Technische Hochschule Deggendorf
Fakultät Angewandte Informatik

Studiengang Master Elektro- und Informationstechnik

Hochpassfilter-Vorverarbeitung bei der Bildverfolgung mit Faltungs-Autoencodern

High-pass Filters Preprocessing in Image Tracing with Convolutional Autoencoders

Masterarbeit zur Erlangung des akademischen Grades:

Master of Science (M.Sc.)

an der Technischen Hochschule Deggendorf

Vorgelegt von:
Zineddine Bettouche
Matrikelnummer: 00809173

Prüfer:
Prof. Dr. Andreas Fischer

Am: 25. November 2021

Abstract

Vector images have a very important advantage when compared to their raster versions: the recognition of shapes is more efficient due to the mathematical formulas they hold, instead of just pixel values on the other hand for raster images. This indicates the potential of getting better results when using vector images in pattern recognition, as for example: face recognition with artificial intelligence.

The abstract aim of this work is to determine the extent to which high-pass filters and artificial intelligence can reduce the complexity of an image, and the effect of this process on the tracing process of this image. Obtaining an abstract representation of an image would be the optimal result, as it can be a step forward to make the application of object recognition algorithms faster, robust, and experiencing less overfitting (lowering the validation losses).

First, the most commonly used filters are going to be investigated and tested on their fitness to be cascaded prior to the vectorization algorithm. Another path would be the integration of autoencoder into the pipeline. This can take two different ways: the autoencoder is fed the default images, or the filtered ones. Therefore, a Python project is going to be implemented to construct the necessary autoencoding models and train them. The structure of this neural network is discussed, and future attempts to improve it, can be done.

Having two distinct preprocessing stages (autoencoding and filtering) leads to different possible combinations. The question here is whether the high-pass filters are more suited to precede or to succeed the autoencoding stage. These two different combinations are going to be placed under experimentation, in order to arrive at the structuring decision that results in the best abstract representation of the images. Noise-reduction algorithms are going to be introduced as an attempt of enhancing the quality of filtering without blurring the images. In addition, experiments are going to be done in order to observe the effect on the different stages when having the two possible versions of a processed image: a white image with black features or a black image with white feature.

The different pipelines built are going to be evaluated in details, as the vectorized images are going to be further rasterized to perform a comparison with their original versions. This gives more options of evaluation algorithms that compare different types of data between the images. Finally, conclusions are going to be drawn based on the results obtained, and further outlooks are to be suggested.

Abbreviations

AI Artificial Intelligence

ANN Artificial Neural Network

CPU Central Processing Unit

GPU Graphics Processing Unit

JSON JavaScript Object Notation

MSE Mean Squared Error

PNG Portable Network Graphics

SSIM Structural Similarity Index

SVG Scalable Vector Graphic

XML Extensible Markup Language

Contents

List of Figures

1 Introduction

Object recognition is considered a complex task in the processing field. Its complexity far exceeds simple arithmetic operations. With the immense size of data generated every year, manual calculations done by hand are not taken into even the slightest consideration. Therefore, data processing and evaluation is automated for all operations.

1.1 Motivation

In the recent years, many studies have emerged to contribute to the already advanced knowledge in the field of object recognition. Two of the most important pillars of this field are image processing and artificial intelligence (AI). AI is a fascinating subject that attracted a lot of attention in the last decade, especially with its use in computer vision. Now not only filter-based models, e.g. Haar Cascade, can be trained to classify images, but also neural network can be wired to learn how to detect various shapes and objects. The models generally learn from the pixel values, and model their structures in mathematical equations, which begs the question on whether would it be more efficient for the models to learn from vector images as they are closer to the nature of the trained models than spatial data in the form of pixel arrays. Thus comes this thesis in place as an attempt to improve the tracing of images by the use of autoencoders and high-pass filters to obtain an abstract representation of images in a vector form. The high-pass filters are chosen due to the fact that they emphasize on the important features of an image. This work is considered as a step forward to achieve a better training rate in object recognition with ANN, as the improvement would not be limited to Boolean decision ratio, however it would extend to the speeds of training and the real-time detection, although the latter would be considered initially not as hypothetically systematic as the learning the speed of networks from XML data rather than pixel values.

1.2 Aim of This Work

At first, the thought process lead to the questioning: if the autoencoding of an image can improve its vectorized format by reconstructing its important features, how can high-pass filters come to play in the process. In other words, the question that opened the door for this thesis is ”Can a high-pass filter be used in combination with an autoencoding model to achieve an abstract representation of the image through the process of vectorization?”. Thus, various ideas branched from this node, leading to the different pipelines that can be built, in order to experiment with high-pass filters integration. For instance, the filters can be put before the autoencoding stage of a model that is already trained with filtered images, to better reconstruct the significant data, leading to a better vectorization. More systematically, the autoencoding stage can act as a smoothing process, removing the noise from the images while reducing their complexity; while the filters come afterwards to further enhance the quality of the important features, leading to a more abstract representation.

1.3 Thesis Structure

Firstly, in Chapter 2 the related work is going to be stated, in order to paint a clear picture on where exactly this thesis takes role. Chapter 3 presents the background of the relevant implementations, such that the technologies used are explained briefly before diving into their applications in the project. Chapter 4 provides the methodology of the work. It includes: the dataset used, the project structure, and a documentation of the various programs built, and closes with the evaluation methods in this thesis. Chapter 5 presents the experiments undergone in this project and their rationale. Chapter 6 is the evaluation part, in which the different pipelines are evaluated in details. Finally the thesis concludes and suggestively directs into further potential development in chapter 7.

2 Related Work

Image vectorization and autoencoding are not new topics. Several studies have been done in the field of dimensionality reduction as well as object recognition with neural networks.

In a paper done in MIT, Solomon and Bessmeltsev [1] explored the use of frame fields. The general idea of their method is to find a smooth frame field on the image plane, where at least one direction is aligned with nearby contours of the drawing. Around X- or T-shaped junctions, the two directions of the field will be aligned with the two intersecting contours. Then, the topology of the drawing is extracted by tracing the frame field and grouping traced curves into strokes. Finally, they created with the extracted topology a vectorization aligned with the frame field.

In another paper titled "Raster-To-Vector Conversion: Problems and Tools" [2] Vinciane Lacroix analyzed some problems of R2V conversion, and a strategy has been proposed involving a preprocessing stage generating a mask, from which edges are removed and lines are kept. A clustering is then performed while considering only the pixels of the mask. A new algorithm, the medianshift, has been proposed in this context. Then comes the labeling process which should also take the pixel type into account. The last step involves a regularization procedure. The importance of the pre-processing ignoring edge pixels while keeping lines has been shown on some examples. Tests also showed the superiority of the median-shift over the mean-shift, and over the clustering method used by Vector-Magic. This paper also showed that a better line vectorization can be obtained from enabling the extraction of dark lines, which can support the use of high-pass filters as a preprocessing stage to put further emphasis on those dark lines.

When it comes to using neural networks in vectorization, a paper titled "A Neural Network Algorithm for Vectorization of 2D Maps" done by Karabork et al. [3] a vectorization algorithm based on Artificial Neural Network method was proposed. The evaluation was a comparison to Sparse Pixel Vectorization (SPV) algorithm. Although SPV algorithm delivers better results, the proposed algorithm also gives acceptable results, which are suitable for mapping purposes.

Another paper in the topic of maps and road networks extraction; titled "Road network extraction and vectorization of remote sensing images based on deep learning" [4] Gong et al. presented an algorithm that successfully completes the automatic extraction and vectorization of the road network. The main obstacles in road extraction in remote sensing images are: first, different scales and strong connectivity; second, complex backgrounds and occlusions; and third, high resolution and a small proportion of roads in the image. The process of road vectorization in this paper is mainly divided into road network extraction and vectorization preservation. This work also shows the advantages of using dense dilation convolution, which points to the possibility of using autoencoding models for vectorization preservation.

Lastly, and most importantly, in the paper "Improving image tracing with artificial intelligence" [5] Fischer and Amesberger showed that preprocessing the raster image with an autoencoder neural network, can reduce complexity by over 70% while keeping reasonable image quality. They proved that autoencoders perform significantly better compared to PCA in this task. Opening the door for future possible development, such as: investigating the refinement of the approach with carefully fine-tuned autoencoder and PCA architectures, and the product of its combination with other preprocessing filters. Finally, they stated that the goal of producing simplified vector descriptions of raster images could help to advance image recognition tasks, thereby pushing artificial intelligence approaches even further in that field.

So it becomes clear that similar topics have already been addressed and researched. However, other than the last paper of Fischer et al. [5], they do not combine the approaches of autoencoders and high-pass filters as an attempt of obtaining an abstract representation of images in a way that improves image recognition through vectorization.

3 Background

In this chapter of the thesis, some processes, structures and programs, that are relevant for the work, are dealt with. This important information is presented to facilitate the understanding of these topics.

3.1 High-Pass Filters

3.1.1 Derivative Masks

Image derivatives can be computed by using small convolution filters of size 2x2 or 3x3, such as the Laplacian, Sobel, Roberts and Prewitt operators [7]. However, a larger mask will generally give a better approximation of the derivative (Gaussian derivatives and Gabor filters). Sometimes high frequency noise needs to be removed and this can be incorporated in the filter so that the Gaussian kernel will act as a band pass filter. The use of Gabor filters in image processing has been motivated by some of its similarities to the perception in the human visual system. A derivative mask has the following properties:

- Positive as well as negative values
- The sum of all the values in a derivative mask is equal to zero
- The edge content is increased by a derivative mask
- As the size of the mask grows, more edge content is increased

Sobel Derivatives:

The derivative kernels, known as the Sobel operator are defined as follows [8]:

$$p_h = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \quad p_v = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix}$$

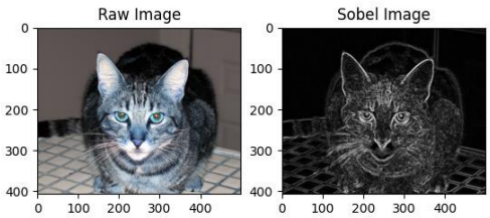

Figure 3.1: sobel derivatives

Gaussian HP Filter:

In this method, instead of a box filter, a Gaussian kernel is used. It is done with the function, cv.GaussianBlur(). We should specify the width and height of the kernel, which should be positive and odd. We also should specify the standard deviation in the X and Y directions, sigmaX and sigmaY respectively. If only sigmaX is specified, sigmaY is taken as the same as sigmaX. If both are given as zeros, they are calculated from the kernel size [9]. Gaussian blurring is highly effective in removing Gaussian noise from an image. To transform this method from a low-pass to a high-pass filter, the blurred image is subtracted from the original image.

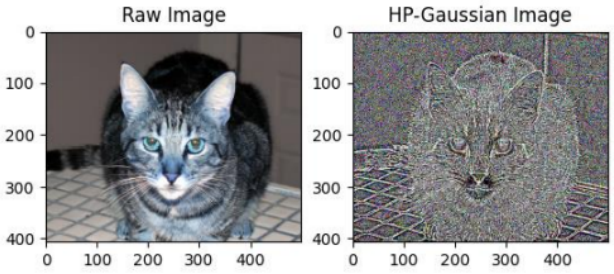

Figure 3.2: gaussian derivatives

3.1.2 Canny Edge Detection

Canny Edge Detection is a popular edge detection algorithm, developed by John F. Canny [10]. It is a multi-stage algorithm:

- **Noise Reduction:** Since edge detection is susceptible to noise in the image, first step is to remove the noise in the image with a 5x5 Gaussian filter.
- **Finding Intensity Gradient of the Image:** Smoothened image is then filtered with a Sobel kernel in both horizontal and vertical direction to get first derivative in horizontal direction Gx and vertical direction Gy Gradient direction is always perpendicular to edges. It is rounded to one of four angles representing vertical, horizontal and two diagonal directions.
- **Non-maximum Suppression:** After getting gradient magnitude and direction, a full scan of the image is done to remove any unwanted pixels, which may not constitute the edge. Every pixel is checked if it is a local maximum in its neighborhood in the direction of gradient Point A is on the edge (in vertical direction). Gradient direction is normal to the edge. Point B and C are in gradient directions. So point A is checked with point B and C to see if it forms a local maximum. If so, it is considered for the next stage, otherwise, it is suppressed (put to zero). In short, the result you get is a binary image with thin edges.

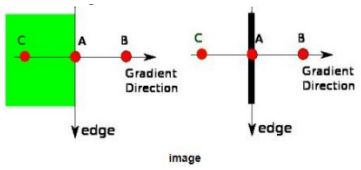

Figure 3.3: non-maximum suppression

- **Hysteresis Thresholding:** This stage omits false positives (detected edges that are not truly edges). For this, we need two threshold values, minVal and maxVal. Any edges with intensity gradient more than maxVal are sure to be edges and those below minVal are sure to be non-edges, so discarded. Those who lie between these two thresholds are classified edges or non-edges based on their connectivity. If they are connected to sure-edge pixels, they are considered to be part of edges. Otherwise, they are also discarded. The edge A is above the maxVal, therefore considered as sure-edge. Although edge C is below maxVal, it is connected to edge A, so that also considered as valid edge and we get that full curve. But edge B, although it is above minVal and is in same region as that of edge C, it is not connected to any sure-edge, therefore discarded. It is very important that we select minVal and maxVal accordingly to get the correct result. This stage also removes small noises on the assumption that edges are long lines. Applying a Canny edge detection on an image leads to the result in figure 3.5:

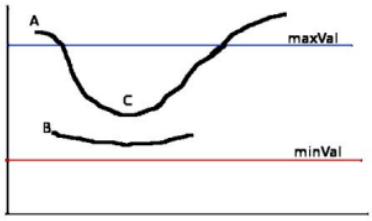

Figure 3.4: hysteresis thresholding

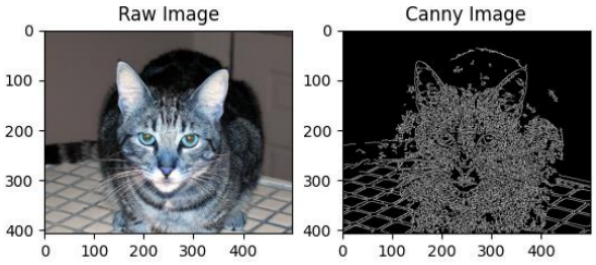

Figure 3.5: canny edge detection

3.2 Image Tracing

Image tracing is the process of converting a bitmap into a vector graphic. As Selinger writes on his tracing algorithm [11], vector graphics are described as algebraic formulas of the contours, typically in the form of Bezier curves. The advantage of displaying an image as a vector outline is that it can be scaled to any size without loss of quality. They are independent of the resolution and are used, for example, for fonts, since these must be available in many different sizes. However, most input and output devices such as scanners, displays and printers generate bitmaps or raster data. For this reason, a conversion between the two formats is necessary. Converting a vector graphic into a bitmap is called rendering or rasterizing. Tracing algorithms are inherently imperfect because there are many possible vector outlines that can represent the same bitmap. Of the many possible vector representations that can result in a particular bitmap, some are clearly more plausible or aesthetically pleasing than others. For example, to render bitmaps with a high resolution, each black pixel is represented as a precise square that creates staircase patterns. However, spikes are neither pleasant to look at, nor are they particularly plausible interpretations of the original image. Bezier curves are used to represent the outlines. As seen in figure 3.6, a cubic Bezier curve consists of four control points, which determine the curvature of the curve [11]. As a rule, the vector graphics are

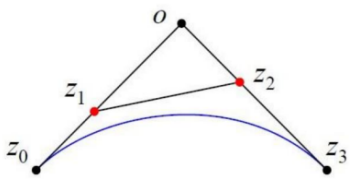

Figure 3.6: bezier curve [11]

saved as SVG files (Scalable Vector Graphic). This file format is a special form of an XML file. XML stands for Extensible Markup Language. It is used to present hierarchically structured data in a human readable format. As can be seen, the structure of this file is based on the Extensible Markup Language scheme. The file header defines which versions of XML and SVG are used. The height and width of the graphic in points are also specified. In this case, the *g* element represents the drawing area on which to draw. The elements

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 20010904//EN"
 "http://www.w3.org/TR/2001/REC-SVG-20010904/DTD/svg10.dtd">
<svg version="1.0" xmlns="http://www.w3.org/2000/svg"
 width="300.000000pt" height="300.000000pt" viewBox="0 0 300.000000 300.000000"
 preserveAspectRatio="xMidYMid meet">
<metadata>
Created by potrace 1.16, written by Peter Selinger 2001-2019
</metadata>
<g transform="translate(0.000000,300.000000) scale(0.100000,-0.100000)"
fill="#000000" stroke="none">
<path d="M0 2795 c0 -113 2 -205 5 -205 13 0 55 25 55 32 0 5 7 2 16 -7 13
```

Figure 3.7: header of an svg file by potrace

to be drawn consist of tags stored as XML elements. They are particularly important in connection with the path elements. Quadratic and cubic Bezier curves, as well as elliptical arcs and lines can be put together as fits best. The entries here determine which forms the path takes.

3.3 Potrace as a Vectorization Tool

Potrace is a tracing algorithm that was developed by Peter Selinger [11]. It is considered to be simple and efficient as it produces excellent results. Potrace stands for polygon tracer, where the output of the algorithm is not a polygon, but a contour made of Bezier curves. This algorithm works particularly well for high-resolution images. Potrace does not generate grayscale images as output on its own, but a threshold vector. The conversion from a bitmap to a vector graphic is done in several steps. First, the bitmap is broken

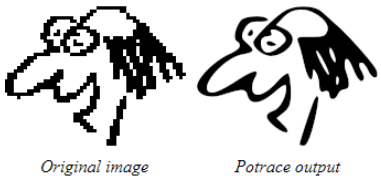

Figure 3.8: potrace vectorization [11]

down into several paths which form the boundaries between black and white areas. The points adjoining four pixels are given integer coordinates. These points are saved as vertices when the four adjacent pixels are not the same color. The connection between two vertices is called the edge. A path is thus a sequence of vertices, whereby the edges must all be different. The path composition in Potrace works by moving along the edges between the pixels. Everytime a corner is found, a decision is made as to which direction the path will continue based on the colors of the surrounding pixels. If a closed path is defined, it is removed from the image by inverting all pixel colors inside the path. This will define a new bitmap on which the algorithm will be applied recursively until there are no more black pixels. Then its optimal polygon is approximately determined for each path. The criterion for optimality with Potrace is the number of segments. A polygon with a few segments is therefore more optimal than one with several segments. In the last phase, the polygons obtained are converted into a smooth vector outline. Here, the vertices are first corrected so that they correspond as closely as possible to the original bitmap. Furthermore, in the main step, the corners and curves are calculated based on the length of the adjacent segments and the angles between them. Optionally, the curves can be optimized after this process, so that they match the original bitmap as closely as possible. Then, in the main step, the corners and curves are calculated based on the length of the adjacent segments and the angles between them. Also optionally, the curves can be optimized after this process, so that they match the original bitmap even more closely. Then, in the main

step, the corners and curves are calculated based on the length of the adjacent segments and the angles between them. Finally, the curves can be optimized after this process [11].

3.4 Auto-encoders

A typical use of a Neural Network is a case of supervised learning. It involves training data, which contains an output label. The neural network tries to learn the mapping from the given input to the given output label. Nevertheless, if the input vector itself replaces the output label, then the network will try to find the mapping from the input to itself. This would be the identity function, which is a trivial mapping. However, if the network is not allowed to simply copy the input, then the network will be forced to capture only the salient features. This constraint opens up a different field of applications for Neural Networks, which was unknown. The primary applications are dimensionality reduction and specific data compression. The network is first trained on the given input. The network tries to reconstruct the given input from the features it picked up and gives an approximation to the input as the output. The training step involves the computation of the error and backpropagating the error. The typical architecture of an Autoencoder resembles a bottleneck [12]. The schematic structure of an autoencoder is displayed in figure 3.9.

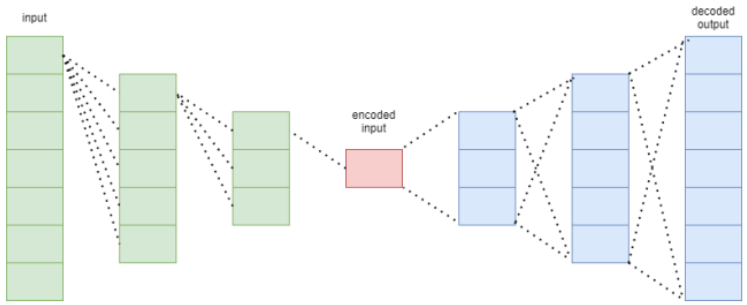

Figure 3.9: general structure of an autoencoder network

The encoder part of the network is used for encoding and sometimes even for data compression purposes although it is not very effective as compared to other general compression techniques like JPEG. Encoding is achieved by the encoder part of the network, which has decreasing number of hidden units in each layer. Thus, this part is forced to pick up only the most significant and representative features of the data. The second half of the network performs the decoding function. This part has an increasing number of hidden units in each layer and thus tries to reconstruct the original input from the encoded data. Therefore Auto-encoders are an unsupervised learning technique. Training of an Auto-encoder for data compression: For a data compression procedure, the most important aspect of the compression is the reliability of the reconstruction of the compressed data. This requirement dictates the structure of the Autoencoder as a bottleneck [12].

1. **Encoding the input data:** The Auto-encoder first tries to encode the data using the initialized weights and biases.
2. **Decoding the input data:** The Auto-encoder tries to reconstruct the original input from the encoded data to test the reliability of the encoding.
3. **Backpropagating the error:** After the reconstruction, the loss function is computed to determine the reliability of the encoding. The error generated is backpropagated. The above-described training process is reiterated several times until an acceptable level of reconstruction is reached.

After the training process, only the encoder part of the Auto-encoder is retained to encode a similar type of data used in the training process. The different ways to constrain the network are:

- **Keep small Hidden Layers:** If the size of each hidden layer is kept as small as possible, then the network will be forced to pick up only the representative features of the data thus encoding the data.
- **Regularization:** In this method, a loss term is added to the cost function which encourages the network to train in ways other than copying the input.
- **Denoising:** Another way of constraining the network is to add noise to the input and teaching the network how to remove the noise from the data.
- **Tuning the Activation Functions:** This method involves changing the activation functions of various nodes so that a majority of the nodes is dormant thus effectively reducing the size of the hidden layers.

3.5 TensorFlow and Keras Framework

Keras is a widely used and very popular deep learning framework [13]. It represents an interface to the TensorFlow back-end. The Keras API was developed with the focus on enabling rapid experimentation. "Getting from the idea to the result as quickly as possible is the key to good research" is a principle of the Keras community. TensorFlow is an end-to-end, open-source machine learning platform [14]. You can think of it as an infrastructure layer for differentiable programming. It has now developed into a quasi-standard in the area of programming neural networks and deep learning. TensorFlow is characterized by its high performance and good scalability. Keras is TensorFlow's high-level API: an understandable, highly productive interface for solving machine learning problems with a focus on modern deep learning. It offers essential abstractions and building blocks for the development and integration of machine learning solutions. Keras enables the cross-platform capabilities of TensorFlow to be exploited: Keras can run on Cloud Tensor Processing Units (TPUs) or on large clusters of GPUs. Keras models can be exported in order to run them in the browser or on a mobile device. Keras thus forms the interface between the

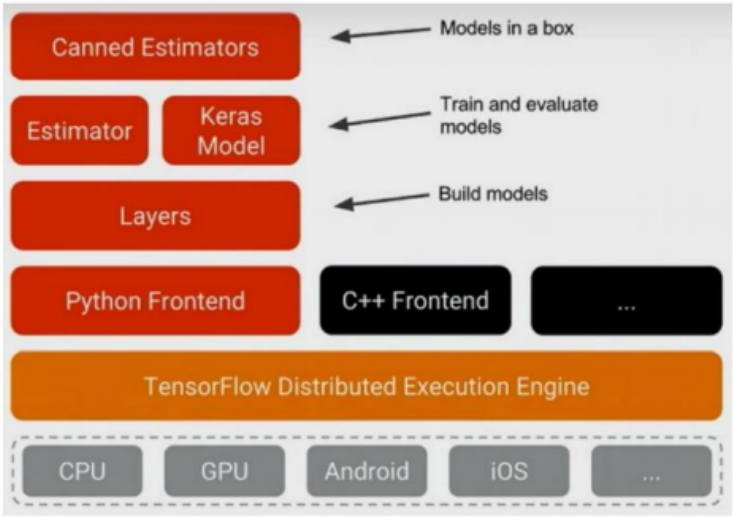

Figure 3.10: structure of layers between Keras and Tensorflow [15]

Python front-end of TensorFlow, the actual implementation of the TensorFlow Execution Engine and the hardware, which is relevant for most users. The great advantage of using TensorFlow as an intermediate layer is the extreme flexibility of this framework. Models can be trained on both the CPU and the GPU. The ability to use NVidia's CUDA library to handle the entire training of the models via the GPU means that the training time can be drastically accelerated compared to processing on CPUs. In addition, many

platforms are supported, such as Windows and Linux. For these reasons, TensorFlow with Keras offers itself as a high level API for creating and training the neural network.

4 Methodology

The following section describes how the autoencoder and the processing pipelines are structured, what functions they fulfill and how they were implemented. A repository was created to contain the software-development side of this thesis [16].

4.1 CAT Dataset - as Data

A dataset with over 10,000 cat images is used as the basis for training the autoencoder for evaluating the results. The CAT dataset [17] was published in 2008 by Zhang, Sun, and Tang. The cat's head was annotated with 9 points for each picture. However, the images are not relevant for this work. The main

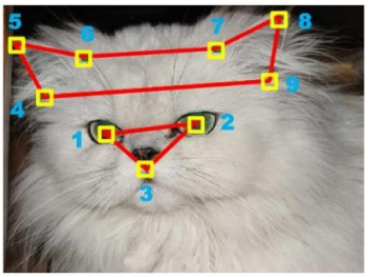

Figure 4.1: display of annotation points in an example image from dataset [17]

reason this dataset is used is the fact that features such as ears, eyes, and noses are relatively easy to see in these images. The autoencoding model can thus be trained on these features and reliably reproduce them.

4.2 Project Folders Structure

In this part, the structure of project folders is shown in figure 4.2. The images are contained initially in dataset folder, in which the images folder should contain the images that are going to be used for evaluation. It is also worth noting, the evaluation images are picked from the eval_set folder.

When running the script *train.py*, the *dataset_filtered* folder is generated automatically. This folder is a replica of the dataset folder, except that the images are all filtered with an appropriate filter, which is set as a preprocessing step prior to training the autoencoder to reconstruct filtered images, as one of the processing pipelines.

The experiments directory holds two main folders. This directory is mainly used in the experiments part, in which the *experiments.py* script is going to be executed and its results are going to be stored in this folder. The evaluation folder is only used in the evaluation part to store the result of the *evaluation.py* script. This script is only run to display the steps of the main script of this project step by step in a clearer way.

The models folder simply contains two directories: one for autoencoding models trained with default images, and the other for the models trained with filtered images. Each folder may contain more than one

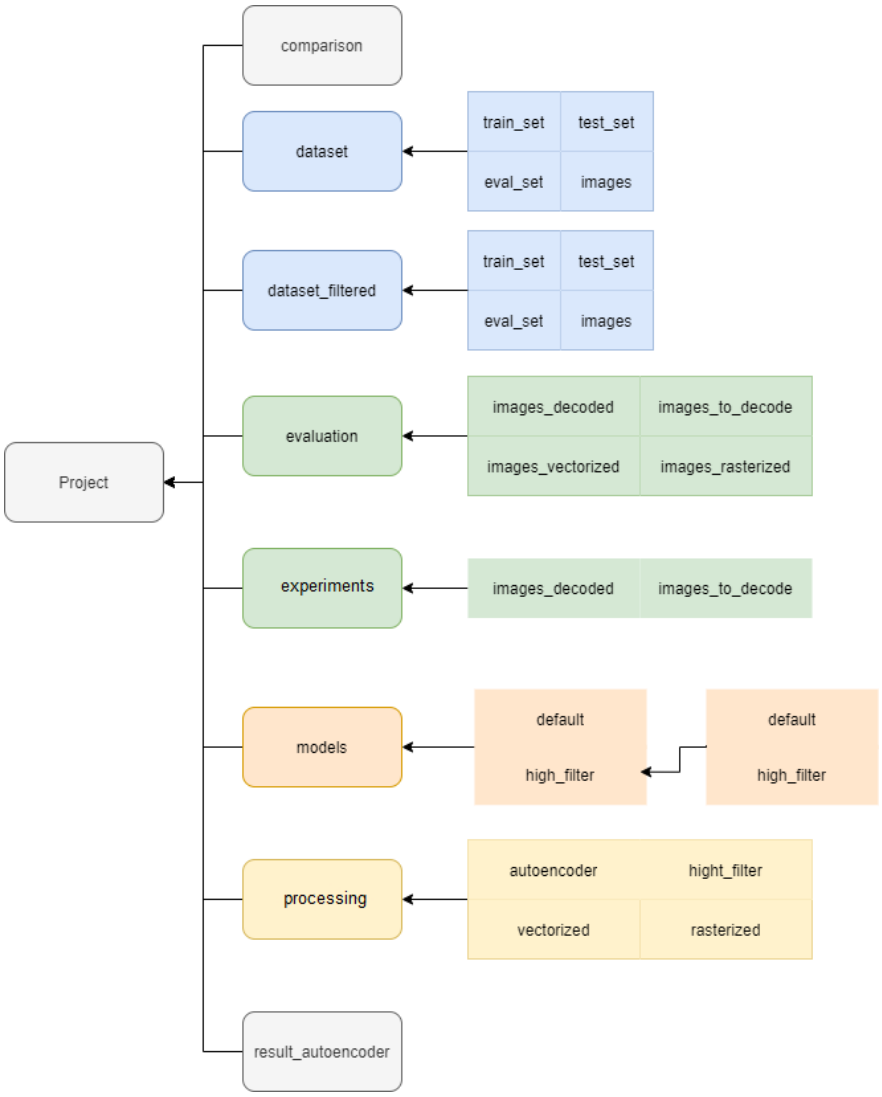

Figure 4.2: project folders structure

model if the training script is run multiple times with different assigned epochs.

The comparison folder is the folder that holds the resulting comparisons at the end of executions. The comparisons are essentially plots to visualize the results of various techniques implemented in this work next to each other.

The processing directory is the largest folder in the project. It contains all the different routes to better divide the approaches in an efficient way, for a better project organization. The lowest subfolders of this directory contain the resulting images.

Almost all of these directories and their subfolders are generated at the beginning of the main pipeline execution.

4.3 Autoencoder Structure

With the help of Keras framework, a Python script was defined to model, train, and use the autoencoder. The functional structure of the neural network will be first discussed, then the developed Python module

within the scope of this work will be examined in more details.

4.3.1 Functional Structure

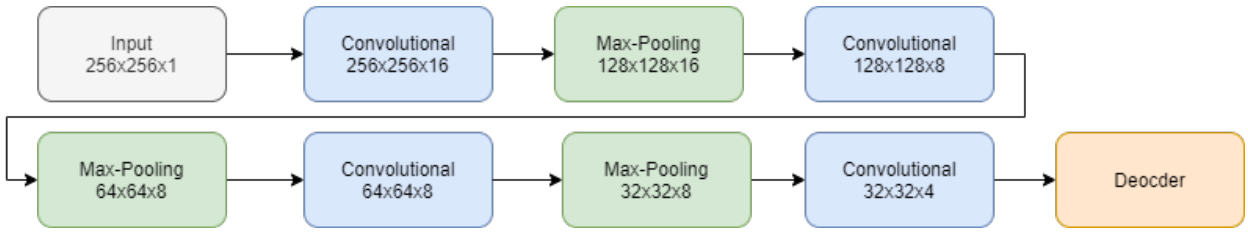

Figure 4.3: encoding part of autoencoder

The structure of the convolutional autoencoder used in this thesis is similar to that used in Dr. Fischer's work [5]. The starting point is an input with the size 256 x 256 x 1 (256 x 256 grayscale image). The first layer of the autoencoder is a convolution layer that contains 16 different trainable filter kernels. Each kernel can result in a different representation of the input image. A Max-Pooling layer is connected to the convolutional layer in order to increase the density of the data and reduce the necessary computing power by reducing the number of trainable neurons. This 2x2 layer halves the size of the original image. This cascade of the layers convolutional-maxPooling is repeated twice for the next two layers with the convolutional layer having 8 different filters and the same 2x2 Max-Pooling layer resulting in the sizes 64x64 and 32x32. In the last convolution layer of the encoder, which receives a 32x32 matrix as input, only 4 convolution kernels are used. The point of highest data density is here reached; therefore, the Max-Pooling layer is omitted. This layer of the autoencoder contains the most compact coding or representation of the data set.

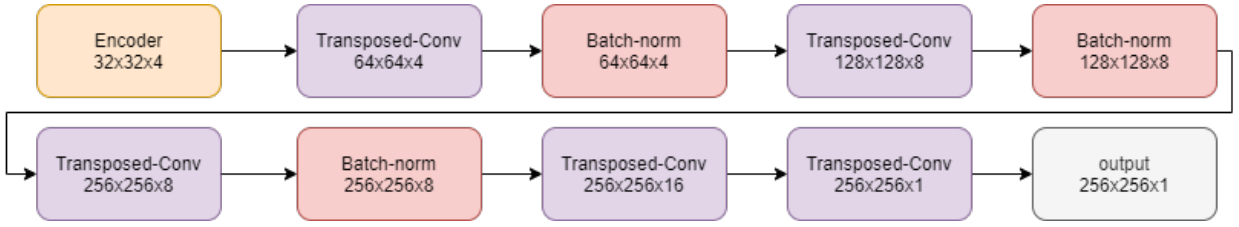

Figure 4.4: decoding part of autoencoder

The decoder follows the layer with the highest data density. This part of the autoencoder is responsible for reconstructing the learned encoding. It uses transposed convolution layers and batch normalization layers. The transposed convolution layer works in a similar way to a convolution layer. The difference between the two is that by transposing the input of the layer is no longer compressed, but decompressed. Here the principles of the convolution layer are reversed. The filter kernel is used to determine how the input value is broken down into the larger grid. By using this layer, the image matrix is enlarged again. The transposed convolution layer is followed by a batch normalization layer. These layers, also known as batch norms, serve to accelerate and stabilize the learning process of neural networks. They reduce the amount by which the values of the neurons can shift. On the one hand, the network can train faster because the batch norm ensures that the activation value is neither too high nor too low. On the other hand, using this layer also reduces overfitting, since less information is lost through dropouts.

The decoder connects directly to the encoder to take over the most compact representation of the data set passed by the encoding layers. First, the decoder receives a tensor with a size of 32x32x4 as input. The first

function that is applied to this tensor is a transposed convolution layer. This results in an enlargement of the image matrix to 64x64. Four 3x3 filter kernels are used here. This is followed by a batch-norm layer to normalize the results and accelerate the learning process. The same process is repeated with a different number of filter kernels to maintain the symmetrical structure of the autoencoder after reaching the original matrix size of 256x256; another transposed convolution layer is added. This ensures that the output of the first layer and the input of the last layer have the same size. The last layer reduces the dimension of the tensor to one in order to get a grayscale image as output.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 256, 256, 1)]     0
_________________________________________________________________
conv2d (Conv2D)              (None, 256, 256, 16)      416
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 128, 128, 16)      0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 128, 128, 8)       1160
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 64, 64, 8)         0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 64, 64, 8)         584
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 32, 32, 8)         0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 32, 32, 4)         292
_________________________________________________________________
conv2d_transpose (Conv2DTran (None, 64, 64, 4)         148
_________________________________________________________________
batch_normalization (BatchNo (None, 64, 64, 4)         16
_________________________________________________________________
conv2d_transpose_1 (Conv2DTr (None, 128, 128, 8)       296
_________________________________________________________________
batch_normalization_1 (Batch (None, 128, 128, 8)       32
_________________________________________________________________
conv2d_transpose_2 (Conv2DTr (None, 256, 256, 8)       584
_________________________________________________________________
batch_normalization_2 (Batch (None, 256, 256, 8)       32
_________________________________________________________________
conv2d_transpose_3 (Conv2DTr (None, 256, 256, 16)      1168
_________________________________________________________________
conv2d_transpose_4 (Conv2DTr (None, 256, 256, 1)       17
=================================================================
Total params: 4,745
Trainable params: 4,705
Non-trainable params: 40
```

Figure 4.5: automatic python display of model structure

4.3.2 Autoencoder – Python Class

This Python script, which was created as part of this work, is responsible for training the autoencoder. It was developed into a Python class to be used more efficiently in an object oriented programming manner. The module was tested with the Python version 3.8.5. A module essentially refers to a Python script that provides functionality, which in return can be imported into another script in order to be used in a more robust way that can be beneficial to the speed of development and credited to increase the reusability of the code. Furthermore, to ensure that the package conflicts do not occur when a user tries to run the code, a requirements file was built through pip-tools to contain all of dependencies, so that the user's Python environment operates on the same versions of libraries that the original code was developed on.

It is far more efficient to use a graphics processing unit (GPUs) for training, rather than using a common central processing unit (CPUs). This is a product of the nature of neural networks, as they rely on passing tensors or more simply matrices, when GPUs are designed to operate on similar tasks. Therefore, it makes more sense to use a GPU as a hardware component, dedicated to training the ANN. In order to accomplish such a task, certain libraries are required depending on the hardware manufacturer. One of the most widely used and supported libraries is NVidia's CUDA, which ensures the optimization of the interaction between Keras, TensorFlow, and the graphics card, resulting in a faster training process.

- *__init__(self, is_trained, models_path):* This is one of the reserved methods in Python. In object-oriented programming, it is known as a constructor. This method can be called when an object

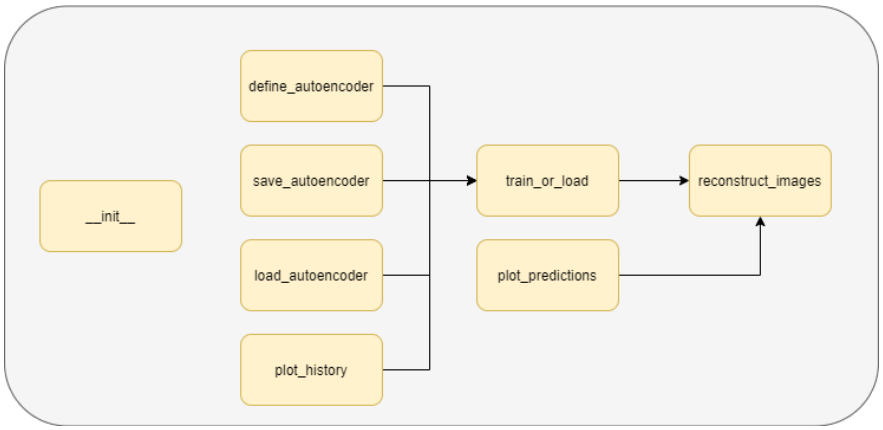

Figure 4.6: autoencoder class structure

is created from the class, and access is required to initialize the attributes of the class. The attributes that the autoencoder object is concerned with, are: a Boolean variable that loads the autoencoder if it is already trained, or trains it if it is set to false. The second attribute is the path the script set the model to load from or save to it.

- *define_autoencoder(self):* This function defines the model of the autoencoder. Here the number and type of the individual model layers are determined. First, the input size is determined. Then the definition of the encoder and the decoder. These are made from built up several Keras layers. The model is then used as an object, saved, and compiled. The function returns a Keras model object.
- *save_autoencoder(self, autoencoder, epoch):* This function saves a Keras model in the specified object attribute *models_path*. The basic structure is in a JSON file saved and the individual weightings of the nodes are separately in an h5 file in the same directory written. The epoch parameter is used for naming the saved files.
- *load_autoencoder(self, epoch):* With this function a Keras model can be loaded from *models_path*. The function loads the JSON file as a structure of the model with a function integrated in Keras or load a default model. The same thing happens afterwards with the weightings of the Nodes. These are initialized from the associated h5 file.
- *plot_history(self, history):* This function can be called after the autoencoder is finished training to plot the history of the losses. It stores the graph in a predefined path.
- *plot_predictions(self, predictions):* This is a function to plot 8 exemplary reconstructed images and saves the plot as well.
- *train_or_load(self, x_train, x_test, epoch):* This function acts as a switch between loading or training the autoencoder, to always return a trained model.
- *reconstruct_images(self, x_train, x_test, epoch, images):* This is the function that should be called by an object to reconstruct images. It uses all of the other functions to define and train a model if necessary, or to just load it, to make the predictions on images, and to plot the results. Finally, it also returns the reconstructed images.

4.4 Python Scripts

4.4.1 The Training Script – *train.py*

This script is responsible for training the autoencoder to reconstruct filtered images. Therefore, it should be run before running the main pipeline script. The script executes as shown in figure 4.7. Each of the

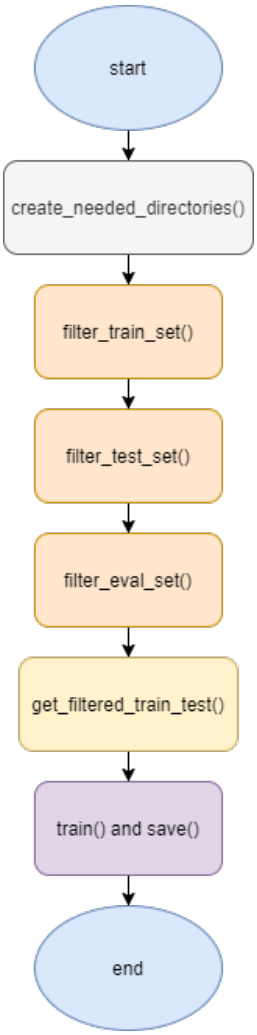

Figure 4.7: flowchart of training script

filtering functions takes the images provided in the dataset directory, applies the appropriate filter, and saves them in the newly created directories of dataset_filtered. Then, the program creates an object of the autoencoder class, trains it with these filtered images and saves it under models/high_filter.

4.4.2 The High-Pass Filters Script – *high_filters.py*

This script was written so that it facilitates filtering images as it is considered a module that provides high-pass filters functions, which forms the structure displayed in figure 4.8.

- *sobel(image, **kwargs):* This is a function that takes an image, applies the Gaussian blur on it, converts it to grayscale, applies horizontal and vertical gradients, and sums up the absolute value of each which implements the sobel derivatives.

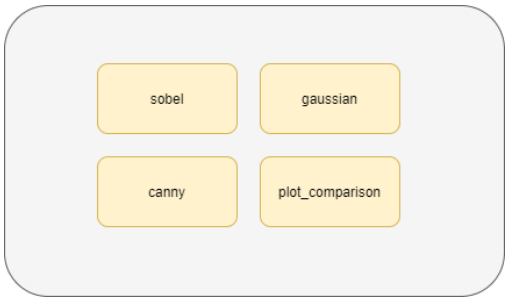

Figure 4.8: high-pass filters script structure

- *gaussian(image, kernel_size):* This functions calculates the Gaussian blur of an image, and subtracts it from the original image so that it acts as a high-pass filter.
- *canny(image, min_value=100, max_value=100, kernel_size=3, L2gradient=False):* This is just a simple function that encapsulates the Canny function from opencv library.
- *plot_comparison(image_0, image_1, title_0, title_1):* This is a plotting function that plots two images next to each other for comparison.

4.4.3 The Potrace Script – *potrace.py*

This is a relatively small script that contains one function: *portrace_png_to_svg(images_path, vectorized_images_path, image_count, temp_name)* This function uses the os library to execute two command lines that use the Potrace tool. It takes two directory paths: one for the input, the other for the desired output directory, with the count of images to be vectorized.

4.4.4 The Utilities Script – *main_utils.py*

This is the largest script in the project, containing all of the functions that will be used in the main script, which is technically the execution of the processing pipelines that produce the results for evaluation and comparison. The script structure is as shown in figure 4.9. The functions bellow the dotted line are the

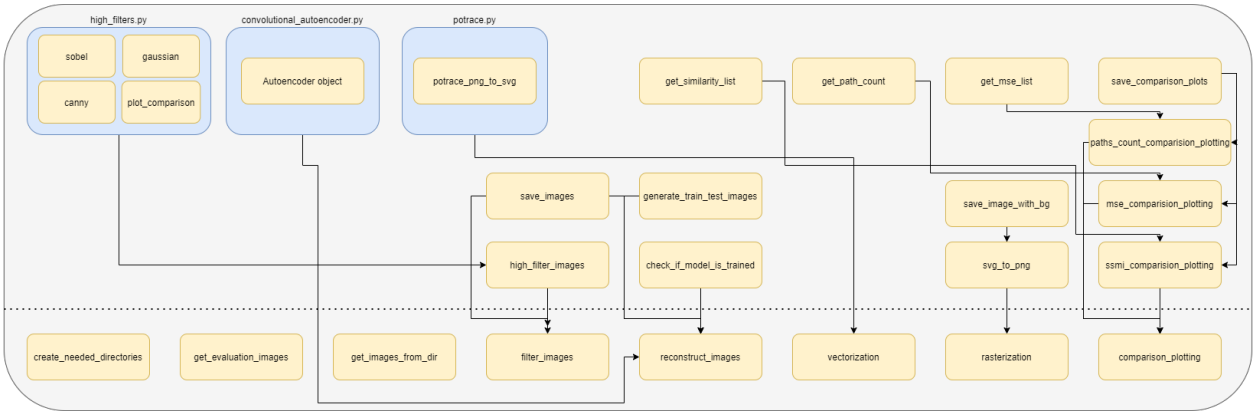

Figure 4.9: utilities script structure

functions that are going to be used in the main script. Following is the description of each exported function:

- *create_needed_directories():* This function uses the pathlib library to create all the necessary directories and paths in order to structure the project properly.
- *get_evaluation_images(images_dir):* This function returns all the images that the directory provided as parameter contains.
- *filter_images(images, filters, output_path):* This is a function that applies every filter passed on the images respectively, and saves the filtered images in the output path directory.
- *reconstruct_images(models_path, train, epoch, evaluation_images):* This function is the function used to utilize the autoencoder class. It creates an autoencoder object, trains it if necessary, or loads it, and reconstructs the images passed through the parameter *evaluation_images*
- *vectorization(images_count):* This function uses the imported function from the project script *potrace.py* to vectorize the different approaches implemented in this work.
- *rasterization(images_count):* This function uses the *svg_to_png()* function to vectorize the different approaches implemented in this work.
- *comparision_plotting(images_count):* This function is implemented lastly to visualize all the results to compare them more efficiently.

4.4.5 The Main Script – *main.py*

As will be explained in section Functional Flow of Project, the main script is the main processing program that implements all of the pipelines. A good practice was to define all of the functions in *main_utils.py* and import them into the main script, therefore maintenance does not go out of control, and it is easier to develop further. Thus, the need is not to introduce the functionality used again, but to show the functional structure, which is demonstrated in figure 4.10.

4.4.6 The evaluation and experiments scripts

The experiments script is the python script containing all the functions used to do the experiments in this thesis, whose results are going to be discussed in the following chapter 5. Experiments. Whereas the evaluation script is simply a minified version of the main script, whose purpose is to display and visualize the overall flow of the project step by step using one image instead of 70 images.

4.5 Functional Flow of Project

For the general functional flow of the thesis project, two scripts must be run. The first one is *train.py*, which is fed the dataset images and applies the high-pass filters on them. It then creates two categories of autoencoder models: one trained with the original images, and a second category trained with the filtered images. These filtered images were at first presumed to be divided into three main folders depending on the applied filter: Canny, Gaussian, or Sobel. However, this is not a final setting, as some filters may get omitted in the experimentation part due to lack of effectiveness, or maybe others may get added according to the found research.

The second and final script running is *the main.py* script. It selects images randomly based on the *images_count* variable, it then resizes them, applies the chosen filters on them and finally stores them in specific folders for later autoencoder-execution. The resulting images are then put through a cascade of vectorization and rasterization. The vectorization is done as previously mentioned by the potrace program, while the rasterization is simply done using the python *cairosvg* library. The vectorized images are rasterized only in order to be able to compare them with their original raster images, and calculate the

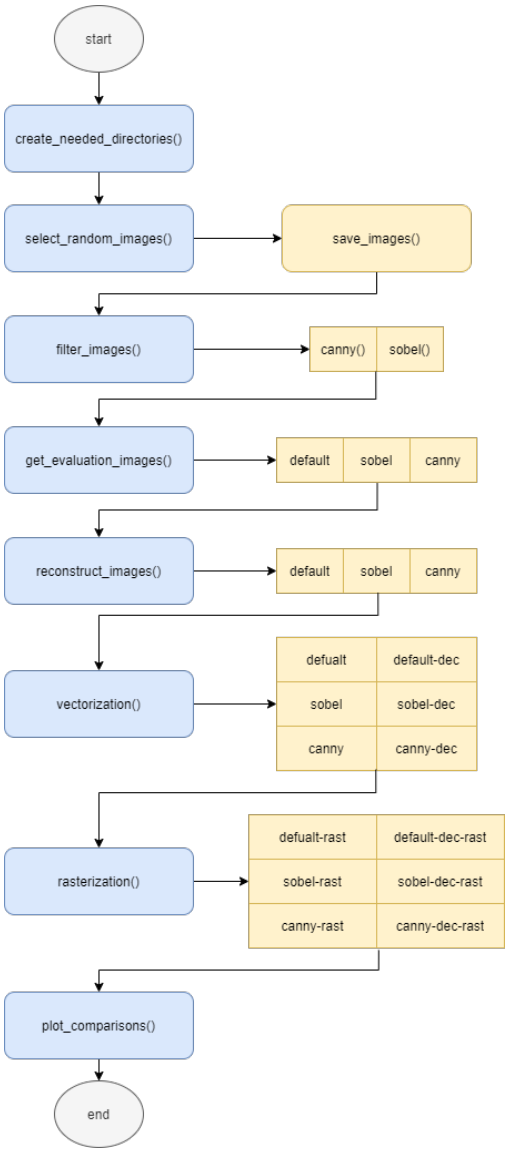

Figure 4.10: main script flowchart

distortion or loss of quality. Finally as already introduced, several comparisons are done between the different results, which are going to be discussed in details in chapter 6. Evaluation.

The flow is displayed in figure 4.11, which contains two boxes, each representing the functional flow of a script. The upper and lower boxes show the flow of the training script and main script executions respectively. At the end of the training process, the main script should be run, as displayed by the lower box: The execution is drawn from left to right, and in two rows (upper to lower sequence)

1. starting by the training images at the top being passed to the default model of the autoencoder
2. or first filtered with the high-pass filters then fed to an autoencoding model to be trained and labeled as the filtered models (one trained with sobel-filtered-images, and the other with canny-filtered-images).
3. now for the main script: it starts by getting the evaluation images and resizing them.
4. the program passes the resized images through the high-pass filters therefore obtaining two groups of images (default and filtered).
5. each group of images is passed then to the appropriate trained-model of autoencoding.

6. and the upper row of execution is finished by storing the resulted images from the autoencoding models (reconstructed images).
7. the dotted line shows the start of the second row of execution in relation to the end of the first one, as the program reads again the stored images.
8. the following processes are the vectorization and rasterization of all groups of the images, which are: the resized images, the filtered images, and the reconstructed images (from the default and the filtered models).
9. finally all of the resized, vectorized, and rasterized images are fed into the comparison process, ending at the bottom right of the chart.

4.6 Evaluation Methods

The project results can be systematically one of two things: vector images or raster images. Now to compare two images of one type, there are different approaches that can be followed:

- **SVG Comparison:** various methods can be used to measure the level of the complexity in a vector image. One can be the size of the file, which can be used to infer the length of the entire path entries in the file. Furthermore, investigating the reduction of complexity can be done through analyzing the longest path tags. The number of path tags can be taken as a characteristic value of the complexity. Thus, it is assumed that the number of path entries is directly related to the complexity.
- **PNG Comparison:** There are mainly two common ways of comparing raster images. The first one is comparing images based on the mean square error (or mean square deviation). MSE value denotes the average difference of the pixels all over the image. A higher value of MSE designates a greater difference between the original image and processed image. Nonetheless, it is indispensable to be extremely careful with the edges. A major problem with the MSE is that large differences between the pixel values do not necessarily mean large differences in content in the images. It is also applied globally to the image. However, the MSE can be calculated very quickly. For these reasons, the MSE is an interesting method that is used to compare the pixel values, but it should not be the only characteristic value to be used for the evaluation. The mean square error does not provide any information about the structural similarity. For this purpose, an algorithm is used that can compare images of similar structures. The Structural Similarity Index (SSIM) is designed to extract structural information from a scene and compare them with one another. The SSIM method is significantly more complex and computationally intensive than the MSE method, but essentially the SSIM tries to model the perceived change in the structural information of the image, while the MSE actually estimates the perceived errors. There is a slight difference between the two methods, but the results are significantly different. Instead of using the entire image as with the MSE, two windows or small sub-samples are compared with each other when calculating the SSIM. This leads to a more robust approach that is able to account for changes in the structure of the image rather than just the perceived change in pixel values across the entire image. The implementation of the SSIM used is contained in the Python library scikit-image.

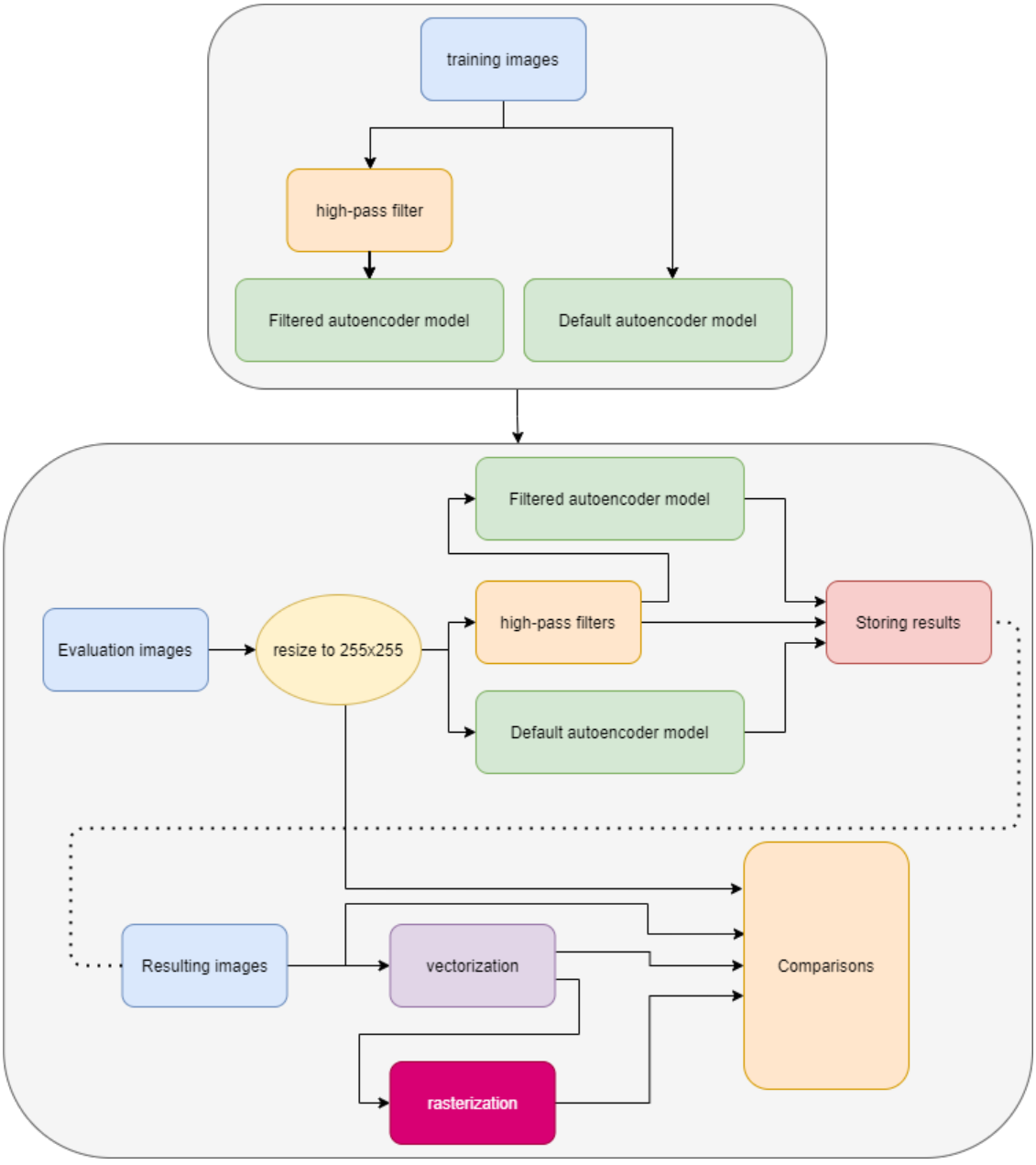

Figure 4.11: project functional flow

5 Experiments

In this section, the experiments undergone in this thesis are discussed. At first the thesis structure was roughly put, therefore it relied on some experiments to better define the path the project should take.

The experiments were mainly done in order to investigate the fitness of the filters as a whole, and the way they should be applied. Systematically speaking, when having a cascade of algorithms representing filters, it is intuitive to look for enhancements. All of the experiments are structured into functions in *experiments.py* script.

Firstly, a sample of five images were filtered with the initial high-pass filters. The results are shown in figure 5.1.

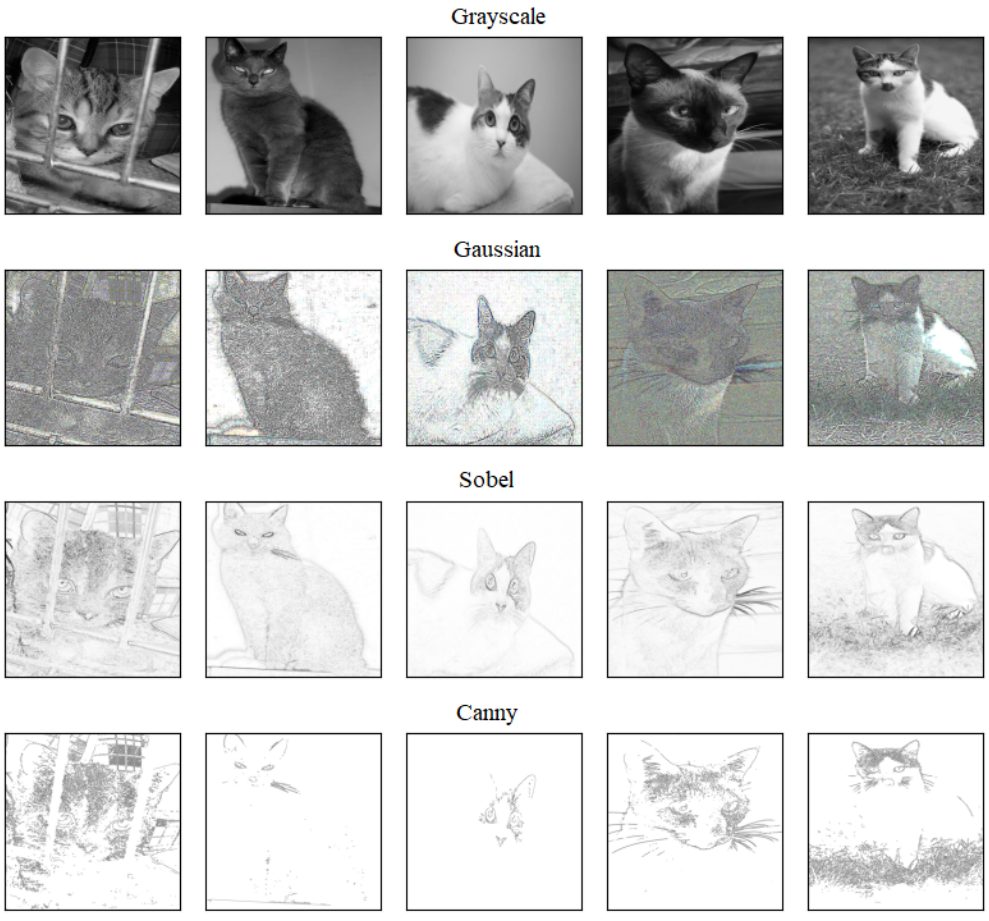

Figure 5.1: applying different filters to five random images

The first impression was that the Gaussian filter results in some significant noise. The sobel and the canny filters were acceptable, with sobel seemingly having better results to the human eye. The three filters were inverted, because it made more sense to have the detected lines drawn black on a white image than having the opposite case.

5.1 Blur-Free Noise-Reduction Filtering

As an attempt to reduce the noise the Gaussian filter was causing, two trials were done. They both were in a form of cascading a filter on top of each high-pass filter. This smoothing filter should result in noise reduction while avoiding blurring the image. Hence, two filters were chosen: difference and grain-extract filters (equations 5.2).

$$Grayscale\ offset = Grayscale\ image + 128$$

$$Difference = Grayscale\ image - Filtered\ image = Grayscale\ image + Inverse\ filtered\ image$$

$$Grain\ extract = Grayscale\ offset - Filtered\ image = Grayscale\ offset + Inverse\ filtered\ image$$

Figure 5.2: filters equations

Using the experiments script, the results in figure 5.3 were obtained.

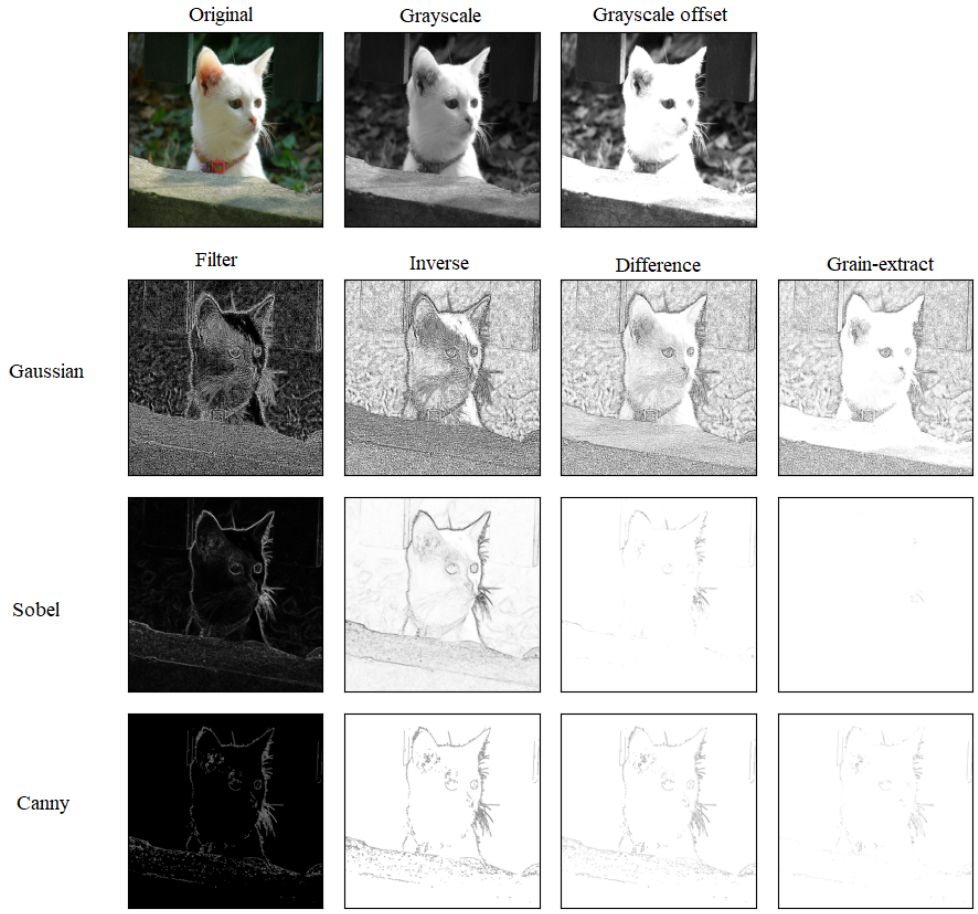

Figure 5.3: applying the difference and grain-extract to a random image after being filtered

As it is visualized, the noise-reduction filters could result in a roughly better version of the Gaussian filter, although the image is still too noisy to be fed into a neural network. Whereas for the sobel and canny filters, the difference and grain-extract filters caused a deterioration in image quality and a significant loss of data. Hence, the experiment implies the unfitness of these two suggested filters as a further preprocessing stage, and also the definite omission of the Gaussian filter from any further use in the project due to its inevitable noise.

5.2 Filter-Inversion Effect on Autoencoding

The second experiment done in this section is obtaining the difference between training an autoencoder with images whose lines are drawn in black on white background, and training it with the same images but inverted.

Therefore, four models of autoencoders were trained with 5000 epochs each in addition to the default model, which makes them five models each trained with the following types of images respectively: grayscale images, sobel-direct images, sobel-inverse images, canny-direct images, and canny-inverse images (direct: dark background and white features. inverse: inverse of direct). Five images were selected randomly and put through the five trained models as shown in figure 5.6:

The first conclusion drawn was that when training an autoencoder, the semi-supervised neural network responds better when the training images have darker lines in their important features. However, a rough estimation with the human eye would not make the cut, rather an exact mathematical calculation. Therefore, a measurement of similarity was done between every image and its decoded version. This was a better way of using the SSIM rather than comparing them with the default images, as the goal was to determine how close was the autoencoding. For this part, 50 images were used, in order to damp the image-specific features, and make the measurement more generalized. The measured values were plotted in figure 5.4.

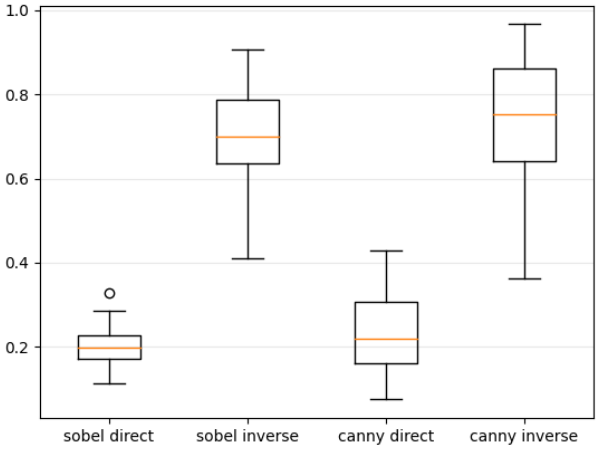

Figure 5.4: SSIM of different autoencoding approaches

For the sobel-direct, the mean and standard-deviation values were 0.202 and 0.044 respectively. Whereas their inverse scored 0.699 and 0.124 respectively. For the canny-direct, the mean and standard-deviation values were 0.234 and 0.090 respectively. Whereas the inverse scored 0.741 and 0.150 respectively.

These values further support our first remark stating a better learning rate for the autoencoder when the most important features of an image are darker than its other contained data. The experiments so far resolved into using the sobel and canny filters and more specifically their inverted results. At the start, it was thought that the experiments would resolve into choosing only one filter as a preprocessing stage for the autoencoding, however as calculated previously, the quality of images between the sobel and canny images is very close that it does not imply the disregard of one of the two filters.

Nevertheless, there is a significant drop in quality when applying a high-pass filter on the original image and then passing it through an autoencoding stage. This raised a flag that perhaps the order of the pipeline might be not thorough. For instance, the autoencoder is perceived to work as a reconstruction algorithm. Simultaneously, it can be considered to smoothen the image, or in other words, represent it with more coherence between the pixel values. Therefore, the high-pass filters might be more efficient if placed after the reconstruction of the images, instead of being applied before the autoencoding stage, which is at a first

glance cancelling out some of the emphasis that the filters generated. Hence, an experiment in the matter should be performed.

5.3 Autoencoders as a Preprocessing Stage to High-Pass Filters

In this experiment, random images were taken, reconstructed with an autoencoder, filtered, and then vectorized. This experiment aims to display the effect of high-pass filters on reconstructed images vectorization. The experiment is encapsulated in a function in the script *experiments.py*, and five random resulting images are as shown in figure 5.7.

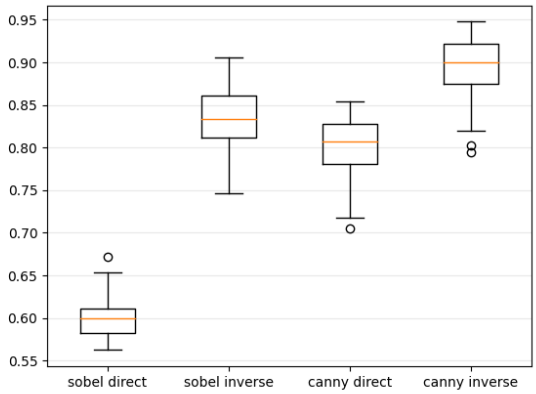

Figure 5.5: SSIM comparison of the vectorization of each of the four groups of images

The first impression the experiment gives off, is that the filters brought more definition to the lines in the images, which made the shapes appear clearer. This can lead to better vectorization, as it depends on the definitions of the shapes represented in the tags.

However, there are two versions of each of the two filters, which suggest an evaluation of the vectorization of the each of the four result-groups. Therefore, an SSIM calculation was done between every filtered image and its vector format, in a pool of 50 images, randomly selected. The results are as displayed in figure 5.5.

The box-plots show the better fitness of white images with black lines when compared to the darker images in vectorization. Visually speaking, the sobel filter results were more recognizable to the naked eye. However, it left more complexity in the image that made it harder for the vectorization to be more exact. Therefore, it is concluded that the darker shapes are going to be used in both filters, while there is not yet a clear end point to resolve into depending on only one of the two filters. Hence, a parallel stage of execution is introduced into the execution, which takes the autoencoded images and filters them with one filter, before passing them to the global vectorization stage.

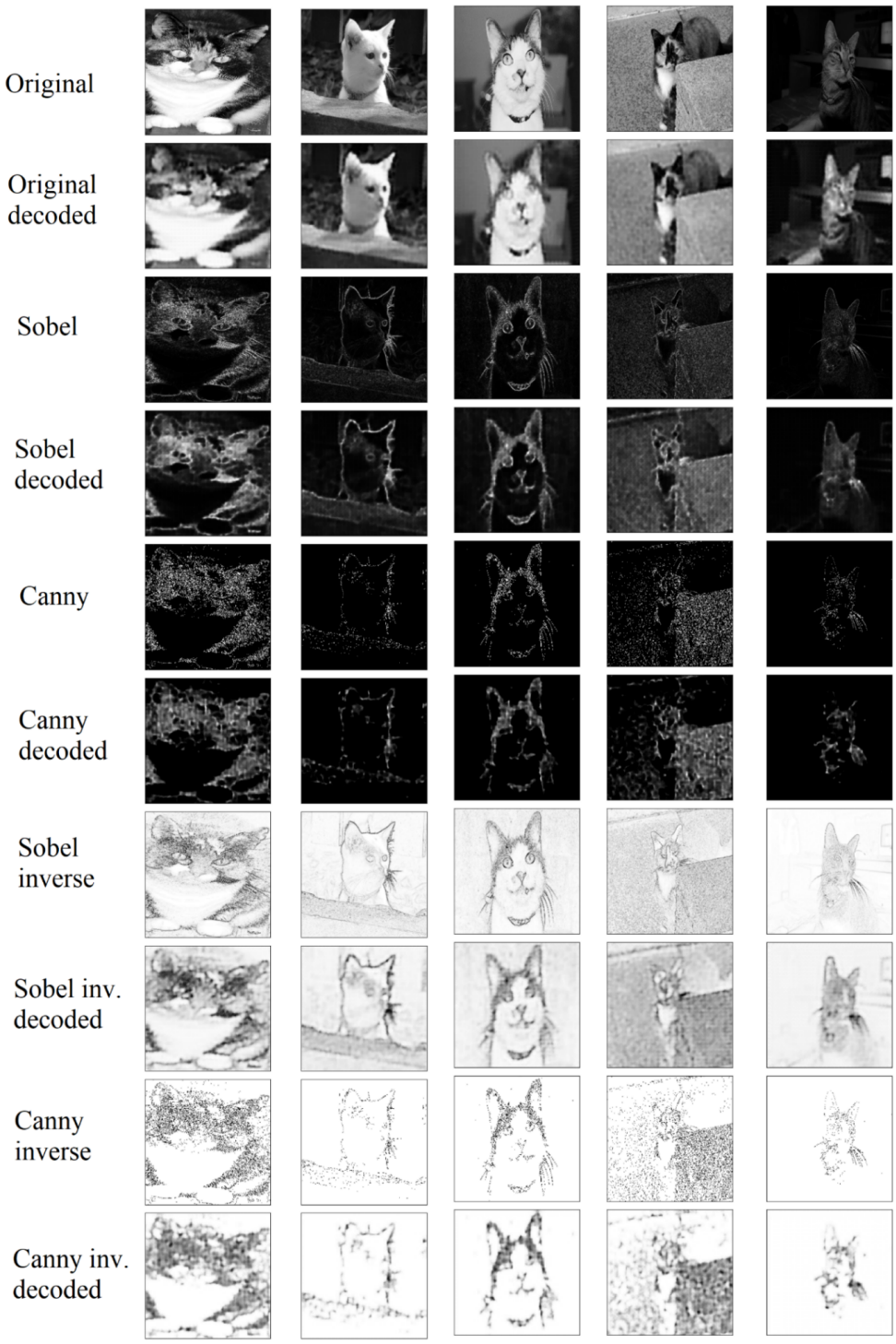

Figure 5.6: comparison between the autoencoding of the sobel and canny filtered images with both of their versions

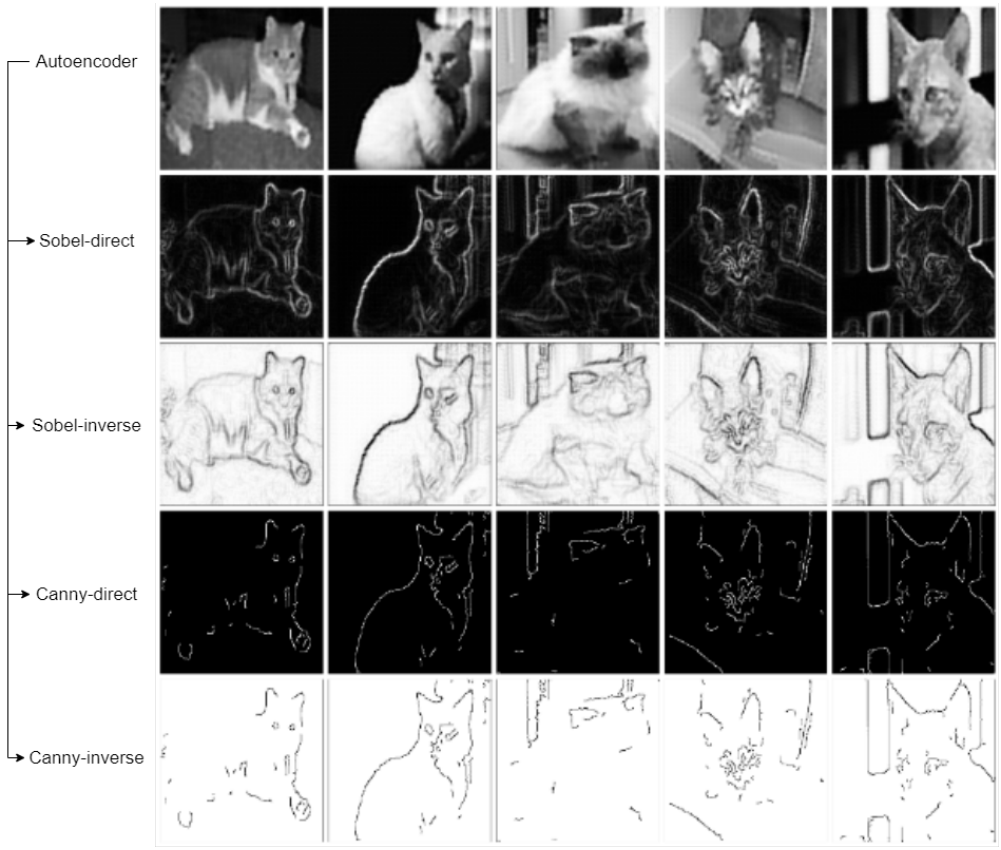

Figure 5.7: filtered autoencoder images with sobel and canny (both versions each)

6 Evaluation

For the evaluation section of this thesis, the pipelines built are going to be put through an evaluation process which is based on the methods introduced in section 4.6 Evaluation Methods. The script uses 70 images, randomly selected, and make them go through the flow of execution. For convenience, one example of the resulting images is visualized throughout the execution in figure 6.1. This was done with the help of the evaluation script, which was referred to earlier as a minified version of the main script of this project.

As a first step, the algorithm used in the main script should be discussed in further details. The program first selects 70 random images. Systematically, any number of images can be selected; however, using 70 images can be considered to achieve a reasonable amount of generalized results without overloading computations unnecessarily. Those images are converted into grayscale, and each filtered by the sobel filter and by the canny filter seperately; therefore three copies of every image are obtained. The three versions of each image are then either vectorized directly or fed into the appropriate autoencoder model and then vectorized. The dotted arrows connect each group and its vectorized version.

After the vectorization, the vector images are rasterized with *cairosvg* [20] in order to perform the spatial calculations that indicates the similarities and errors between the original images and the vectorized ones. The results are shown in figure 6.1.

At first glance, it can be picked up from the evaluation chart, that the two first rows of vector images do not align with the desired goal of this work (obtaining an abstract representation of an image) for one of two reasons:

- The filtered image (regardless whether it's passed through an autoencoder) experiences a significant drop in quality that cannot be accepted.
- When the image is only reconstructed and then vectorized, although it gets smaller in size, it does not go down to its primal features, making it too complex to be considered abstract.

On a positive note, the evaluation chart shows initially that the vectorization was optimal when the image was reconstructed then filtered with one of the two high-pass filters (canny and sobel), which is seen in the third and last row of vector images.

Before engaging in the results of evaluation methods, it is good to elaborate on the column naming of the upcoming plots:

- default: the default image
- sobel, canny: the filtered version of the image by the respective filter
- dec: the decoded version
- vect: the vectorized version
- A combination of two or more indicates the case of cascaded stages: e.g. default-dec-sobel: the default image is reconstructed with the autoencoder then filtered with the sobel filter.

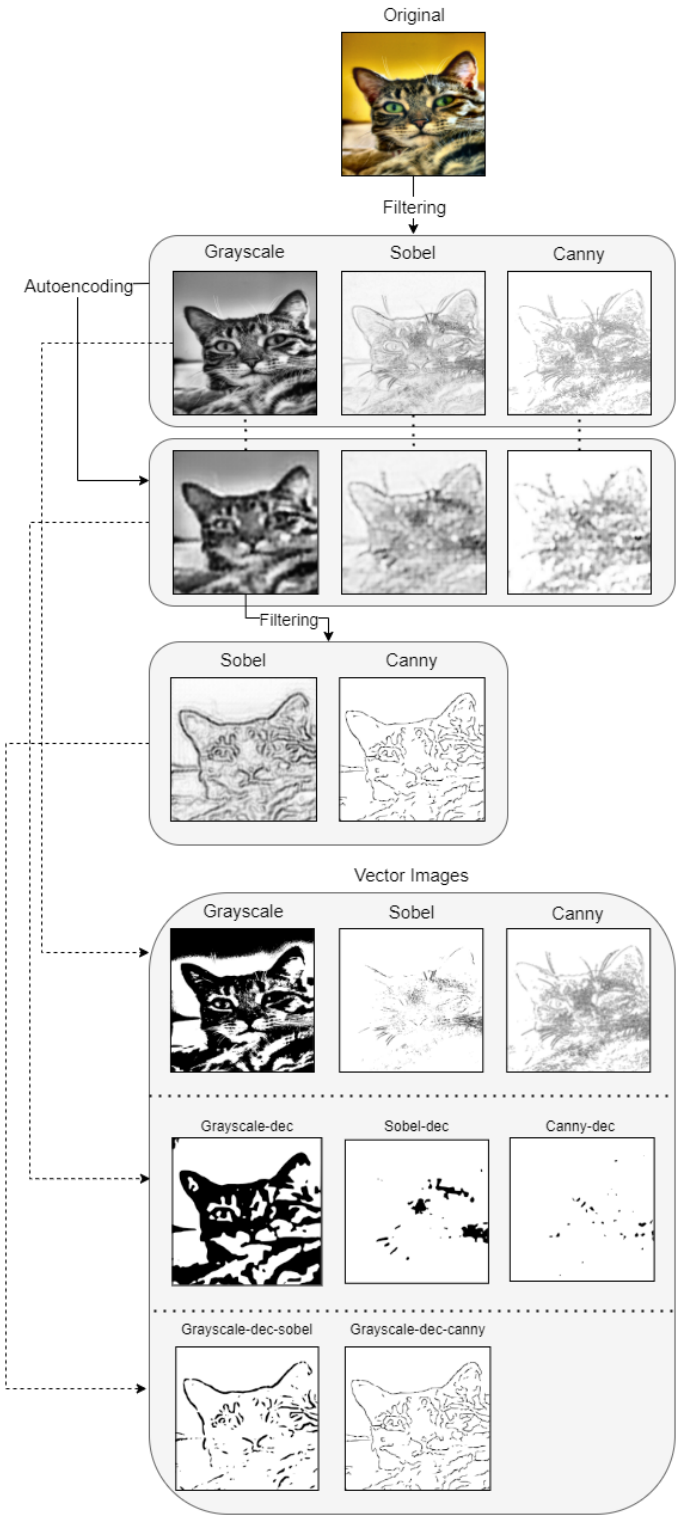

Figure 6.1: evaluation steps with one random image

6.1 Vector Comparison (Path Count)

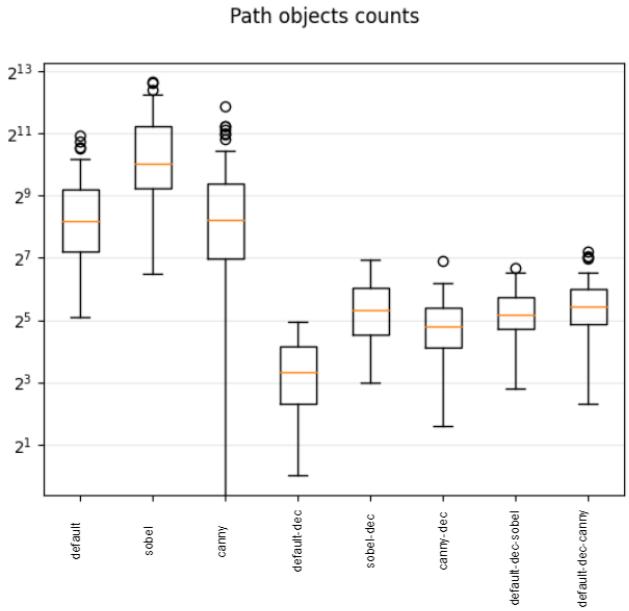

Figure 6.2: path-count of the resulted vector images

Path count can be utilized to measure the size of a vector image. It is clear from the calculations in figure 6.2 that the autoencoder (the five columns from the right, i.e. *-dec-*) significantly reduced the size of images. The autoencoding can be represented here as the stage that keeps only the most important features of an image, therefore its vectorized version contained fewer paths. One anomaly can be spotted in the vectorized canny images (canny), as the filter resulted in a significantly bigger deviation. This is explicable by the fact that the canny filter produced images with a very low number of features or with a small sized that they were ignored by the vectorization algorithm (empty image, i.e. zero paths). Note that the difference in the graph is in logarithmic scale, hence it is way bigger than what is perceived at first glance. This logarithmic scale was used due to the fact that the different columns had so big of a difference that it made some values unable to be read directly from the graph. One of the other remarks that can be seen from the box-plots, is that the autoenocoded images hold less deviation (standard-deviations) as a whole. This can be interpreted as the consistency the autencoder models provide, as they get trained to reconstruct the critical building features of the training images. Furthermore, the sobel and canny filtered images that have been put through the autoencoding step (canny-dec, sobel-dec), had a similar path count, although it was much smaller than the ones that did not go through that step, it was still above the default images that were reconstructed and vectorized without any filtering. Finally, when filters were applied onto the default images that were put through an autoencoding stage (default-dec-sobel, default-dec-canny), these images scored in size calculations very similarly to the filtered images when only reconstructed (canny-dec, sobel-dec). The only thing that would appear as an anomaly, is that the default images when reconstructed (default-dec), were smaller in size than the other images filtered before or after reconstruction (four groups from the right). This can be due to the fact that the inverse of the filtered images was used in every filtering stage. This means that the resulted images had way more brighter pixels than darker ones, leading to an increase in size.

6.2 Raster Comparisons (SSIM & MSE)

When it comes to raster images, the image pool for comparison contains all of the images before and after the vectorization stage. As a reminder, a score of 1 in SSIM indicates perfect matching, and a score of 0 in MSE indicates zero difference (hence no errors).

First, a comparison between the original (or default) images and their processed versions is shown in figure 6.3.

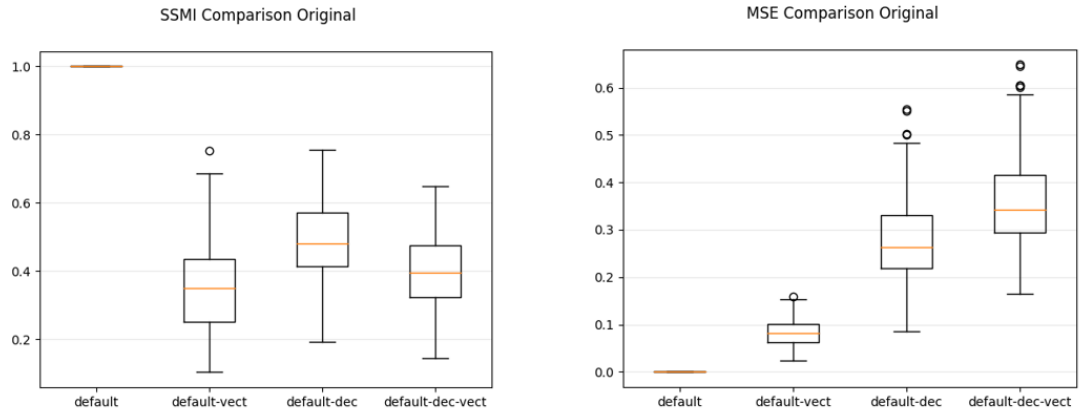

Figure 6.3: SSIM and MSE of default images and their processed versions

The closer the median of the plot (the orange line in the middle of the box) to the perfect score (1 in SSIM, and 0 in MSE), the higher the similarity between these images and their default version. default-vect group represents the default images when vectorized, and default-dec group is the default images that were reconstructed by an autoencoder, which in return when vectorized, are represented in the first column from the right default-dec-vect. There isn't much of consistency between the default images and their processed versions, as one interpretation can be: the information they contain, saturates the fitness of the autoencoder model and the vectorization algorithm, leading to having more errors when autoencoded before vectorization. However, it is hard to correlate between the similarity index and the MSE, as they do not follow similar calculation procedures. This further proves the essence of this work, which is introducing a cascade of an autoencoder model and high-pass filters in a way that enhances the vectorization process by preprocessing images in a way that transforms them into their abstract representations.

Next we have the calculations for the sobel and the canny filters, and each of their processed versions are displayed respectively in Figure 6.4 and figure 6.5. These calculations show the closeness of the filtered images (canny or sobel) to their processed version (reconstructed and/or vectorized).

The results for both filters were very similar, although there is more deviation in the images processing with the Canny edge detection filter. The structure of the plots is similar, and can state that when filtered images are passed through an autoencoding stage, their size get reduced significantly; however, this does not affect the vectorization accuracy. As from the graphs, it is clear that both the images, when reconstructed with an autoencoder (sobel-dec, canny-dec), scored when vectorized (sobel-dec-vect, canny-dec-vect) similarly to the ones that were not reconstructed (sobel-vect, canny-vect). This is a clear advantage of using a neural network to autoencode the images in a way that preserves the important features in an image, in a way that does not deteriorate their vector versions.

When introducing a high-pass filter to an image, the similarity between the filtered image (sobel, canny) and its vectorized version (sobel-vect, canny-vect) is way bigger than when vectorizing the default version of that image without any filtering (default-vect in comparison to default). This was seen when the SSIM

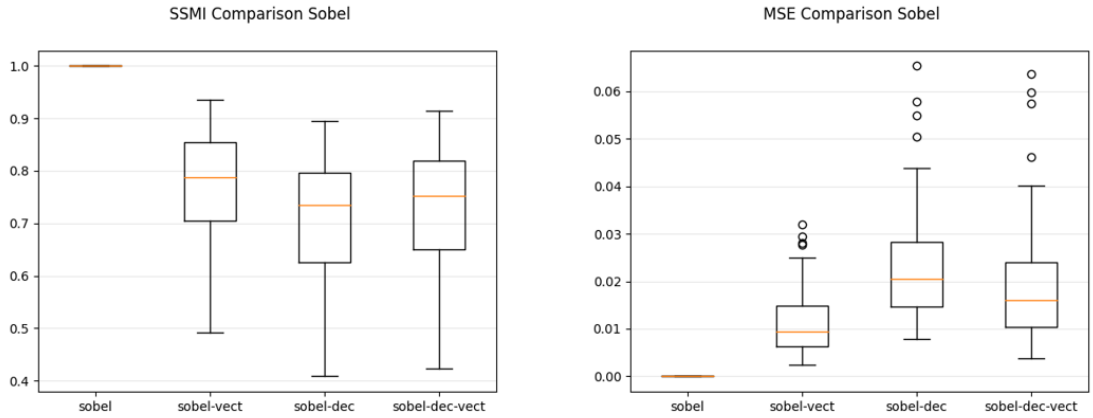

Figure 6.4: SSIM and MSE of sobel images and their processed versions

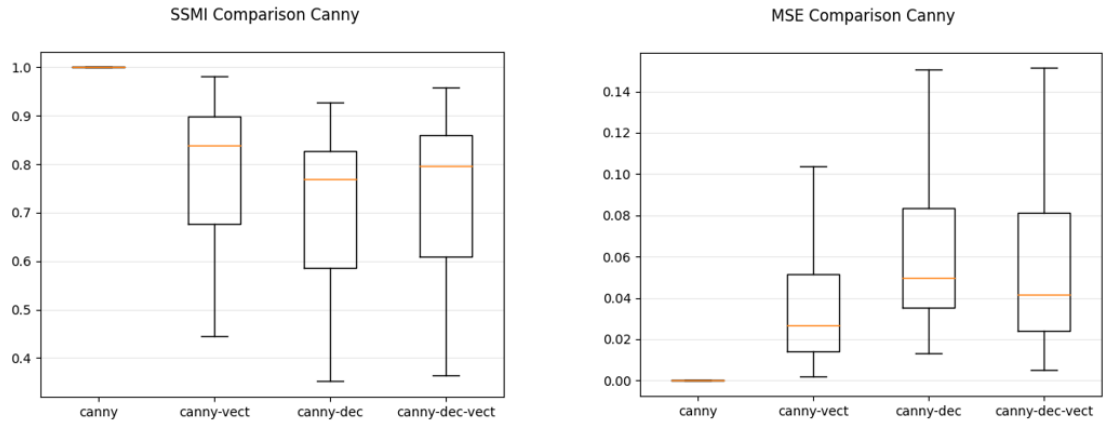

Figure 6.5: SSIM and MSE of canny images and their processed versions

of a filtered image and its vectorized version are put next to the SSIM of a default image and its vectorized version, as the first was in the range of [0.8- 0.9] while the latter was only in the range of [0.3-0.4].

However the consistency provided by the filters can have a downside. Although the images when being filtered, seemed more fit for autoencoding and vectorization, the difference in quality that the filters introduce should be examined. Therefore, a comparison was done between the default images and their filtered versions as figure 6.6 shows. It is clear from the two plots, that there was somewhat of a significant drop in quality when the images were filtered, as only 40% for images on average was similar to their default versions. This was expected, as the high-pass filters only keep the important features of an image such as lines and edges, and the calculations were based on the similarity and the mean squared error. However, the aim of this work revolves around the use of high-pass filters in a way that increases the efficiency of vectorization and resulting in an abstract representation of the images.

A more accurate way of examining the efficiency of the vectorization process of each of the pipelines, is to perform SSIM/MSE calculations between the images and their vector versions (figure 6.7). This means that all images resulting in every pipeline are taken, vectorized and compared with their versions before vectorization (rasterization is included to allow comparison). The two groups on the most right in both plots, are the result of the experiment in section 5.3. As a quick recap, the default images were passed through an autoencoder, then filtered with one of the two high-pass filters, and finally vectorized. This pipeline of execution seems to have the smallest MSE and the highest SSIM, which indicates its fitness in vectorization. The experiment was put together because it made more sense for the autoencoder to

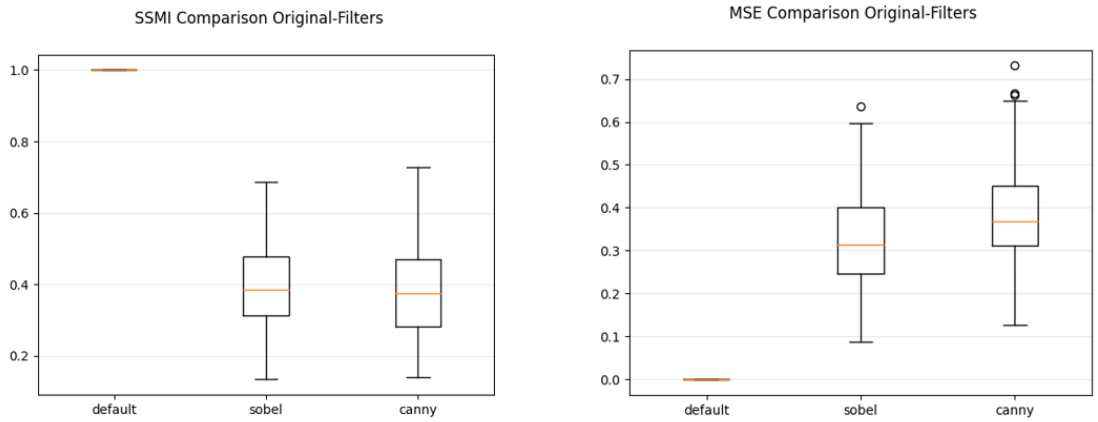

Figure 6.6: SSIM and MSE of default images and their filtered versions

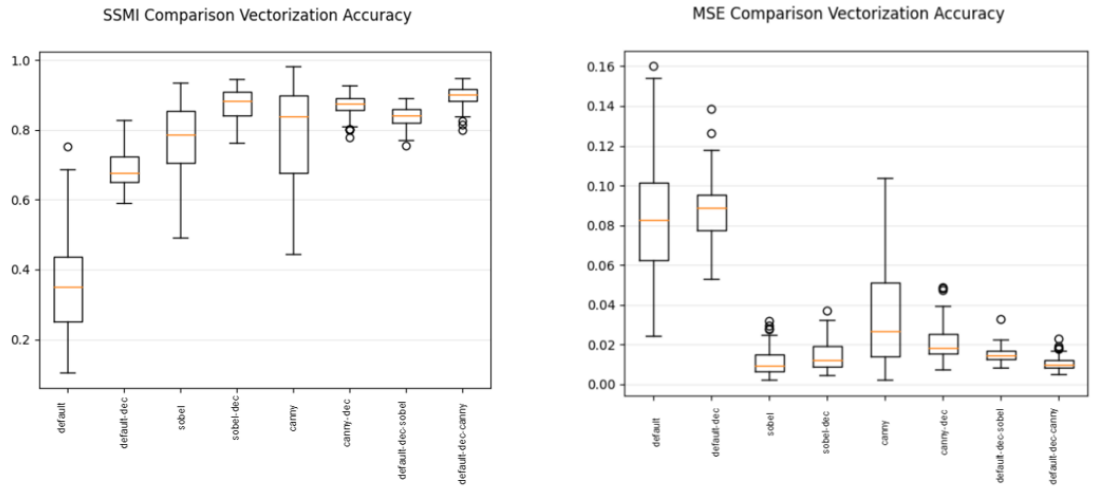

Figure 6.7: SSIM and MSE of images and their vector versions

reconstruct the images and then for the filters to come afterwards, putting emphasis on the important features of each image. As it can be also seen from the evaluation figure displaying a random image going through the various pipelines, the lattest results (i.e. autoencoder-filter cascade) are the most satisfactory.

6.3 Interpretation

This evaluation section was intended to go over the details of the results, in order to evaluate the different pipelines built in this work. Therefore this closing section is going to put forward an interpretation-bullets block as a summary of each pipeline evaluation; otherwise, the section elaboration would be overwhelming to get a good grasp of. The naming of every bullet point is the cascade of stages: e.g. *Filtering-Vectorization* is the pipeline that filters the images then vectorizes them.

- **Autoencoding-Vectorization:** This pipeline was based on the work of Fischer [5]. However, the implementation was different (in terms of programming aspects), and the evaluation was about the abstractness of the results. The quality of the vectorization is acceptable only in terms of general similarity. However, the abstract representation of the image is not achieved.

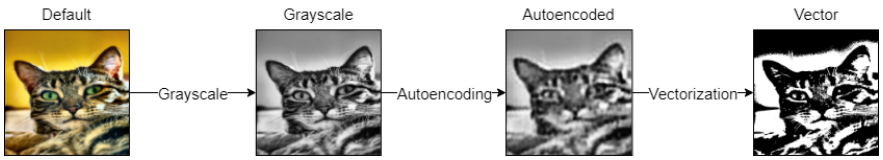

Figure 6.8: Autoencoding-vectorization pipeline

- **Filtering-Vectorization:** In this pipeline, the vectorization algorithm finds a difficulty in vectorizing the filtered images. This is due to the noises caused by the applied filters. Although the experiments showed that the quality of the vectorization increased when the images where taken as a light background with dark features, the noise involved created an obstacle for Potrace to convert thoroughly the images into a vector format, which resulted into losing data.

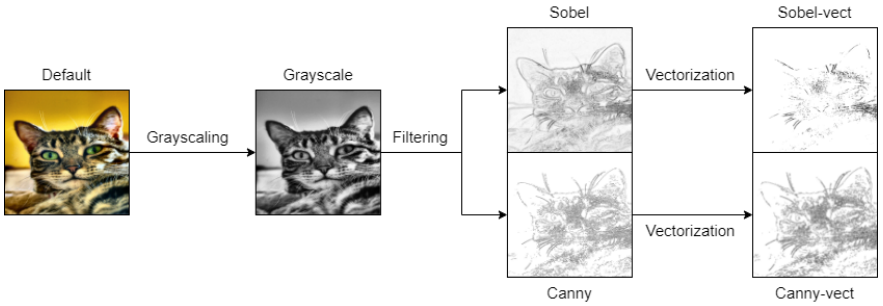

Figure 6.9: Filtering-vectorization pipeline

- **Filtering-Autoencoding-Vectorization:** This pipeline was built as an attempt to enhance the *Autoencoding-Vectorization* pipeline. Although the autoencoding stage was efficient in reducing the size of the images, it did not result in an abstract view of the image features. Therefore, a filtering stage was placed prior the autoencoding process. Unfortunately, this pipeline does not achieve the result intended. The autoencoding stage was supposed to reconstruct the filtered images in a lower complexity; but the case at hand is that, the autoencoding model is attempting to smooth the images, canceling the effect of the high-pass filters. This has resulted in a significant drop in the quality of the vector images, which is seen figure 6.10.

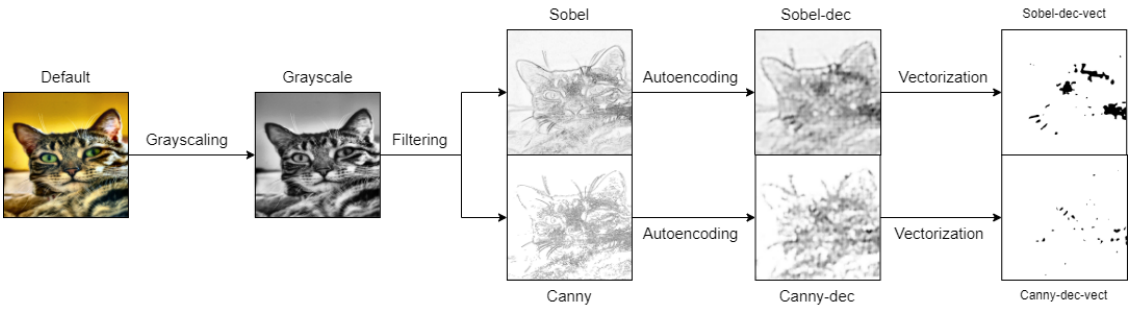

Figure 6.10: Filtering-autoencoding-vectorization pipeline

- **Autoencoding-Filtering-Vectorization:** Due to the results in the *Filtering-Autoencoding-Vectorization* pipeline, it was clear that the filtering stage would act in a more proper way if it succeeded the autoencoding process, rather than preceding it. This was concluded when the autoencoding model was

seen to reduce the complexity of the images while introducing a smoothing effect. The filters were placed after the reconstruction stage to preserve the important features of the reduced-complexity image. This cascade shows an acceptable vectorization quality while resulting in the intended abstract representation of the images as shown in figure 6.11.

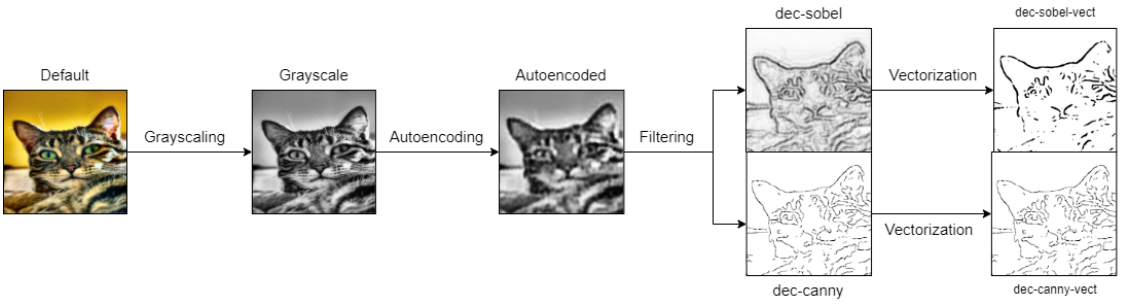

Figure 6.11: Autoencoding-filtering-vectorization pipeline

7 Conclusion & Outlooks

At the very beginning of this thesis, there was a question on whether high-pass filters would enhance the vectorization process of Potrace, and whether they could be used in combination with autoencoders in a way that facilitates for the vectorization algorithm to construct an abstract representation of the processed image. The use of an autoencoder was already introduced in the related work section, as an attempt to reduce the size of the vector images.

As the high-pass filters are known for their vast use in edge detection algorithms, they became a candidate to be set as a preprocessing stage for the vectorization, in both cases of presence and absence of an autoencoding stage prior to the Potrace execution. First was the search for reliable high-pass filters, which resulted into choosing three of the most commonly used filters: canny, sobel, and gaussian. Python was chosen to be the programming language that the thesis work is going to be implemented in. Systematically, a repository was created to contain all of the various scripts used in the thesis, and more importantly, their results. The framework Keras was also included in the requirements of the Python repository, as it simplifies the construction of the autoencoder model used. Finally, the repository is hosted on the institute's Git services [16].

An initial test was done to evaluate the noise resulted when using the filters, as noise can be very detrimental when it comes to both of vectorizing an image and training an autoencoder. In the first case (vectorization), noise would result in unnecessary paths, which implies no size reduction or even magnification in some severe cases, and in the second (autoencoding), it would make the training process give importance to irrelevant data that was added through noisy filters. This leads to models that do not reconstruct images based on their significant data, therefore lack of similar results. The test revealed that the gaussian filter generated some noise in its filtered images, and as a first experiment, the difference and grain-extract filters were introduced as an attempt to reduce the noise. However, their effect was not efficient to the point of considering to cascade them with the initially-chosen high-pass filters. Therefore, the gaussian filter was flagged as unfit to be used in the preprocessing stages.

Having both sobel and canny filters in place to use, it was only systematic to go through an experiment to decide which filter is more fit to be set to work. The autoencoding stage was going to have initially two parallel executions, each having a different model: one trained with default images, and the other trained with filtered images. Therefore, two versions of each image in the dataset were created, whereas each version was filtered by one of the two high-pass filters. These filtered images with each of their versions, trained a different model of the autoencoder; and at the end a calculation of similarity was done to evaluate the quality of the two models, in order to choose between the two filters. The evaluation was not decisive as the results were close, which lead to having both of high-pass filters in parallel in the final flow of the project. Another side of this experiment, is having to choose whether to take the filtered image directly or inversely. It was more intuitive for the image to have a light background with darker lines defining its important data. However, this is not the case that the chosen filters are implemented in programming libraries; in fact, the case is the opposite. Therefore, the training of the autoencoder was put to test on which version of an image it responds best to. It was indeed the inverted versions of the filtered images that scored significantly higher on the similarity between the reconstructed images and their original versions. Thus, both filters were taken in use and with their inverted results. Finally, as the autoencoding stage was perceived to smooth the images by reconstructing them, it was only intuitive to put an experiment in which the default images were passed through an autoencoding stage, then later filtered and vectorized. Each

of the two filters had two versions: a direct one and an inverse one. Therefore a similarity comparison was done to compare how efficient was it for the vectorization algorithm to vectorize each version. It was cleared out that the images with a brighter background and darker lines were much favored by the algorithm in both filters. This concludes the experiments of this thesis.

Lastly, a more detailed evaluation was done to observe the results of the various pipelines of the project. The path-count calculations were done on the vector images obtained, and the SSIM and MSE calculations came in place after the rasterization of these vector images.

Overall, the images that were reconstructed by the autoencoder models were all smaller in size while not scoring a significant drop in quality. This was consistent in all of the models used. The filtered images on one hand scored very similarly, although the canny filter is perceived as more complex than the sobel. They both had close results in the various pipelines built. On the other hand, the high-pass filters kept the important data of the images, which resulted in a better training set of images that helped in building a more accurate autoencoding model; thus, better reconstruction and less error. This was seen in the reduction of standard deviation of images autoencoded with models trained with filtered images. One result that did not align with the initial speculation is the size reduction of default images when passed through an autoencoder, in comparison to filtered images. It was expected a drop is image size when an image was to be filtered before reconstruction; however, this was not the case. An explanation can be that the images filtered had brighter background and more white pixels then the default images.

After evaluating the efficiency of the vectorization algorithm in every pipeline, it was clear that the images that went through the cascade of autoencoding-filtering, scored the highest in similarity, and the lowest in error. This points to the fact that the images that were reconstructed, preserved the most important features, which were brought up even more by the filtering part succeeding the reconstruction, which leads to not only a better vectorization but also a more abstract representation of the image. This was also remarked when the result of each stage of every pipeline was displayed on a random image as in the evaluation figure.

This thesis overall discussed the use of high-pass filters in vectorization pipelines, along with the autoencoding stage. As already concluded in this chapter, high-pass filters can enhance the training of an autoencoder, which in return make the vectorization process more efficient by preserving the important features of an image. More importantly, the best result of this work can be regarded in the cascade of autoencoding-filtering stage that had the highest efficiency in vectorization. Furthermore, this can be experimented with, especially in the area of object recognition, as the performance can be enhanced not only concerning the true-false decision-ratio, but also in execution speed.

The cascade of autoencoding-filtering gave decent results that matched the initial expectations; however, further work must be put into the structure of the models built, and their training dataset.

Finally, it is only systematic to state that this can be regarded as only a solidified beginning of development.

Bibliography

[1] Mikhail Bessmeltsev and Justin Solomon; *Vectorization of Line Drawings via PolyVector Fields*. ACM Trans. Graph. 0, 0, Article 0 (2017), 12 pages.

[2] Vinciane Lacroix; *Raster-to-Vector Conversion: Problems and Tools Towards a Solution A Map Segmentation Application*. 2009 Seventh International Conference on Advances in Pattern Recognition

[3] H. Karaborka, B.Kocerb, I.O. Bildiricia, F.Yildiza, E.Aktasc; *Raster-to-Vector Conversion: Problems and Tools Towards a Solution A Map Segmentation Application*. The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences. Vol. XXXVII. Part B2. Beijing 2008

[4] Zhe Gong; Li Xu; Zhenpo Tian; Jingyuan Bao; Delie Ming; *Road network extraction and vectorization of remote sensing images based on deep learning*. 2020 IEEE 5th Information Technology and Mechatronics Engineering Conference (ITOEC)

[5] Andreas Fischer; Manuel Amesberger; *Improving Image Tracing with Artificial Intelligence*. 2021 11th International Conference on Advanced Computer Information Technologies (ACIT)

[6] John Wiley; *Digital image processing*. Inc. pp. 465–522, 4th edition, 2007.

[7] Image Derivatives, . https://docs.opencv.org/3.4/d2/d2c/tutorial_sobel_derivatives.html, (accessed on 21.11.2021)

[8] Sobel *Filtering in the Time and Frequency Domains*. https://docs.opencv.org/3.4/d2/d2c/tutorial_sobel_derivatives.html (accessed on 21.11.2021)

[9] Herman J. Blinchikoff, Anatol I. Zverev; *Filtering in the Time and Frequency Domains*. The Institution of Engineering and Technology, 2013

[10] John Canny *A Computational Approach to Edge Detection*. IEEE Transactions on Pattern Analysis and Machine Intelligence (Volume: PAMI-8, Issue: 6, Nov. 1986)

[11] Peter Selinger *Potrace: a polygon-based tracing algorithm*. September 20, 2003, (accessed on 21.11.2021)

[12] Francois Chollet; *Building Autoencoders in Keras*. https://blog.keras.io/building-autoencoders-in-keras.html, Sat 14 May 2016, (accessed on 21.11.2021)

[13] Keras https://keras.io, (accessed on 21.11.2021)

[14] Luber and Litzel; *What is TensorFlow* https://www.tensorflow.org/, (accessed on 21.11.2021)

[15] Jeff Dean, Rajat Monga, Megan Kacholia; *TensorFlow Dev Summit 2017 - Keynote* https://www.youtube.com/watch?v=4n1AHvDvVvw (accessed on 21.11.2021)

[16] Andreas Fischer, Zineddine Bettouche; *Image Tracing Repository*. https://mygit.th-deg.de/zb14173/image-tracing, (accessed on 25.11.2021)

[17] Weiwei Zhang, Jian Sun, and Xiaoou Tang; *TensorFlow Dev Summit 2017 - Keynote* Proc. of European Conf. Computer Vision, vol. 4, pp.802-816, 2008, https://archive.org/details/CAT_DATASET

[18] Bharath Ramsundar, Reza Bosagh Zadeh; *TensorFlow for deep learning. From linear regression to reinforcement learning*. O'Reilly Media; 1st edition (April 3, 2018)

[19] Zhou Wang; A.C. Bovik; H.R. Sheikh; E.P. Simoncelli; *Image quality assessment: from error visibility to structural similarity*. IEEE transactions on image processing : a publication of the IEEE Signal Processing Society, 13(4):600–612

[20] *CairoSVG*. https://cairosvg.org/, (accessed on 23.11.2021)