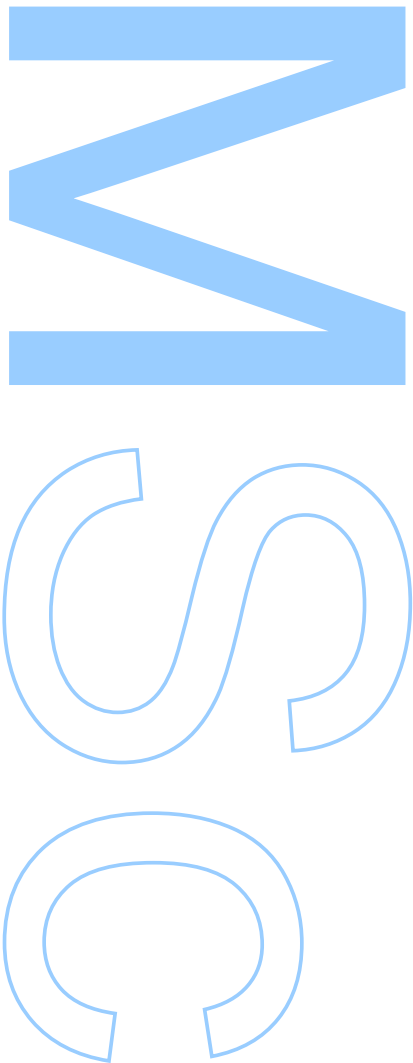
# Segmentation Label Generation Based on Self-Supervised and Semi-Supervised Learning

Berna Araújo
Mestrado em Engenharia Matemática
Departamento de Matemática
2026

**Orientador**
Prof. Dra. Rita Gaio, Faculdade de Ciências



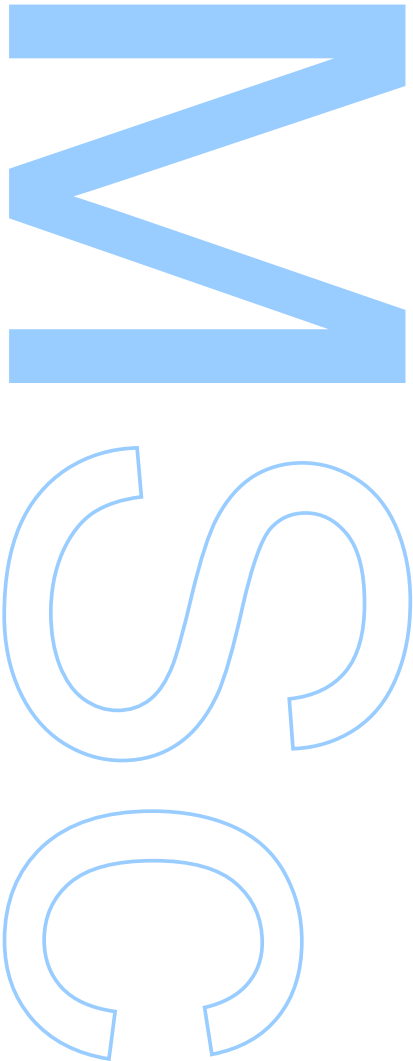
U.PORTO

FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Todas as correções determinadas pelo júri, e só essas, foram efetuadas.

O Presidente do Júri,

Porto, _____/_____/_____



UNIVERSITY OF PORTO

MSC

# Segmentation Label Generation Based on Self-Supervised and Semi-Supervised Learning

*Author:*
Berna ARAÚJO

*Supervisor:*
Prof. Dra. Rita GAIO

*A thesis submitted in fulfilment of the requirements*
*for the degree of MSc. Engineering Mathematics*

*at the*

Faculty of Sciences of the University of Porto
Departament of Mathematics

December 3, 2022

*" If you know you are on the right track, if you have this inner knowledge, then nobody can turn you off...*
*no matter what they say. "*

Barbara McClintock

# *Sworn Statement*

I, Berna de Araújo Nogueira, enrolled in the Master Degree Engineering Mathematics at the Faculty of Sciences of the University of Porto, hereby declare, in accordance with the provisions of paragraph a) of Article 14 of the Code of Ethical Conduct of the University of Porto, that the content of this internship report reflects perspectives, research work and my own interpretations at the time of its submission.

By submitting this internship report, I also declare that it contains the results of my own research work and contributions that have not been previously submitted to this or any other institution.

I further declare that all references to other authors fully comply with the rules of attribution and are referenced in the text by citation and identified in the bibliographic references section. This internship report does not include any content whose reproduction is protected by copyright laws.

I am aware that the practices of plagiarism and self-plagiarism constitute a form of academic offense.

BERNA ARAÚJO
12TH OCTOBER 2022

# *Acknowledgements*

"I would like to sincerely thank my internal adviser, Prof. Dra. Rita Gaio, for her continued support of my M.Sc. thesis and throughout my master's program. I am grateful for the opportunity of working with her, and benefiting from her knowledgeable advice and guidance.

I would also like to thank SMARTEX, and every member of staff for my internship experience, with special regards to my external advisor Miguel Gomes, and colleague Pedro Fernandes, for their guidance and expertise.

I wish to further thank my family for their support, my siblings Beatriz and Bruno for their encouragement, and my friend André Mendes, from the Physics department of this institute, for always inspiring me to further explore my scientific interests.

Lastly, I am profoundly grateful for the friends I have made throughout my years of study. Unknowingly, they played an immense role in my motivation and hopefulness during some grueling personal years."

BERNA ARAÚJO

*University of Porto, Porto*
*September 2022*

UNIVERSITY OF PORTO

# *Abstract*

Faculty of Sciences of the University of Porto

Departament of Mathematics

MSc. Engineering Mathematics

**Segmentation Label Generation Based on Self-Supervised and Semi-Supervised Learning**

by Berna ARAÚJO

This thesis serves as the internship report, with theoretical support, of my six-month internship at SMARTEX. This company develops real-time solutions powered by artificial intelligence to the problem of fabric defects in textile factories. The main project presented here develops various approaches for generating labels, in the form of image masks, for one specific fabric defect, internally denominated the *vertical defect* (for its semblance to a vertical line in the fabric images). Other projects were done, in the context of initial integration, which are included here due to being relevant enough to grasp the inner workings of an industry-grade product.

All work presented in this thesis was conducted under the SMARTEX internal framework, which included, but was not limited to, their image database, all their previous work on similar projects, already-created programs, models, and methods incorporated into my projects, and their machines. Every project was developed using Python3 running in a virtual environment on a Linux machine with the Mint distribution. The first project required SQL knowledge while every project required at least some database intuition. Machine learning and computer vision libraries were also used extensively.

**Keywords:** Machine learning, convolutional neural network, computer vision, U-NET, random forest, textile defects, label generation.

UNIVERSIDADE DO PORTO

# *Resumo*

Faculdade de Ciências da Universidade do Porto
Departamento de Matemática

Mestrado em Engenharia Matemática

**Geração de Rótulos de Segmentação Baseada em Aprendizagem Auto-Supervisionada e Semi-Supervisionada**

por Berna ARAÚJO

Esta tese serve como o relatório de estágio, com suporte teórico, do meu estágio de seis meses na SMARTEX. Esta empresa desenvolve soluções, em tempo real, com recurso a inteligência artificial para o problema de deteção de defeitos em tecidos produzidos em fábricas têxteis. O projeto principal aqui apresentado incorpora várias abordagens para a geração de rótulos, na forma de máscaras de imagem, para um defeito específico do tecido, denominado internamente de defeito vertical (pela sua semelhança a uma linha vertical nas imagens do tecido). Outros projetos foram realizados no contexto de integração inicial, que também são incluídos aqui por serem relevantes para a compreensão do funcionamento interno de um produto de nível industrial.

Todo o trabalho apresentado nesta tese foi realizado sob a estrutura interna da SMARTEX, que incluiu, mas não se limitou a, bases de dados de imagens, todos os seus trabalhos anteriores em projetos semelhantes, programas, modelos e métodos já criados e incorporados nos meus projetos, e as suas máquinas. Todos os projetos foram desenvolvidos utilizando Python3 num ambiente virtual numa máquina Linux com a distribuição Mint. O primeiro projeto exigiu conhecimentos de SQL, enquanto que os restantes projetos exigiram, pelo menos, alguma intuição sobre bases de dados. Várias bibliotecas de aprendizagem de máquina e visão computacional foram também usadas extensivamente.

**Palavras-chave:** Aprendizagem de máquina, rede neuronal convolucional, visão computacional, U-NET, floresta aleatória, defeitos têxteis, geração de rótulos.

# Contents

# List of Figures

# List of Algorithms

# Glossary

| | |
|---|---|
| **DB** | Database |
| **ML** | Machine Learning |
| **CNN** | Convolutional Neural Network |
| **MNV2** | MobileNetV2 |
| **MCC** | Mathews Correlation Coefficient |
| **ROC** | Receiver Operating Characteristic |
| **AUC** | Area Under the Curve |
| **API** | Application Programming Interface |
| **PPHT** | Progressive Probabilistic Hough Transform |
| **RGBA** | Red, Green, Blue, and Alpha (colour model) |
| **RGB** | Red, Green, and Blue (colour model) |
| **GUI** | Graphical User Interface |
| **PCA** | Principal Component Analysis |
| **SVD** | Singular Value Decomposition |
| **t-SNE** | t-Distributed Stochastic Neighbor Embedding |
| **DBSCAN** | Density-Based Spatial Clustering of Applications with Noise |
| **GPU** | Graphics Processing Unit |
| **DGB** | Distributed Gradient Boosting |
| **IoU** | Intersection-Over-Union |

# Chapter 1

# Introduction

The objective of this thesis is to report on the work, which mostly involved label generation and machine learning, done this year in the context of a curricular internship at SMARTEX. This recently founded company intends to significantly reduce the production of defects in textile factories all around the world, using artificial intelligence and computer vision technologies for defect detection, while also offering real-time monitoring software.

SMARTEX technologies are already present in several countries and to date, they estimate their product is responsible for saving more than 33.5 million liters of water, more than 650 thousand kilograms of $CO_2$, around 300 thousand kilograms of fabric, and more than 2.5 million kWh of energy [1]. I was inspired to select this organization as my internship locale because of their use of artificial intelligence techniques which are very relevant to my own career goals, and their present and potential positive global impact in sustainability. Working with companies which care about the environment and that do not place profit over the negative impact they can exert on our planet is one of my most essential priorities and SMARTEX did not disappoint me in this regard.

During my six-month internship experience, I worked with the machine learning developer team. After some initial projects which lasted for 2 months, to integrate with the team and get comfortable with their developer environment, I was tasked with creating classifiers and generating labels for one of the possible textile defects, a *vertical* defect normally caused by lack of fabric in a line due to a needle breaking in the knitting machine. This project required computer vision and machine learning techniques which certainly allowed me to develop my skills in these areas, as well as giving me experience with a professional developer environment and its various technologies.

Concerning the textile industry and SMARTEX's involvement with it, SMARTEX develops cameras that are usually placed in a row and mounted at the end of a knitting machine. These cameras continuously take photos at specified intervals, from two different angles, and send them via the internal WiFi offered by SMARTEX to the factory's main processor, equipped with a production model tasked with predicting various textile defects. In case of continuing defects and depending on each factory's parameters, the SMARTEX system can automatically stop a currently faulty machine, reducing the production of defective material. One of the main advantages of this system is its up-time. While a person responsible for quality control only checks a sample of around 10% of the produced material in any given day, the SMARTEX system has the ability to quality control nearly 100% of the output of a machine.

## 1.1 Outline

Chapter 2 describes the first on-boarding project, the implementation of an image loader that resorts to caching via an SQL database.

Chapter 3 reports the final on-boarding project, the building of a pruned convolutional neural network based on an internal production model. It also presents a theoretical foundation for this type of neural networks.

Chapter 4 gives a detailed description of the main project of this thesis, segmentation label generation of *vertical* defects in textiles.

Section 4.1 presents our simple defect classification model, responsible for creating initial labels used in fitting the next models.

Section 4.2 describes the implementation of a graphic application used in producing labels by hand, to assist in fitting future models.

Section 4.3 details the clustering and the dimensionality reduction process used to improve the training of the ensemble machine learning models.

Section 4.4 reports the random forest model and its results. It also gives an overview of another ensemble classifier, the distributed gradient boosting classifier.

Section 4.5 describes in-depth the final machine learning project, the U-Net classifier.

Chapter 5 offers some final remarks about the internship experience and the projects that we worked on, as well as some possible future extensions of the work in this thesis.

# Chapter 2

# Limited Caching Loader

As an initial on-boarding project, I was tasked with implementing an image loader responsible for download, storage, and eventual use in the process of training and testing any internal classification model. This project lasted for, approximately, my first month of internship. It was built on top of the company's base loaders using Python's class hierarchy. This loader was internally called limited caching loader. As the name suggests, the loader would store images in cache up to a certain set file size. This cache file is an SQLite [2] database and would be available in the shared file system for SMARTEX's developers.

SQLite is a C-language library that implements an SQL database engine, and is supported in Python with the sqlite3 module. SQLite as a project started on the 29th of May in the year 2000 [3] and it is, currently, one of the most deployed database engines in the world, as well as one of the most deployed softwares [4]. Even though it does not directly support multi-threading since SQLite only supports one write transaction at a time (but multiple read transactions at the same time), these write transactions are fast enough (in the millisecond range) so that this library can handle some writer concurrency. However, the implemented loader did not perform as well as intended in its multi-threading capabilities when more advanced testing was done by my external supervisor.

Saving images in cache is a good approach for the internal image loaders due to its potential to improve the download speed of images and reduce the redundant monetary cost of downloading from the cloud recently downloaded images, by the same user or by another developer. The cache database and the accompanying loader are accessible through the internal server by various developers which means a caching loader can be very advantageous in this context by reducing these redundant costs.

## 2.1 Architecture

To ensure the database (DB) does not surpass the stipulated maximum size, before introducing an image in the DB, checks are performed and the oldest entries are deleted until there is enough space. However, if the oldest image is not older than a predefined time (set by the class attribute *threshold* when initializing the loader class, as exhibited in algorithms 2.1 and 2.2), the new images downloaded from cloud storage are simply not introduced in cache, as evidenced by lines 20–26 of algorithm 2.3. Due to an internal decision, no warnings are emitted when an image fails to be inserted into the cache. Since images can always be downloaded from the cloud and, usually, no image is more important than any other, we did not find it necessary to warn the user in such cases.

**Algorithm 2.1** Limited caching loader class

```
class LIMITED CACHING LOADER
    source : CLASS                    ▷ Source loader
    cache : STRING                    ▷ Cache location
    limit : INTEGER                   ▷ Cache size limit in GB
    threshold : INTEGER               ▷ Threshold for deleting old files in hours
    dt : INTEGER                      ▷ Threshold for updating date of last use in cache
    thread_local : CLASS              ▷ Local thread for read/writes to DB
end class
```

To introduce some specificities about SQL databases, transactions is the name given to the read and write processes an user establishes with the database. A read transaction is started by a SELECT statement, while a write transaction starts with SQL statements like CREATE, DROP, INSERT, or UPDATE. The main commands for a transaction are *commit* and *rollback*. As the names suggest, these commands are used to commit changes to the database or rollback any changes made since the transaction was open. Because no changes are made to the database until the transaction is committed, these processes are very useful when using an SQL database engine [5].

To clarify, and specifically in Python's implementation of SQLite and with the settings used in this project, if any error occurs during a transaction, the transaction is automatically rolled back and any previous change is reverted, reducing the chances for faulty or incomplete write processes to the database. In the case there are no errors, we manually commit the transaction via the loader's connection object to the SQLite database, as in line 14 of algorithm 2.2.

**Algorithm 2.2** Limited caching loader method

```
1:  procedure INITIALIZE(source, cache, limit, threshold)
2:      self.source ← source
3:      self.cache ← cache as path object
4:      self.limit ← limit * 1024^3 as integer                ▷ Conversion to bytes
5:      self.threshold ← threshold * 3600 as integer          ▷ Conversion to seconds
6:      self.dt ← self.threshold/20 as integer
7:      connect to self.cache via sqlite3 and store connection in con
8:      call connection cursor and store in cur
9:      if database is empty then
10:         create tables for images and frame information linked via blob_id
11:         create index on frame information table for date_last_use
12:         create table for database metadata
13:         initialize metadata table with a sample record
14:         commit transaction
15:     end if
16:     execute select statement for size_limit from metadata table
17:     fetch data to size_limit
18:     self.limit ← max(size_limit, self.limit)
19:     update size_limit in metadata table with self.limit
20:     commit transaction
21:     close cursor and connection
22:     self.thread_local ← THREADING.LOCAL()
23: end procedure
```

One specific process of this loader is, when downloading files from cache, to update the image's date of last use only if a predefined set time has passed since its last use. This time is set by the class attribute *dt*, which was internally decided to be a function of the class *threshold* attribute. This process is displayed in lines 7–15 of algorithm 2.3 and it saves unnecessary executions of update transactions to the SQL database for recent images in cache that would most certainly not be considered for deletion right away. The date of last use of an image is also important in the deletion process given by algorithm 2.4 to sort the images from the oldest to the most recent.

Aside from the INITIALIZE method of algorithm 2.2 and the methods to get and delete images (algorithms 2.3 and 2.4, respectively), this loader class has one other auxiliary method, algorithm A.1's _GETSIZE in appendix A, created to return the current size of the database.

With respect to the architecture of the SQL database, there is a table for frame information called *frames* with the fields *key* (for the unique text identifier of a frame), *date_last_use*, *image_size*, and *blob_id*, which links this table, in a one-to-one relationship, with the table

**Algorithm 2.3** Limited caching loader method

```
1: procedure _GET(key)
2:     if self.thread_local does not have connection attribute then
3:         connect to self.cache via sqlite3 and store connection in self.thread_local.con
4:         call connection cursor and store in self.thread_local.cur
5:     end if
6:     try
7:         execute select statement for image and date_last_use for key = key
8:         fetch data to record
9:         output, last_use ← record
10:        store current time in now
11:        delta ← now − last_use
12:        if delta.seconds > self.dt then
13:            update data_last_use with now for key = key
14:            commit transaction
15:        end if
16:        return output as BytesIO                    ▷ Python class for bytes objects
17:    catch type error                ▷ Get image from source DB if not stored in cache
18:        output ← self.source[key]
19:        store size of output in size
20:        while self._GETSIZE()+size > self.limit do
21:            try
22:                self._DELETE()        ▷ Delete images from cache until space is available
23:            catch type error                ▷ In case no image can be deleted from DB
24:                return output
25:            end try
26:        end while
27:        try
28:            updated_size ← self._GETSIZE()+size
29:            update size in metadata table with updated_size
30:            store current time in now
31:            store output as binary in frame          ▷ BytesIO is not supported in sqlite3
32:            insert frame into images table
33:            insert key, now, and size into frame information table
34:            commit transaction
35:            return output
36:        catch sqlite3 integrity error            ▷ In case of error inserting image into DB
37:            return output
38:        end try
39:    end try
40: end procedure
```

**Algorithm 2.4** Limited caching loader method

1: **procedure** _DELETE
2: **execute** select statement of 1 record for key and image_size in table for frame information ordered by date_last_use where date_last_use is less than the current time subtracted by $self.threshold$
3: **fetch** data to *record*
4: $key, image\_size \leftarrow record$
5: **execute** delete statement in table for frame information for key $= key$
6: $updated\_size \leftarrow$ SELF._GETSIZE() $- size$
7: **update** size in table metadata with $updated\_size$
8: **commit** transaction
9: **end procedure**

called *blobs*, responsible for storing the actual image objects in its field *image*. Additionally, there is a table called *metadata*, designed to hold only one entry at a time, with two relevant fields: *size_limit*, which stores the current maximum limit for the database; and *size*, which stores the current size of the database.

Examining this section's algorithms, it would seem this loader is construed fairly simply and does not have the sufficient complexity a technology company should get from a loader possibly used by various developers and in various projects, each with possibly different needs from it. Yet, this represents a situation where Python's class hierarchy is absolutely convenient. Put simply, one class may be created with a dependence on another class, inheriting every method and attribute of the base class. In this case, our caching loader class inherits the company's internal *BaseLoader* and its useful existing methods, such as the self-explanatory methods __CONTAINS__ and COPY.

One other intricacy vis-à-vis Python classes displayed in the algorithms presented in this thesis is the way a class references its attributes and methods in its internal methods. Python uses the *self* prefix, as evidenced, for example, in line 2 of algorithm 2.2. This is especially useful for the ability to dynamically change a class' attribute or call one of its methods within another internal method.

## 2.2 Performance

This loader was first evaluated with an advanced test implemented by my external adviser, the Machine Learning (ML) team leader, which focused on the loader's multithreading capabilities and performance, i.e. how this loader fared in terms of speed when handling multiple concurrent reads and writes. In addition, we also created a simple

speed test, plotted in figure 2.1. This test aimed to compare the speed of downloading every image in the testing set in a loader with a full cache vs. a loader that did not have a database beforehand, while varying the percentage of the cache file size limit to the total testing set size, to ascertain the performance improvements attributed by the caching capability of the loader.

The testing set was comprised of some tens of random images totalling 98.4 MB in size. The percentage of the cache size limit to the total test set size ranges from around 10% to around 102% with approximately 5% steps. Figure 2.1 portrays the significant performance improvement that caching provides. Plotted in orange are the tests run for a loader that used a database filled randomly to the limit, and plotted in blue are the test results for a loader that did not have a database beforehand.

This figure clearly demonstrates the improvement that caching can create. While the left-most data points of the figure closely estimate the download time of a non-caching loader (save for the overhead in trying to delete images to get space to insert into cache and failing) and are thus similar for the two loaders, the right-most data points show a significant 10 second difference between the two loaders. For the loader plotted in blue, its speed plateaus at around 10 seconds, while the download speed of the loader plotted in orange consistently decreases till the millisecond range per MB.

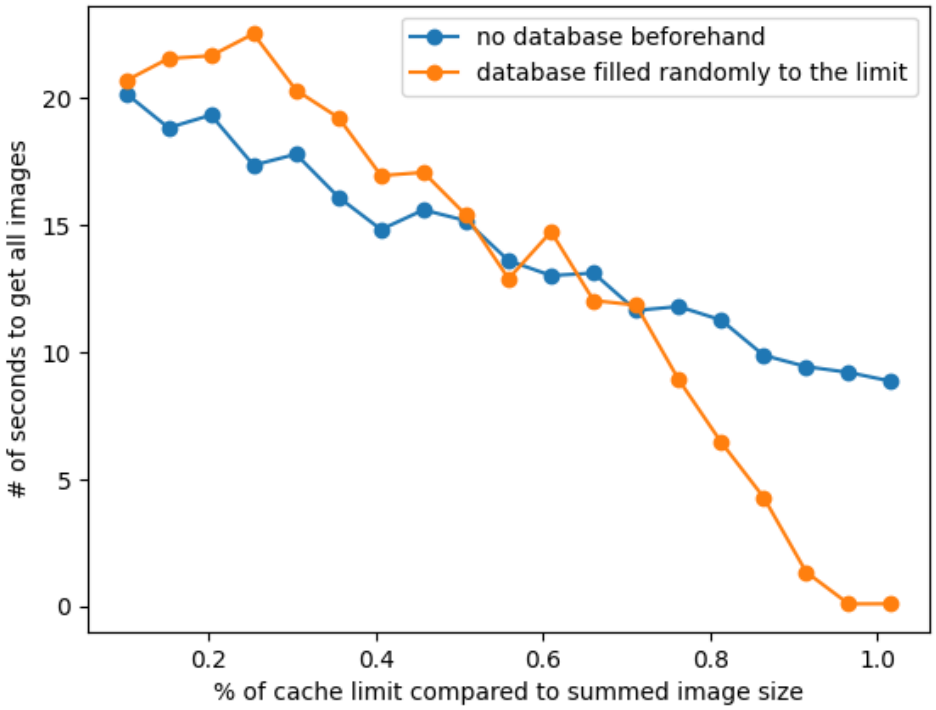



FIGURE 2.1: Download speed comparison test.

A caching loader integrated into the SMARTEX environment would use a database size limit in the tens, or maybe hundreds, of GB, that could swiftly be populated by the various machine learning training sessions run by the multiple developers. This simple test implied that a limited caching loader could be an efficient alternative to other internal image loaders in terms of speed, if it could handle the read/write concurrencies expected of a multiple-developer team.

Alas, SQLite's lack of multi-threading support ultimately revoked the loader's apparent speed improvements. The limited caching loader under-performed in the more advanced multi-threading tests, but future alternatives using different SQL database engines that can handle concurrency could be created, employing the research and development presented in this chapter.

# Chapter 3

# Convolutional Neural Networks

This chapter is dedicated to providing an essential understanding of convolutional neural networks (CNNs), which is critical for our introduction of the MobileNetV2 architecture, our second on-boarding project, our dimensionality reduction pipeline of section 4.3, and the final segmentation U-Net model described in section 4.5. A preliminary knowledge on neural networks is assumed, such that concepts which do not pertain solely to CNNs, such as dense layers or backpropagation, will not be comprehensively explained. The second on-boarding project of the internship is also described in-depth in section 3.3 of this chapter. Every CNN created for this internship was implemented with the use of two popular neural network libraries: TensorFlow [6] and Keras [7], both available in Python.

## 3.1 Fundamentals

A convolutional neural network is a class of neural networks originally introduced by K. Fukushima in 1980 with the name of ”neocognitron” with the initial purpose of pattern recognition [8]. In this paper, Fukushima introduced the most important component of a CNN, the convolution layer, and the pooling (or down-sampling) layer as well. The CNN has since been repurposed and improved on for many applications such as natural language processing, financial time-series, image classification and segmentation, or any other application using data with a grid-like topology. Fukushima's novel model architecture was inspired by work from the 1960s regarding visual cortices of cats and how some neurons individually responded to small regions of a visual field.

Likewise, a CNN is able to explore the local structure of data. This is an appealing feature in the context of image problems, since sections of an image which are close are more likely to vary together than distant sections. This network architecture achieves this by substituting the usual matrix multiplication of neural network layers for the convolution operation in at least one of its layers. The resulting so-called convolution layer (further explained in subsection 3.1.1) restricts its neurons to receive input only from neighboring neurons of the previous layers, while reducing the number of trainable parameters since not all neurons are interconnected.

Each convolution layer also shares the same limited number of weights amongst its neurons, further reducing the number of parameters, and preventing over-fitting, and both the vanishing and exploding gradient problems during the backpropagation process. This parameter sharing between neurons in a layer also allows each filter to locate its specific feature anywhere in the input map [9]. These weight limitations lower memory requirements for training and predicting with a network as well, allowing for the training of more complex and powerful networks. In the following subsections we describe the most used layers in this type of neural networks, which are critical to comprehend CNN models presented throughout the rest of this thesis.

### 3.1.1 Convolution Layer

In a CNN, the convolution layer takes a tensor (i.e. multidimensional array) of shape $(batch\ size, height, width, \#\ of\ channels)$ and outputs an abstraction of the input, called a feature map or an activation map. This output is obtained by convolving the input with the layer's filters (also called convolutional kernels or feature detectors). Formally, the convolution operation combines two functions, where one is generally called the kernel or filter, into one function $s(t)$ which can be defined for discrete variables as

$$s(t) = (x * \omega)(t) = \sum_{a} x(t-a)\omega(a), \tag{3.1}$$

where the function $x$ is typically referred to as the input and the function $\omega$ as the kernel, in the context of CNNs. Practically, we assume that these functions are zero everywhere except a finite set of points (the stored values of the input and the kernel weights) and thus, the convolution operation becomes a finite summation [9].

To clarify, the convolution layer creates a feature map by sliding a kernel, a grid of discrete numbers, along the width and height of the input and storing the inner products

between the kernel and parts of the input of the same size, as figure 3.1 clearly conveys with a low-dimensional example with one filter. In this example, a $2 \times 2$ filter convolved with a $4 \times 4$ single-channel input map resulted in a $3 \times 3$ feature map. A convolution layer always outputs maps smaller in height and width than the input map, unless padding of the input map is used. While information is then usually lost, the network trains its weights to extract only the integral features of an input map and to exclude the information not relevant to the problem at hand. However, the number of channels of the output map can be larger since it is equal to the number of applied filters in a layer.

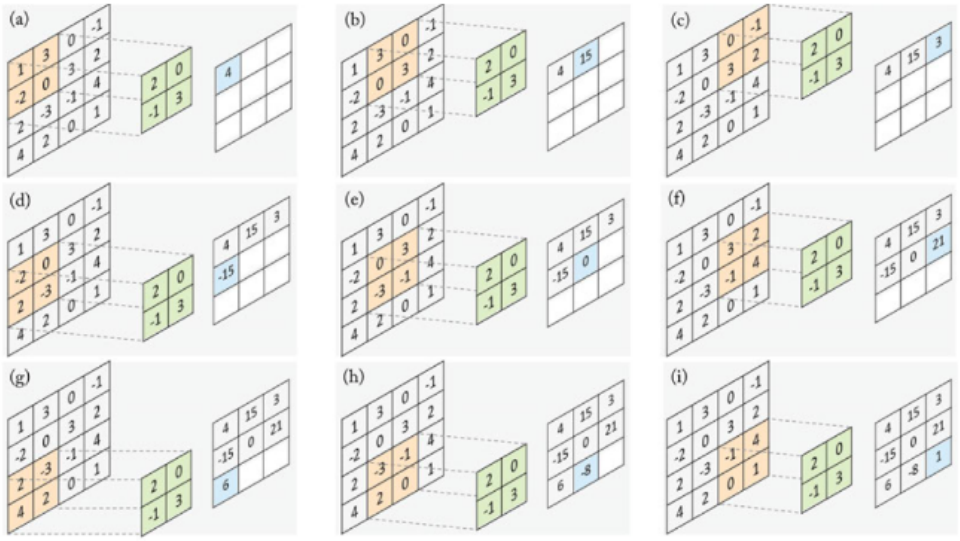



FIGURE 3.1: Operation of a convolutional layer (adapted from [9]).

Given an input with multiple channels, either the same kernel or different kernels for each channel can be applied to each "depth slice" and the sum of the inner products between the kernel(s) and the input map of each depth slice is stored instead. This ensures each layer filter outputs a feature map with only one channel. Each filter may also have a bias parameter [9]. Thus, the maximum number of weights (or trainable parameters) for a single filter of size $h \times w$ of a convolution layer is $c \times (h \times w) + 1$, where $c$ is the number of channels of the input map.

Lastly, in the example of figure 3.1, the kernel takes a step of 1 when sliding along the width and height of the input map. This step is referred to as the stride of the convolutional filter and is a non-trainable parameter of the network. Additionally, with the goal of preserving spatial sizes and building deeper networks, the padding (or zero-padding) parameter can be applied to input maps by padding zeros (or other values) around the original map, as the name suggests [9]. These two parameters allow us to represent the

dimensions of a feature map resulting from the convolution of an input map of size $h \times w$ with a kernel of size $k \times k$, with stride $s$ and symmetric padding $p$ (the increase in the input map in each direction) as such:

$$h' = \lfloor \frac{h - k + s + p}{s} \rfloor, \quad w' = \lfloor \frac{w - k + s + p}{s} \rfloor. \tag{3.2}$$

### 3.1.2 Pooling Layer

One other frequently used layer in a CNN which also possesses the ability to down-sample an input map is the pooling layer. This layer can be much less complex than a convolution layer, as it switches out the convolution process with typically simple activation functions, such as the average or max functions, also applied to different regions of the input map in succession. The pooling layer, therefore, has no trainable parameters. Albeit less complex, these layers are important for easily down-sampling an input map, while still obtaining a compact feature representation invariant to moderate changes in object scale, pose, and translation in a map [9].

Although the pooling layer down-samples an input map, it does not decrease nor increase its number of channels, instead applying the same activation function and creating one feature map for each channel. In these layers, instead of the parameters for the filter sizes, there are parameters for the size of the pooled region by the activation functions. Similarly to the convolution layers however, if we denote the size of an arbitrary square pooled region as $k \times k$, the dimensions of the feature map resulting from the pooling of an input map of size $h \times w$ with stride $s$ are given by

$$h' = \lfloor \frac{h - k + s}{s} \rfloor, \quad w' = \lfloor \frac{w - k + s}{s} \rfloor. \tag{3.3}$$

### 3.1.3 Batch Normalization Layer

As a stepping stone to introducing the batch normalization layer, we should first summarize the batch, or mini-batch, concept commonly used with neural networks. In the backpropagation (i.e. gradient descent) process, each input data $\mathbf{x}_i$ produces a gradient direction $\nabla \mathcal{L}(\mathbf{x}_i)$, where $\mathcal{L}$ is the network's cost function.

Although it would be more accurate to continuously update the trainable parameters of a model with each gradient direction given by an input data, this practice can become time-prohibitive. Thus, as a trade-off between accuracy and computation time, the average gradient directions of subsequent batches of $N$ randomly chosen inputs from the

training set, $\mathcal{B} = \{\mathbf{x}_i, \ldots, \mathbf{x}_N\}$, can be computed instead, for each update of a model's weights and biases. The average gradient direction is then defined as

$$\widetilde{\nabla\mathcal{L}}(\mathcal{B}) = \frac{1}{N}\sum_{i=1}^{N}\nabla\mathcal{L}(\mathbf{x}_i). \tag{3.4}$$

Furthermore, if the input data are independent random variables, by the Central Limit Theorem, $\widetilde{\nabla\mathcal{L}}(\mathcal{B})$ has the same mean as $\nabla\mathcal{L}(\mathbf{x}_i)$ and an $N$ times smaller variance [9].

Given this, a batch normalization layer is a popular regularization technique which prevents the exploding gradient problem. Similarly to the practice of scaling input data to ensure a stable start to the training of a network by preventing large early fluctuations of weight values (which could immediately result in exploding gradients), the batch normalization layer normalizes the output of its preceding layer across the current batch. This layer has the ability to stabilize a network's weights and significantly reduce the covariate shift problem, i.e. a large growth of weights during training which can lead to exploding gradients [9].

This layer accomplishes its goal by normalizing each channel of an input map with the means and standard deviations of each channel of the corresponding input batch. The mean and variance of a mini-batch $\mathcal{B}$ of size N are given by

$$\mu_{\mathcal{B}} = \frac{1}{N}\sum_{i=1}^{N}\mathbf{x}_i, \quad \sigma^2_{\mathcal{B}} = \frac{1}{N}\sum_{i=1}^{N}(\mathbf{x}_i - \mu_{\mathcal{B}})^2, \tag{3.5}$$

and the normalized input and final output of a data point of the batch $\mathcal{B}$ are defined as

$$\widehat{\mathbf{x}}_i = \frac{\mathbf{x}_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}}, \quad \mathbf{y}_i = \gamma\widehat{\mathbf{x}}_i + \beta, \tag{3.6}$$

where $\epsilon$ is a small constant used to avoid dividing by zero (defaulted to 0.001 in Keras' implementation of the layer), $\gamma$ is the scale parameter, and $\beta$ is the shift parameter. The scale and shift parameters are trainable parameters for each channel of the previous layer.

Additionally, the layer stores the moving averages of the mean and standard deviation for each channel to be able to normalize inputs during inference. The weight of the previous value when calculating the moving averages at each step can also be set by the momentum parameter (available in Keras). Thus, a batch normalization layer contains two trainable parameters and two non-trainable parameters (moving averages) for each channel in the preceding layer [9].

### 3.1.4 Dropout Layer

As another commonly used network regularization technique, a dropout layer is able to prevent overfitting simply by setting to zero the output of a random set of units from the previous layer during training. This mechanism manages to reduce overfitting by preventing a network from becoming overdependent on arbitrary units or groups of units, thus better spreading information across the network and improving generalization [9].

During prediction, this layer does not drop any units and the full network is used. In addition, the dropout layer does not use any trainable or non-trainable parameters [9]. In Keras' implementation of this layer, a rate parameter dictates the proportion of (randomly selected) units from the preceding layer to drop.

## 3.2 MobileNetV2

The MobileNetV2 (MNV2) is a convolutional neural network specifically designed for mobile models and resource constrained environments, like SMARTEX's systems in factories. It is an effective feature extractor for image detection and segmentation, ready for immediate use, since an MNV2 model pretrained with more than a million images from the ImageNet database [10] can be loaded directly from the TensorFlow/Keras libraries. In the context of SMARTEX's production models and CNN models developed during this internship, an MNV2 model was usually used at the beginning of their architecture, for its immediate and fast ability as a feature extractor.

This architecture was first introduced by M. Sndler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen in 2018 [11]. Its goals were to improve on their previously introduced architecture, the MobileNetV1, and to strike a better balance between accuracy and performance, a crucial task for mobile and embedded applications. The authors accomplished this by introducing a novel layer: the inverted residual with linear bottleneck, which ensures that the number of parameters and mathematical operations stay as low as possible.

To better understand this novel layer, we should first introduce the concept of residual blocks. These are convolution blocks that connect layers which have other layers in between them. This connection, typically a simple add operation, allows a network to access earlier activations that haven't been modified by the convolutional block. The original residual block typically follows a "wide-narrow-wide" approach in regard to the number of channels of its layers, evidenced by the example representation of this block of

figure 3.2. The residual block has been essential for CNN architectures of great depth, for its ability to better propagate gradients across multiplier layers [11].

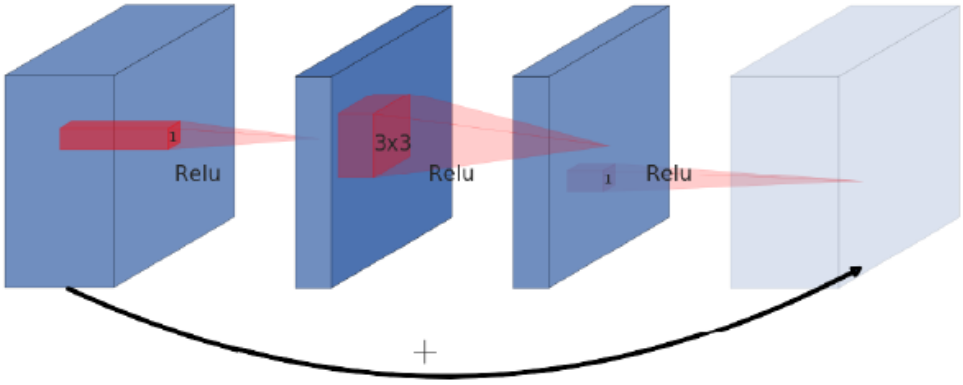



FIGURE 3.2: Residual block example (adapted from [11]).

With the same goal, the authors introduced the inverse residual block, which empirically works better and is significantly more memory efficient than the original residual block in the context of the MNV2 model. Unlike the original block, the inverted residual block follows a "narrow-wide-narrow" approach with $ReLU6$ activations and a connection between the low-dimensional feature maps (if $stride = 1$). Besides the activation of the final classification layer, the $ReLU6$ function is always used as the activation function in the MNV2 architecture, instead of other $ReLU$ functions or different activations, for its robustness when used with low-precision computation [11], and is defined by

$$ReLU6(x) = \min(\max(0, x), 6). \tag{3.7}$$

The inverted residual block contains a depthwise separable convolution layer (an example is depicted in figure 3.3). This approach considerably reduces the number of parameters of the block, comparatively to the original residual block. In fact, by using $3 \times 3$ depthwise separable convolutions, the MNV2 model achieves computational costs 8 to 9 times smaller than that of standard convolutions with only a small decrease in accuracy.

To clarify, a depthwise separable convolution layer is a key building block for efficient CNN architectures which replaces a full convolution operation with two separate layers. The first layer, the depthwise convolution, performs lightweight filtering by applying only one filter for each input channel. The second layer, the pointwise convolution, is a simple $1 \times 1$ convolution, which builds new features by computing linear combinations of the input channels. For an $h_i \times w_i \times c_i$ input tensor resulting in an $h_i \times w_i \times c_j$ output tensor, a standard convolution with a filter of shape $k \times k \times c_i \times c_j$ has a computation cost

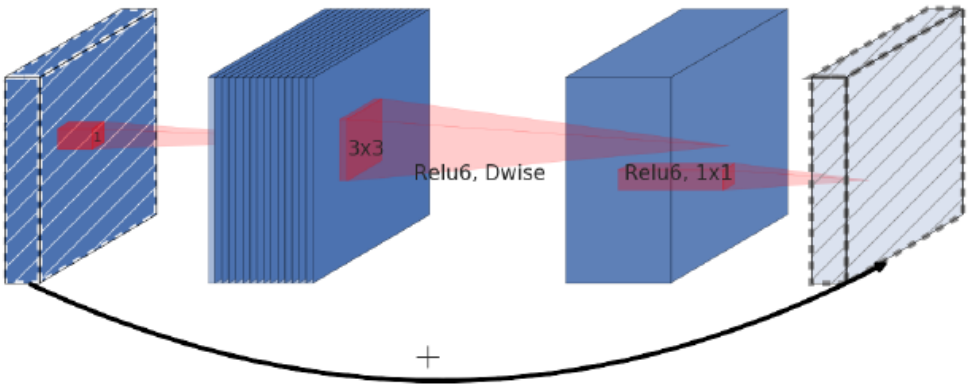



FIGURE 3.3: Inverted residual block example (adapted from [11]).

of $h_i \cdot w_i \cdot c_i \cdot c_j \cdot k^2$, while a depthwise separable convolution costs $h_i \cdot w_i \cdot c_i(k^2 + c_j)$. This constitutes a decrease in computation cost by a factor of $k \cdot c_j / (k^2 + c_j)$ [11].

Finally, MobileNetV2 introduces the use of linear bottlenecks in its inverted residual blocks. Non-linear activation functions like *ReLU*, commonly used in neural networks, can lead to loss of information (for *ReLU*, for example, values less than 0 are discarded), and ultimately, worse performance by a network. This can be rectified by increasing the number of channels at various stages of the network. However, the MNV2 model attempts to fix this problem by discarding the activation function of the last convolution of an inverted residual block, therefore obtaining a linear output before it is added to the initial activation. Empirically, the linear bottlenecks significantly benefit the performance of the network, compared with the use of non-linear layers [11].

An inverted residual with linear bottleneck block transforming a map from $k$ to $k'$ channels, with stride $s$ and expansion factor $t$, can then be described by figure 3.4. Additionally, MNV2 implements dropout and batch normalization layers in its architecture as regularization techniques during training. The complete MobileNetV2 architecture, described in figure 3.5, consists of an initial $3 \times 3$ convolution layer with 32 filters followed by 17 inverted residual with linear bottleneck blocks, a regular $1 \times 1$ convolution layer, a global average pooling layer, and a final $1 \times 1$ convolution layer with $k$ filters for the classification of $k$ classes. The Keras implementation of this network replaces the final $1 \times 1$ convolution layer with a dense layer with $k$ units, as their behaviour is practically equal. Each line in figure 3.5 represents a sequence of identical layers (with the exception of the stride $s$) repeated $n$ times, with an expansion factor $t$ applied to the input. Every layer in a sequence has the same number $c$ of output channels and the first layer uses stride $s$ while all others use stride 1.

| Input | Operator | Output |
|---|---|---|
| $h \times w \times k$ | 1x1 conv2d , ReLU6 | $h \times w \times (tk)$ |
| $h \times w \times tk$ | 3x3 dwise s=$s$, ReLU6 | $\frac{h}{s} \times \frac{w}{s} \times (tk)$ |
| $\frac{h}{s} \times \frac{w}{s} \times tk$ | linear 1x1 conv2d | $\frac{h}{s} \times \frac{w}{s} \times k'$ |

FIGURE 3.4: Architecture of the bottleneck residual block (adapted from [11]).

| Input | Operator | $t$ | $c$ | $n$ | $s$ |
|---|---|---|---|---|---|
| $224^2 \times 3$ | conv2d | - | 32 | 1 | 2 |
| $112^2 \times 32$ | bottleneck | 1 | 16 | 1 | 1 |
| $112^2 \times 16$ | bottleneck | 6 | 24 | 2 | 2 |
| $56^2 \times 24$ | bottleneck | 6 | 32 | 3 | 2 |
| $28^2 \times 32$ | bottleneck | 6 | 64 | 4 | 2 |
| $14^2 \times 64$ | bottleneck | 6 | 96 | 3 | 1 |
| $14^2 \times 96$ | bottleneck | 6 | 160 | 3 | 2 |
| $7^2 \times 160$ | bottleneck | 6 | 320 | 1 | 1 |
| $7^2 \times 320$ | conv2d 1x1 | - | 1280 | 1 | 1 |
| $7^2 \times 1280$ | avgpool 7x7 | - | - | 1 | - |
| $1 \times 1 \times 1280$ | conv2d 1x1 | - | k | - | |

FIGURE 3.5: Architecture of the MobileNetV2 model (adapted from [11]).

## 3.3 Pruned Network Model

My final on-boarding project involved machine learning, with the aim of integrating me into the developer environment and the company methodologies. More specifically, I was tasked with studying and implementing pruning capabilities in an existing internal production model. This version of the model would hopefully achieve a better inference speed (very useful in SMARTEX's production context) because of the significant simplification of the mathematical operations that occur in an inference, brought about by the pruning of the network. Additionally, a pruning network is easier to compress which is a good feature to have for networks deployed to mobile devices, such as SMARTEX does.

Specifically, pruning a CNN consists of gradually replacing model weights with zeros until the desired sparsity is reached at the model's final fitting epoch (or at a custom epoch given by a pruning parameter). As a regularization technique, pruning can improve generalization and prevent over-fitting, while empirically losing only a small amount of accuracy in the case of the Keras' pruning package we used (available through TensorFlow's model optimization library) [12].

Regading Keras' implementation of MobileNetV2, used in our pruned CNN model, an additional $\alpha$ parameter is present which represents a width multiplier, to allow for a reduction or increase in total number of model parameters as needed. For example, for an $\alpha = 0.35$, which we use in all of our MNV2 models, the number of output channels of the first convolution layer is 16 instead of 32, further changing the dimensionality of the outputs of every subsequent layer, except for the penultimate convolution layer if $\alpha \leq 1$. If $\alpha > 1$, the number of extracted features before the classification layer increases (e.g. for an $\alpha = 1.4$, 1792 features are extracted per input map). In addition, the final fully

connected layer can be excluded via the *include_top* argument of Keras' MNV2 model so as to use it directly as a feature extractor.

### 3.3.1 Model Architecture

The updated model is a multi-label binary classifier based on an internal CNN model which uses the max function in its pooling layer. The first section of the model is constituted by a MobileNetV2 model with width multiplier $\alpha = 0.35$ and default pretrained weights. The model handles a normal input of a 3-dimensional tensor for an image with 3 colour channels. Note that the dimension relating to the batch does not need to be specified in the input definition.

This MNV2 model, which extracts 1280 features for each input point, is then pruned with the use of TensorFlow's model optimization library. The construction of this base model is presented in algorithm 3.1 in Python instead of pseudocode, as the rest of the algorithms in this section and corresponding appendix, for the comprehensive use of TensorFlow and Keras methods.

**Algorithm 3.1** MobileNetV2 base model

```
mnv2 = MobileNetV2(alpha=0.35, pooling="max", include_top=False)
x = Input((None, None, 3))
base_model = Model(x, mnv2(x, training=False), model_name="mnv2")

import tensorflow_model_optimization as tfmot
pruning_params = {
    "pruning_schedule": tfmot.sparsity.keras.PolynomialDecay(**kwargs)
}
prune_low_magnitude = tfmot.sparsity.keras.prune_low_magnitude
base_model = prune_low_magnitude(base_model, **pruning_params)
```

Afterwards, an input of shape $(2, None, None, 3)$ is created and scaled between -1 and 1. The *None* values reference the variable values for the height and width of an image, and the first dimension corresponds to the FRONT and TOP angles taken by the SMARTEX cameras. Then, a custom layer named distributed layer handles integrating this unconventional tensor shape into the MNV2 model. Thus, the output of this layer has shape $(b, 2, 1280)$, where $b$ represents the batch size.

This output is flattened to obtain a shape of $(b, 2560)$ and passed through a pruning-activated dense layer with the *sigmoid* activation function (equation 3.8) which outputs a tensor of shape $(b, k)$, where $k$ stands for the number of different labels. In this case,

$k = 8$, for the 8 distinct fabric defects the SMARTEX production models search for. The algorithm for this final section of the model, the one used for deploy after training, is available in algorithm B.1 of appendix B. This model expects inputs between 0 and 1 to properly scale them between -1 and 1 (line 2 of the algorithm).

The *sigmoid* activation function, or $S(x)$, is commonly used with neural networks to introduce non-linearity in the models (thus improving model generalization), and due to its usefulness in backpropagation calculations because of its simple derivative $S'(x) = S(x)(1 - S(x))$. The *sigmoid* function, which returns values between 0 and 1, is given by

$$S(x) = \frac{1}{1 + e^{-x}}. \tag{3.8}$$

Lastly, the training model, which expects two 3-channel images for each input point with values in the interval 0–255, is set up by creating a tensor of shape $(2, None, None)$. Feeding the model with the 2 angles (first dimension of this tensor) for each data point improves the generalization capability of the model. This tensor is expanded one dimension at the end (line 3 of algorithm B.2 of appendix B) and its last dimension is repeated 3 times (line 4) to account for 3 colour channels and a tensor of shape $(b, 2, None, None, 3)$. This tensor is then cast to the data type *float32* (floating-point format which occupies 32 bits in memory) and scaled between 0 and 1 (line 5). The final training model is obtained by passing this input tensor through the deploy model.

The model was compiled with the *Adam* optimizer and the Mathews Correlation Coefficient as its metric, as well as the binary cross entropy as its loss function. The used optimizer was published in 2015 by D. Kingma and J. Ba in 2015 [13]. It is a stochastic gradient descent method well-suited for various non-convex optimization problems in machine learning, and especially for tasks with large datasets and/or high-dimensional parameter spaces. The algorithm is also computationally efficient with little memory requirements and has been shown effective for deep CNNs [13].

The model was trained, with validation and batch size of 40, for 10 epochs with a learning rate of $5 \times 10^{-3}$. It was set to fine-tune for 100 epochs with a learning rate of $5 \times 10^{-5}$, before it auto-stopped at epoch 60. To clarify, the first 10 epochs train the model as a general feature extractor, without updating the pretrained weights of the MNV2 model. The fine-tuning process instructs the model to learn more specific features, using a smaller learning rate and updating all the weights of the model. The dataset consisted of 25584

samples (70%/15%/15% split) and was augmented by usual methods: random brightness, Gaussian noise, salt and pepper noise, random quality, random flip, random perspective, and random warp. The fitting of this model took around one day and a half in a dedicated SMARTEX machine.

Pruning was set to start at epoch 30 and end at epoch 110 with an initial sparsity of 0.01 and a final sparsity of 0.75. These hyper-parameters were set by my external adviser, based on the number of epochs the company usually trained their models and their previous research on the pruning process. The pruning of the whole network was not possible since some layers of the network were not supported by Keras' pruning package. Thus, pruning was only applied to the MobileNetV2 base model and the final dense layer.

### 3.3.2 Metrics and Results

Model inference speed is a very important property for a good production model tasked with predicting defects in real-time and stopping machines to reduce textile waste. Compared with the current production model (figure 3.6), the pruned model (figure 3.7) performed worse by an average of 9ms, albeit with a lower standard deviation, in a proprietary test of inference speed. This unexpectedly worse performance may be attributed to the pruned model not being quantized and converted to a compressed .tflite model as advised by the official TensorFlow pruning guide [14] (quantization and conversion seem to be bugged in TensorFlow 2.4). The worse performance may also be attributed to training the pruned model in the *float32* data type. Unfortunately, it was not possible to train it in the (mixed) *float16* (floating-point format which occupies 16 bits in memory) data type as usual within SMARTEX, due to incompatibilities with Keras' pruning package.

```
116 ms ± 275 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

FIGURE 3.6: Inference times for the production model.

```
125 ms ± 238 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

FIGURE 3.7: Inference times for the pruned model.

Another factor which may have influenced the worse inference speed performance was the auto stoppage of the training at 60 epochs. As a consequence, the model was only trained for 30 epochs with active pruning and its final sparsity was far from the desired final sparsity of 75%. Unfortunately, due to time constraints, we did not have the opportunity to improve on this network after the initial fitting.

The pruned model achieved worse results than the usual production models in accurately identifying real defects, although it seems the model could be improved with more training. While a performance decrease is to be expected from the pruning of a network, this model still did not meet its goals of improving inference speed and compression ability compared with the regular production model.

While the scores for true negatives are high (figure 3.8), the production models consistently achieve higher scores. This pruned network is also worse than the production model at correctly predicting the existence of defects, especially for the *point* defect and the *oil* defects. Nonetheless, without the context of the very high performances of SMARTEX's main models, this CNN, with little tinkering of its hyper-parameters, still performed well. For any one true defect, the model did not wrongly predict it more than 10% of the times.

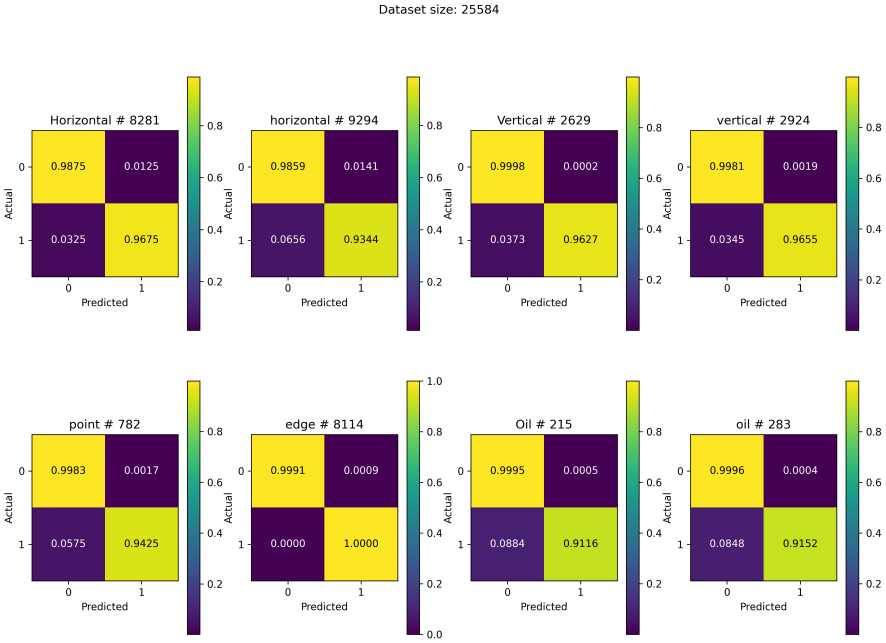



FIGURE 3.8: Confusion matrices for the pruned model.

The Mathews Correlation Coefficient (MCC) is a popular metric, especially for imbalanced classification problems, since a high score in this metric is only achievable if the model does well on the classification of both the negatives and the positives. Regarded as one of the best measures to summarize a confusion matrix [15], the metric is defined as

$$MCC = \frac{TN \times TP - FP \times FN}{\sqrt{(TN + FN)(FP + TP)(TN + FP)(FN + TP)}}, \tag{3.9}$$

where $TN$ stands for true negatives in a confusion matrix, $TP$ true positives, $FN$ false negatives, and $FP$ false positives. This metric is symmetric (switching positives and negatives leads to the same value), and ranges from -1 to 1. A value of 1 ($FP = FN = 0$) indicates perfect positive correlation and a value of 0 implies the classifier is no better than a random coin flip.

The MCC curve is a method for visualizing this metric according to the binarization threshold, to better ascertain the thresholds to choose for each class, which SMARTEX routinely employs in their model analysis. The curve pertaining to the trained pruned network of this section is displayed in figure 3.9. The threshold values where the MCC is maximum for each of the five main defects are displayed in the legend of this figure. This metric achieved very high maximum vales for each class, ranging from 0.931 for the *Oil* defect to 0.999 for the *edge* defect.

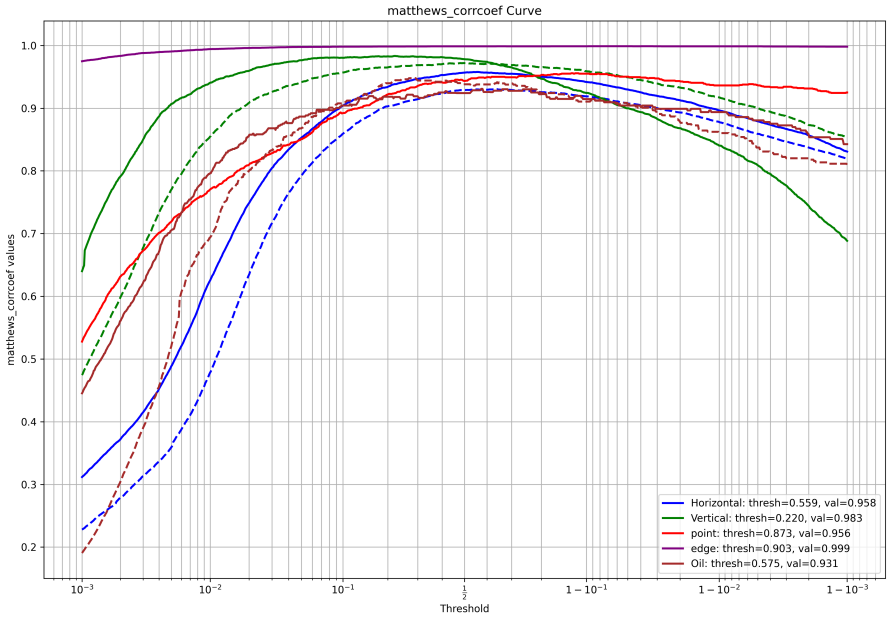



FIGURE 3.9: MCC curve for the pruned model.

The cross-entropy loss function is a commonly used loss function in ML which translates the inefficiency of using a predicted label, $\widehat{\mathbf{y}}_k$, instead of the true label $\mathbf{y}_k$, for a $k$ class classification problem [9]. The cross-entropy loss function $\mathcal{L}$ is given by

$$\mathcal{L} = -\sum_k \mathbf{y}_k \ln \widehat{\mathbf{y}}_k, \tag{3.10}$$

and the binary cross-entropy loss function used in our model is, therefore, given by

$$\mathcal{L} = -\mathbf{y} \ln \hat{\mathbf{y}} - (1 - \mathbf{y}) \ln (1 - \hat{\mathbf{y}}). \tag{3.11}$$

The model was fitted with a local logger callback for metric and loss values, the TensorBoard log callback, a model checkpoint which saves the weights of each epoch, an automatic stop callback, and a pruning summary logger. Epoch 59 was elected as the best epoch and sustained values of around 0.0232 and 0.0227 for the loss function, and values of 0.9625 and 0.9622 for the overall MCC metric, for training data and validation data, respectively.

The Receiver Operating Characteristic (ROC) curve is another widely-used classifier metric, especially for binary classification. The ROC curve plots the true positive rate ($TP/(TP + FN)$) against the false positive rate ($FP/(FP + TN)$) for different threshold values. Since we aim to maximize the true positive rate and to minimize the false positive rate, a classifier improves as its ROC curve gets "steeper" i.e. as the area under its curve grows [16]. This is another popular metric, the Area Under the Curve (AUC), which ranges from 0 to 1. Regarding the evaluation of the pruned network by this metric (figure 3.10), the area under the ROC curve of each label is very high (in the 0.986–1 range), with the *vertical* defects and the *edge* defect classification performing better than the rest.

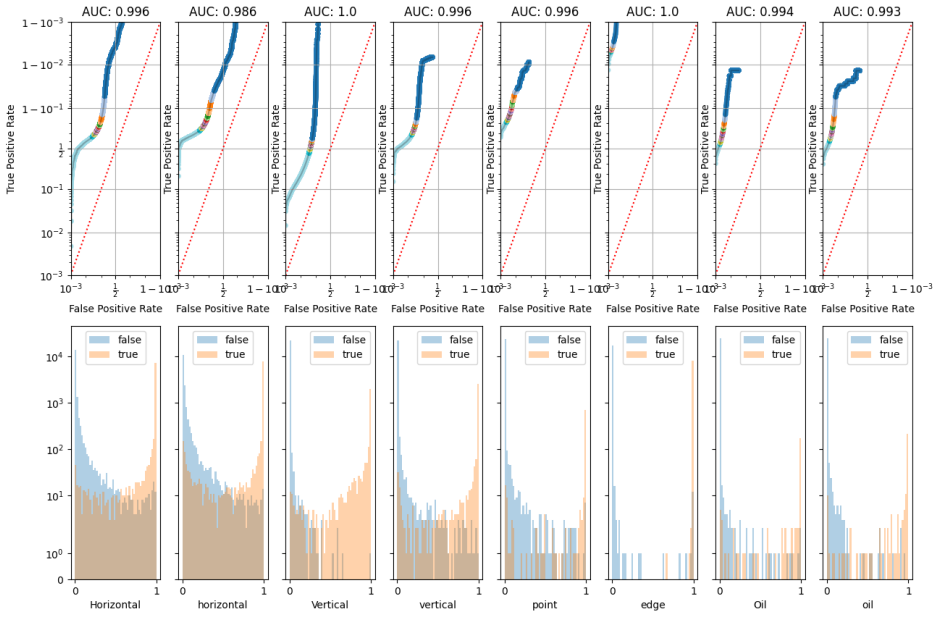



FIGURE 3.10: ROC and AUC evaluation for the pruned model.

# Chapter 4

# Needle Defect Segmentation

In this chapter, we describe in-depth the main and largest project of this internship, which spanned for approximately 3 months. This project consisted of developing increasingly complex classifiers designed to identify the areas of images of fabric (from SMARTEX cameras installed in various factories) containing a specific textile defect. This type of defect was internally named a *vertical* defect, or a *needle* defect, because it resembles a black vertical line along the fabric and is typically caused by the collapse of a sewing needle.

The purpose of this project was to obtain a classifier capable of effectively segmenting these particular defects and to eventually obtain segmentation labels for the entire SMARTEX database. Due to starting this project with an unlabeled dataset, we initially had to rely on self-supervised learning. Specifically, our first classifier, analysed in the following section, depends on the underlying structure of data to predict outcomes. This initial model resulted in a labeled fraction of the dataset, allowing us to then apply semi-supervised learning techniques. In section 4.4 we explain our, mostly unproductive, first approach with semi-supervised learning and in section 4.5 we describe our final classifier, which provided good results and could later be recursively trained with increasingly better-labeled datasets by itself.

Regarding the dataset used throughout this project, it consisted of 12245 samples from fabrics with self-evident *vertical* defects, previously classified by SMARTEX. Every sample contained an image for the FRONT angle, which was the only angle used for this project. Every fabric was single jersey (formed by one set of needles) and greige (not bleached). The different materials of yarn present in this dataset were cotton (8005 samples), lyocell (2577), polyester (1118), viscose (70), and one specific type internally named '50.50' (475).

## 4.1 Simple Vertical Defect Classifier

Besides resolving our specific problem of starting with an unlabeled dataset, building up simple models and then progressively increasing complexity according to our performance needs and goals is a good practice in machine learning. This practice is beneficial to any project in two ways. First, model interpretability usually correlates negatively with complexity. Second, more complex models may bring additional costs, e.g. with data storage necessities or power consumption during model fitting and predicting.

The self-supervised model we created to classify *vertical* defects in textiles is described in-depth in this section. This model is based on computer vision techniques implemented by the Python library *cv2*, a wrapper for the OpenCV C++ Application Programming Interface (API) [17]. Succinctly, we construct a pipeline consisting of an image preprocessor, a line detector method, and a final classification algorithm. Every image is converted to grayscale (1-channel image) prior to going through this pipeline. Image processing functions are described in subsection 4.1.1 and line detector methods are analysed in subsection 4.1.2. The classification algorithm and interface are explained in subsection 4.1.3.

This classifier model and the line detector module, as well as the graphic tools of subsection 4.1.4 and section 4.2, were built as Python classes for the modularity advantages of these objects i.e. the ability to embed classes in other classes. The methods employed to select the hyper-parameters of the final model are described in subsection 4.1.4 and a detailed overview of the performance of the classifier is given in subsection 4.1.5.

As the first step of the *vertical* defect classification process, a significant amount of time and attention was spent in this part of the project, due to its self-supervised nature and the eventual use of the generated labels in the later semi-supervised models. Obtaining many acceptable labels in this step would improve the fitting the next models, both as less time would have to be spent labeling images by hand with the graphic tool of section 4.2, and as the results of the next models would largely depend on the results of this model.

### 4.1.1 Image Processing

As recommended in the guidelines of most line detector algorithms, better results can be achieved by employing some previous feature extraction. With this intent, we experimented with several image processors until we found some methods which were capable of extracting the most relevant features for the detection of the *vertical* defect.

These methods were the Sobel derivatives, the Laplacian operator, and the Canny edge detector. Aside from these methods, the k-means clustering and the contour detection methods were also used for the hyper-parameter set comparisons of subsection 4.1.4. These two methods, while achieving mediocre results, still performed better than some other edge detection algorithms, briefly examined in subsection 4.1.1.5. The main feature extractors, however, will be analysed in the following subsections. Ultimately, the use of image processing proved beneficial, as is directly demonstrated in subsection 4.1.4 with the use of the hyper-parameter sets comparison tool.

In order to seamlessly integrate these methods into the processor-detector-classifier pipeline, various procedures were created for each method (with the template given by algorithm 4.1). These procedures accommodate for the extractors' different arguments via the use of the $**kwargs$ variable, which accounts for any out-of-place keyword argument that may be imputed. This particularly allowed for a more readable and generalized code in the IMAGE_PROCESSOR procedure of the simple classifier class (algorithm 4.8).

**Algorithm 4.1** Image processing methods

```
1: procedure PROCESSOR WRAPPER TEMPLATE(image, blur, sigma, **kwargs)
2:     transform image to grayscale image
3:     output ← PROCESSOR(image, **kwargs)
4:     if blur is True then
5:         apply Gaussian blur to output with blur σ = sigma
6:     end if
7:     return output
8: end procedure
```

#### 4.1.1.1 Gaussian Blur

Prior to the theoretical review of the following subsections on the image processors we used in this project, an introductory analysis of one of the most used filters in image processing, the Gaussian blur, is warranted. This filter was always used before image processing, and available as an option before line detection and after image processing. Some visual experiments, however, indicated that the concurrent use of the two options was excessive. Only the latter option was part of our search for the optimal hyper-parameters of the simple classifier, as the test models and the final model did not use the former option.

The Gaussian blur works by convolving an image with a Gaussian kernel, and is well-suited for noise reduction due to the weights of the kernel smoothly decreasing to zero with distance from the origin. This means that the values of the central location of a pixel

are more important than the values of more remote pixels. The $\sigma$ parameter determines the spread of this central location, as 95% of the total weight will be contained within $2\sigma$ of the center [18]. Specifically, the value of a pixel $(x, y)$ is set to an average of its neighborhood, given by the two-dimensional symmetric Gaussian function

$$g(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{d^2}{2\sigma^2}}, \tag{4.1}$$

where $d = \sqrt{(x - x_c)^2 + (y - y_c)^2}$ is the distance between the pixel $(x, y)$ and the center pixel $(x_c, y_c)$ of the input image. Computationally, values from this distribution are then used to create a finite Gaussian kernel for convolution. Even though the Gaussian function is theoretically non-zero everywhere, it is practically zero at more than $3\sigma$ from the mean. Therefore, a discrete approximation of the function can be produced by truncating the kernel beyond this point.

Taking the center of the input image as $(0, 0)$, it can be shown that $g(x, y)$ is a particular separable function, i.e. the product of two 1D Gaussian functions, one for each direction (equation C.1 of appendix C). OpenCV takes advantage of this fact in its implementation of the method, significantly decreasing the number of operations needed for the method by replacing a 2D convolution with 2 subsequent 1D convolutions. In fact, for a $k \times k$ Gaussian kernel, 2D convolution requires $k^2$ operations for each output pixel while 2 separable 1D filters require $k + k = 2k$ operations per output pixel.

Explicitly, one $k \times 1$ kernel is computed for each direction, where $k$ does not need to be equal for both directions. Every row of the original image is first filtered by convolving the image with the kernel related to the x-direction, and then every column of the output is filtered by convolving the image with the (transposed) kernel related to the y-direction. The coefficients of these 1D kernels, for an arbitrary $\sigma$, are given by

$$G_i = \alpha \cdot e^{-\frac{(i-(k-1)/2)^2}{2\sigma^2}}, \quad i = 0 \ldots k-1, \tag{4.2}$$

where $\alpha$ is a scale factor such that $\sum_i G_i = 1$ [19]. In addition, the $\sigma$ parameter can be defined as a tuple $(\sigma_x, \sigma_y)$ to account for different standard deviations for each direction.

#### 4.1.1.2 Sobel Derivatives

Regarding image processors designed for edge detection problems, the computation of derivatives of an image (or an approximation of them) is a common and successful approach. As substantiated by the official OpenCV documentation [20], in an edge of an

image the pixel intensity of the surrounding areas can be significantly different. Thus, the Sobel operator detects edges in an image by locating pixel locations where the gradient is higher than its neighbors' (or than a specific threshold). This operator achieves this by combining Gaussian smoothing and discrete differentiation, computing an approximation of the gradient of the input image's pixel intensity function.

Explicitly, to approximate the gradient of an image, the Sobel operator traditionally calculates the horizontal changes $G_x$ and the vertical changes $G_y$ of an image and combines these results into the approximated gradient $G$ either as the square root of the sum of the squares of the horizontal and vertical changes, or the sum of the absolutes of these changes. $G_x$ and $G_y$ are obtained by convolving the image with pre-specified kernels. The user can choose the size of these kernels, although it must be an odd number. As an example, for the usual use of this operator with a kernel of size 3, and given a grayscale input image with pixel intensity values $I(x, y)$, the horizontal and vertical changes are given by:

$$G_x = I * \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix}, \quad G_y = I * \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix}. \tag{4.3}$$

Additionally, because Sobel kernels are separable, OpenCV's implementation benefits from the computational advantages of subsequent 1D convolutions [19], akin to its implementation of the Gaussian blur. This implementation of the Sobel operator is capable of calculating first, second, third, and mixed image derivatives by convolving the original image with the appropriate kernels.

In our experiments with this operator, we found that accounting mostly for the horizontal changes $G_x$ understandably achieved better results in detecting vertical features. Thus, the "x-direction percentage" parameter of this method was considered in our search for the optimal hyper-parameter set.

Figure 4.1 portrays the first derivative of an example image only in the x-direction, while figure 4.2 depicts the second derivative in the x-direction of the same image. Generally, and noticeably with this example image, the second derivative calculated by the Sobel operator substantially reduced the noise and better isolated the *vertical* defect. This later proved beneficial (as will be analysed in subsection 4.1.4) when applying the subsequent line detectors.

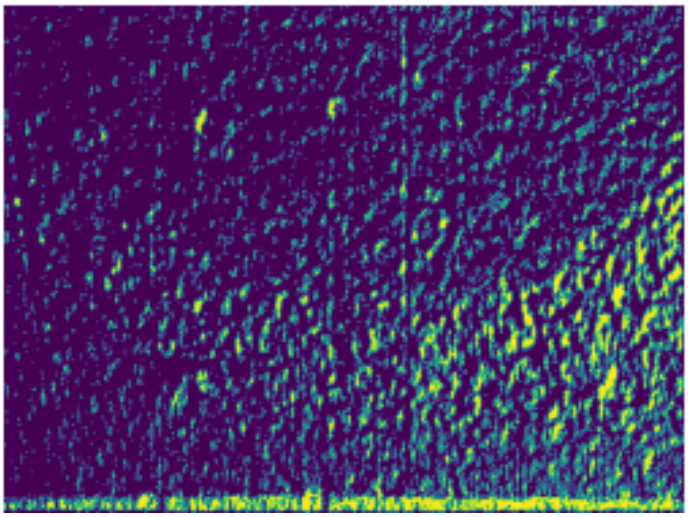


FIGURE 4.1: Sobel (1st derivative) processed image example.

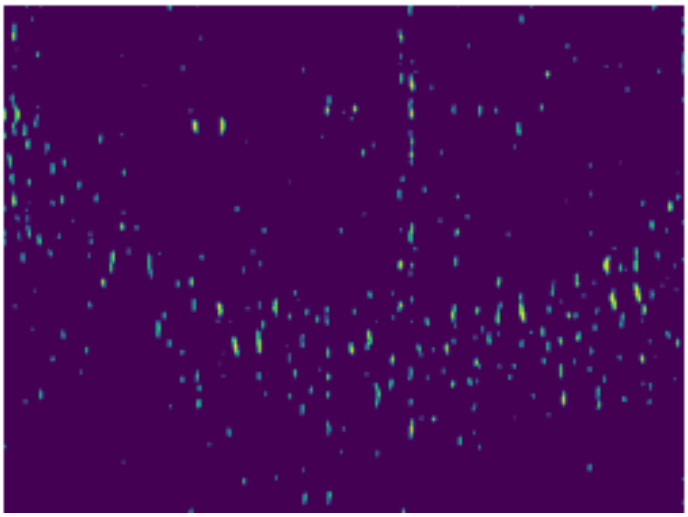


FIGURE 4.2: Sobel (2st derivative) processed image example.

#### 4.1.1.3 Laplacian Operator

The Laplacian image processor implemented by OpenCV also uses the gradient of images, and therefore internally calls the Sobel method to calculate the second derivatives needed for this processor. Formally, given a grayscale image with pixel intensity values $I(x, y)$, its Laplacian $L(x, y)$ is:

$$L(x, y) = \frac{\partial^2 I}{\partial x^2} + \frac{\partial^2 I}{\partial y^2}. \tag{4.4}$$

$L(x, y)$ represents the image returned by the processor with $I(x, y)$ as an input [21]. Figure 4.3 depicts an example of an output of this method, where the *needle* defect (center-right) appears visibly more pronounced than the surrounding area.

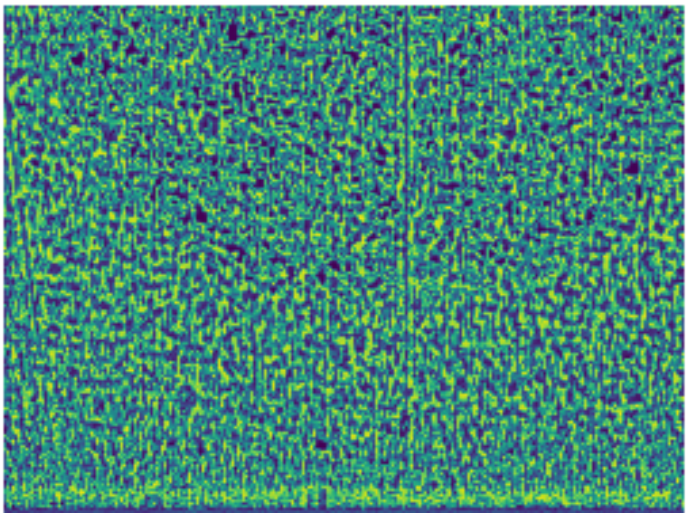


FIGURE 4.3: Laplacian processed image example.

#### 4.1.1.4 Canny Edge Detector

One other feature extractor that achieved significant efficacy was the Canny method, published in an article by John Canny in 1986 [22]. As with the Laplacian operator, OpenCV uses its Sobel method to obtain image gradients used in its implementation of the Canny edge detector [23]. After an initial noise reduction step, where a $5 \times 5$ Gaussian blur is applied, the first derivatives in the horizontal direction ($G_x$) and in the vertical direction ($G_y$) are obtained via OpenCV's Sobel method, as described in subsection 4.1.1.2. These derivatives are combined to form the image gradient and the angle of the gradient direction for each pixel is obtained rounded to one of four angles (representing vertical, horizontal and two diagonal directions) from the following equation:

$$Angle(\theta) = \tan^{-1}\left(\frac{G_y}{G_x}\right) \tag{4.5}$$

As the gradient direction is always perpendicular to edges [23], in the next step, the Canny method checks every pixel for if it is a local maximum w.r.t gradient intensity in its neighborhood of pixels that are in the perpendicular direction of the gradient relative to the original pixel. If a pixel is indeed a local maximum, it is considered for the next stage. Of these selected pixels, preliminary edges are found as groups of connected pixels.

The method then locates "sure edges", where the gradient intensity of its pixels are all above a certain threshold, and discards edges where any of the gradient intensity of its pixels falls below a specified lower bound. For edges that do not contain pixels with

gradient intensity below this lower bound, if they contain any pixel with gradient intensity higher than the upper bound they are selected as final edges. Otherwise, they are discarded.

Both the upper threshold and the lower threshold can be selected by the user in OpenCV's implementation of the Canny method, as tuning these values is crucial due to the variety of computer vision tasks. After some experimentation, we used 200 and 100 as the upper and lower thresholds, respectively, for the final search of the optimal model hyper-parameters of subsection 4.1.4.

Figure 4.4 exhibits an example of this method used with the same image from the examples of the previous subsections. While this method is clearly able to isolate a *vertical* defect, it sometimes fails due to discarding real edges that do not have enough gradient intensity. This weakness stems from not being viable to determine appropriate thresholds for all the various subsets of images in our dataset, coming from different cameras, different factories, or made with different fabrics. Nonetheless, the Canny method was one of the best feature extractors we found for our problem.

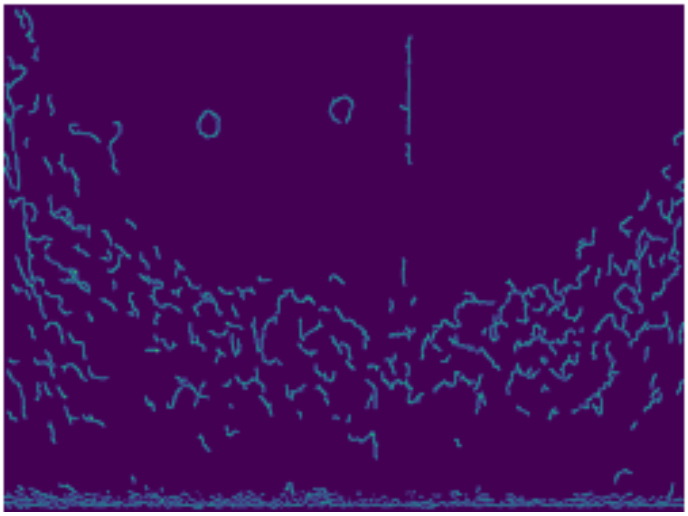


FIGURE 4.4: Canny processed image example.

#### 4.1.1.5 Unused feature extractors

Not every feature extractor achieved good results and in this subsection, we briefly introduce some methods that we experimented with and show some image examples that demonstrate the inefficacy of some of these feature extractors. While these methods did not produce tolerable results in the context of this project, they can surely succeed in other contexts and problems.

Pictured in figure 4.5, we tried out the popular k-means clustering method but it mostly found changes in the lighting exposure of the original images. Likewise, the countour detection method, displayed in figure 4.6, often found the edges of the areas with different lighting exposure instead of the vertical edge we were looking for. Other feature extractors that we experimented with and that are worth noting were the niblack threshold, the superpixel, and the structured edge detector. Every experimentation was made with the OpenCV implementation of these methods.

FIGURE 4.5: K-means processed image example.

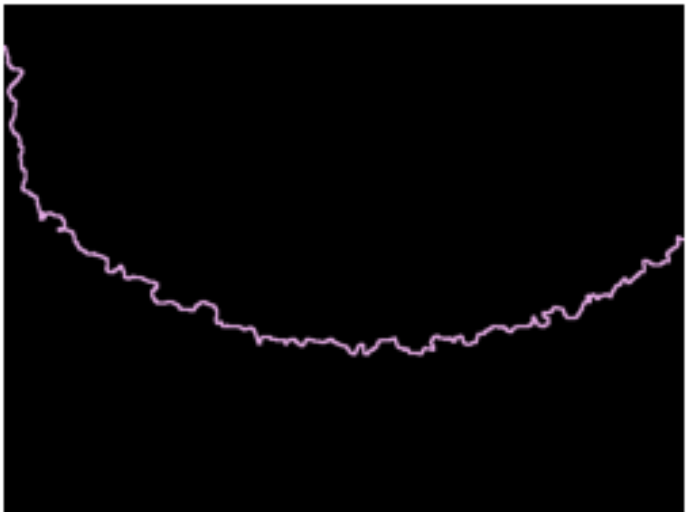


FIGURE 4.6: Contour detection processed image example.

### 4.1.2 Line Detectors

After exploring OpenCV's possible methods for line detection and some visual experiments, four line detectors were selected as possible hyper-parameters for the simple classifier for being the only detectors that produced acceptable results. These are the *Fast Line Detector*, the *Edge Drawing* algorithm, and both the standard and probabilistic implementations of the *Hough Line Transform*. In the context of line detectors and this chapter as a whole, we informally use the word lines to refer to lines, rays, and line segments, unless the distinction is necessary and thus specified.

With the goal of an easier integration of these detectors into our later implemented classifier class, and given that these detectors are similar in their actions, a Python wrapper class (algorithm 4.2) was created to streamline their use. This class provides the expected procedure to detect lines in an image (algorithm 4.3) as well as a procedure to draw lines in an image (algorithm 4.4), capable of showing or saving the output image.

Particularly for the use of the Gaussian blur before line detection (line 9 of algorithm 4.3), as well as before image processing (line 9 of algorithm 4.8, which represents a method of the simple classifier), we set $\sigma = (3.1, 1.1)$ following previous experiments conducted by the ML team leader.

**Algorithm 4.2** Line detectors wrapper class

```
class LINE WRAPPER
    detector : STRING
    blur : BOOLEAN                          ▷ Gaussian blur before line detection
    image                                   ▷ Image ready for line detection
    **kwargs                                ▷ Keyword arguments for main line detector
    lines : TENSOR                          ▷ Tensor to store detected lines
    edge_detector : CLASS                   ▷ Class for Edge Drawing
    houghline_detector : CLASS              ▷ Class for Hough Line Transform
    line_detector : CLASS                   ▷ Class for Fast Line Detector
    save_id : INTEGER                       ▷ Counter of masks in mask folder
    edgemap : IMAGE                         ▷ Image with drawn lines
end class
```

The DRAW procedure is responsible for the process of generating labels for images classified by this simple model. The method CONVERTTOMASK (line 13) of this procedure is a generic function that binarizes an image. This allows the procedure to save binary masks by passing an empty image to the variable *image*. Another similar function was created to convert a binary mask into an RGBA mask. The recourse to the alpha channel (representing the transparency information of the image) was beneficial for later

**Algorithm 4.3** Line detectors wrapper method

```
1:  procedure DETECT(image)
2:      if image was not introduced then
3:          image ← self.image
4:      else
5:          if image has 3 colour channels then
6:              transform image to grayscale image
7:          end if
8:          if self.blur is True then
9:              apply Gaussian blur to image
10:         end if
11:     end if
12:     if self.edge_detector is not None then
13:         self.edge_detector.DETECTEDGES(image)
14:         self.lines ← self.edge_detector.DETECTLINES()
15:     else if self.houghline_detector is not None then
16:         self.lines ←self.houghline_detector.DETECT(image)
17:     else
18:         self.lines ←self.line_detector.DETECT(image)
19:     end if
20:     if self.lines is None then
21:         set self.lines as empty tensor of shape (0, 1, 4)
22:     end if
23:     return self.image
24: end procedure
```

visualizing labels on top of the original colour images. These functions are available, written in Python for clarity, in algorithms C.1 and C.2 of appendix C.

Additionally, this wrapper class can set its image and detector dynamically, as evidenced by algorithms C.4 and C.5, respectively. As a side note, to better understand lines 14-16 of the initialize procedure of this class (algorithm C.3), the main line detector (*Fast Line Detector*) is always initialized as it is its method DRAWSEGMENTS that handles line drawing, regardless of the detector used to obtain those lines.

An additional wrapper class for the two implementations of the *Hough Line Transform* was created (algorithm 4.5) to facilitate their use. This class is equipped with a procedure to detect lines in an image, choosing between the two implementations of this detection algorithm based on a boolean variable (*prob*). Even though this class may seem unnecessary given the designed line detectors wrapper class (algorithm 4.2), it greatly improved code readability by self-containing the process of line detection and storage, which is vastly different between these two implementations. These differences, noticeable in algorithms C.6 and C.7, will be made clear in subsections 4.1.2.1 and 4.1.2.2.

**Algorithm 4.4** Line detectors wrapper method

```
1: procedure DRAW(image, lines, show, save, frame)
2:     if image was not introduced then
3:         image ← self.image
4:     end if
5:     if lines was not introduced then
6:         lines ← self.lines
7:     end if
8:     if frame was not introduced then
9:         frame ← self.save_id
10:    end if
11:    self.edgemap ←self.line_detector.DRAWSEGMENTS(image, lines)
12:    if save is True then
13:        mask ← CONVERTTOMASK(self.edgemap)
14:        save mask to masks folder with name frame
15:        if frame == self.save_id then
16:            increment self.save_id
17:        end if
18:    end if
19:    if show is True then
20:        display self.edgemap
21:    end if
22:    return self.edgemap
23: end procedure
```

**Algorithm 4.5** Hough line wrapper class

```
class HOUGHLINE WRAPPER
    prob : BOOLEAN          ▷ If true apply probabilistic method
    new_lines : TENSOR      ▷ Tensor to store detected lines
    edges : IMAGE           ▷ Image used for detection
end class
```

**Algorithm 4.6** Hough line wrapper method

```
1: procedure DETECT(image)
2:     set self.new_lines as empty tensor of shape (0, 1, 4)
3:     self.edges ← image
4:     if self.prob is True then
5:         self.PROBABILISTIC()
6:     else
7:         self.STANDARD()
8:     end if
9:     return self.new_lines
10: end procedure
```

#### 4.1.2.1 Standard Hough Transform

The Hough Transform is a feature extractor used in image analysis and computer vision. The original method was first patented in 1962 by P. Hough, while the algorithm as it is universally used today was introduced in 1972 by R. Duda and P. Hart (with its $(\rho, \theta)$ parametrization) [24]. By describing any arbitrary straight line by the angle $\theta$ of its normal vector and its distance $\rho$ from the origin, this method can represent any straight line by a single point in the $(\rho, \theta)$ plane. Restricting $\theta$ to the interval $[0, \pi]$, the normal parameters for a line are unique and every line in the $(x, y)$ plane corresponds to an unique point in the $(\rho, \theta)$ plane [24]. The equation of a line resulting from this parametrization is

$$x \cos\theta + y \sin\theta = \rho. \tag{4.6}$$

In turn, each point $(x_i, y_i)$ transforms into a sinusoidal curve in the $(\rho, \theta)$ plane given by

$$\rho = x_i \cos\theta + y_i \sin\theta. \tag{4.7}$$

Thus, detecting points in a straight line becomes a problem of finding concurrent curves. The algorithms first transforms every figure point into a curve in the $(\rho, \theta)$ plane. If the curves of any set of figure points intersect, these points belong to the same line [24]. The final lines obtained by this algorithm are the lines which have a greater number of intersections in the $(\rho, \theta)$ plane than a pre-specified threshold [25]. In our case, the threshold was set at 100 intersections.

Additionally, simply by changing the algorithm's parametrization, this algorithm is capable of detecting arbitrary shapes, most commonly circles or ellipses. The algorithm we used, implemented by OpenCV and designed for straight lines, returns the detected lines of an image in their $(\rho, \theta)$ coordinates. These coordinates are then converted into $(x, y)$ coordinates that define the straight lines found (lines 5–12 of algorithm C.6).

#### 4.1.2.2 Progressive Probabilistic Hough Transform

The Hough Transform is a popular statistical method for geometric features extraction and due to experimenting with the standard Hough Line Transform, we felt it would be relevant to also test the probabilistic version of the algorithm. The version of the probabilistic Hough Line Transform implemented by OpenCV is the algorithm conceived by J. Matas, C. Galambos, and J. Kittler and published in the year 2000 [26].

This implementation, differently from the implementation of the standard Hough Transform, directly returns the coordinates of the edges of each line segment found in an image. In addition, it allows for rejection parameters of minimum line length and maximum gap between points to allow linkage, which we empirically set at 25 and 150, respectively, for our experiments with this algorithm.

The algorithm, officially entitled *Progressive Probabilistic Hough Transform* (PPHT), differs from the usual probabilistic Hough Transform because it minimizes the computation needed to detect lines [26]. Whereas the usual algorithm applies the standard Hough Transform to a preselected random fraction of input points, the PPHT algorithm achieves better performance by accounting for possible noise in the data in its decision to make a random chosen pixel a part of a new line. By comparing the size of a line given by a point to a predefined threshold, and instead restarting with another random point if the threshold value is not met, the algorithm is faster than the usual probabilistic Hough Transform, as well as the standard Hough Transform, and well-suited for real-time tasks [26].

Besides empirically achieving faster detection times than the rest of the line detector algorithms presented in this section, the PPHT algorithm was also the best at detecting only the most salient features. This characteristic of the algorithm naturally made it an excellent candidate in the context of our problem, where the line detector should bypass as much image noise as it can to find the usually small amount of *vertical* defect in an image. Unsurprisingly, the PPHT algorithm was ultimately selected (more details in section 4.1.4) as the line detector of our final model, as it achieved better results than the standard Hough Transform algorithm, and the more outdoors-focused *Fast Line Detector* and *Edge Drawing* algorithms described in the following subsections.

#### 4.1.2.3 Fast Line Detector

OpenCV implements the algorithm introduced by J. H. Lee, S. Lee, G. Zhang, J. Lim, W. K. Chung, and I. H. Suh in 2014 [27] in its *Fast Line Detector* class. This visual place recognition algorithm uses only straight lines instead of the usual point features of other place recognition methods, and was designed specifically for outdoors environments, where line features are easily found in man-made structures.

Given that this algorithm uses only line features, we found it a good candidate to detect the *vertical* defects in the images of our problem, since these defects are usually not curved. Nonetheless, even in the case of slightly curved defects, this algorithm usually

picked up on two or more small straight lines to represent the bigger curved defect. Some visual experiments later corroborated the effectiveness of this algorithm with our defects. The algorithm, however, was the slowest detector amongst the ones described in this section.

#### 4.1.2.4 Edge Drawing

The Edge Drawing algorithm devised in 2012 by C. Topal and C. Akinlar [28] was implemented in OpenCV's *Edge Drawing* class and was selected as a line detector candidate for our problem due to its use of straight line features and acceptable preliminary experiments. This algorithm employs a novel approach in edge detection, by first locating sparse points along so-called anchors, and then joining these anchors via an heuristic edge tracing procedure [28].

### 4.1.3 Classifier Algorithm and Interface

The next step in the simple model pipeline would be devising an algorithm capable of parsing a tensor of detected lines and returning a tensor of acceptable lines, according to the context of our problem. Since the line detectors could find lines of any orientation, an immediate feature of this algorithm would have to be the orientation of the lines. This is done by setting the minimum absolute angle of a line with an imaginary x-axis that divides it horizontally in half as a parameter. To clarify, setting a $\frac{\pi}{4}$ angle would allow lines with positive angles from the $\frac{\pi}{4}$–$\frac{3\pi}{4}$ range with the x-axis.

We also found it useful for the algorithm to be able to control for the minimum vertical length (i.e. the absolute difference of the y-coordinates of the edges). In addition, a third condition was implemented that controls for lines contained in the first and and last 30-pixel vertical strips (of 640-pixel wide images). This condition was added after noticing non-existing defects were sometimes, and unexpectedly, found by some line detectors in these sectors. This algorithm is portrayed in pseudocode in lines 13–26 of algorithm 4.10.

The previously mentioned algorithm was integrated into the API of the simple classifier, with the use of the Python class object (algorithm 4.7). This API was created to be able to handle the processing and the line detection of an image, the classification of acceptable lines, and the classification of several images at once. The modularity of the Python class object heavily facilitated its use in the search for the best hyper-parameter sets and its integration in the comparison tool of subsection 4.1.4.

With this in mind, this classifier class has four methods: an image processor (algorithm 4.8) which processes and sets images for line detection; a procedure to get detected lines (algorithm 4.9) which sends images to processing and detects and returns their lines; a line classifier method (algorithm 4.10) that handles parsing detected lines and classifying them, as well as drawing the lines on the original image (or creating an RGB mask that can then be converted into a binary 1-channel label); and the predict method of algorithm 4.11, which is a wrapper method for classifying a set of images.

**Algorithm 4.7** Simple classifier class

```
class SIMPLE CLASSIFIER
    function : FUNCTION                          ▷ Processing function
    linedetector : STRING
    blur : BOOLEAN                               ▷ Gaussian blur before line detection
    iterations : INTEGER                         ▷ Number of images to predict
    imageindex : INTEGER                         ▷ To predict only one image
    wrapper : CLASS                              ▷ Line detectors wrapper class
    **kwargs                                     ▷ Keyword arguments for main line detector
    original_image : IMAGE
    image : IMAGE                                ▷ Image to process and classify
    lines : TENSOR                               ▷ Tensor to store detected lines
    good_lines : TENSOR                          ▷ Tensor to store accepted lines
    good_lines_counter : LIST                    ▷ List to store number of accepted lines per image
    angle : RADIANS                              ▷ Acceptable absolute angle of a line with the x-axis
    min_size : INTEGER                           ▷ Acceptable minimum size of a line
    edgemap : IMAGE                              ▷ Image with drawn lines
    i : INTEGER                                  ▷ Counter for prediction iterations
end class
```

### 4.1.4 Hyper-parameter Sets Comparison Tool

Given the number of usable preprocessing functions, line detectors, and each of their parameters (including the optional use of Gaussian blur after processing), a method had to be employed to find acceptable hyper-parameter combinations. Initially, a grid search method was created, which coupled the classification of images and the collection of some metrics: the average number of detected lines per image, the rate of zero-defect images after the model prediction (useful since every image in the dataset contains at least one *vertical* defect), and a list of the number of detected lines per image, to check for irregularities in the distribution of the number of lines. This grid search method, while not optimal, substantially limited our search to a handful of hyper-parameter sets due to the much worse performance of other sets in these metrics.

**Algorithm 4.8** Simple classifier method

```
1:  procedure IMAGE_PROCESSOR(blur, sigma, kernelsize, xweight)
2:      if self.iterations == 1 and self.imageindex was introduced then
3:          frameindex ← self.imageindex
4:      else
5:          frameindex ← self.i
6:      end if
7:      load RGB image as array to self.original_image
8:      copy self.original_image to self.image
9:      apply Gaussian blur to self.image
10:     if self.function is not None then
11:         self.image ← self.function(self.image, blur, sigma, kernelsize, xweight)
12:     else
13:         transform self.image to grayscale image
14:     end if
15:     self.wrapper.SET_IMAGE(self.image)
16:     return self.image
17: end procedure
```

**Algorithm 4.9** Simple classifier method

```
1: procedure GET_LINES(**kwargs)
2:     self.IMAGE_PROCESSOR(**kwargs)
3:     self.lines ←self.wrapper.DETECT()
4:     return self.lines
5: end procedure
```

However, since this current step is a self-supervised learning model for image classification, the previous metrics are understandably unreliable. The best method we found to compare hyper-parameter sets was a tool that could visually display any parameter combination. With this purpose, we developed a Graphical User Interface (GUI) capable of exhibiting different images and sorting through different parameter sets in real-time, based on user input. This graphical application, created with the *matplotlib* library, can also call the simple classifier class of algorithm 4.7 using the currently selected hyper-parameter set and then display the classified defects on top of the original image.

This GUI was created as a callable Python class, represented in algorithm 4.12, and accepts user input making use of *matplotlib*'s button and sliders objects. Figures 4.7 and 4.8 display this functionality, as well as the visualization of 4 processed images and their subsequent classification. Note that there are apparently misclassified lines at the left side of 3 images in figure 4.8. These occurrences were the basis for the third condition of the line classifier method of algorithm 4.10 regarding the control of lines detected in the first and last 30-pixel vertical strips of the $480 \times 640$ images.

**Algorithm 4.10** Simple classifier method

**procedure** LINE_CLASSIFIER(*lines*, *angle*, *min_size*, *show*, *save*, *mask*, ***kwargs*)
  **set** $self.good\_lines$ as empty tensor of shape (0, 1, 4)
  **if** *lines* was not introduced **then**
    **try**
      $lines \leftarrow self.lines$
    **catch** name error ▷ Triggers in case $self.lines$ does not exist
      **alert** that lines have not been set
      **return** empty tensor of shape (0, 1, 4) and zero tensor of shape (480, 640, 3)
    **end try**
  **end if**
  $self.angle \leftarrow angle$
  $self.min\_size \leftarrow min\_size$
  **for** $j \leftarrow 0, length(lines) - 1$ **do**
    **try**
      $x_1, y_1, x_2, y_2 \leftarrow lines[j][0]$
      $condition1 \leftarrow |y_2 - y_1| > |x_2 - x_1| \times |\tan(self.angle)|$
      $condition2 \leftarrow |y_2 - y_1| > self.min\_size$
      $condition3 \leftarrow max(x_1, x_2) > 30$ **and** $min(x_1, x_2) < 610$
      **if** *condition*1 **and** *condition*2 **and** *condition*3 is True **then**
        **increment** $self.good\_lines\_counter[self.i]$
        **concatenate** $x_1, y_1, x_2, y_2$ to $self.good\_lines$ as a tensor of shape (1, 1, 4)
      **end if**
    **catch**
      **return** empty tensor of shape (0, 1, 4) and $self.original\_image$
    **end try**
  **end for**
  **if** *mask* is True **then**
    **store** zero tensor of shape (480, 640, 3) in *image*
  **else**
    $image \leftarrow self.original\_image$
  **end if**
  $self.edgemap \leftarrow self.wrapper$.DRAW(*image*, $self.good\_lines$, *show*, *save*, $self.frame$)
  **return** $self.good\_lines$ and $self.edgemap$
**end procedure**

**Algorithm 4.11** Simple classifier method

**procedure** PREDICT(***kwargs*)
  **for** $i \leftarrow 0, self.iterations - 1$ **do**
    $self.i \leftarrow i$
    $self$.GET_LINES(***kwargs*)
    $self$.LINE_CLASSIFIER(None, ***kwargs*)
  **end for**
  **return** $self.good\_lines\_counter$
**end procedure**

The hyper-parameter set exhibited in these figures was ultimately deemed as the best set and used for the generation of labels for the whole dataset with the simple classifier. This set consists of the Sobel (2nd derivative) image processor, followed by a $3 \times 3$ Gaussian blur with $\sigma = (1.5, 1.5)$ and 100% in the x-direction, and the *Progressive Probabilistic Hough Transform* line detector. Figures C.1 through C.6 of appendix C display processed and classified images by the three next best hyper-parameter sets, originally found by the grid search method.

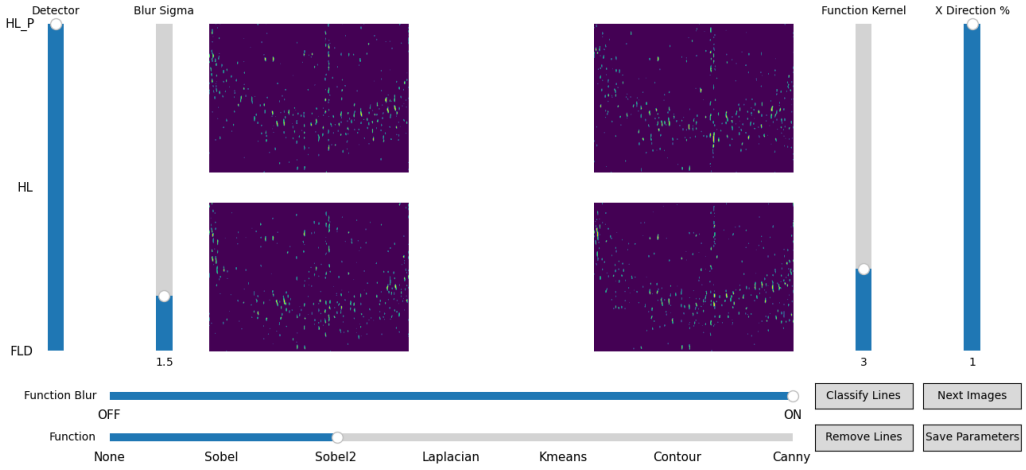



FIGURE 4.7: Tool interface and visualization of processed images with the best hyper-parameter set.

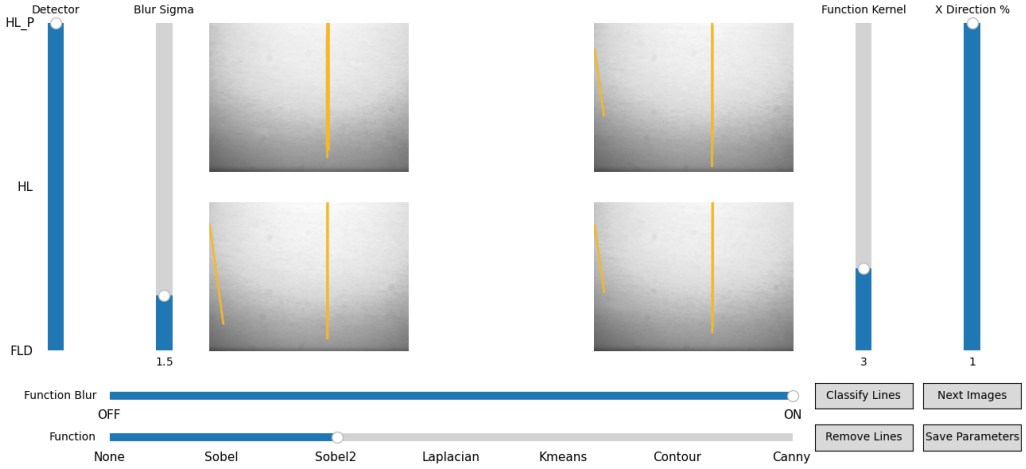



FIGURE 4.8: Tool interface and visualization of classified images with the best hyper-parameter set.

Algorithm 4.13 describes the procedure of calling the class of this graphic tool. There are two passable arguments, for the rows and columns of the image grid (although the tool's design was optimized for a $2 \times 2$ grid). This procedure is responsible for initializing the *matplotlib* figure, the simple classifier model, creating the plots, the sliders, and the buttons, and finally, showing the figure, ready to accept user input.

Aside from the methods to create the initial plots and all sliders (lines 10 and 11 of algorithm 4.13), this tool comes equipped with a method to re-create sliders that were removed (when a blur is not used after the processor or the processor does not use the x-direction % parameter); a method that updates the plots after a parameter change; a method to classify and update the plots with the original images and the detected lines on top, called by the "Classify Lines" button; a method to save the current hyper-parameter set to a file, called by the "Save Parameters" button; a method to cycle to the next set of images, called by the "Next Images" button; and an auxiliary method used to change the line detector of the simple classifier model.

**Algorithm 4.12** Comparison tool class

```
class PARAMETER COMPARISON TOOL
    rows : INTEGER                    ▷ Rows of image grid
    cols : INTEGER                    ▷ Columns of image grid
    axes : LIST                       ▷ List of subplots
    fig : FIGURE                      ▷ Main interface
    model : CLASS                     ▷ Defect classifier model
    functions : LIST                  ▷ List of processing functions
    detectors : LIST                  ▷ List of line detectors
    removed : LIST                    ▷ Boolean checks for removed sliders
end class
```

**Algorithm 4.13** Comparison tool method

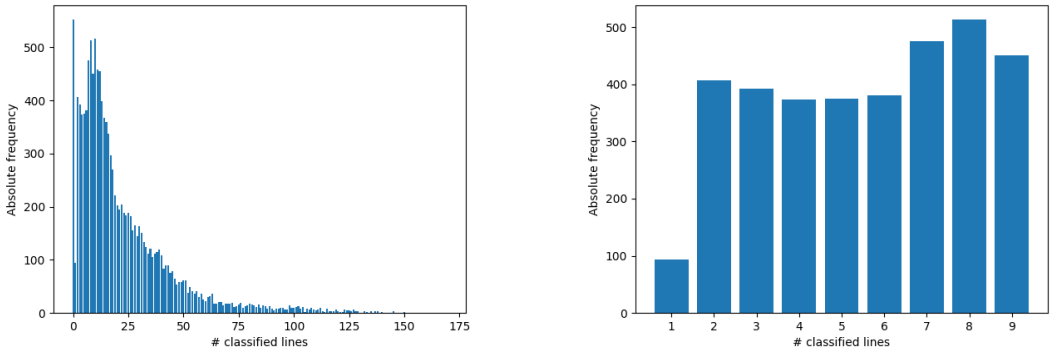
```
1:  procedure INITIALIZE(rows, cols)
2:      self.rows ← rows
3:      self.cols ← cols
4:      set self.axes as empty list
5:      initialize self.fig with Figure module from matplotlib
6:      initialize self.model with simple classifier
7:      define list of processing functions in self.functions
8:      define list of line detectors in self.detectors
9:      set self.removed as list of zeros of length 4
10:     self.CREATE_PLOTS()
11:     self.CREATE_SLIDERS()
12:     show figure
13: end procedure
```

### 4.1.5 Metrics and Results

In evaluating the performance of this simple classifier, our options for a metric were limited due to our model input being unlabeled. Apart from the eye test, which was applied using both the graphic tool of section 4.2 and the comparison tool of subsection 4.1.4, no usual metric could be applied. While the model performed moderately well in the couple hundred individually analysed pictures, it could be risky to extrapolate that performance to pictures possibly originating in different factories, with different cameras or different fabrics.

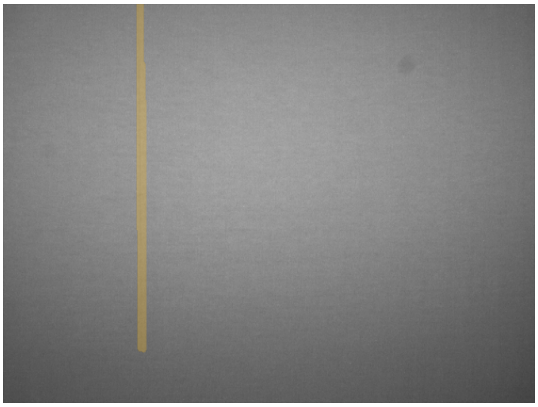
The metric that we would ultimately use to evaluate this model was the number of classified lines. This metric, coupled with a decision boundary of 9 lines, inclusive, would give us 3460 acceptable labels (which exclude labels with zero defects). While this decision boundary seems excessive, we note that this model often detects small lines very close to each other, which convey an accurate bigger line. The dataset also consists of some samples which have two or more *vertical* defects. Given this and the relatively uniformity of bar plot (B) of figure 4.9, we decided this decision boundary would be appropriate.

Figures 4.10 and 4.11 convey some of the best and worst produced masks, respectively. In the best results, our simple classifier was able to locate with precision the location of the defects. In the worst results, the model was, more often that not, detecting extra lines for defects that did not exist or detecting mild natural ridges of the fabric, rather than failing to detect existing *vertical* defects. This issues stem from the inner variability between images classified with a static hyper-paramenter set. While the comparison tool of the previous subsection significantly aided our search, it was unviable to check the success of the chosen set with a majority of the dataset.

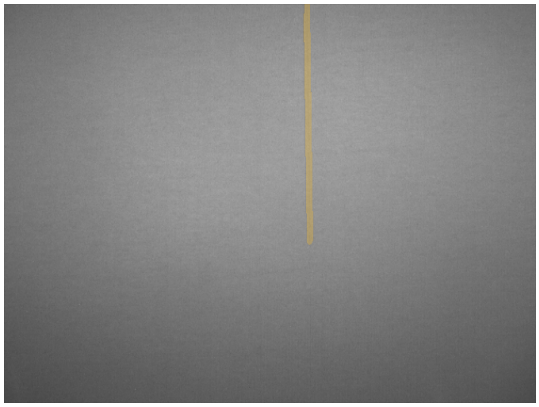



(A) Full bar plot. (B) Maximum of 9 lines.

FIGURE 4.9: Bar plots of the number of lines classified by the simple classifier.

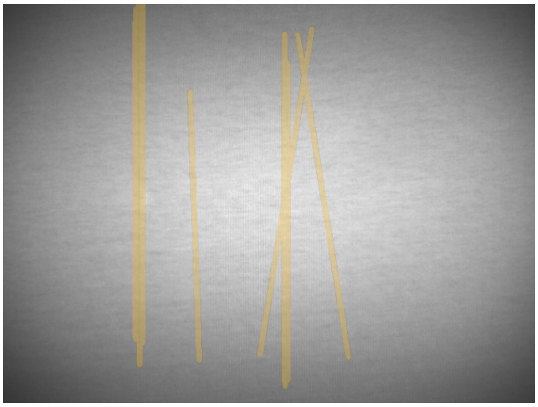


(A) Example 1.

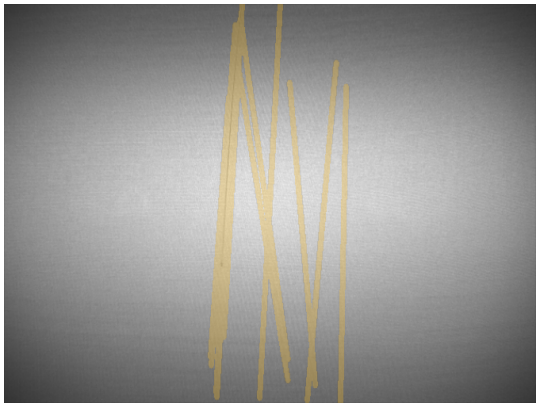


(B) Example 2.

FIGURE 4.10: Examples of the best masks produced by the simple classifier.



(A) Example 1.



(B) Example 2.

FIGURE 4.11: Examples of the worst masks produced by the simple classifier.

## 4.2 Graphic Tool for Human Labeling

After the initial simple defect classifier, the next step would be fitting a more complex ML model, an ensemble classifier. However, due to the mixed results of our first self-supervised classifier, we decided to extend our labeled dataset by visually checking labels given by the simple classifier consisting of 10–15 classified lines. In the case a label was unsatisfactory, we generated a label by hand to bolster the chances of the next models to yield good results. This was an extensive part of the project in two ways. First, creating the interactive graphic application used to generate the defect labels. Second, generating around 1000 masks while checking over 1500 labels given by the simple classifier.

The interface of the tool was adapted from a previous SMARTEX project, while the logic behind line construction was created for this task. The tool has the ability to draw either lines, line segments, or rays, using only the left and right mouse buttons. Line width can be increased or decreased by the ”up” and ”down” buttons, respectively. Initially, the

user sets either a reference point or an origin point. Depending on the next point, one of these geometric constructions is produced. In addition, as seen in figure 4.14, the tool has the ability to produce more than one construction per image. Under the hood, a line is drawn by colouring circles of the user-defined size along the line that passes through the two input points. Rays and line segments are drawn with the same logic, accounting for where they start and if they end.

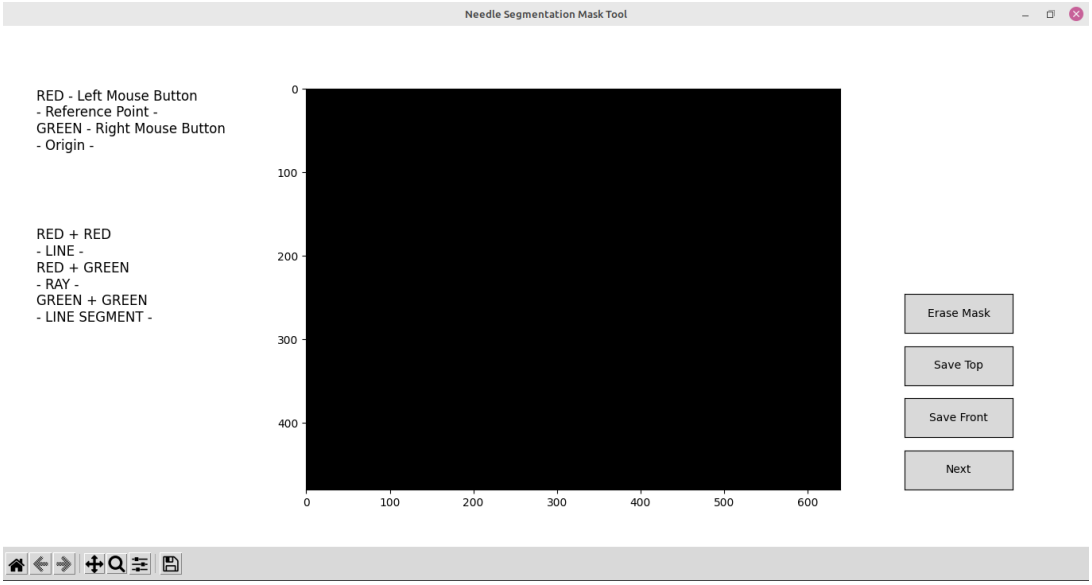



FIGURE 4.12: Tool initialization and interface.

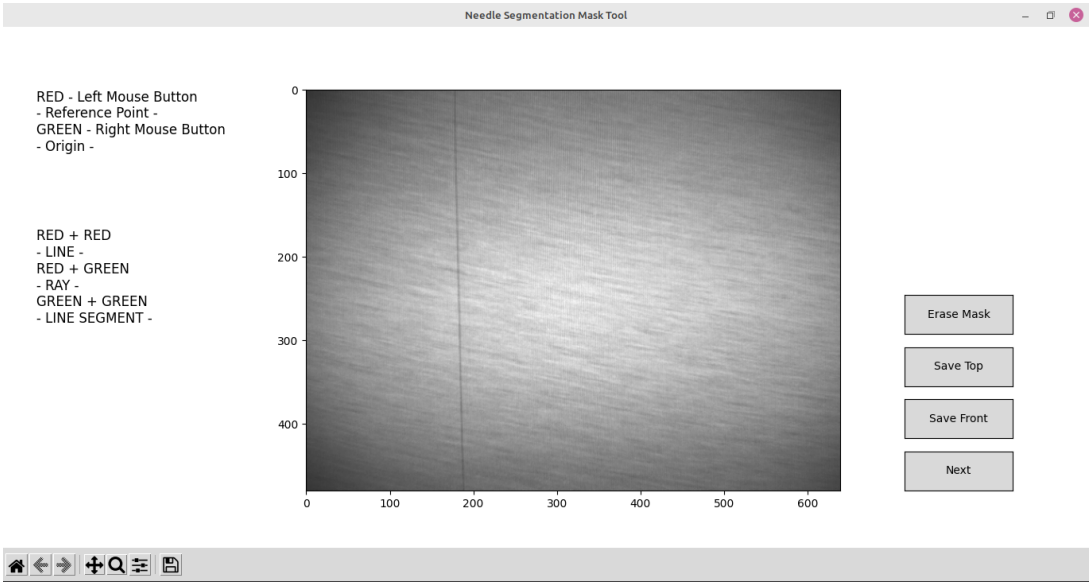
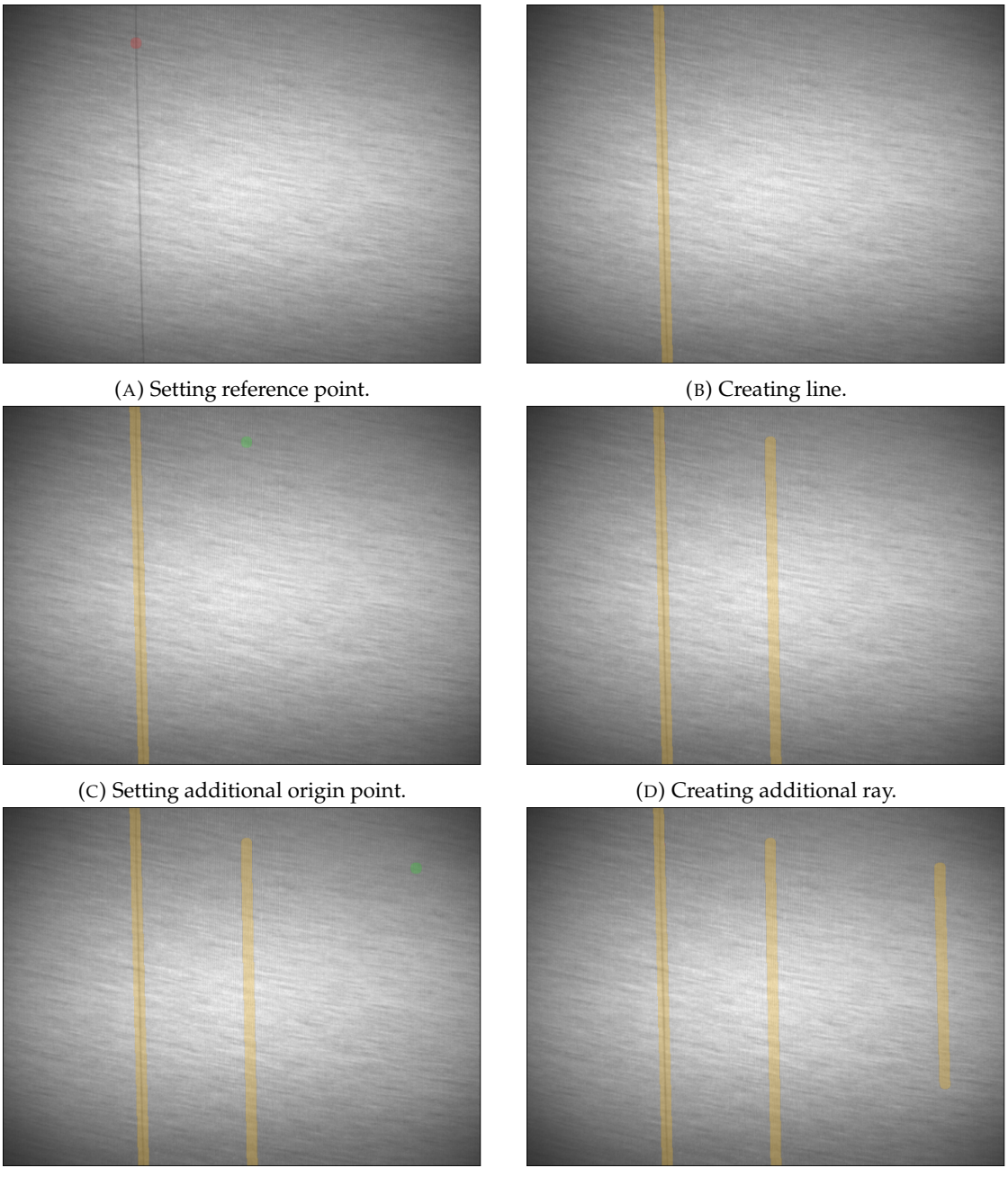



FIGURE 4.13: GUI with image interface.



(A) Setting reference point.

(B) Creating line.

(C) Setting additional origin point.

(D) Creating additional ray.

(E) Setting additional origin point.

(F) Creating additional line segment.

FIGURE 4.14: Example of the use of the graphic tool.

## 4.3 Dimensionality Reduction and Clustering

Following the simple classifier model of section 4.1, the next step was implementing a more complex machine learning model, fitted with the acceptable results of the simple classifier, based on the metric of subsection 4.1.5, and the masks produced with the use of the graphic tool of section 4.2.

This ML model would be an ensemble classifier and the decision was made for the model to be a random forest due to its use in previous and similar internal projects. Given the inherent memory limitations of the architecture of the model (analysed in-depth in subsection 4.4.1), the training dataset would have to be split in subsets of approximately 50 images to fit several different models. To generate a label for a new image, it would have to be assigned to one of the existing clusters and then classified by the corresponding random forest model.

Initially, random forest models were fit with simple sequential partitions of the dataset into subsets with 40 images. However, this method did not produce tolerable results at all, in part because of the variability of the source of the images in each subset (e.g. different factories or fabrics). More complex clustering and dimensionality reduction techniques were thus applied in an attempt to improve the results of the ensemble classifier. These techniques will be described in-depth in this section.

Succinctly, features were first extracted from the images using a pretrained MNV2 model. Afterwards, Principal Component Analysis (PCA) substantially reduced the features' dimensionality. The resulting features were then given a two-dimensional location using t-distributed Stochastic Neighbor Embedding (t-SNE), which was helpful to visualize the various groupings of the data points (as shown in figure 4.15) and ascertain the efficacy of the final clustering.

Lastly, the data points that resulted from PCA were clustered using the Density-Based Spatial Clustering of Applications with Noise (DBSCAN) method. The final clusters given by this method were then visually compared (with the use of the 2D locations obtained by the t-SNE method) with the clusters given by the reference groupings obtained with SMARTEX's methods for splitting a dataset by fabric, camera, and time. Employing DBSCAN as our final clustering method seemed to attain better results than the generic grouping with less overlap between clusters in each t-SNE agglomerate, as evidenced by plots (A) and (B) of figure 4.16.

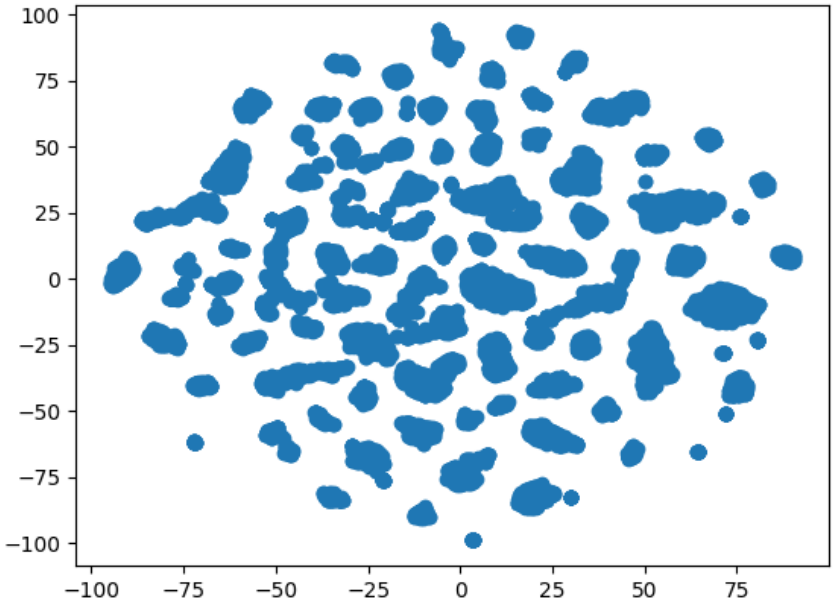
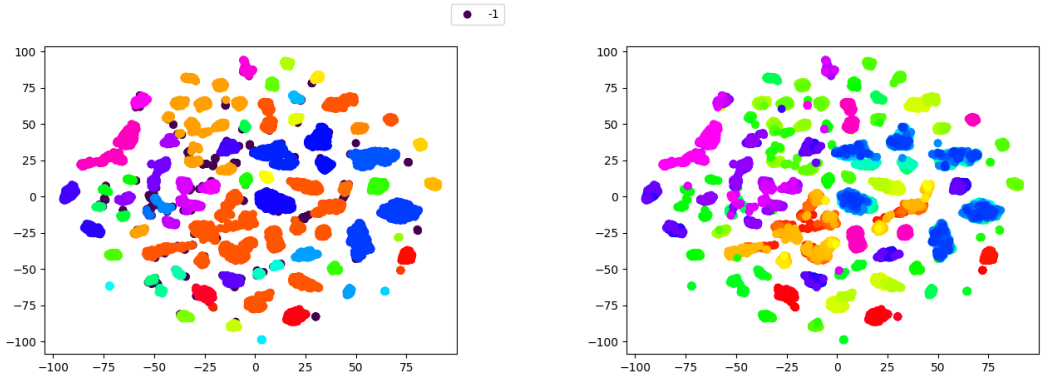



FIGURE 4.15: t-SNE scatter plot.



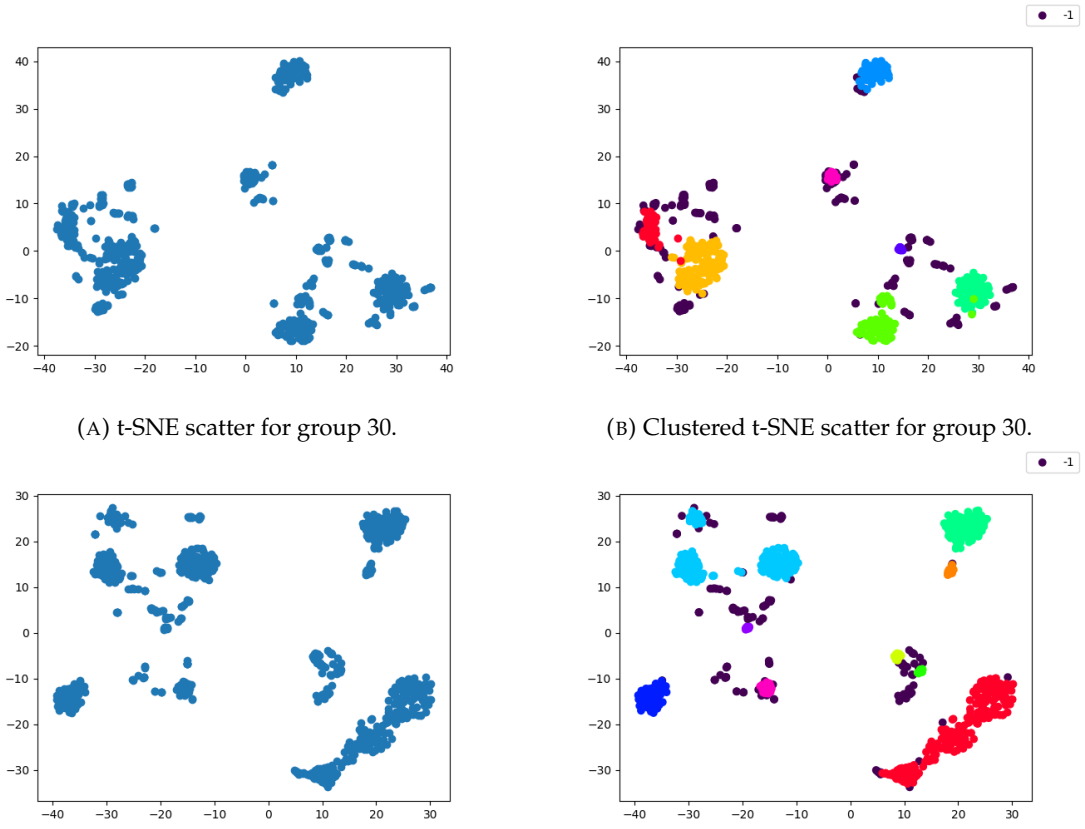

(A) DBSCAN clustered t-SNE scatter.

(B) Reference clustered t-SNE scatter.

FIGURE 4.16: Clustered t-SNE scatter plots.

To achieve even better results in the clustering process, an initial separation of the dataset was conducted via a proprietary method of splitting by camera, fabric, and time of the images. This resulted in 45 initial groups. The clustering and dimensionality reduction pipeline described above and more in-depth in the next subsections was then applied to each of these groups. This specific clustering pipeline was explored based on the recommendations of my external advisor. Some interesting examples of the obtained DBSCAN clusters are shown in figure 4.17, while other interesting t-SNE scatter plots, both clustered and unclustered, are exhibited in figure C.7 of appendix C. The DBSCAN labels

created 185 different clusters which were later split for a more balanced image quantity distribution. In total, this process of clustering and dimensionality reduction resulted in 258 clusters.




(A) t-SNE scatter for group 30.

(B) Clustered t-SNE scatter for group 30.

(C) t-SNE scatter for group 31.

(D) Clustered t-SNE scatter for group 31.

FIGURE 4.17: Scatter plots for DBSCAN clusters in the initial groups.

### 4.3.1 MNV2

The first step in the pipeline was to obtain embeddings (a lower dimensional space with which to represent the higher dimensionality of an initial map) through an MNV2 model. This untrained CNN used the library's standard ImageNet-trained weights, an $\alpha = 0.35$, and average pooling, to considerably reduce the colour images' dimensionality from $720 \times 960 \times 3 = 2073600$ features to 1280 features for each image. Before classification, the data, scaled between -1 and 1, was augmented with typical processes: random lighting, random vertical flip, random zoom, random contrast, and image noise. The straightforward Python code to build this particular model is available in algorithm C.8 of appendix C.

### 4.3.2 PCA

The next step in this process was changing the basis of our data containing 1280 features, obtained by the MNV2 model, via the use of Principal Component Analysis (PCA). Heuristically, and using the implementation of PCA by *sklearn* [29], we decided to extract 45 components from the initial features. In total, these 45 resulting features accounted for more than 99% of the variance in the data.

While PCA is a well-estabilished method for dimensionality reduction with a lot of different derivations, *sklearn* in particular implements a probabilistic PCA devised by M. Tipping and C. Bishop and published in 1999 [30]. In their article, the authors derive PCA through the lens of a maximum-likelihood procedure based on a probability density model of the observed data. Specifically, a latent variable model is used which relates a set of $d$-dimensional data vectors $\{\mathbf{t}_n\}$ to a corresponding set of $q$-dimensional latent variables $\{\mathbf{x}_n\}$ ($q < d$, generally) [30]. This model is of the form

$$\mathbf{t} = \mathbf{W}\mathbf{x} + \boldsymbol{\mu} + \boldsymbol{\epsilon}, \tag{4.8}$$

where $\mathbf{W}$ is a $d \times q$ parameter matrix, $\boldsymbol{\mu}$ is a constant whose maximum-likelihood estimator is the mean of the data, and $\mathbf{x} \sim N(0, \mathbf{I})$. Given an isotropic noise model, $\boldsymbol{\epsilon} \sim N(0, \sigma^2\mathbf{I})$, equation 4.8 implies the following conditional probability distribution over the $\mathbf{t}$-space:

$$p(\mathbf{t}|\mathbf{x}) = (2\pi\sigma^2)^{-d/2} \exp\{-\frac{1}{2\sigma^2}||\mathbf{t} - \mathbf{W}\mathbf{x} - \boldsymbol{\mu}||^2\}. \tag{4.9}$$

Given the Gaussian distribution over the latent variables defined by

$$p(\mathbf{x}) = (2\pi)^{-q/2} \exp\{-\frac{1}{2}\mathbf{x}^T\mathbf{x}\}, \tag{4.10}$$

the marginal distribution of $\mathbf{t}$ is obtained:

$$\begin{aligned} p(\mathbf{t}) &= \int p(\mathbf{t}|\mathbf{x})p(\mathbf{x})d\mathbf{x} \\ &= (2\pi)^{-d/2}|\mathbf{C}|^{-1/2} \exp\{-\frac{1}{2}(\mathbf{t} - \boldsymbol{\mu})^T\mathbf{C}^{-1}(\mathbf{t} - \boldsymbol{\mu})\}, \end{aligned} \tag{4.11}$$

where the model covariance is $\mathbf{C} = \sigma^2\mathbf{I} + \mathbf{W}\mathbf{W}^T$. Applying Bayes' rule, the *posterior* distribution of $\mathbf{x}$ given the observed data $\mathbf{t}$ takes the form

$$\begin{aligned} p(\mathbf{x}|\mathbf{t}) = (2\pi)^{-q/2}|\sigma^{-2}\mathbf{M}|^{1/2} \times \\ \exp[-\frac{1}{2}\{\mathbf{x} - \mathbf{M}^{-1}\mathbf{W}^T(\mathbf{t} - \boldsymbol{\mu})\}^T(\sigma^{-2}\mathbf{M})\{\mathbf{x} - \mathbf{M}^{-1}\mathbf{W}^T(\mathbf{t} - \boldsymbol{\mu})\}], \end{aligned} \tag{4.12}$$

where the posterior covariance matrix is $\sigma^2\mathbf{M}^{-1} = \sigma^2(\sigma^2\mathbf{I} + \mathbf{W}^T\mathbf{W})^{-1}$. Therefore, the log-likelihood of observing the data under this model is

$$\begin{aligned}\mathcal{L} &= \sum_{n=1}^{N} \log\{p(\mathbf{t}_n)\} \\ &= -\frac{Nd}{2}\log(2\pi) - \frac{N}{2}|\mathbf{C}| - \frac{N}{2}\text{tr}[\mathbf{C}^{-1}\mathbf{S}],\end{aligned} \tag{4.13}$$

where $\mathbf{S}$ is the sample covariance matrix for the observed data. Thus, this model's parameters can be estimated by maximising the log-likelihood $\mathcal{L}$. Additionally, the authors showed that the columns of the maximum likelihood estimator $\mathbf{W}_{ML}$ contain the principal eigenvectors of $\mathbf{S}$, with arbitrary rotation and scaling determined by the parameter $\sigma^2$ and the corresponding eigenvalue.

Deriving PCA from the perspective of density estimation offers some importance advantages, such as the possibility of comparison with other density estimation methods via the likelihood measure, easier statistical testing, and the use of PCA in classification problems via the posterior probabilities of class membership [30].

For the eigendecomposition of a matrix, *sklearn*'s implementation of the PCA method uses the randomized truncated Singular Value Decomposition (SVD) published by N. Halko, P. G. Martinsson, and J. A. Tropp in 2011 [31]. This algorithm is especially well-suited for finding a relatively small amount of principal components in large datasets with a large amount of features versus other SVD algorithms.

The algorithm significantly shortens its computational operations by random sampling the original matrix and thus computing a partial matrix decomposition. In fact, to find $k$ dominant components of the SVD of a dense input $m \times n$ matrix, the randomized SVD requires $\mathcal{O}(mn\log(k))$ floating-point operations while classical algorithms require $\mathcal{O}(mnk)$ operations [31]. As we were looking to extract a small amount of components from our dataset which had over 12000 input points with 1280 features each at this point, this feature of the randomized truncated SVD made this algorithm appropriate from our problem and the PCA model was fit in a matter of a couple of seconds.

### 4.3.3 t-SNE

The t-distributed Stochastic Neighbor Embedding (t-SNE) method, devised by L. Van der Maaten and G. Hinton and published in 2008 [32], is used for the visualization of high-dimensional data, typically in two or three dimensions. It works by computing joint probabilities for the high-dimensional data and for the low-dimensional map from similarities

between data points and by minimizing the Kullback-Leibler divergence between the distributions of these joint probabilities [32]. The joint probabilities of the high-dimensional data, are represented by $p_{ij}$ and are set as $p_{ij} = \frac{p_{j|i}+p_{i|j}}{2n}$, where $n$ is the number of data points and $p_{j|i}$ the conditional probability that data point $x_i$ would choose $x_j$ as its neighbor if neighbors were chosen in proportion to their probability density under a Gaussian centered at $x_i$ with variance $\sigma_i$. This conditional probability is given by

$$p_{j|i} = \frac{\exp\left(-||x_i - x_j||^2/2\sigma_i^2\right)}{\sum_{k\neq i}\exp\left(-||x_i - x_k||^2/2\sigma_i^2\right)}. \tag{4.14}$$

In t-SNE, the similarity between low-dimensional data points $y_i$ and $y_j$ is modeled as joint probabilities $q_{ij}$, using a Student t-distribution with one degree of freedom as the distribution to convert distances into probabilities in the low-dimensional map, and are therefore defined as

$$q_{ij} = \frac{(1 + ||y_i - y_j||^2)^{-1}}{\sum_{k\neq l}(1 + ||y_k - y_l||^2)^{-1}}. \tag{4.15}$$

This method minimizes a single Kullback-Leibler divergence between a joint probability distribution, $P$, in the high-dimensional space, and a joint probability distribution, $Q$, in the low-dimensional map using a gradient descent method. The cost function $C$ is given by

$$C = KL(P||Q) = \sum_i \sum_j p_{ij} \log \frac{p_{ij}}{q_{ij}}. \tag{4.16}$$

The gradient of the Kullback-Leibler divergence between $P$ and the t-distribution based joint probability distribution $Q$ is given by

$$\frac{\partial C}{\partial y_i} = 4\sum_j (p_{ij} - q_{ij})(y_i - y_j)(1 + ||y_i - y_j||^2)^{-1}. \tag{4.17}$$

Since this method's cost function is not convex, and thus different initializations can produce different results, the method usually iterates several times in its optimization.

From the 45 components derived by the PCA analysis, the t-SNE method translated each images' information into a two-dimensional space, as seen in figure 4.15. The method employed was standard from the *sklearn* library, with an Euclidean metric and 1000 maximum iterations.

### 4.3.4 DBSCAN

The Density-Based Spatial Clustering of Applications with Noise (DBSCAN) method was introduced by E. Schubert, J. Sander, M. Ester, H. P. Kriegel, and X. Xu, in 1996 at the

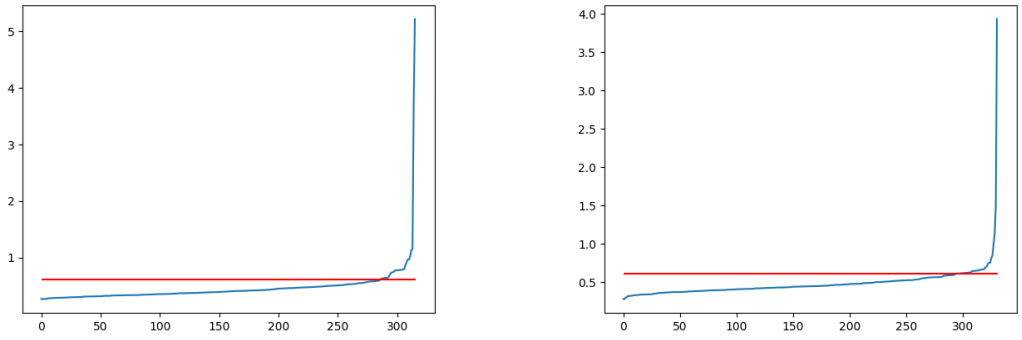
KDD data mining conference [33]. It is a popular density-based clustering algorithm that finds a cluster by starting with an arbitrary point $p$ and finding every point called *density reachable* from $p$ w.r.t *eps* (the maximum distance between two data points for one to be considered as in the same neighborhood of the other), and *min_samples* (the minimum number of samples in a neighborhood of a point for that point to be considered a *core point*) [34].

More specifically, when $p$ is a core point, all its neighbors within the *eps* radius are called *direct density reachable* and are part of the same cluster as $p$. For any neighbor point that is itself a core point, its neighbors are transitively included in this cluster and are called *density reachable*. Non-core points are called *border points* and any point that is not *density reachable* from any core point is considered a *noise point* [35].

If $p$ is a core point then the method yields a cluster else, if no other point is *density reachable* from $p$, it is a border point, and the procedure progresses to the next sample in the dataset [33]. The original DBSCAN algorithm has complexity $\mathcal{O}(n)$ (for examining every point in the dataset) and its labels can significantly change just by the reordering of the dataset. However, *sklearn*'s implementation of this algorithm bulk-computes all neighborhood queries to find the best cluster combinations, although its complexity raises to $\mathcal{O}(n \cdot d)$ where $d$ is the average number of neighbors of a point [34].

Lastly, a DBSCAN analysis, from *sklearn*, was employed on the 45 features per data point originated by the PCA. To find the *eps* for the analysis of each initial group, a Nearest Neighbors model, also implemented by *sklearn*, was used to find the distance between each data point and its closest neighbor. Sorting these distances, the *eps* can heuristically be defined as the distance of the data point at the "corner" of the graph. This distance can be approximated as the distance where the growth from the distance of the previous data point is greater than 1%, according to a 2016 article [36] and corroborated by experiments within our dataset. Examples of this process for groups 22 and 23 are depicted in figure 4.18. In addition, attending to the context of our clustering needs and our small labeled dataset, the *min_samples* parameter was heuristically set at a maximum of 10, varying for groups with a low amount of samples as nearly half of the size of these groups.

After obtaining the labels provided by the DBSCAN analysis, each of the original 45 groups were split based on these labels, creating 185 clusters. The undefined data points (i.e. the noise points) of each initial group were clustered together. Then, due to the size of some clusters, each that had more than 45 images was split into clusters of 40 images




(A) Plot of the distances for group 22.

(B) Plot of the distances for group 23.

FIGURE 4.18: Plots of the sorted distances and the obtained values of *eps*.

maximum, resulting in 258 final clusters. Some clusters that had a lack in quantity of samples labeled by the simple classifier were reinforced with the use of the graphic tool of section 4.2, as much as time allowed, and we were at last ready to fit the random forest models and generate labels for the full dataset.

## 4.4 Ensemble Classifiers - Random Forest

The random forest classifier, introduced by L. Breiman in an article published in 2001 [37], is a popular ensemble learning method for classification and regression tasks that rivaled Adaboost's (Freund & Schapire, 1996) efficacy since its inception, without progressively changing the training set as the boosting algorithms do. Random forests fit various decisions trees on various sub-samples of the dataset and use averaging to improve accuracy and prevent over-fitting. In fact, Breiman proved random forests do not overfit as more trees are added, using the Strong Law of Large Numbers, and instead produce a limiting value of the generalization error [37].

By Breiman's definition, random forests used for classification tasks, as is the case of our project, consist of decision tree classifiers $\{h(\mathbf{x}, \Theta_k), k = 1, \dots\}$ where the $\{\Theta_k\}$ are independent identically distributed random vectors and each tree casts a unit vote for the most popular class at input $\mathbf{x}$. These random vectors can be used to characterize the bagging and random subset of features techniques present in random forests.

Succintly, bagging (or bootstrap aggregating) is the selection of a random sample of the original dataset with replacement, of the same size of the original dataset, for each decision tree in the random forest. Bagging is useful to improve accuracy in conjunction

with the random features technique, and to obtain ongoing estimates of the model's generalization error, strength, and correlation. Similarly, using the random subset of features technique, random forests avoid correlation between their tree classifiers by selecting a random subset with replacement of input features at each candidate split of each decision tree, leading to better accuracy [37].

As explained in the last section, the next step of this segmentation project would be fitting a random forest model for each of the obtained clusters of images. This ensemble classifier was elected for this step due to a random forest framework from a previous SMARTEX project already being present that could be adapted for this project. The ensemble classifiers' architecture, described in subsection 4.4.1, is the same for every cluster and the framework has the ability to continuously train the classifier with datasets containing labels obtained from the previous iterations. A metric for this model is discussed and its performance analysed in subsection 4.4.2. Our experimentation with another ensemble classifier, the distributed gradient boosting model, is outlined in subsection 4.4.3.

### 4.4.1 Model Architecture

A random forest model was fit for each of the clusters obtained in section 4.3 using the random forest classifier from *sklearn*'s ensemble package. In total, 5145 labeled images were used to fit these models. The *Gini* impurity method was employed to determine how well a decision tree was split, for all 100 tree estimators, each with a max depth of 50. For each split, the number of features considered was the square root of the total number of features, while the minimum number of samples needed to split an internal node was 2 and the minimum number of samples required to be at a leaf node was 1.

Each training sample, which contained the information of a single pixel of a single image, was data augmented by a custom implementation (which improves computational performance) of a basic feature extractor from *scikit-image* [38], resulting in 24 features per input. This "pixel-by-pixel" classification was expected to be sub-optimal for this specific classification problem given that it would not take into account the correlations of "neighbor pixels", which would undoubtedly improve the classification of a vertical line. Nevertheless, fitting the model with samples containing the information of an entire image proved impossible due to memory limitations, as each of these samples contained $480 \times 640 \times 24 = 7372800$ features, including prior data augmentation.

In an attempt to solve this problem, we experimented with the classifier chain class from *sklearn*'s multioutput package. This chain model would contain a random forest classifier for each of the pixels of an image and every classifier would be trained with the original features and the predictions of the preceding classifiers in the chain as features. This allows a classifier chain to explore correlations among the labels, which are, in this case, each of the pixels of an image. However, this attempt failed due to memory limitations, given the number of different labels ($480 \times 640 = 307200$) and, once more, the exceedingly high number of features of the input data (7372800 features).

As a last resort to improve the pixel-by-pixel classification on such a scarce and imbalanced dataset (as the *vertical* defects cover only a small part of an image), the weights of each class were set as follows: 1% for class 0 and 99% for class 1. The full call to the random forest model is available, written in Python, in algorithm C.9 of appendix C.

### 4.4.2 Metrics and Results

Given the semi-supervised context of this problem, in that there are no true labels to rely on (aside from the hand-drawn labels), and in addition to having been impossible to check the accuracy of many of the labels obtained by the simple classifier of section 4.1, there was not a clear way to determine the performance of this model. While the preliminary eye-tests were not favorable even for some of the best masks (figure 4.19), which depicted various loosely-connected clusters of pixels instead of the desired connected lines, we still debated on a metric that could quantify in some way the model's performance with a broader range of images.

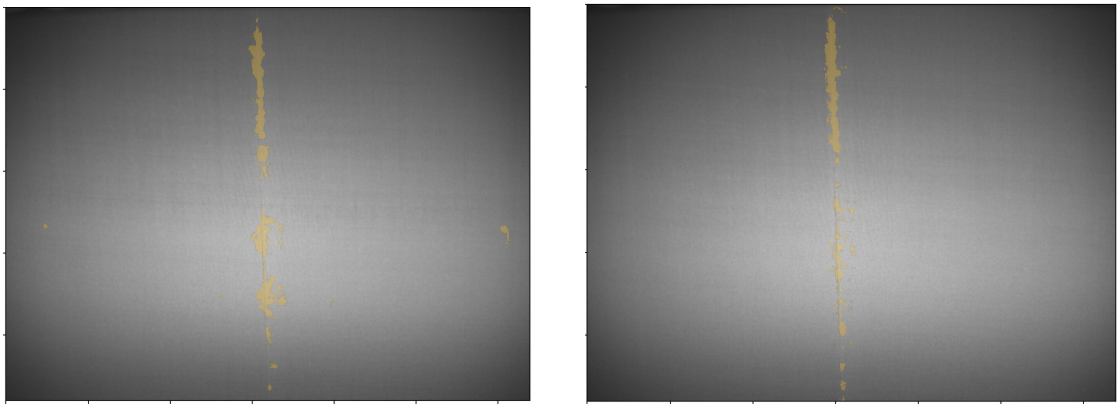
With this purpose, we found it made sense to quantify the model's confidence (i.e. how sure the model is of each prediction). The formula used was

$$\frac{\sum_{i=1}^{N} |p_i - 0.5|}{N} \times 2, \tag{4.18}$$

where $p_i$ is the predicted probability of class 0 given by the model to sample $i$, and $N$ is the total number of samples. A high score in this metric (near 1) represents high confidence in the model's predictions while a low score (near 0) represents the opposite. To exemplify, given 2 different models, one with $p_1 = 0.8$ and $p_2 = 0.3$, and another with $p_1 = 0.7$ and $p_2 = 0.4$, the first would be considered to have more confidence in its predictions than the second model, with scores of $\frac{0.3+0.2}{2} \times 2 = 0.5$ and $\frac{0.2+0.1}{2} \times 2 = 0.3$, respectively.

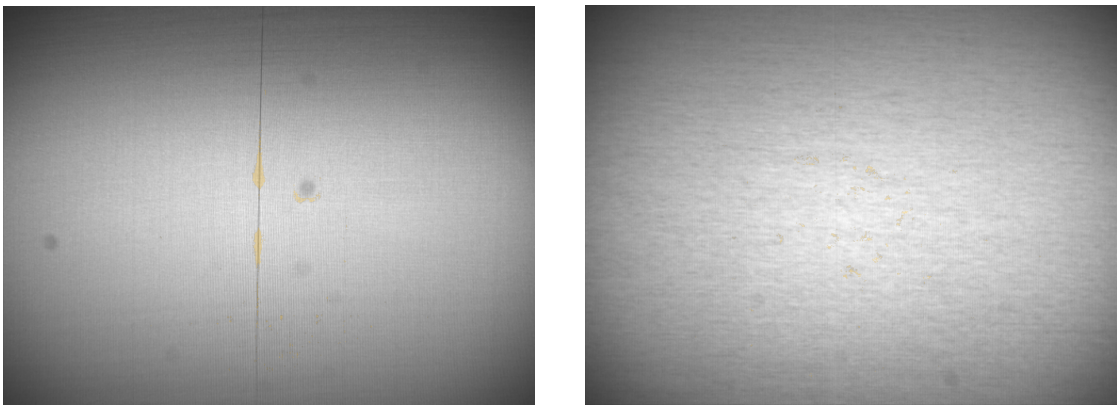
This metric surpassed 80% in most of the cluster models, with an average of around 91% among all clusters, a standard deviation of around 9%, and a median of 95%. These high confidence scores, however, did not translate into high model performance. Unlike the low specificity of the simple classifier model, this model had trouble in classifying positives, even with the class weights parameter over-correction.

The worst masks produced by the random forest models, like figure 4.20 depicts, barely portray any inter-connectedness between the scarce and small groups of positive pixels. As expected, given the graphic nature of our problem and the pixel-by-pixel classification of this random forest model, this model was not suitable for the task at hand.



(A) Example 1. (B) Example 2.

FIGURE 4.19: Examples of the best masks produced by the random forest.



(A) Example 1. (B) Example 2.

FIGURE 4.20: Examples of the worst masks produced by the random forest.

### 4.4.3 Distributed Gradient Boosting

In light of our recent research about the boosting method (particularly the AdaBoost algorithms), we sought to experiment and adapt a boosting algorithm to our defect classification problem. We focused on the distributed gradient boosting (DGB) algorithm, popularized by its success in machine learning competitions, and available via the XGBoost (eXtreme Gradient Boosting) library. This library, originally written in C++, is parallelizable onto Graphics Processing Units (GPUs), making it capable to process large datasets.

The classifier architecture would be largely similar to the architecture of the random forest classifier, also with 100 estimators, for a more direct comparison. Unfortunately, due to time constraints, it was not possible to fit this classifier for each cluster. There is no indication, however, that it would have performed significantly better than the random forest classifier.

As is the case with the previous random forest model, this DGB model would be a pixel-by-pixel classifier and would not process any relationship between the output pixels, seriously undermining the task at hand of finding relatively big and connected lines. Attempts were made to circumvent this problem, also with the use of the classifier chain class from *sklearn*'s multioutput package. Nonetheless, these attempts were not successful due to the same memory limitations .

## 4.5 U-Net Classifier

As the last model we would develop in the scope of this project, and given the poor results of the random forest classifier (section 4.4), a lot of expectations were set on the U-Net classifier. Thankfully, this more complex model yielded good results with limited preparation (due to time constraints). These results demonstrate the capabilities of convolutional neural networks (given a more in-depth examination in chapter 3), and specifically the U-Net network, when applied to image classification and segmentation, and the potential use for these types of models in an industrial production context.

The U-Net model is a CNN architecture developed by O. Ronneberger, P. Fischer, and T. Brox in 2015, and initially designed for biomedical image segmentation [39]. This architecture has the ability to train from fewer training samples, which was optimal for the task at hand in which we only had around 5000 usable labels to train this model.

This networks implements an expansive path, which increases the resolution of the output and is responsible for localization, after the usual contracting path of a CNN, responsible for classification. The expansive path replaces pooling layers with upsampling operators, thus giving the network an U-shape, hence its name. The connections between the two paths, seen in figure 4.21 and similar to the connections of residual blocks described in section 3.2, improve the network in two ways: feeding context from the contracting path to the expansive path by combining high resolution features with upsampled outputs; and better gradient flow between layers during backpropagation.

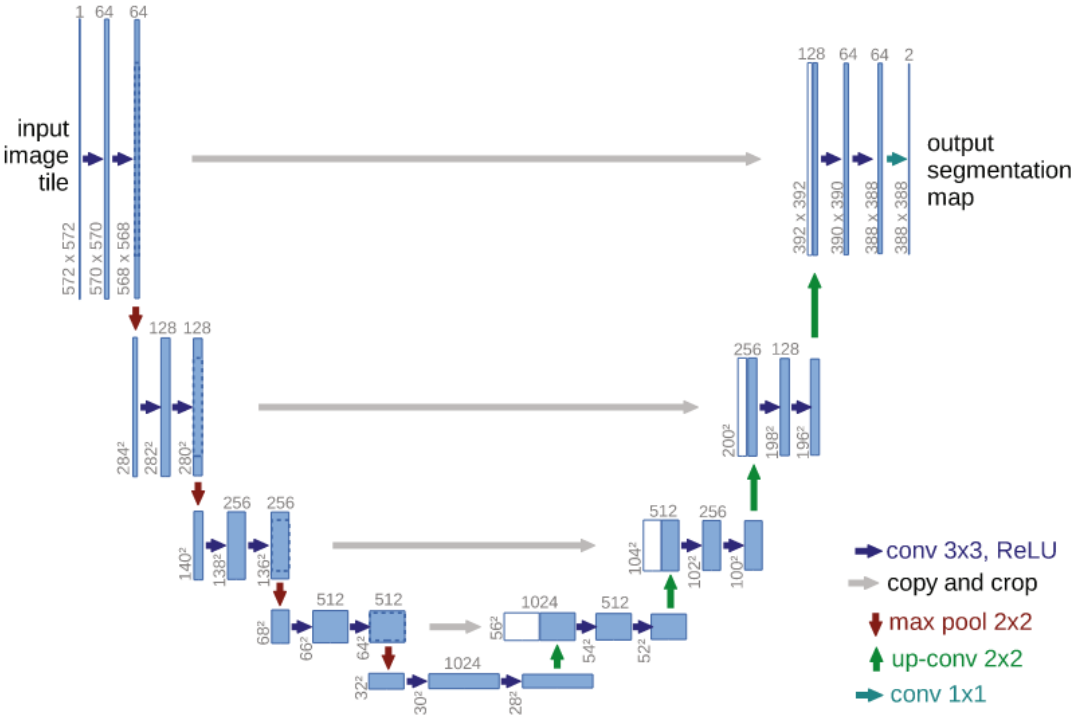



FIGURE 4.21: First U-Net architecture proposed (adapted from [39]).

The upsampling operator (available in Keras) increases the size of the dimensions of a feature map by the given size factors. For typical U-Net models, the $2 \times 2$ upsampling layers double the number of rows and columns of a map. The algorithm used for this expansion depends on the interpolation argument. For example, the bilinear interpolation fills empty spaces with the average of the filled surrounding values. In the original U-Net architecture, this operator is followed by a $2 \times 2$ convolution layer (creating the "up-convolution" operation [39]) which halves the number of feature channels to match the number of feature channels of the feature map from the corresponding step in the contracting path, before their concatenation. The up-convolution operations are followed by two $3 \times 3$ convolutions, activated by the *ReLU* function.

The final layer of the network is a $1 \times 1$ convolution layer specified with the number of filters to map each input map to the desired number of classes (or labels). This layer is coupled with an activation function, usually the *sigmoid* function (equation 3.8), and outputs a segmentation mask representing the pixel-wise classification [39]. For the network to better learn positional invariance, and due to typically being trained with small training sets, the U-Net network usually relies on data augmentation of its input data. The architecture outperformed other proposed networks for the same task with reasonable training times and has been a staple in the field of image segmentation since its inception.

### 4.5.1 Model Architecture

The U-Net model described in this subsection was adapted from an earlier model devised at SMARTEX for a previous project which also entailed image segmentation (where the goal was to classify which parts of an image contain fabric). The model's contracting path consists of a pretrained MNV2 model without a final fully connected layer (i.e. dense layer) which outputs 1280 features per data point. The activated outputs of specific inverted residual blocks of the MNV2 (blocks 13, 6, 3, and 1) are also stored for the connections between the contracting and expansive paths via concatenation.

The initial 3-channel input is cast to the data type *float32* and scaled to values between -1 and 1. For the expansive path, the output of the MNV2 model is repeatedly passed through a $2 \times 2$ upsampling layer with a bilinear interpolation, then concatenated to the corresponding MNV2 block output (in the order given above), and passed through a custom 2D convolution block with a predefined number of filters. The final output of this process is also upsampled and passed through a $1 \times 1$ convolution layer with 1 filter activated by the *sigmoid* function.

The custom 2D convolution block consists of a 2D spatial dropout layer, followed by two equal layers. These layers first pad their input and then pass it through a separable 2D convolution layer with a $3 \times 3$ kernel. Optionally, a batch normalization layer, with default momentum of 0.99, is used before the output is activated by the *ReLU6* function (equation 3.7). The output of the first of these two layers is stored as a residual. The final output of the custom convolution block is the addition of the second of these two layers with this residual. For a more concise description of the U-Net model architecture, refer to the algorithms C.10 and C.11 of appendix C, presented in Python for their extensive use of TensorFlow and Keras methods.

The model was trained for 5 epochs with a learning rate of $4 \times 10^{-3}$ and was fine-tuned for another 55 epochs with a learning rate of $8 \times 10^{-5}$, a dropout rate of 0.1, a batch size of 64, and used batch normalization in the custom convolution blocks. 128 filters were used for the first custom convolution block, 64 for the second block, and 32 for the last two custom blocks. The training data consisted of 5145 samples (80% for training and 10% each for validation and testing), augmented by typical processes: random brightness, Gaussian noise, salt and pepper noise, random quality, random flip, and random perspective. The model was compiled with the *Adam* optimizer (described in subsection 3.3.1), as well as a custom loss function based on F1 scores internally named macro soft F1 score. The metric used was Keras' implementation of the binary Intersection-Over-Union (IoU) metric (*BinaryIoU*).

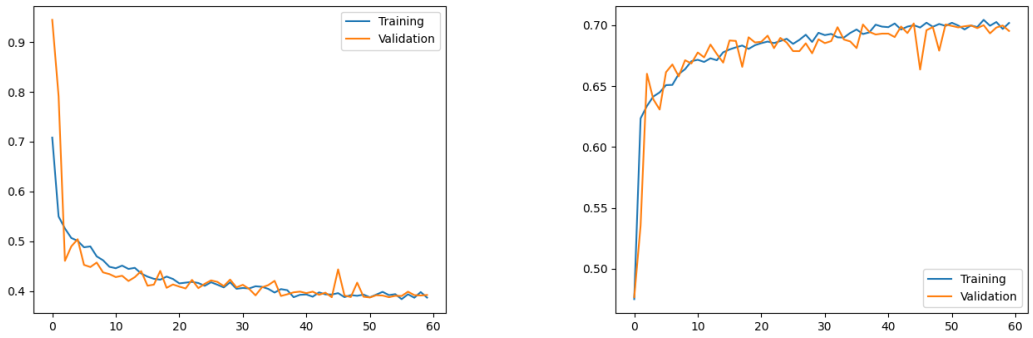
### 4.5.2 Metrics and Results

The IoU metric is a common metric for image segmentation problems. For one class the metric is simply defined as $IoU = TP/(TP + FP + FN)$, where $TP$ stands for true positives in a confusion matrix, $FP$ false positives, and $FN$ false negatives. Given a threshold to convert *logit* predictions for a binary classification task, IoUs are computed for both classes and the mean of these values is returned as the final metric [40].

The loss function used is a custom adaptation of the F1 score metric, which combines information about the precision and the recall of a model. This custom score is defined as $MSF1 = 2 \times TP/(2 \times TP + FP + FN)$. The final loss function is given by $1 - MSF1$ such that by reducing it, the macro soft F1 score increases. Both this score and the IoU metric are appropriate in the context of defect detection and segmentation due to prioritizing the correct classification of positives over the correct classification of negatives, or over a more balanced approach, as with the MCC metric (equation 3.9) used for image classification.

The fitting of the model ran with a model checkpoint callback which saves the weights of each epoch, a local logger callback for metric and loss values, and the TensorBoard log callback, in addition to a memory garbage collector. Epoch 56 was selected as the best epoch and achieved an accuracy of 48.55% on the test set. This epoch sustained values of 0.384 and 0.390 for the loss function, and values of 0.704 and 0.700 for the IoU, for training data and validation data, respectively. Figure 4.22 depicts these values for every epoch and shows a stabilization over the last 10 epochs, which indicates this model's

performance would not improve significantly with more fine-tuning, barring architecture changes or the addition of training samples.




(A) Macro soft F1 loss function. (B) *BinaryIoU* metric.

FIGURE 4.22: Plots of loss function and metric values by epoch.

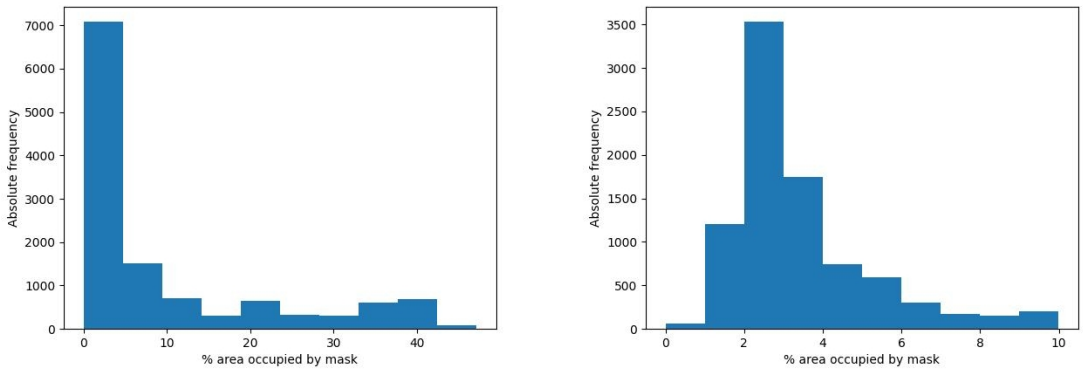
The network could, however, be improved by sequentially training it on more accurate datasets, given by the addition of the labels of the previous model to the set of true labels (the accepted labels from the simple classifier of section 4.1 and/or the hand-drawn labels with the tool of section 4.2). Nevertheless, this step did not fit into our internship schedule and was set aside for future experimentation within SMARTEX.

Influenced by the metric of number of classified lines used to evaluate the simple classifier of section 4.1, one metric which seemed natural to evaluate the labels of the whole dataset, obtained with the weights of the best training epoch, was the percentage of the total area of an image occupied by its mask. Given that every image in the dataset is assumed to have at least a defect, we would expect this metric to be around 1%–5% for every label. To clarify, assuming a mask with one perfectly vertical line 10 pixels wide, the percentage of the area of the image occupied by the mask would be $(10 \times 480)/(640 \times 480) \approx 1.56\%$. Binary classification of the pixel labels given by the *sigmoid* activation function were obtained with a standard threshold of 0.5.

Figures 4.23 and 4.24 display histograms of the results of this metric for the whole dataset and for the dataset that excludes the lyocell yarn, which was the worst-performing yarn material. Histogram (B) of each figure limits the sample to images that resulted in a mask which occupies less than 10% of the area of its image. Inspecting these histograms, our expectations were met regarding the most common interval of this metric.

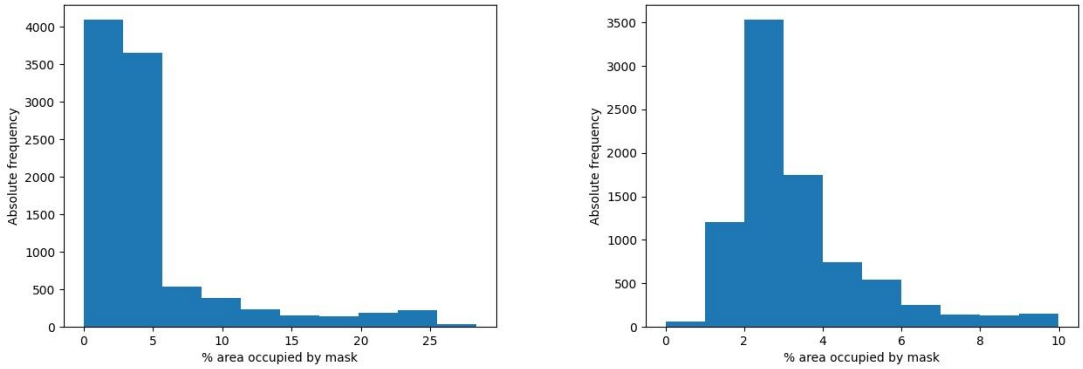
In fact, 7287 images resulted in a label that was in the 0%–5% range of this metric (excluding empty labels). Out of 12245 total samples, approximately 59.51% of them seem

to be labeled correctly, and the, albeit limited, eye tests corroborated this claim. In addition, the next interval with the most frequency, the 5%–10% interval, is comprised of 1416 images, around 11.56% of the total set. According to a limited eye test with the use of the graphic tool of section 4.2, labels belonging to this range were correct more often than not.




(A) Full histogram. (B) Zoomed histogram.

FIGURE 4.23: Histograms of percentage of area covered by mask.

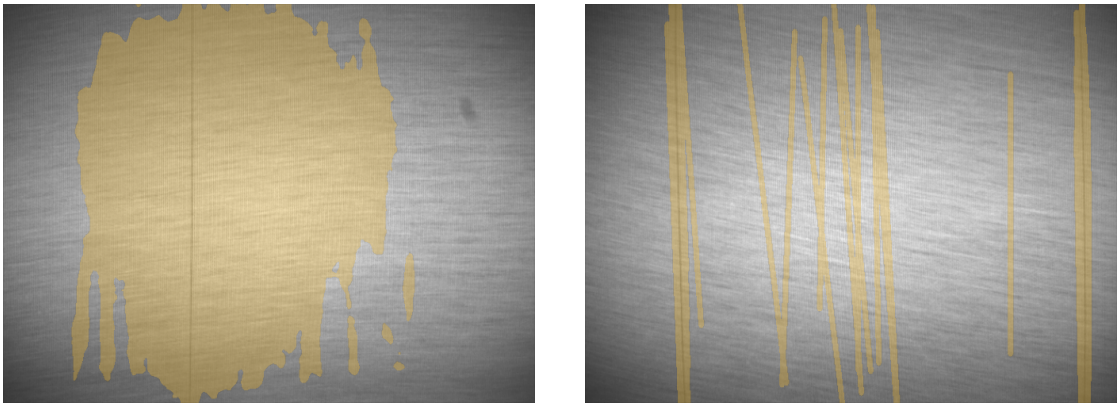



(A) Full histogram. (B) Zoomed histogram.

FIGURE 4.24: Histograms of percentage of area covered by mask (w/o lyocell).

When we analysed the images of the worst performing labels concerning this metric, we quickly realized most of these labels were outputs of images of the lyocell yarn. In fact, of the 500 worst performing labels, all of them were from images of this material. Examining some of the original images containing this type of yarn material (labeled examples are portrayed in figure 4.25), however, it seems expected our models would have a difficult time correctly labeling them due to the salient vertical ridges of this material.

Another factor could be the lack of labels available for training containing this material due to the inability of the simple classifier concerning the lyocell yarn. While this material



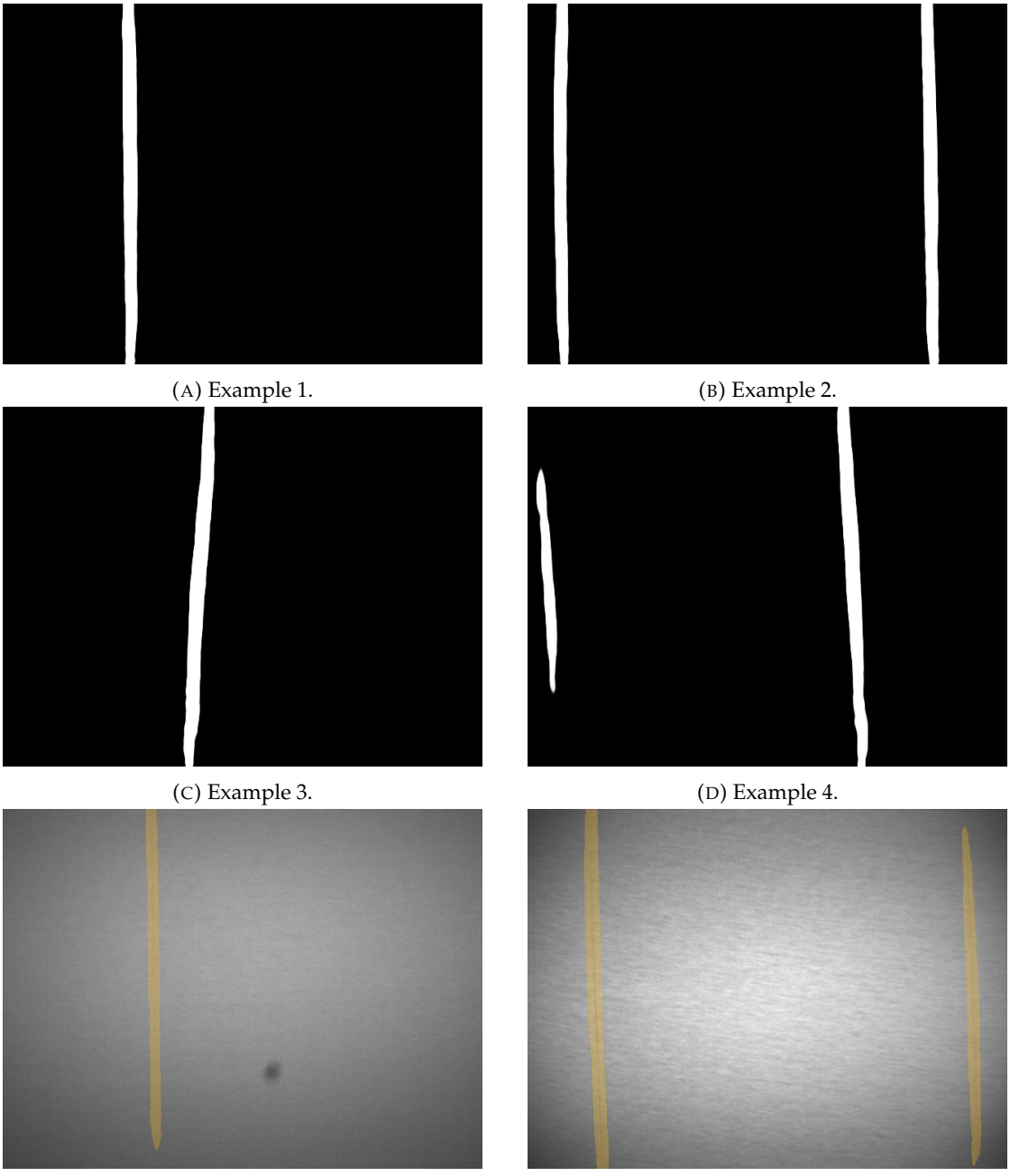
(A) Labeled by the U-Net. (B) Labeled by the simple classifier.

FIGURE 4.25: Examples of labeled samples of the lyocell yarn.

accounted for around 21% of the complete dataset, it only accounted for around 3.63% (187 samples) of the training dataset for the U-Net classifier.

Comparing histograms (A) of figures 4.23 and 4.24, the maximum value of this metric decreases substantially (from approximately 47.19% to around 28.36%), while the frequencies of the lowest intervals remain largely unaffected, as evidenced by comparing histograms (B). If the lyocell yarn is disregarded from the dataset, the share of labels in the 0%–5% interval jumps from 59.51% to approximately 75.34%. Labels from the 5%–10% interval of this metric account for around 12.64% of the total, resulting in nearly 88% of possibly accurate labels.

This value, coupled with an eye test performed on a limited sample of the labels, indicates that this model mostly performed as intended and is, without a doubt, the best-performing model created for this project. Figure 4.26 exhibits some of the best masks produced by the U-Net, including masks for distinct images with two *vertical* defects, while figure 4.27 displays some of the worst masks produced by the model. These examples include final binary masks as well as labeled images for a clearer understanding of the defect detection of this final model.

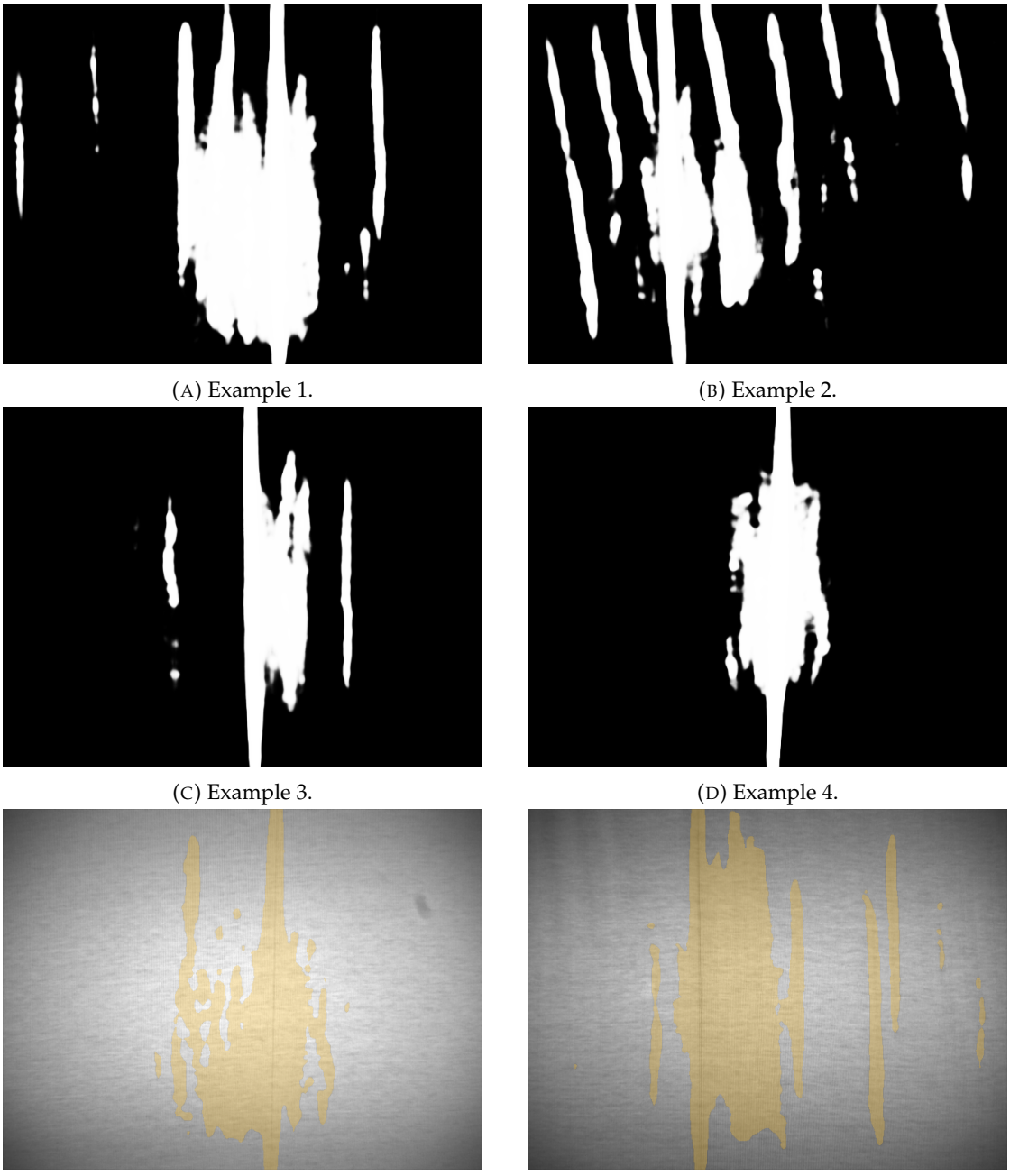


(A) Example 1. (B) Example 2.

(C) Example 3. (D) Example 4.

(E) Example 5. (F) Example 6.

FIGURE 4.26: Examples of the best masks produced by the U-Net.



(A) Example 1. (B) Example 2.

(C) Example 3. (D) Example 4.

(E) Example 5. (F) Example 6.

FIGURE 4.27: Examples of the worst masks produced by the U-Net.

# Chapter 5

# Final Remarks

This internship project was decidedly a great and vast learning experience for my own personal and professional growth. In addition to getting used to the full-time work routine and the added personal responsibility that invariably comes with it, I was flooded with knowledge one could only get in a professional environment, geared for an industry-level output. I was fortunate to work with a wide variety of current technologies like TensorFlow and Keras, OpenCV, pandas, and SQLite, while being fully integrated in a developer environment well-suited to the technologies' and the company's needs. Regarding this developer environment, my knowledge and skill greatly improved in using a Linux operating system and Git/GitLab for code integration in a team project.

I was also often astonished when witnessing first-hand the current capabilities of industry-standard machine learning models, and, by extension, the immense potential of predictive models as new and improved hardware and theory comes into fruition, both in the wildly variable applications of these models as well as in their processing power and speed. With the use of dedicated and powerful machines and GPUs, and complex and thought-out model pipelines, as well as a massive and continuously updated dataset, their levels of accuracy are nowhere near the "good-enough" accuracy rates (most around 75%) I would often see in my own university projects.

Witnessing the potential of industry-level artificial intelligence definitely inspired me to keep learning and creating in this area. This experience certainly cemented this area of mathematical applications as one of the areas I will delightfully focus on the most throughout my professional career, and undoubtedly in my personal projects. I hope one day I have the opportunity and the means to use my knowledge in some way so as to make the world a better place.

## 5.1 Future Work

As most projects, the main project of this internship experience, or any of the on-boarding projects, would have certainly fared better given more time to develop. Some of the things I would have liked to have the time to do are: optimizing the algorithm responsible for drawing lines in the graphic tool for human labeling, to streamline human labeling; fitting the distributed gradient boosting model to compare with the results of the random forest (although more out of curiosity in the boosting methodology, seeing as though the U-Net model would still certainly perform better for most of the same reasons it out-performs the random forest model); and tinkering with the U-Net, its layers and hyper-parameters.

Nevertheless, the main project of this thesis is surely a solid foundation and guideline for any future work at SMARTEX involving label generation pertaining to specific kinds of defects. In addition, our algorithms have the potential to be repurposed, almost effortlessly, to detect horizontal defects and perhaps oil defects as well (these defects appear as dark circles in the fabric and obtaining an initial labeling akin to the goal of section 4.1 could be attainable via circle detectors that OpenCV implements, similar to the line detectors we used and referenced in subsection 4.1.2).

The other two projects of this internship experience could certainly be built upon and improved too. First, given the loader framework already present at SMARTEX, more methods could be straightforwardly introduced to bring our limited caching loader on par with the more developed and utilized image loaders. For example, more direct and common features in SMARTEX's loaders like PUT (i.e. insert a new object into loader, overwriting any previous object with the same key) or POP (i.e. remove an object and return it) were not implemented, even though some of the methods which we created do the same things but in different conditions.

Second, all the research and work done on the pruned network model can undoubtedly prove useful in later experiments with the production model. Given more time, and support by the TensorFlow and Keras libraries concerning pruning, this model could find its way into the production stage if it meets its envisioned performance advantages in inference speed over the main production model.

Coming back to the main project however, these kinds of "isolated" algorithms and models (meaning they concentrate on a limited set of the fabric defects), which are simpler than the complex production model, could also be implemented "in-the-field" directly, if

proven efficient and fast enough. Aside from being used for their main purpose of one-time label generation for an existing dataset, these models would have the ability to stop the factory machines or warn the operators by testing quickly for their specific assigned defects, before handing the problem over to the more complex model.

Using simpler and less resource-intensive predictors in conjunction with the production CNN could possibly cut costs, both in time and power, as well as in fabric defects, the ultimate goal of SMARTEX. These aspirations, however, would require further study and experiments, encompassing various technologies and teams, that will be left for future works at SMARTEX.

# Appendix A

# Limited Caching Loader

**Algorithm A.1** Limited caching loader method

```
1: procedure _GETSIZE
2:     execute select statement for size in table metadata
3:     fetch data to db_size
4:     return db_size
5: end procedure
```

# Appendix B

# Convolutional Neural Networks

**Algorithm B.1** Final section of the model

```
x = Input((2, None, None, 3))
y = 2 * x - 1
y = DistributedLayer(base_model)(y)
y = Flatten()(y)
y = prune_low_magnitude(Dense(n_cls, activation="sigmoid"),
        **pruning_params)(y)
model = Model(x, y, model_name=model_name)
```

**Algorithm B.2** Input creation and training model

```
x = Input((2, None, None))
y = x
y = tf.expand_dims(y, axis=-1)
y = tf.repeat(y, repeats=3, axis=-1)
y = tf.cast(y, dtype=tf.float32) / 255.0
y = model(y)

gym = Model(x, y, name="training_setup", model_name=model_name)
```

# Appendix C

# Needle Defect Segmentation

$$
\begin{aligned}
g(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \\
&= \frac{1}{(\sqrt{2\pi}\sigma)^2} e^{-\frac{x^2}{2\sigma^2} - \frac{y^2}{2\sigma^2}} \\
&= \frac{1}{(\sqrt{2\pi}\sigma)^2} e^{-\frac{x^2}{2\sigma^2}} e^{-\frac{y^2}{2\sigma^2}} \\
&= \frac{1}{(\sqrt{2\pi}\sigma)} e^{-\frac{x^2}{2\sigma^2}} \times \frac{1}{(\sqrt{2\pi}\sigma)} e^{-\frac{y^2}{2\sigma^2}} \quad \square
\end{aligned}
\tag{C.1}
$$

**Algorithm C.1** Conversion to binary

```
1 def ConvertToMask(image):
2     converted = numpy.zeros((480,640), numpy.uint8)
3
4     converted[image[:,:,0] != 0] = 1
5     return converted
```

**Algorithm C.2** Conversion to RGBA

```
1 def ConvertToRGBA(mask):
2     converted = numpy.zeros((480,640,4), numpy.uint8)
3
4     converted[mask[:,:] >= 0.5] = (245, 185, 40, 80)
5     return converted
```

**Algorithm C.3** Line detectors wrapper method

```
1: procedure INITIALIZE(detector, blur, image, **kwargs)
2:     set self.lines as empty tensor of shape (0, 1, 4)
3:     self.blur ← blur
4:     if image was introduced then
5:         self.image ← image
6:         if self.image has 3 colour channels then
7:             transform self.image to grayscale image
8:         end if
9:         if self.blur is True then
10:            apply Gaussian blur to self.image
11:        end if
12:    end if
13:    self.SET_DETECTOR(detector, **kwargs)
14:    if self.line_detector is not set then
15:        initialize the main line detector with predefined arguments to self.line_detector
16:    end if
17:    if there are masks in mask folder then
18:        store the number of masks in mask folder to self.save_id
19:    else
20:        self.save_id ← 0
21:    end if
22: end procedure
```

**Algorithm C.4** Line detectors wrapper method

```
1: procedure SET_IMAGE(image)
2:     self.image ← image
3:     return self.image
4: end procedure
```

**Algorithm C.5** Line detectors wrapper method

```
1: procedure SET_DETECTOR(detector, **kwargs)
2:     set self.edge_detector as None
3:     set self.houghline_detector as None
4:     if detector == 'ED' then
5:         initialize the edge drawing detector to self.edge_detector
6:     else if detector == 'HL' then
7:         self.houghline_detector ← HOUGHLINE(False)
8:     else if detector == 'HL_P' then
9:         self.houghline_detector ← HOUGHLINE(True)
10:    end if
11:    if **kwargs were introduced then
12:        initialize the main line detector with **kwargs to self.line_detector
13:    end if
14: end procedure
```

**Algorithm C.6** Hough line wrapper method

**procedure** STANDARD
  $lines \leftarrow$ CV2.HOUGHLINES$(self.edges, 1, \frac{\pi}{180}, 100)$
  **if** *lines* is not None **then**
    **for** $i \leftarrow 0, length(lines) - 1$ **do**
      $\rho \leftarrow lines[i][0][0]$
      $\theta \leftarrow lines[i][0][0]$
      $a \leftarrow \cos\theta$
      $b \leftarrow \sin\theta$
      $x \leftarrow a * \rho$
      $y \leftarrow b * \rho$
      $pt_1 \leftarrow (x + 1000 \times (-b), y + 1000 \times a)$ ▷ Both tuple elements as integers
      $pt_2 \leftarrow (x - 1000 \times (-b), y - 1000 \times a)$ ▷ Both tuple elements as integers
      **concatenate** $pt_1, pt_2$ to $self.new\_lines$ as a tensor of shape (1, 1, 4)
    **end for**
  **end if**
  **return** $self.new\_lines$
**end procedure**

**Algorithm C.7** Hough line wrapper method

**procedure** PROBABILISTIC
  $minLineLength \leftarrow 25$
  $maxLineGap \leftarrow 150$
  $lines \leftarrow$ CV2.HOUGHLINESP$(self.edges, 1, \frac{\pi}{180}, 100, minLineLength, maxLineGap)$
  **if** *lines* is not None **then**
    $self.new\_lines \leftarrow lines$
  **end if**
  **return** $self.new\_lines$
**end procedure**

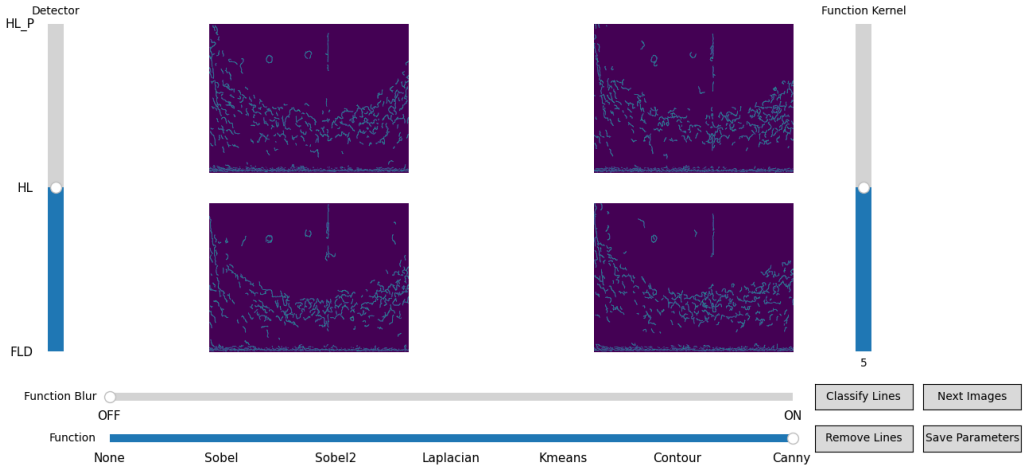



FIGURE C.1: Tool interface and visualization of processed images with one of the best hyper-parameter sets.

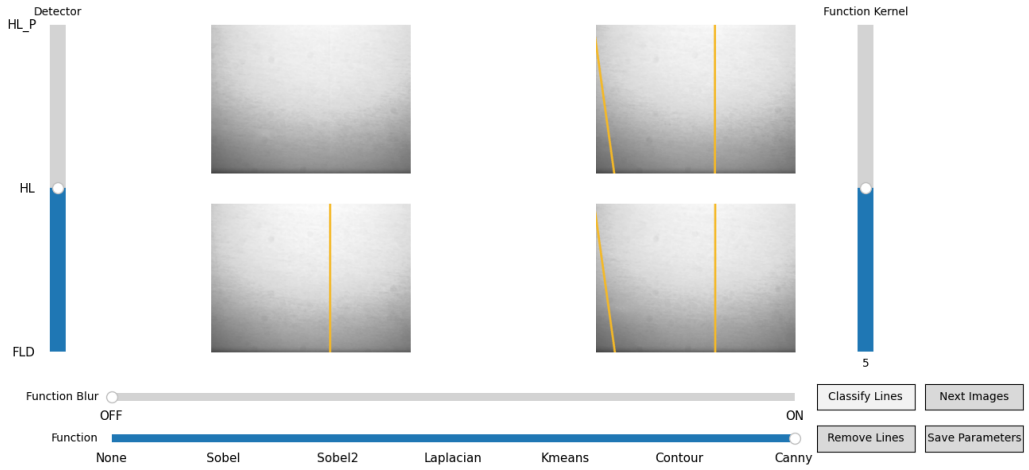



FIGURE C.2: Tool interface and visualization of classified images with one of the best hyper-parameter sets.

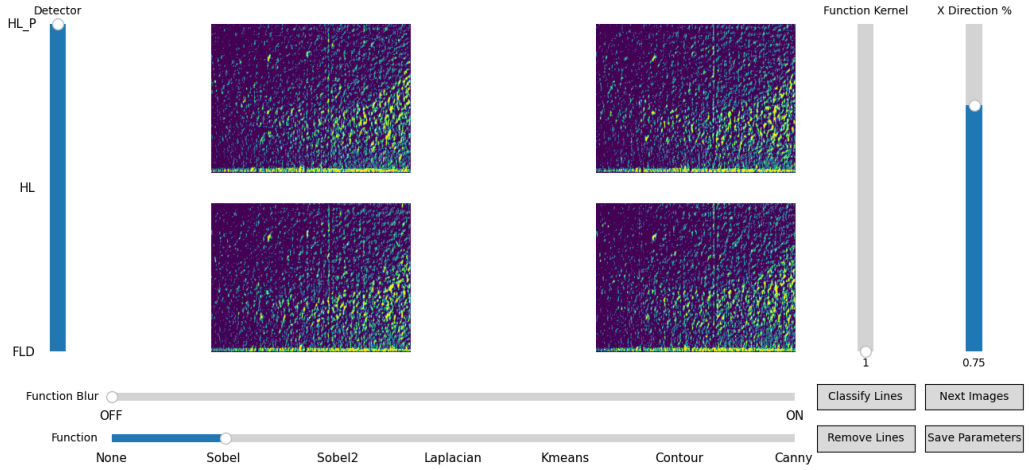



FIGURE C.3: Tool interface and visualization of processed images with one of the best hyper-parameter sets.

**Algorithm C.8** MNV2 model architecture

```
def mnv2_model():
    zoom = 0.75
    height = int(960 * zoom)
    width = int(1280 * zoom)
    mnv2 = MobileNetV2(alpha=.35, pooling='avg', include_top=False)
    x = Input((height, width, 3))
    y = 2 * x - 1
    y = data_augmentation(y, zoom=zoom)
    y = mnv2(y, training=False)
    model = Model(x, y)

    model.summary()
    return model, mnv2
```

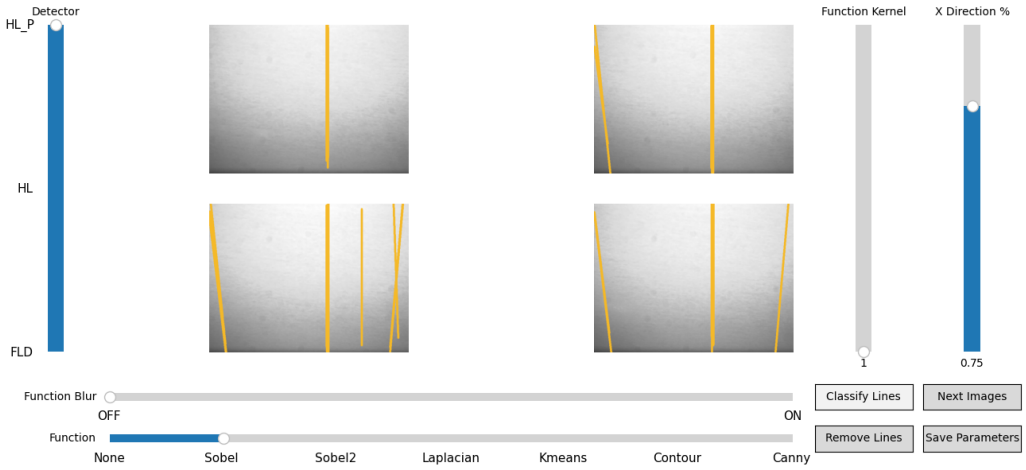



FIGURE C.4: Tool interface and visualization of classified images with one of the best hyper-parameter sets.

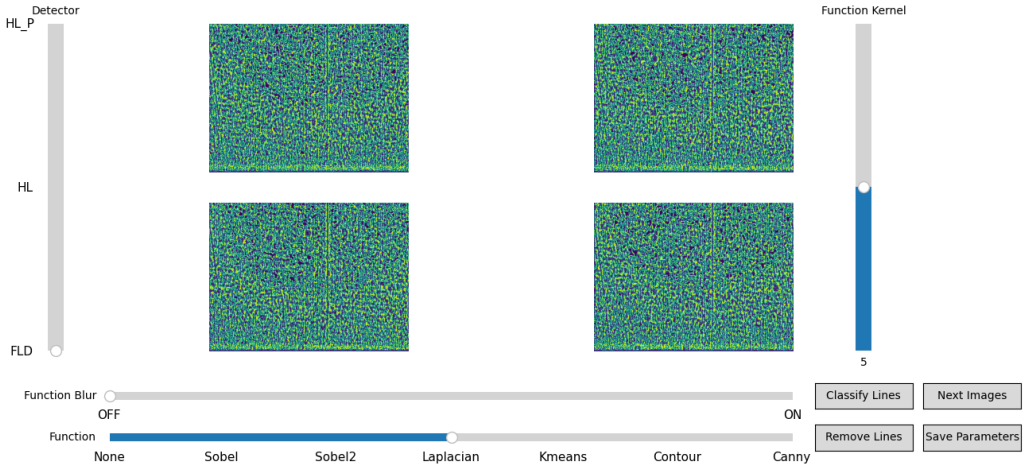



FIGURE C.5: Tool interface and visualization of processed images with one of the best hyper-parameter sets.

**Algorithm C.9** Random forest classifier

```
RandomForestClassifier(n_estimators=100, min_samples_split=2,
                       min_samples_leaf=1, max_features='auto',
                       oob_score=True, max_depth=50, criterion='gini',
                       bootstrap=True, random_state=7, n_jobs=-1, verbose=1,
                       class_weight={0:0.01, 1:0.99})
```

**Algorithm C.10** U-Net model architecture

```
def unet_model(name, dropout=0.5, n_filters=16, batchnorm=True):
    zoom = 0.5
    width = int(1280 * zoom)
    height = int(960 * zoom)
    mnv2 = MobileNetV2(input_shape=(height, width, 3), alpha=0.35,
                       pooling=None, include_top=False)
    model = Model(mnv2.input, [
        mnv2.output,
        mnv2.get_layer('block_13_expand_relu').output,
        mnv2.get_layer('block_6_expand_relu').output,
        mnv2.get_layer('block_3_expand_relu').output,
        mnv2.get_layer('block_1_expand_relu').output
    ], model_name=name + '_mnv2')

    x = Input((height, width))
    y = tf.expand_dims(x, axis=-1)
    y = tf.repeat(y, repeats=3, axis=-1)
    y = tf.cast(y, tf.float32) / 255.0
    y = 2 * y - 1

    y, c4, c3, c2, c1 = model(y, training=False)

    u6 = UpSampling2D(size=(2, 2), interpolation="bilinear")(y)
    u6 = concatenate([u6, c4])
    c6 = conv2d_block(u6, n_filters=n_filters * 8, dropout=dropout,
                      kernel_size=3, batchnorm=batchnorm)

    u7 = UpSampling2D(size=(2, 2), interpolation="bilinear")(c6)
    u7 = concatenate([u7, c3])
    c7 = conv2d_block(u7, n_filters=n_filters * 4, dropout=dropout,
                      kernel_size=3, batchnorm=batchnorm)

    u8 = UpSampling2D(size=(2, 2), interpolation="bilinear")(c7)
    u8 = concatenate([u8, c2])
    c8 = conv2d_block(u8, n_filters=n_filters * 2, dropout=dropout,
                      kernel_size=3, batchnorm=batchnorm)

    u9 = UpSampling2D(size=(2, 2), interpolation="bilinear")(c8)
    u9 = concatenate([u9, c1])
    c9 = conv2d_block(u9, n_filters=n_filters * 2, dropout=dropout,
                      kernel_size=3, batchnorm=batchnorm)

    u10 = UpSampling2D(size=(2, 2), interpolation="bilinear")(c9)
    outputs = Conv2D(1, (1, 1), activation='sigmoid')(u10)[..., 0]

    unet = Model(inputs=x, outputs=outputs, model_name=name)
    return unet, model, mnv2
```

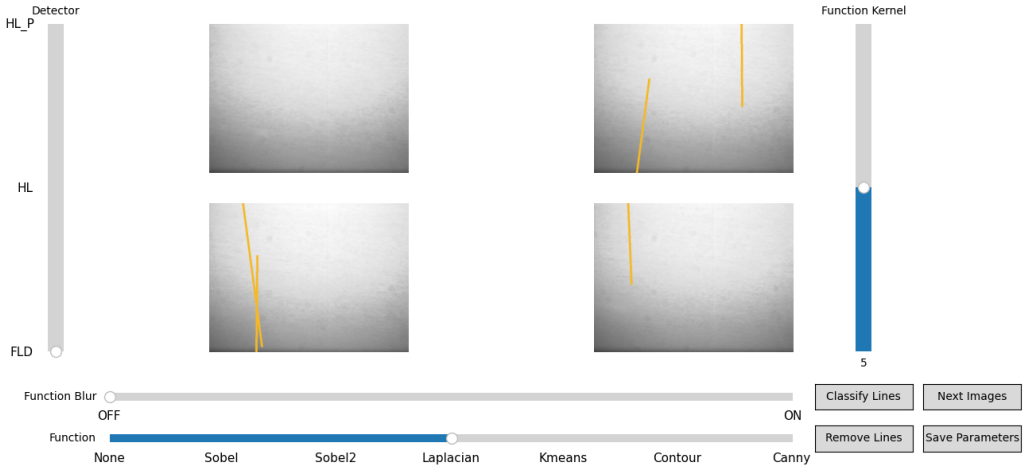
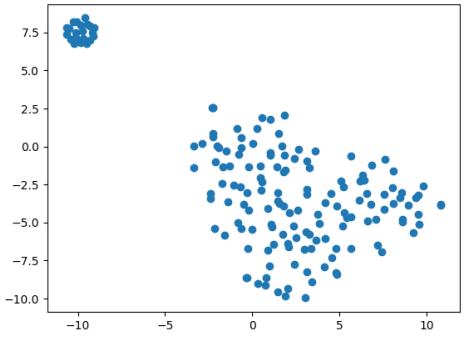


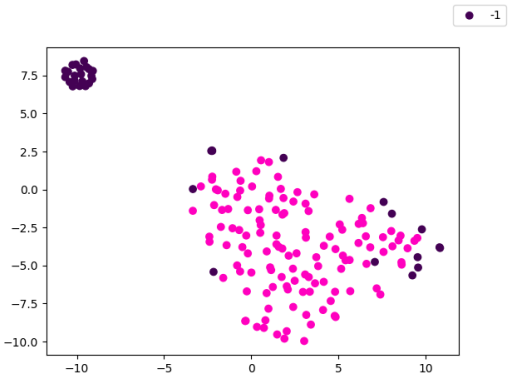

FIGURE C.6: Tool interface and visualization of classified images with one of the best hyper-parameter sets.

**Algorithm C.11** U-Net model block

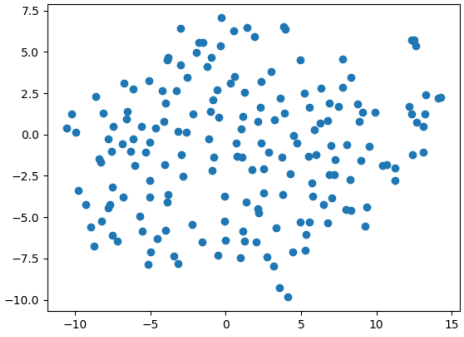
```
def conv2d_block(input_tensor, n_filters, dropout, kernel_size=3,
                 batchnorm=True):
    u = SpatialDropout2D(dropout)(input_tensor)

    # first layer
    x = tf.pad(u, [[0, 0], [1, 1], [1, 1], [0, 0]], mode="SYMMETRIC")
    x = SeparableConv2D(filters=n_filters,
                        kernel_size=(kernel_size, kernel_size),
                        padding="valid")(x)

    if batchnorm:
        x = BatchNormalization()(x)
    x = ReLU(6)(x)

    x_residual = x

    # second layer
    x = tf.pad(x, [[0, 0], [1, 1], [1, 1], [0, 0]], mode="SYMMETRIC")
    x = SeparableConv2D(filters=n_filters,
                        kernel_size=(kernel_size, kernel_size),
                        padding="valid")(x)

    if batchnorm:
        x = BatchNormalization()(x)
    x = ReLU(6)(x)

    x = x + x_residual
    return x
```



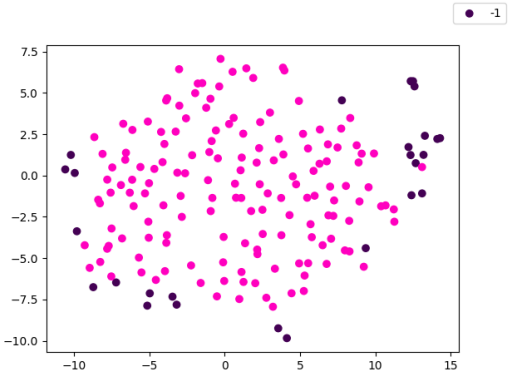
(A) TSNE scatter for group 14.



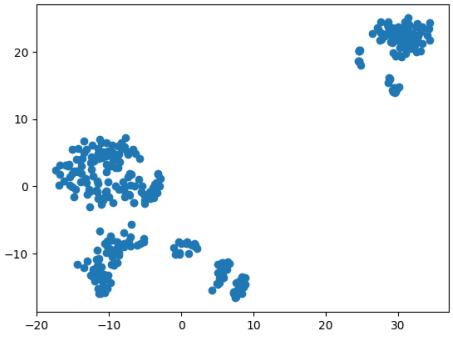

(B) Clustered TSNE scatter for group 14.



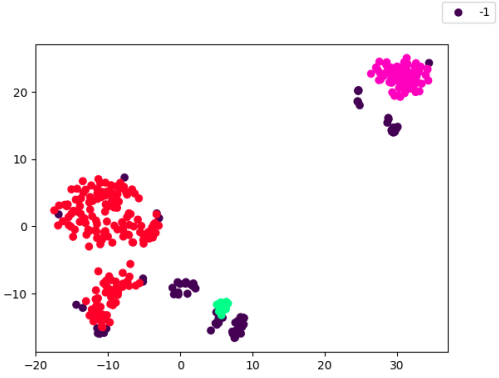
(C) TSNE scatter for group 16.




(D) Clustered TSNE scatter for group 16.

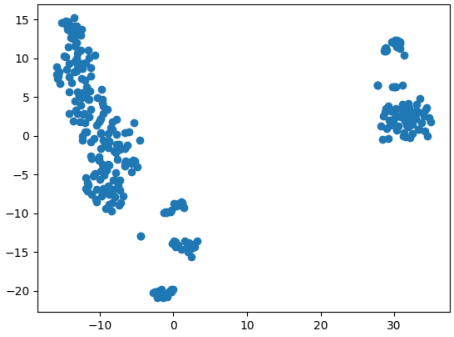


(E) TSNE scatter for group 20.

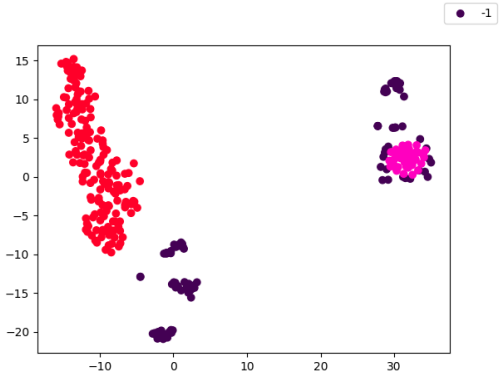



(F) Clustered TSNE scatter for group 20.

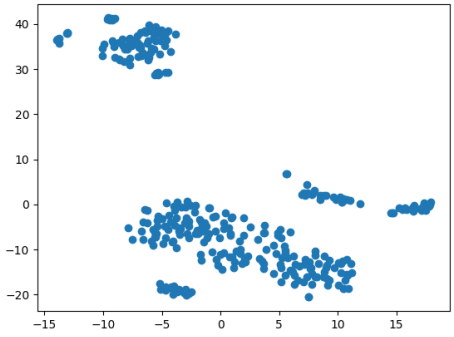
FIGURE C.7: Scatter plots for DBSCAN clusters in the initial groups.



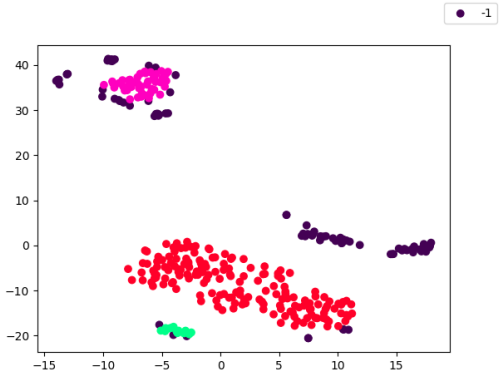
(G) TSNE scatter for group 21.



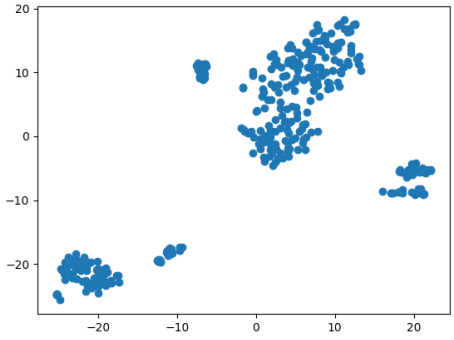

(H) Clustered TSNE scatter for group 21.



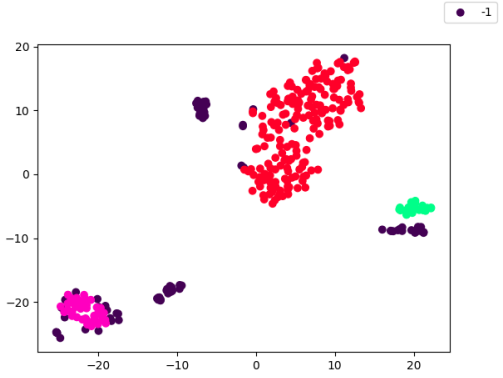
(I) TSNE scatter for group 22.




(J) Clustered TSNE scatter for group 22.

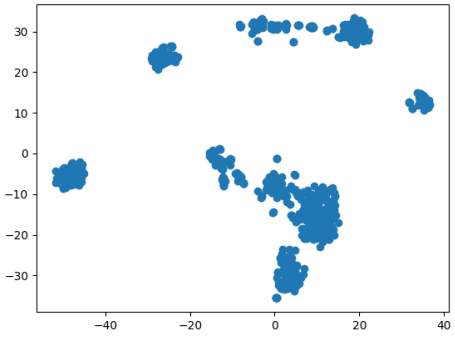


(K) TSNE scatter for group 23.

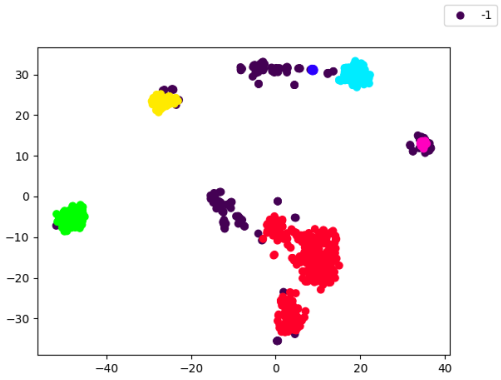



(L) Clustered TSNE scatter for group 23.

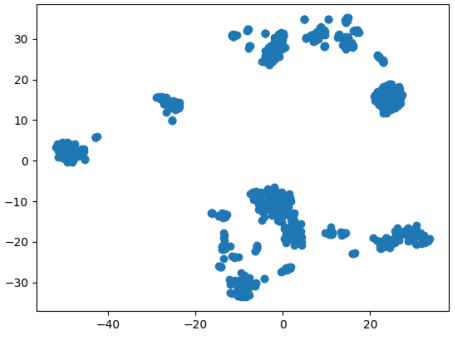
FIGURE C.7: Scatter plots for DBSCAN clusters in the initial groups.



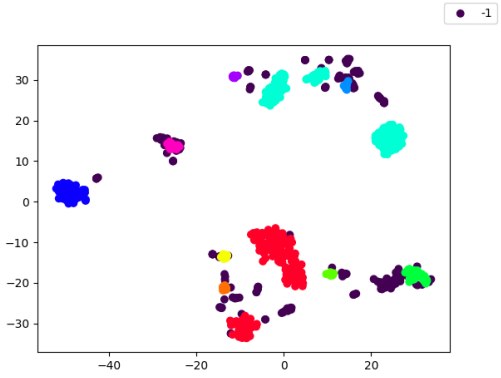
(M) TSNE scatter for group 28.



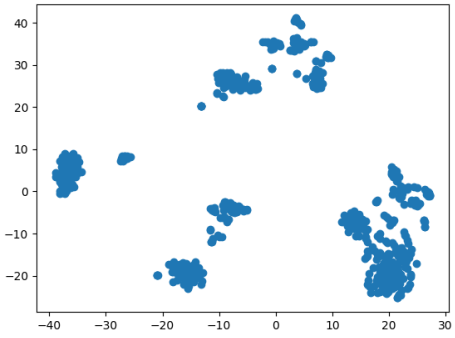

(N) Clustered TSNE scatter for group 28.



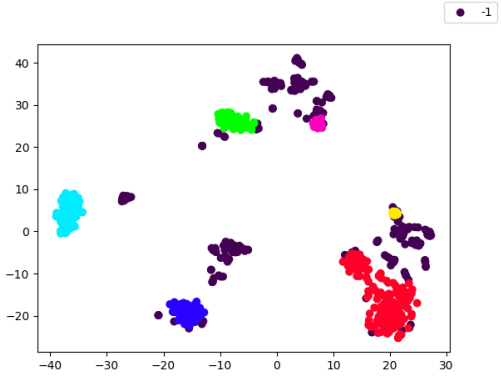
(O) TSNE scatter for group 29.




(P) Clustered TSNE scatter for group 29.

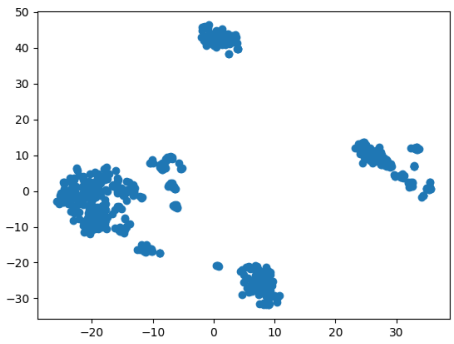


(Q) TSNE scatter for group 32.

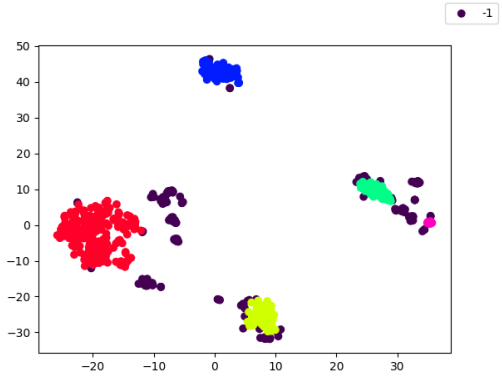



(R) Clustered TSNE scatter for group 32.

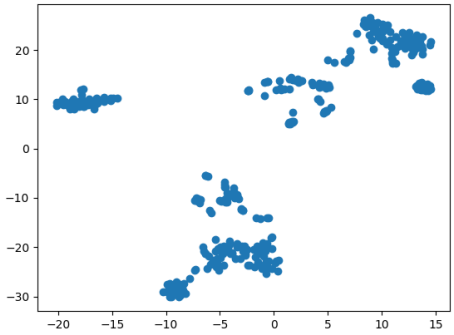
FIGURE C.7: Scatter plots for DBSCAN clusters in the initial groups.



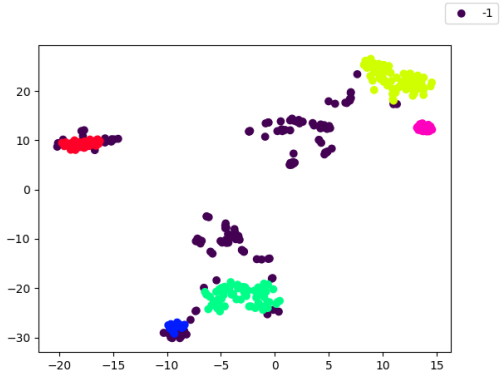

(S) TSNE scatter for group 33.



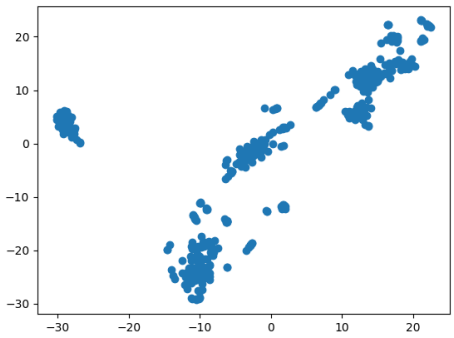

(T) Clustered TSNE scatter for group 33.



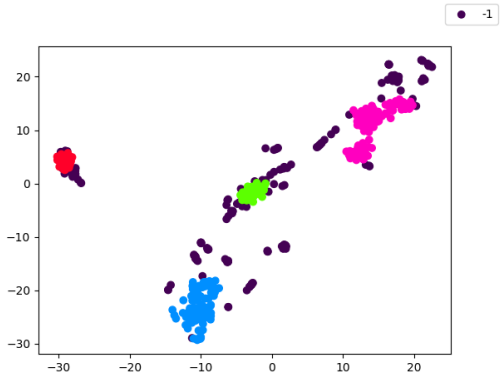

(U) TSNE scatter for group 34.




(V) Clustered TSNE scatter for group 34.

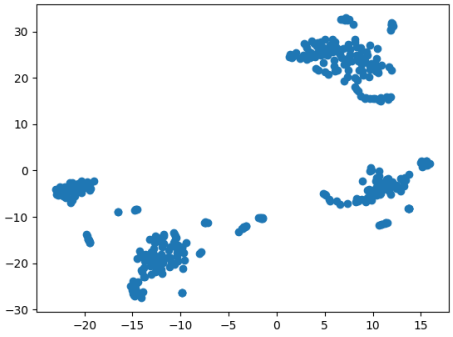



(W) TSNE scatter for group 35.

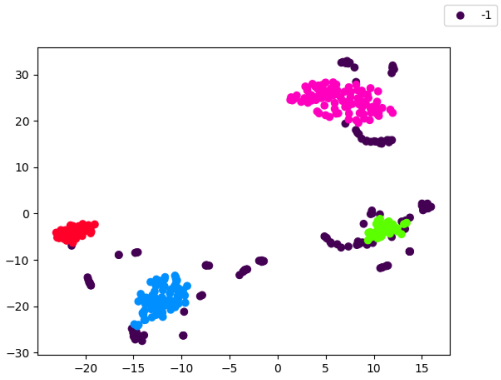



(X) Clustered TSNE scatter for group 35.

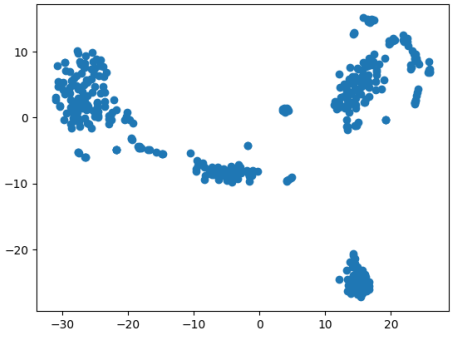
FIGURE C.7: Scatter plots for DBSCAN clusters in the initial groups.



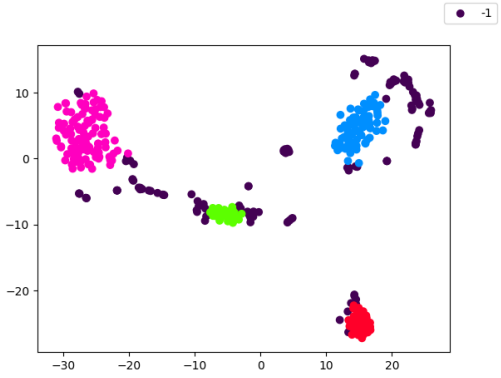
(Y) TSNE scatter for group 36.



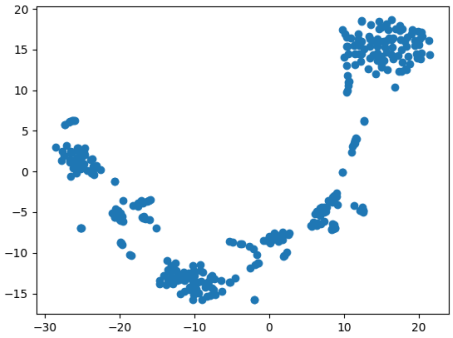

(Z) Clustered TSNE scatter for group 36.



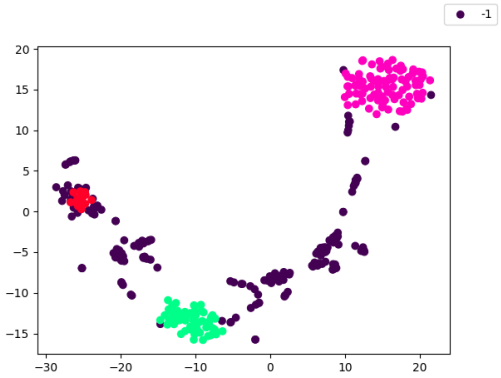
(AA) TSNE scatter for group 37.




(AB) Clustered TSNE scatter for group 37.

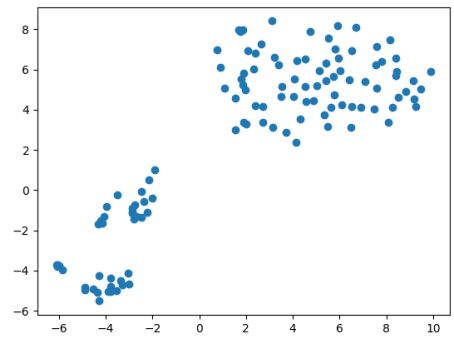


(AC) TSNE scatter for group 38.

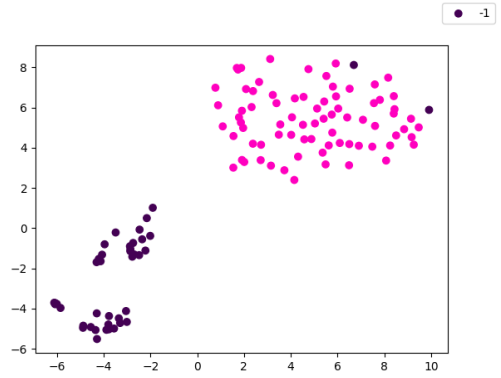



(AD) Clustered TSNE scatter for group 38.

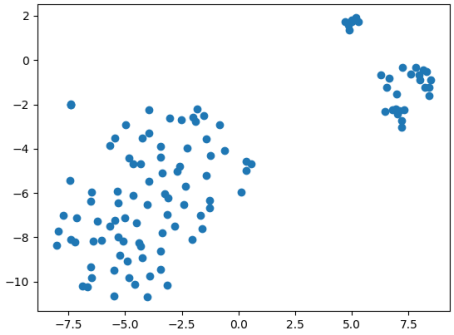
FIGURE C.7: Scatter plots for DBSCAN clusters in the initial groups.



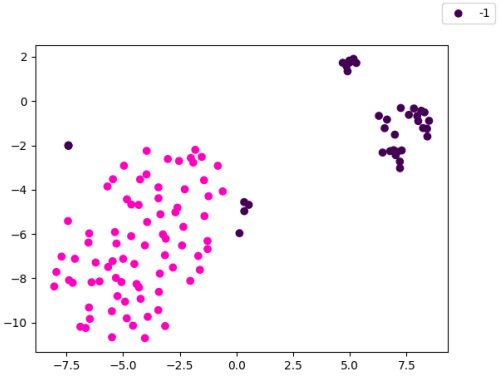
(AE) TSNE scatter for group 40.



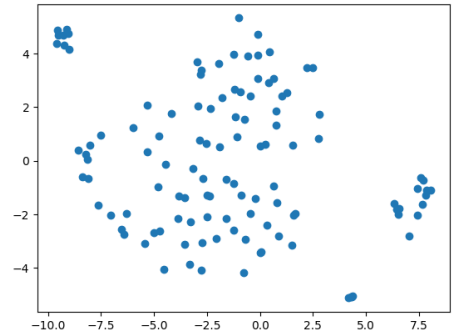

(AF) Clustered TSNE scatter for group 40.



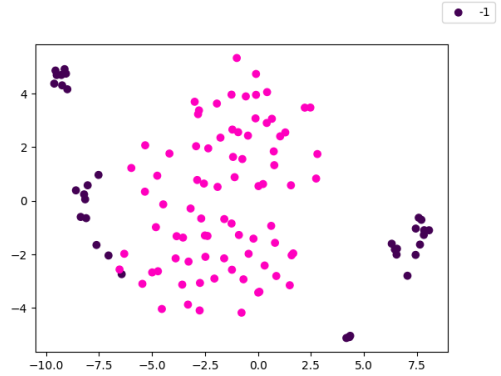
(AG) TSNE scatter for group 41.




(AH) Clustered TSNE scatter for group 41.



(AI) TSNE scatter for group 42.




(AJ) Clustered TSNE scatter for group 42.

FIGURE C.7: Scatter plots for DBSCAN clusters in the initial groups.

# Bibliography


[1] "Sustainability - Smartex," 2022. [Online]. Available: https://smartex.ai/sustainability/ [Cited on page 1.]

[2] R. D. Hipp, "SQLite," 2022. [Online]. Available: https://www.sqlite.org/index.html [Cited on page 3.]

[3] "About SQLite," 2022. [Online]. Available: https://www.sqlite.org/about.html [Cited on page 3.]

[4] "SQLite: Most widely deployed SQL database engine," 2022. [Online]. Available: https://www.sqlite.org/mostdeployed.html [Cited on page 3.]

[5] "SQLite: Transactions information," 2022. [Online]. Available: https://www.sqlite.org/lang_transaction.html [Cited on page 4.]

[6] M. Abadi *et al.*, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. [Online]. Available: https://www.tensorflow.org/ [Cited on page 11.]

[7] F. Chollet *et al.*, "Keras," https://keras.io, 2015. [Cited on page 11.]

[8] K. Fukushima, "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position," *Biological Cybernetics*, vol. 36, pp. 193–202, 1980. [Online]. Available: https://link.springer.com/article/10.1007/BF00344251 [Cited on page 11.]

[9] J. N. Tavares, "Lecture notes from the course 'Statistical Learning'," FCUP, 2022. [Cited on pages xv, 12, 13, 14, 15, 16, and 24.]

[10] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A large-scale hierarchical image database," in *2009 IEEE conference on computer*

*vision and pattern recognition*. IEEE, 2009, pp. 248–255. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/5206848 [Cited on page 16.]

[11] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "MobileNetV2: Inverted residuals and linear bottlenecks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 4510–4520. [Online]. Available: https://arxiv.org/abs/1801.04381 [Cited on pages xv, 16, 17, 18, and 19.]

[12] "TensorFlow: Trim insignificant weights," 2022. [Online]. Available: https://www.tensorflow.org/model_optimization/guide/pruning [Cited on page 19.]

[13] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *2015 International Conference on Learning Representations (ICLR)*, 2015. [Online]. Available: https://arxiv.org/abs/1412.6980 [Cited on page 21.]

[14] "TensorFlow: Pruning in Keras example," 2022. [Online]. Available: https://www.tensorflow.org/model_optimization/guide/pruning/pruning_with_keras [Cited on page 22.]

[15] D. Chicco, N. Tötsch, and G. Jurman, "The matthews correlation coefficient (MCC) is more reliable than balanced accuracy, bookmaker informedness, and markedness in two-class confusion matrix evaluation," *BioData mining*, vol. 14, no. 1, pp. 1–22, 2021. [Online]. Available: https://biodatamining.biomedcentral.com/articles/10.1186/s13040-021-00244-z [Cited on page 23.]

[16] "Scikit-learn: Receiver operating characteristic," 2022. [Online]. Available: https://scikit-learn.org/stable/auto_examples/model_selection/plot_roc.html [Cited on page 25.]

[17] G. Bradski, "The OpenCV Library," *Dr. Dobb's Journal of Software Tools*, 2000. [Online]. Available: https://opencv.org/ [Cited on page 28.]

[18] M. S. Nixon and A. S. Aguado, "Image processing," in *Feature Extraction and Image Processing for Computer Vision*, 4th ed., M. S. Nixon and A. S. Aguado, Eds. Academic Press, 2020, pp. 83–139. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780128149768000038 [Cited on page 30.]

[19] "OpenCV: Image filtering," 2022. [Online]. Available: https://docs.opencv.org/4.x/d4/d86/group__imgproc__filter.html [Cited on pages 30 and 31.]

[20] A. Huamán, "OpenCV: Sobel derivatives," 2022. [Online]. Available: https://docs.opencv.org/4.x/d2/d2c/tutorial_sobel_derivatives.html [Cited on page 30.]

[21] ——, "OpenCV: Laplace operator," 2022. [Online]. Available: https://docs.opencv.org/4.x/d5/db5/tutorial_laplace_operator.html [Cited on page 32.]

[22] J. Canny, "A computational approach to edge detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-8, no. 6, pp. 679–698, Nov 1986. [Online]. Available: https://ieeexplore.ieee.org/document/4767851 [Cited on page 33.]

[23] "OpenCV: Canny edge detection," 2022. [Online]. Available: https://docs.opencv.org/4.x/da/d22/tutorial_py_canny.html [Cited on page 33.]

[24] R. O. Duda and P. E. Hart, "Use of the hough transformation to detect lines and curves in pictures," *Commun. ACM*, vol. 15, no. 1, p. 11–15, jan 1972. [Online]. Available: https://dl.acm.org/doi/10.1145/361237.361242 [Cited on page 39.]

[25] A. Huamán, "OpenCV: Hough line transformation," 2022. [Online]. Available: https://docs.opencv.org/4.x/d9/db0/tutorial_hough_lines.html [Cited on page 39.]

[26] J. Matas, C. Galambos, and J. Kittler, "Robust detection of lines using the progressive probabilistic hough transform," *Computer Vision and Image Understanding*, vol. 78, no. 1, pp. 119–137, 2000. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1077314299908317 [Cited on pages 39 and 40.]

[27] J. H. Lee, S. Lee, G. Zhang, J. Lim, W. K. Chung, and I. H. Suh, "Outdoor place recognition in urban environments using straight lines," in *2014 IEEE International Conference on Robotics and Automation (ICRA)*, 2014, pp. 5550–5557. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/6907675 [Cited on page 40.]

[28] C. Topal and C. Akinlar, "Edge drawing: A combined real-time edge and segment detector," *Journal of Visual Communication and Image Representation*, vol. 23, no. 6, pp. 862–872, 2012. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1047320312000831 [Cited on page 41.]

[29] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos,

D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011. [Online]. Available: https://scikit-learn.org/stable/ [Cited on page 54.]

[30] M. E. Tipping and C. M. Bishop, "Probabilistic principal component analysis," *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, vol. 61, no. 3, pp. 611–622, 1999. [Online]. Available: https://rss.onlinelibrary.wiley.com/doi/abs/10.1111/1467-9868.00196 [Cited on pages 54 and 55.]

[31] N. Halko, P. G. Martinsson, and J. A. Tropp, "Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions," *SIAM Review*, vol. 53, no. 2, pp. 217–288, 2011. [Online]. Available: https://epubs.siam.org/doi/10.1137/090771806 [Cited on page 55.]

[32] L. van der Maaten, G. Hinton, and G. Hinton, "Visualizing data using t-SNE," *Journal of Machine Learning Research*, vol. 9, no. 86, pp. 2579–2605, 2008. [Online]. Available: http://jmlr.org/papers/v9/vandermaaten08a.html [Cited on pages 55 and 56.]

[33] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, "A density-based algorithm for discovering clusters in large spatial databases with noise," in *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, ser. KDD'96. AAAI Press, 1996, pp. 226–231. [Online]. Available: https://dl.acm.org/doi/abs/10.5555/3001460.3001507 [Cited on page 57.]

[34] "Scikit-learn: DBSCAN," 2022. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.cluster.DBSCAN.html [Cited on page 57.]

[35] E. Schubert, J. Sander, M. Ester, H. P. Kriegel, and X. Xu, "DBSCAN revisited, revisited: Why and how you should (still) use DBSCAN," *ACM Trans. Database Syst.*, vol. 42, no. 3, Jul 2017. [Online]. Available: https://dl.acm.org/doi/10.1145/3068335 [Cited on page 57.]

[36] N. Rahmah and I. S. Sitanggang, "Determination of optimal epsilon (eps) value on DBSCAN algorithm to clustering data on peatland hotspots in sumatra," *IOP Conference Series: Earth and Environmental Science*, vol. 31, jan 2016. [Online]. Available: https://iopscience.iop.org/article/10.1088/1755-1315/31/1/012012 [Cited on page 57.]

[37] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, Oct 2001. [Online]. Available: https://link.springer.com/article/10.1023/A:1010933404324 [Cited on pages 58 and 59.]

[38] "Scikit-image: Image processing in Python," 2022. [Online]. Available: https://scikit-image.org/docs/dev/api/skimage.feature.html#skimage.feature.multiscale_basic_features [Cited on page 59.]

[39] O. Ronneberger, P. Fischer, and T. Brox, "U-Net: Convolutional networks for biomedical image segmentation," in *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015*. Springer International Publishing, 2015, pp. 234–241. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-319-24574-4_28 [Cited on pages xv, 62, 63, and 64.]

[40] "TensorFlow: Binary IoU," 2022. [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/keras/metrics/BinaryIoU [Cited on page 65.]