data accuracy is checked via parity bits at the beginning and end of every data line, as well as by a strip checksum. This unique design results in an undetected bit error rate of less than one bit error per 10 billion bits.*
> — The Art of Stripping: Marketing Material [115]

As section 3.4 on page 3.4 and chapter 7 on page 7 showed, the error detection methods are not sufficient to detect all errors and a fail is relatively common. Does this only occur while using the digital reader or also with the optical reader?

The last interesting point to test is whether the digital and optical reader deal differently with specific errors. For example, which reader performs better if:

- the rack is damaged or even missing?
- the checkerboard is damaged or even missing?
- the printer ink is smeared in the data area?
- some dibits are damaged or even missing?

Bibliography

[1] A. Denso, "QR Code Essentials," *Denso Wave*, vol. 900, 2011.

[2] N. Tredennick, "Microprocessor-based computers," *Computer*, vol. 29, no. 10, pp. 27–37, 1996, ISSN: 0018-9162. DOI: `10.1109/2.539718`.

[3] M. Real and R. Warren, *A Revolution in Progress: A History of Intel to Date*. Intel Corporation, Corporate Communications Department, 1984, p. 14. [Online]. Available: `https://www.intel.com/Assets/PDF/General/15yrs.pdf` (visited on 10/30/2018).

[4] M. R. Betker, J. S. Fernando, and S. P. Whalen, "The history of the microprocessor," *Bell Labs Technical Journal*, vol. 2, no. 4, pp. 29–56, 1997.

[5] Apple Computer Inc., *Apple II Reference Manual*, 1979, p. 88. [Online]. Available: `https://archive.org/details/Apple_II_Reference_Manual_1979_Apple` (visited on 10/30/2018).

[6] Commodore Business Machines Inc., *PET Personal Computer User Manual*, Oct. 1978. [Online]. Available: `https://archive.org/details/PET_2001-8_Personal_Computer_User_Manual_1978-10_Commodore` (visited on 10/30/2018).

[7] P. Diskin, "Nintendo Entertainment System Documentation," *Tokyo: Nin-tendo*, 2004.

[8] *Compute!'s Gazette*, vol. 2, no. 2, 168ff, 1984. [Online]. Available: `https://archive.org/stream/1984-02-computegazette/Compute_Gazette_Issue_08_1984_Feb#page/n169` (visited on 10/30/2018).

[9] Apple Computer Inc., "Introducing Disk II," *Contact 2*, p. 2, Jun. 1978.

[10] "Apple II Accessories," *The Apple Orchard*, p. 66, 1980. [Online]. Available: `https://archive.org/stream/Apple_Orchard_1980_Fall_v1n2#page/n67` (visited on 10/30/2018).

[11] K. Budnick, *Bar Code Loader*. BYTE Publications Inc., 1977.

[12] K. Savetz, D. Picard, K. Garretson, and N. Enzenauer. (Dec. 2017). Databar OSCAR, ANTIC The Atari 8-bit Podcast, [Online]. Available: `https://ataripodcast.libsyn.com/antic-interview-321-databar-oscar` (visited on 10/30/2018).

[13] J. Romero. (no date). Bongo's Bash, [Online]. Available: `https://rome.ro/bongosbash` (visited on 10/30/2018).

[14] Cauzin Systems Inc., *Softstrip Business Advantage*, 1986. [Online]. Available: `https://archive.org/stream/CauzinSoftstrip/Softstrip%20Business%20Advantage#page/n0/mode/2up` (visited on 10/30/2018).

[15] L. G. Johnson, "New Solutions to Storage Hassles," *ABAJ*, vol. 72, p. 90, 1986.

[16] Cauzin Systems Inc., "It's amazing what you can reveal when you strip," *Byte*, vol. 11, no. 3, pp. 294–295, Mar. 1986.

[17] "This issue's Softstrip: Income tax spreadsheet," *II Computing*, vol. 2, no. 3, pp. 58–59, Feb. 1987.

[18] "Softstrip Programs are everywhere," *inCider*, vol. 4, no. 9, pp. 78–79, Sep. 1986.

[19] M. P. Barnett and S. J. Barnett, *Animated Algorithms: A Self-teaching Course in Data Structures and Fundamental Algorithms*. New York, NY, USA: McGraw-Hill, Inc., 1986, ISBN: 0-07-003797-3.

[20] "The Editors' choice awards for 1986," *MacUser*, vol. 3, no. 1, pp. 55–56, Jan. 1987.

[21] J. Glaberson and S. Santulli, "Card reader for receiving a card bearing an imprinted data strip, self positioning the card in a pre-determined position and scanning the imprinted data strip in two directions," pat., US Patent 4,886,957, 1989.

[22] K. Savetz, B. Brass, and P. D'Amato. (Jan. 2016). Cauzin Softstrip, ANTIC The Atari 8-bit Podcast, [Online]. Available: `https://ataripodcast.libsyn.com/antic-interview-115-bob-brass-and-peter-damato-cauzin-softstrip` (visited on 10/30/2018).

[23] "It's amazing what you can reveal when you strip," *Byte*, vol. 11, no. 2, pp. 154–155, Feb. 1986.

[24] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, `http://www.deeplearningbook.org`.

[25] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-Based Learning Applied to Document Recognition," vol. 86, pp. 2278 –2324, Dec. 1998.

[26] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet Classification with Deep Convolutional Neural Networks," in *Advances in Neural Information Processing Systems 25*, Curran Associates, Inc., 2012, pp. 1097–1105. [Online]. Available: `http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf` (visited on 10/30/2018).

[27] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[28] Y. Yang, Y. Li, C. Fermüller, and Y. Aloimonos, "Robot Learning Manipulation Action Plans by"watching"unconstrained Videos from the World Wide Web.," in *AAAI*, 2015, pp. 3686–3693.

[29] F. Chollet, *Deep Learning with Python*, eng, 1st ed. Manning Publications, 2017, ISBN: 9781617294433.

[30] M. D. Zeiler and R. Fergus, "Visualizing and Understanding Convolutional Networks," *CoRR*, vol. abs/1311.2901, 2013.

[31] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 1–9.

[32] A. Cauchy, "Méthode générale pour la résolution des systemes d'équations simultanées," *Comp. Rend. Sci. Paris*, vol. 25, no. 1847, pp. 536–538, 1847.

[33] S. Ruder, "An overview of gradient descent optimization algorithms," *arXiv preprint arXiv:1609.04747*, 2016.

[34] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[35] T. Tieleman and G. Hinton, "Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude," *COURSERA: Neural networks for machine learning*, vol. 4, no. 2, pp. 26–31, 2012.

[36] P. J. Werbos, "Backpropagation through time: What it does and how to do it," *Proceedings of the IEEE*, vol. 78, no. 10, pp. 1550–1560, 1990, ISSN: 0018-9219. DOI: `10.1109/5.58337`.

[37] Y.-L. Boureau, J. Ponce, and Y. LeCun, "A theoretical analysis of feature pooling in visual recognition," in *Proceedings of the 27th international conference on machine learning (ICML-10)*, 2010, pp. 111–118.

[38] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *The Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[39] R. C. Gonzalez, R. E. Woods, *et al.*, *Digital image processing*, 2002.

[40] T. Kumar and K. Verma, "A Theory Based on Conversion of RGB Image to Gray Image," *International Journal of Computer Applications*, vol. 7, no. 2, pp. 7–10, 2010.

[41] T.-M. Rhyne, *Applying Color Theory to Digital Media and Visualization*. CRC Press, 2016.

[42] M. Grundland and N. A. Dodgson, "Decolorize: Fast, contrast enhancing, color to grayscale conversion," *Pattern Recognition*, vol. 40, no. 11, pp. 2891–2896, 2007.

[43] OpenCV, "Color conversions," 2018. [Online]. Available: `https://docs.opencv.org/3.4.2/de/d25/imgproc_color_conversions.html#color_convert_rgb_gray` (visited on 10/30/2018).

[44] L. Shapiro and G. C. Stockman, "Computer Vision. 2001," *ed: Prentice Hall*, 2001.

[45] Gimp, *Histogram dialog*, unknown. [Online]. Available: `https://docs.gimp.org/2.10/en/gimp-histogram-dialog.html` (visited on 10/30/2018).

[46] E. R. Dougherty, "An Introduction to Morphological Image Processing," *DC O'Shea, SPIE Optical Engineering Press, Bellingham, WA, USA*, 1992.

[47] L. Vincent, "Morphological grayscale reconstruction in image analysis: Applications and efficient algorithms," *IEEE transactions on image processing*, vol. 2, no. 2, pp. 176–201, 1993.

[48] R. Szeliski, *Computer vision: algorithms and applications*. Springer Science & Business Media, 2010.

[49] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2009.

[50] F. P. Preparata and S. J. Hong, "Convex hulls of finite sets of points in two and three dimensions," *Communications of the ACM*, vol. 20, no. 2, pp. 87–93, 1977.

[51] R. L. Graham, "An efficient algorithm for determining the convex hull of a finite planar set," *Info. Pro. Lett.*, vol. 1, pp. 132–133, 1972.

[52] R. A. Jarvis, "On the identification of the convex hull of a finite set of points in the plane," *Information processing letters*, vol. 2, pp. 18–21, 1973.

[53] J. Sklansky, "Finding the convex hull of a simple polygon," *Pattern Recognition Letters*, vol. 1, no. 2, pp. 79–83, 1982.

[54] ——, "Measuring concavity on a rectangular mosaic," *IEEE Transactions on Computers*, vol. 100, no. 12, pp. 1355–1364, 1972.

[55] G. T. Toussaint and H. El Gindy, "A counterexample to an algorithm for computing monotone hulls of simple polygons," *Pattern Recognition Letters*, vol. 1, no. 4, pp. 219–222, 1983.

[56] A. Bykat, "Convex hull of a finite set of points in two dimensions," *Information Processing Letters*, vol. 7, no. 6, pp. 296–298, 1978.

[57] OpenCv. (2018). Python convex hull, [Online]. Available: `https://docs.opencv.org/2.4/modules/imgproc/doc/structural_analysis_and_shape_descriptors.html?highlight=convex%20hull#cv2.convexHull` (visited on 10/30/2018).

[58] S. van der Walt, J. L. Schönberger, J. Nunez-Iglesias, F. Boulogne, J. D. Warner, N. Yager, E. Gouillart, T. Yu, and the scikit-image contributors, "Scikit-image: Image processing in Python," *PeerJ*, vol. 2, e453, Jun. 2014, ISSN: 2167-8359. DOI: `10.7717/peerj.453`. [Online]. Available: `http://dx.doi.org/10.7717/peerj.453`.

[59] N. Otsu, "A threshold selection method from gray-level histograms," *IEEE transactions on systems, man, and cybernetics*, vol. 9, no. 1, pp. 62–66, 1979.

[60] OpenCV, *Otsu's binarization*, 2018. [Online]. Available: `https://docs.opencv.org/3.4/d7/d4d/tutorial_py_thresholding.html` (visited on 10/30/2018).

[61] R. Jain, R. Kasturi, and B. G. Schunck, *Machine vision*. McGraw-Hill New York, 1995, vol. 5.

[62] M. K. Vairalkar and S. Nimbhorkar, "Edge detection of images using sobel operator," *International Journal of Emerging Technology and Advanced Engineering*, vol. 2, no. 1, pp. 291–293, 2012.

[63] J. Canny, "A computational approach to edge detection," *IEEE Transactions on pattern analysis and machine intelligence*, no. 6, pp. 679–698, 1986.

[64] B. Green, "Canny edge detection tutorial," *Retrieved: March*, vol. 6, p. 2005, 2002.

[65] P. V. Hough, *Method and means for recognizing complex patterns*, US Patent 3,069,654, 1962.

[66] R. C. Palmer, *The bar code book: A comprehensive guide to reading, printing, specifying, evaluating, and using bar code and other machine-readable symbols*. Trafford Publishing, 2007.

[67] P. Service, "POSTNET Barcode Discontinuation," 2012. [Online]. Available: `https://www.gpo.gov/fdsys/pkg/FR-2012-03-02/pdf/2012-5050.pdf` (visited on 10/30/2018).

[68] Y. P. Wang, *System for encoding and decoding data in machine readable graphic form*, US Patent 5,243,655, 1993.

[69] International Air Transport Association, "Bar Coded Boarding Pass (BCBP)," 2016.

[70] N. Degara-Quintela and F. Perez-Gonzalez, "Visible encryption: Using paper as a secure channel," in *Security and Watermarking of Multimedia Contents V*, International Society for Optics and Photonics, vol. 5020, 2003, pp. 413–423.

[71] Microscan Systems, "Fundamentals of PDF417 Symbology," 2015.

[72] Y. J. Kim and J. Y. Lee, "Algorithm of a perspective transform-based PDF417 barcode recognition," *Wireless Personal Communications*, vol. 89, no. 3, pp. 893–911, 2016.

[73] M. Trummer and J. Denzler, "Reading out 2D Barcode PDF417," *Progress in Pattern Recognition*, p. 179, 2007.

[74] R. L. Brass, J. Glaberson, R. W. Mason, A. J. L'Heureux III, S. Santulli, G. T. Roth, J. Frega, and H. S. Imiolek, "Optical reader for printed bit-encoded data and method of reading same," pat., US Patent 4,692,603, 1987.

[75] R. L. Brass, J. Glaberson, R. W. Mason, S. Santulli, G. T. Roth, W. M. Feero, and R. K. Balaska Jr, "Method and apparatus for transforming digitally encoded data into printed data strips," pat., US Patent 4,754,127, 1988.

[76] R. L. Brass, J. Glaberson, R. W. Mason, S. Santulli, and G. T. Roth, "Printed data strip including bit-encoded information and scanner control," pat., US Patent 4,782,221, 1988.

[77] Cauzin Systems Inc., *Reader Spec: Technical Specification for the Softstrip Reader Interface*, 1986. [Online]. Available: `https://archive.org/details/CauzinSoftstrip` (visited on 10/30/2018).

[78] Unknown, "Softstrip Magazine Strips Collection," [Online]. Available: `https://www.apple2scans.net/tag/cauzin-softstrip-reader/` (visited on 10/30/2018).

[79] Cauzin Systems Inc., "Softstrip System Application Notes," 1986. [Online]. Available: `https://archive.org/stream/CauzinSoftstrip/Softstrip%20System%20Application%20Notes#page/n0` (visited on 10/30/2018).

[80] C. Osborn, *Distripitor*, Github, 2016. [Online]. Available: `https://github.com/FozzTexx/Distripitor` (visited on 10/30/2018).

[81] D. Nichols, "OSCAR," *Enthusiast'99*, vol. 1, no. 4, p. 26, Nov. 1983. [Online]. Available: `https://archive.org/details/E99-1183` (visited on 10/30/2018).

[82] "Programming The Home Computer," *Databar Magazine Atari Edition*, p. 101, 1983. [Online]. Available: `https://archive.org/stream/DatabarMagazineAtariEdition/Databar_Magazine_Atari_Edition#page/n99` (visited on 10/30/2018).

[83] Databar Corporation, *Optical Scanning Reader Box*, circa 1983. [Online]. Available: `https://archive.org/details/DatabarOSCARbox` (visited on 10/30/2018).

[84] "Collecting bottles of broken things, pastor manul laphroaig with theory and praxis could be the man who sneaks a look behind the curtain!" *International Journal of Proof-of-Concept or Get The Fuck Out*, pp. 53,54, Jun. 2016.

[85] P. Teuwen, *OSCAR*, Github, 2016. [Online]. Available: `https://github.com/doegox/Oscar` (visited on 10/30/2018).

[86] Databar Corporation, *An Introduction to OSCAR and Bar Code Scanning*, 1983. [Online]. Available: `http://www.mainbyte.com/ti99/hardware/oscar/read_first_oscar.pdf` (visited on 10/30/2018).

[87] C. Osborn. (2016). Encoding Software in Barcodes, the Eight-Bit Magazine Way, [Online]. Available: `http://www.insentricity.com/a.cl/265/encoding-software-in-barcodes-the-eight-bit-magazine-way` (visited on 10/30/2018).

[88] Cauzin Systems Inc., *Cauzin StripWare Stripper Software (DOS 3.3)*. [Online]. Available: `http://www.apple2scans.net/2015/12/20/cauzin-softstrip-reader-manuals-software-etc/` (visited on 10/30/2018).

[89] C. Osborn. (2016). Another Retro Challenge Has Arrived: CauzCoin! [Online]. Available: `http://www.insentricity.com/a.cl/266/another-retro-challenge-has-arrived-cauzcoin` (visited on 10/30/2018).

[90] W. Sowerbutts. (2016). Solving the CauzCoin Retro BattleStations Challenge, [Online]. Available: `http://sowerbutts.com/cauzcoin/` (visited on 10/30/2018).

[91] E. M. D. R. N. Taylor N. Medvidovic, *Software Architecture: Foundations, Theory, and Practice*. Wiley, 2009.

[92] Tensorflow. (2018). About Tensorflow, [Online]. Available: `https://www.tensorflow.org/` (visited on 10/30/2018).

[93] Keras. (2018). Keras: The Python Deep Learning library, [Online]. Available: https://keras.io/ (visited on 10/30/2018).

[94] M. Buda, A. Maki, and M. A. Mazurowski, "A systematic study of the class imbalance problem in convolutional neural networks," *Neural Networks*, vol. 106, pp. 249–259, 2018.

[95] Datastrip, *About Us: History*, 2003. [Online]. Available: https://web.archive.org/web/20030624032546/http://www.datastrip.com:80/English/history.asp (visited on 10/30/2018).

[96] ——, *Readers*. [Online]. Available: https://web.archive.org/web/19970110194617/http://www.datastrip.com:80/readers.htm (visited on 10/30/2018).

[97] ——, *FAQ: What is the difference between 2DSuperscript and Datastrip2D bar codes?* 2003. [Online]. Available: https://web.archive.org/web/20030620205648/http://www.datastrip.com:80/English/faqs.asp (visited on 10/30/2018).

[98] ——, *Products*, 2002. [Online]. Available: https://web.archive.org/web/20030204114404/http://www.datastrip.com:80/products.htm (visited on 10/30/2018).

[99] ——, *Continental Trust Bank Implements Biometric Savings Card Application*, 2000. [Online]. Available: https://web.archive.org/web/20010419021332/http://www.datastrip.com:80/PR-ContTrustBank.htm (visited on 10/30/2018).

[100] ——, *Datastrip Provides ID Card System For Sri Lanka's International Airport*, 2001. [Online]. Available: https://web.archive.org/web/20021204152134/http://www.datastrip.com:80/pr-srilanka.htm (visited on 10/30/2018).

[101] ——, *Datastrip's 2D Superscript Bar Code System Integrated with Synercard's Asure Photo Card Software To Permit Addition of Biometrics*, 2002. [Online]. Available: https://web.archive.org/web/20021204150310/http://www.datastrip.com:80/pr-synercard.htm (visited on 10/30/2018).

[102] ——, *Cambodia National ID: High density printed code provides economic solution for photo-based application*, 1998. [Online]. Available: https://web.archive.org/web/20021219024913/http://www.datastrip.com:80/casestudies/pr-cambodia.htm (visited on 10/30/2018).

[103] ——, *Diamond Bank Implements Biometric ID Card Security System*, 2002. [Online]. Available: https://web.archive.org/web/20021219024817/http://www.datastrip.com:80/casestudies/pr-diamond_savings.htm (visited on 10/30/2018).

[104] ——, *APEC Business Travel Card Program For Hong Kong*, 2002. [Online]. Available: https://web.archive.org/web/20021219030012/http://www.datastrip.com:80/casestudies/pr-hongkong.htm (visited on 10/30/2018).

[105] ——, *Yemen Chooses Innovative Printed Code For National ID Cards*, 2002. [Online]. Available: https://web.archive.org/web/20021219025153/http://www.datastrip.com:80/casestudies/pr-yemen.htm (visited on 10/30/2018).

[106] ——, *Datastrip's Printed Code Chosen For Zimbabwe's Social Security ID Card*, 2002. [Online]. Available: https://web.archive.org/web/20021219031025/http://www.datastrip.com:80/casestudies/pr-zimbabwe.htm (visited on 10/30/2018).

[107] ——, *Safe Student ID Cards*, 2002. [Online]. Available: https://web.archive.org/web/20060314194302/http://www.datastrip.com/english/news_detail.asp?id=101 (visited on 10/30/2018).

[108] ——, *LISCR Unveils World's First Biometric Seafarer's Identity Card*, 2003. [Online]. Available: https://web.archive.org/web/20060314194552/http://www.datastrip.com/english/news_detail.asp?id=165 (visited on 10/30/2018).

[109] ——, *Datastrip's 2D Bar Code Selected for Colombia's San Andrés Island New Resident ID Cards*, 2003. [Online]. Available: https://web.archive.org/web/20060314194327/http://www.datastrip.com/english/news_detail.asp?id=256 (visited on 10/30/2018).

[110] E. P. Gerety, R. A. Strempski, and S. G. Sardi, *Two-dimensional printed code for storing biometric information and integrated off-line apparatus for reading same*, US Patent 6,560,741, 2003.

[111] P. E. Black, *Gray code*, 2005.

[112] A. B. Raj, *Bar codes: technology and implementation*. Tata McGraw-Hill Publishing Company, 2001.

[113] Datastrip, *Products: 2DSuperscript*, 2003. [Online]. Available: https://web.archive.org/web/20030410123442/http://www.datastrip.com:80/english/products_detail.asp?id=90 (visited on 10/30/2018).

[114] ——, *Smart Media Innovations (SMI) to aquire Datastrip assets*, 2011. [Online]. Available: https://web.archive.org/web/20130831222730/http://datastripsystems.com/press/BusinessTransfer.html (visited on 10/30/2018).

[115] Cauzin Systems, *The Art of Stripping*, 1986. [Online]. Available: https://archive.org/stream/CauzinSoftstrip/The%20Art%20Of%20Stripping#page/n5 (visited on 10/30/2018).

List of Figures

List of Tables

Listings

Abbreviations

LSB	Least Significant Byte
MSB	Most Significant Byte
CNN	Convolutional neural network
MLP	MultiLayer perceptron
ReLU	Rectified Linear Unit
CRC	Cyclic Redundancy Check
QR	Quick Response
UPC	Universal Product Code
LED	Light Emitting Diode
PC	Personal Computer
OSCAR	Optical Scanning Reader
ROI	Region of Interest
DPI	Dots per inch
MLP	Multilayer perceptron
RNN	Recurrent neural network
SLP	Single Layer Perceptron
EAN	European Article Number
GUI	Graphical User Interface

Appendix

A File Information

This appendix provides more details about the file information encoded in the Cauzin Softstrip which are described in table 3.1 and are defined in the Cauzin Reader Spec [77]. Table A.1 shows the different Softstrip types. However, only the standard Softstrip is defined. The possible operating system with their hexadecimal values are described in table A.2 and the corresponding file types are shown in table A.4. The general file types are listed in table A.3.

Hexadecimal Value	**Description**
0x00	Standard Softstrip data strip
0x01	Special key strip (Not implemented)
0x02 - 0xFF	Other formats (Not implemented)

Table A.1: Strip Types [77]

Hexadecimal Value	**Description**
0x00	Cauzin Generic Strip Format
0x01	COLOS (Cauzin's Own Little Operating System)
0x10	Apple DOS 3.3
0x11	Apple DOS ProDOS
0x12	Apple CPM 2.0
0x14	PC/MS-DOS (2.1)
0x15	Macintosh - MacBinary
0x20	Reserved (Interpreted as PC/MS-DOS)

Table A.2: Operating System [77]

Hexadecimal Value	**Description**
0x00	Unknown (text file if the operating system is Cauzin)
0x01	Text file
0x02	Binary, executable file
0x04	BASIC
0x10	Cauzin's compressing technique (not implemented)

Table A.3: Cauzin File Type [77]

Operating System	Hexadecimal Value	Description
Apple DOS 3.3	0x00	Text file
	0x01	Integer BASIC file
	0x02	Applesoft BASIC file
	0x04	Binary file
	0x10	Relocatable object module file
	0x20	A type file (not supported)
	0x40	B type file (not supported)
Apple ProDos	0x04	ASCII text file
	0x06	General binary file
	0xFA	Integer BASIC file
	0xFC	Applesoft BASIC file
	0xFE	Relocatable object module file
	0xFF	System file
IBM PC-DOS/MS-DOS	0x00	Executable DOS file
	0x01	Any other DOS file
Macintosh	0x00	MacBinary
	0x01	Data fork text file document, non MacBinary

Table A.4: Operating System File Type [77]

B Contents of the attached CD

The following content is included on the CD:

- Source code of the digital Softstrip reader
- Installation guide
- Example Softstrips
- CNN models
- CNN training scripts

Kolophon

Dieses Dokument wurde mit der LaTeX-Vorlage für Abschlussarbeiten an der htw saar im Bereich Informatik/Mechatronik-Sensortechnik erstellt (Version 2.1). Die Vorlage wurde von Yves Hary und André Miede entwickelt (mit freundlicher Unterstützung von Thomas Kretschmer, Helmut G. Folz und Martina Lehser). Daten: (F)10.95 – (B)426.79135pt – (H)688.5567pt