University Institute of Lisbon

Department of Information Science and Technology

Quality Control in Clothing Manufacturing with Machine Learning

Gonçalo Laginha Serafim San-Payo

Dissertation submitted as partial fulfillment of the requirements for the degree of

Master in Computer Engineering

Supervisor

PhD João Carlos Amaro Ferreira, Assistant Professor

ISCTE-IUL

Co-Supervisor

PhD João Pedro Afonso Oliveira da Silva, Assistant Professor

ISCTE-IUL

October 2018

Resumo

O controlo de qualidade é vital para um negócio e a aprendizagem automática tem provado ser bem-sucedida neste tipo de área. Neste trabalho propomos e desenvolvemos um sistema de controlo de qualidade para o fabrico de roupas utilizando aprendizagem automática. O sistema consiste em usar fotografias, tiradas através de dispositivos móveis, para detetar defeitos em peças de roupa. Um defeito pode ser a falta de um componente ou um componente errado numa peça de roupa. A função do sistema é, portanto, classificar os objetos que compõem uma peça de roupa através do uso de um modelo de classificação. À medida que um negócio fabril progride, novos objetos são criados, assim, o modelo de classificação deve ser capaz de aprender as novas classes sem perder conhecimento prévio. No entanto, a maioria dos algoritmos de classificação não suporta um aumento de classes, estes precisam ser treinados a partir do zero com todas as classes. Neste trabalho, utilizamos um algoritmo que suporta aprendizagem incremental para resolver este problema. Este algoritmo classifica características extraídas de imagens das peças de roupa usando uma rede neural convolucional, que tem provado ser uma técnica muito bem sucedida no que toca a resolver problemas de classificação de imagem. Como resultado deste trabalho, desenvolvemos um sistema de controlo de qualidade que combina uma aplicação móvel para tirar fotografias de peças de roupa e um servidor que executa os processos de deteção de defeitos usando um modelo de classificação de imagens preciso, capaz de aumentar o seu conhecimento a partir de novos dados nunca antes vistos. Este sistema pode ajudar as fábricas a melhorar seus processos de controlo de qualidade.

Palavras-chave: Controlo de qualidade, aprendizagem incremental, classicação de imagem.

Abstract

Quality control is vital for business and machine learning has proven to be successful in this type of area. In this work we propose and develop a classification model to be used in a quality control system for clothing manufacturing using machine learning. The system consists of using pictures taken through mobile devices to detect defects on clothing items. A defect can be a missing component or a wrong component in a clothing item. Therefore, the function of system is to classify the objects that compose a clothing item through the use of a classification model. As a manufacturing business progresses, new objects are created, thus, the classification model must be able to learn the new classes without losing previous knowledge. However, most classification algorithms do not support an increase of classes, these need to be trained from scratch with all classes. In this work, we make use of an incremental learning algorithm to tackle this problem. This algorithm classifies features extracted from pictures of the clothing items using a convolutional neural network (CNN), which have proven to be very successfully in image classification problems. As the result of this work, we have developed a quality control system that combines a mobile application to take pictures of clothing items and a server that performs defect detection processes using an accurate image classification model capable of increasing its knowledge from new unseen data. This system can help factories improve their quality control processes.

Keywords: Quality control, incremental learning, image classification.

Acknowledgements

I would like to acknowledge my supervisors, Professors João Ferreira and João Oliveira, for their guidance and support. To all the people at INOV INESC Inovação for providing me a great work environment as well as the tools to develop this work.

A special thanks to my parents, Margarida and Miguel, for motivating me to achieved greater goals.

And last, but not least, to my closest friends that shared long nights of writing with me. Without them this thesis would not be finished on time.

Contents

List of Figures

List of Tables

Abbreviations

AI	**A**rtificial **I**ntelligence (see page 1)
CNN	**C**onvolutional **N**eural **N**etwork (see page 2)
PCA	**P**rincipal **C**omponent **A**nalysis (see page 19)
SVM	**S**upport **V**ector **M**achine (see page 7)
k-NN	**k-N**earest **N**eighbors (see page 7)
ReLUs	**R**ectified **L**inear **U**nits (see page 10)
EWC	**E**lastic **W**eight **C**onsolidation (see page 16)
QCSCM	**Q**uality **C**ontrol **S**ystem for **C**lothing **M**anufacturing (see page 21)
DD Server	**D**efect **D**etection **S**erver (see page 22)
TP	**T**rue **P**ositive (see page 58)
FP	**F**alse **P**ositive (see page 58)
FN	**F**alse **N**egative (see page 58)

Chapter 1

Introduction

In this chapter, we introduce the scope of this work by presenting the overview, the motivation and the objectives we proposed to achieve. The document structure is also presented in this chapter.

1.1 Overview

Machine learning has been a hot topic in recent times. Although it is an old concept, the computational improvements of the last decade made it become one of the most trending areas of research in artificial intelligence (AI), as well as one of the fields most used by business companies to solve real world problems. Financial trading, health-care, recommendations systems, anomaly detection are some of the business and problems that use machine learning applications (Marr, 2016).

One of the machine learning areas that is drawing a lot of interest nowadays is deep learning, due to its state-of-the-art results when solving computer vision related problems. Computer vision is used to process, analyze and understand digital images, and is aimed to solve problems like object detection, image classification and video tracking (Szeliski, 2010).

Computer vision problems can be applied to quality control tasks, more precisely in defect detection and classification. There are many quality control systems of manufacturing processes that can be improved with the right use of machine learning algorithms, such as mobile phone cover glass production (Li, Liang, & Zhang, 2014), fabric production (Chan & Pang, 2000), etc.

Many machine learning algorithms can be used for image classification problems, as shown in the next chapter, but most of them have a fixed number of classes and the algorithms cannot learn new classes incrementally. This can be a problem for applications and processes where new data and classes are created, because it would require training the algorithm again from scratch with the old and new data together. The present work addresses this issue as it plays a major part in the proposed system.

1.2 Motivation

Quality control is a key factor in all major manufacturing businesses, as costumers and investors are increasingly demanding for higher quality. It is vital for a company to ensure that the number of defective products is kept to a minimum, otherwise it can have a big impact on the company's sales and business.

Most of the quality control processes are still made by humans and although these processes have improved over the years, human based processes can lead to a few disadvantages. For example, a human usually works approximately 8 hours a day, along which its focus level varies significantly. These levels of concentration may vary due to fatigue, lack of motivation and other factors that can lead to unnoticed defects and, therefore, hurt the business. A computer, however, can keep the same levels of "concentration" throughout the day.

In the textile industry, where humans are responsible for the quality control processes, only 70% of the defects are detected (Kumar, 2003). Therefore, there is still room from improvement.

Training an image classification model such as a convolutional neural network (CNN) from scratch is time consuming and resource intensive. The CNN proposed in (Krizhevsky, Sutskever, & Hinton, 2012) took five or six days to train on two GTX 580 GPUs using the ImageNet dataset (Krizhevsky & Hinton, 2009). In quality control realities new types of defects can appear and in an ordinary classification model this would require training it again from scratch with data from all classes. Therefore, it would be tremendously helpful to have a classification model that can learn new classes without having to be trained from scratch.

Taking these facts into account, an application or system capable of helping factory workers improve the quality control processes can be very useful for manufacturing factories.

In this work, we propose and develop a system for defect detection, to improve the quality control processes of a clothing factory. This system makes use of machine learning algorithms capable of gaining knowledge over time and it was made possible by the collaboration ISCTE-IUL/INOV INESC Inovação.

1.3 Objectives

The goal of this work is to develop a collaborative system capable of identifying defects on clothing products and improving the efficiency of the quality control process of a clothing factory. This system must contain an image classification model capable of learning new classes incrementally and increase its knowledge.

To better understand the purpose of the proposed system, the classification model and the objectives of the present work, is important to define which are the types of defect this system is capable of detecting. A defect can be one of two types, the lack of one or more components used to produce a clothing item leading to an incomplete clothing item, or a wrong choice of components, which leads to an incorrect clothing item. For example, for the first type, a defect could be a shirt that should have fives buttons only having four buttons, as for the second type, a defect could be the choice of a black zipper in a jacket that only has white zippers.

In order to detect the defects of a clothing item, the system, given an image or a set of images, must be able to identify its components and check if the identified components correspond to the ones present on the product data sheet. These identifications correspond to the classifications made by the classification model.

From time to time, new components are made to create new products. Therefore, the classification model must also have the ability to learn new classes from new data without losing the knowledge learned from previous training and previous data. The creation of data is made by the workers responsible for the quality control in a collaborative way.

The present work can be divided in two sub-problems, the task of correctly classifying the components of the clothing items (image classification) and the task of the classification model increase its knowledge over time (incremental learning).

By the end of the present work it must be possible to answer the following question:

- How to develop a system capable of identifying defects and gain more knowledge over time in a robust and efficient way using machine learning algorithms to improve the quality control process of a factory?

To answer this question, due to its complexity, the best approach is to divide it into smaller and simpler questions that can be answer more easily, based on the scope of the present work:

- **Which are the best algorithms to classify objects in an image?** To detect the defects on a clothing item, the system must classify its components, thus the need of image classification algorithms.
- **Can algorithms capable of incremental learning be adapted to a quality control reality maintaining previous knowledge?** Since the factory can produce new clothing items with new and different components, the system must be able to learn new classes maintain its knowledge over the previous ones.
- **Can the incremental learning algorithms to be used in the quality control system be applied to an image classification problem?** The algorithms capable of incremental learning must be applied to image classification.

1.4 Document Structure

The present work consists of 6 chapters, structured as follows:

- In **Chapter 2** a literature review of works made in the same research areas of the present work is presented. The research areas include, quality control, image classification and incremental learning.
- **Chapter 3** describes the proposed quality control system and classification model, where a high-level view of context of the system is presented.
- In **Chapter 4** the development of the quality control system and classification model is detailed. The tools, techniques and libraries that were used are presented as well as results of the experiments made to determine the path to follow.
- In **Chapter 5** the proposed classification model is evaluated, and the results of experiments are presented.
- Finally, **Chapter 6** presents the conclusions of this work as well as some considerations of what can be done to improve the development of quality control system and classification model in future work.

Chapter 2

Literature Review

In this chapter, we provide an analysis of the work previously performed in the areas of study of the present work, to better understand the algorithms and tools that were used to develop the proposed quality control system. The review is divided in three main topics, quality control, image classification, incremental learning.

2.1 Image Classification

Our objective is to develop a quality control system that detects defects in clothes. This system classifies the components of a clothing item and checks if they are correct, therefore our problem can be considered as an image classification problem.

Image classification can be described as the process of taking an image as input to predict a class from a set of classes. It is an old research topic, with work dating back to 1973 (Haralick, Shanmugam, & Dinstein, 1973).

Before deep learning techniques started to be used for image classification tasks, many works used other algorithms to perform the classification of images, such as, support vector machines (SVM)((Chapelle, Haffner, & Vapnik, 1999; Anthony, Greg, & Tshilidzi, 2007) or k-nearest neighbors (k-NN) (Kim, Kim, & Savarese, 2012).

For example, in (Chapelle et al., 1999) SVM were used to perform image classification on color histograms of the images. They tested their approach on two different datasets, one contained 14 classes and the other 7 classes. The results then achieved, although satisfactory at the time, cannot be compared to the more recent results achieve with deep learning.

More detailed information about image classification techniques and approaches, before the use of deep learning, can be found in (Lu & Weng, 2007).

2.1.1 Deep Learning

In more recent years, deep learning techniques have achieved state-of-the-art results in image classification problems with the development of a handful of neural network architectures.

Neural networks are algorithms inspired in biological nervous systems like the brain. They consist in input, hidden and output layers that are connected through sets of neurons, each neuron has a weight, a bias and an activation function associated with it. A neuron when receiving an input performs some calculations using its activation function and outputs the result, which will serve as input to next neuron. deep learning consists in neural networks with multiple hidden layers. Figure 2.1 shows a representation of a neural network with an input layer, two hidden layers and an output layer.

Nowadays, the neural networks architectures used in image classification problems consist in a series of convolutional, pooling and fully-connected layers, used to extract relevant features from the images, followed by one or more fully-connected layers used to perform the classification task. This type of neural networks is called convolutional neural networks (CNN) (Wu, 2017).

Convolutional layers apply convolutions to the input matrix by sliding a filter over it. This filter consists of a kernel, which the width and height are usually the same, a stride and a padding. The stride determines how the filter convolves over the input, for example, if the stride is set as 1, the filter will shift around the input one unit at a time. The padding adds zeros to the input in cases where the filter exceeds the input size, this only happens when the stride is greater than 1. In Figure 2.2 is possible to see a filter with a kernel of size 3x3 and a stride

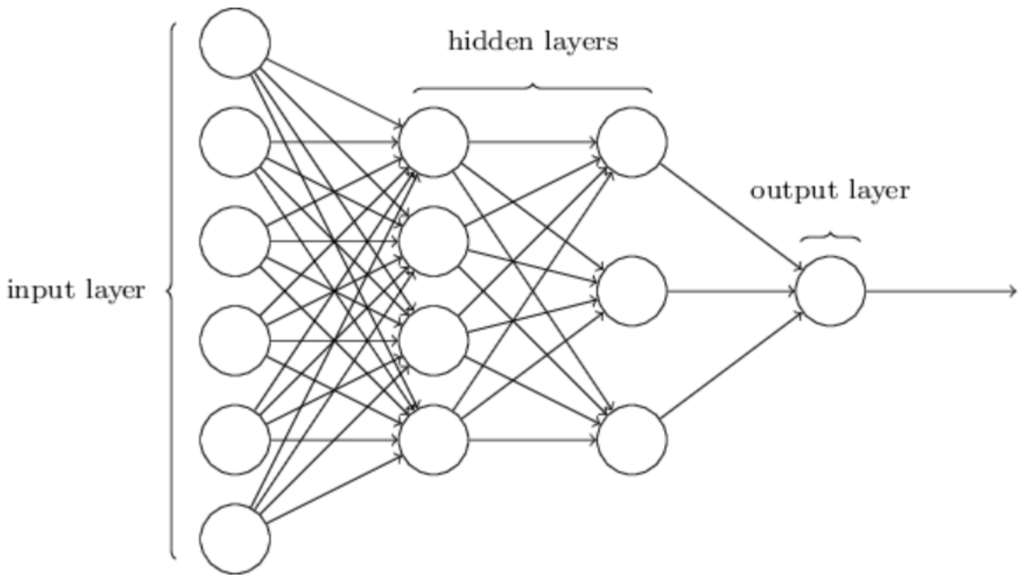

FIGURE 2.1: Neural network representation. Source: (Nielsen, 2015)

of 2 convolving over an input matrix of size 4x4, in this case a padding of 1 is necessary.

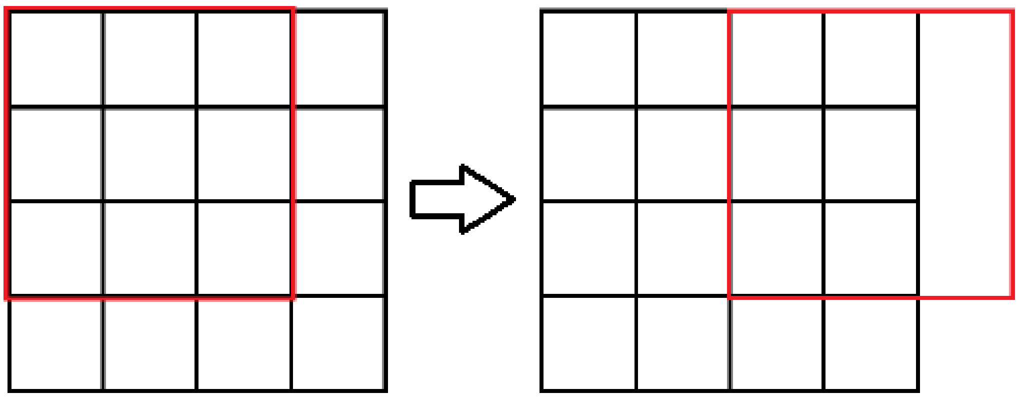

FIGURE 2.2: Convolutional layer filter example.

Pooling layers are used to reduce the size of the image representation and therefore reduce the number of parameters and the computational time in the network. Like the convolutional layers, a pooling layer also uses a kernel and stride, normally with the stride being set to match the width or height of the kernel. The most common type of pooling is the max pooling, this way it keeps the maximum value of the values being process by the kernel.

Finally, the fully connected layers are normal neural networks layers where all the neurons of the layer are connected to the previous layer. They are commonly the last layers used in a CNN and perform the classification task, where the final layer, the output layer, has one neuron per class.

One of the first CNNs was introduced in (LeCun, Bottou, Bengio, & Haffner, 1998) and it was called LeNet5. The architecture of the LeNet5 consists of seven layers, not counting the input. Three of those layers are convolutional layers use to extract features and the first two convolutional layers are each followed by a subsampling layer (pooling). The third convolutional layer extracted features that are then processed by two fully-connected layers where the classification is performed. The LeNet5 was a pioneer in the deep learning field and the works that followed were based in this work.

In (Krizhevsky et al., 2012) AlexNet was introduced with an architecture that use the principles of the LeNet5 to create a larger neural network. AlexNet is composed by five convolutional layers, which some are followed by max-pooling layers and three fully-connected. The three convolutional layers used filters of size 11x11, 5x5 and 3x3 respectively. The authors of this work performed some introductions like the use of Rectified Linear Units (ReLUs) for the activation functions, which improved the training time, the use of dropout layers and data augmentation to reduce *overfitting*.

The authors in (Simonyan & Zisserman, 2014) continue the exploration of using CNN for image classification by proposing two new architectures, the VGG16 and the VGG19. Both architectures consist of five convolutional blocks, each block composed by a set of convolutional layers followed by a max-pooling layer, rather than having a pooling layer after each convolution. Instead of using 11x11 filter like the AlexNet, the VGG networks use 3x3 filters. By doing these alterations the authors showed that doing smaller consecutive convolutions have equivalent results to a single convolution with a bigger filter, but with the benefits of using a smaller filter.

Seeking to improve the computational cost of CNNs, the authors in (Szegedy et al., 2015) presented GoogLeNet or Inception. The Inception model contains 22 blocks of layers, that when summed up result in more than one hundred layers, and rather than having the layers stack sequentially, parallel convolutional layers compose these blocks. The authors called these blocks with parallel layers Inception module. Each Inception module consist of 1x1, 3x3 and 5x5 convolutions, this way the model can decide which is the best convolution operation for each layer. The authors also proposed the use of 1x1 convolutional layer, before each 3x3 and 5x5 convolutions, in order to reduce the number of features and therefore reducing computations.

Continuing from the previous work, some alterations to the Inception network were proposed in (Ioffe & Szegedy, 2015). Batch-normalization was introduced, by normalizing the output of some layers for each mini-batch the model was able to use higher learning rates and consequently reduce training time.

Two new CNN architectures driven from the first Inception network, InceptionV2 and InceptionV3, were introduced in (Szegedy, Vanhoucke, Ioffe, Shlens, & Wojna, 2016). The authors propose factorizing large convolutions into smaller ones, showing that a 5x5 convolution is 2.78 times more expensive than a 3x3 convolution, therefore, replacing a 5x5 convolution by two 3x3 convolutions increases the performance. Additionally, the authors found that nxn convolutions are equivalent to combination of 1xn and nx1 convolutions, and by doing this factorization the computational cost is reduced. A label smoothing technique is also presented, this technique helps prevent *overfitting* by preventing the model from becoming too confident about a class.

Around the same time InceptionV2 and InceptionV3 were introduced a new CNN called ResNet was introduced in (He, Zhang, Ren, & Sun, 2016). ResNet consists of a neural network substantially deeper than the previous ones and introduced residual learning. Residual learning consists of blocks of layers and shortcut connection where the input of these layers is added to the output. This allows information to be more easily propagated through the network and contributes to a reduction of *overfitting*. A representation of a residual learning block is showed in Figure 2.3

Inspired by the Inception and ResNet architectures, a hybrid version CNN combining the first two was proposed in (Szegedy, Ioffe, Vanhoucke, & Alemi, 2017) called InceptionResNet. This combination is achieved by adding a shortcut connection, like in the ResNet, to each Inception module. By doing this, although the accuracy did not have significant improvements, the training time was reduced.

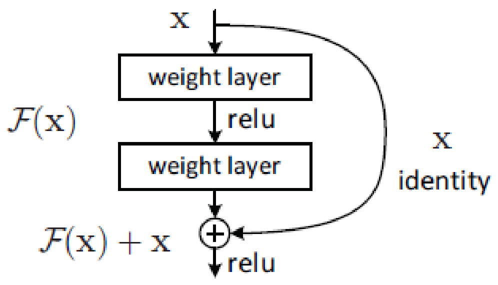

FIGURE 2.3: Residual learning. Source: (He et al., 2016)

In the same work the authors also present a new version of the Inception network, InceptionV4, which is a simplified version of InceptionV3.

Mobile phones are now more than ever part of our lives, therefore efforts to make these deep learning models able to be used on mobile phones, which have smaller process capabilities and less memory than a normal computer, have been performed. In (Howard et al., 2017), a smaller CNN optimized for mobile application, called MobileNet, is proposed. MobileNet consists of *depthwise* convolutions and a 1x1 convolution the authors call *pointwise*. This *pointwise* convolution is applied to combine the outputs of the *depthwise* convolution. Two hyper-parameters that help reduce the model, are proposed as well. The first hyper-parameter, the width multiplier, reduces the number of channels of each layer. The second hyper-parameter, the resolution multiplier, reduces the input image of the model.

Most of these CNN architectures were evaluated in the ImageNet Large Scale Visual Recognition Challange (ILSVRC), where as of 2017, the InceptionResNet architecture achieved the best results.

2.1.1.1 Transfer Learning

Most of the CNNs reviewed take a long time to train even on last generation GPUs. However, there is a way to use the knowledge of a CNN gained when trained in a large dataset, like the ImageNet, and adapt it to a similar classification problem. This is called transfer learning.

Transfer learning consists of using a CNN with the parameters, weights and biases obtained when trained in a large dataset, use the first layers for feature extraction and replace the last layers (fully-connected layers) use for classification with new layers adapted to the desire classification task. This way there is only need to train the new layers, which will save time and resources (Pratt, Mostow, Kamm, & Kamm, 1991; Pratt, 1993).

In (Yosinski, Clune, Bengio, & Lipson, 2014) the author aims to answer the question "how transferable are features in deep neural networks?". Two problems that affect transfer learning, depending on the layers where features are transferred from, are presented in this work.

The first problem addressed by the authors is "optimization difficulties related to splitting networks between co-adapted neurons", this problem is more accentuated when splitting the network in the middle layers then on the bottom or top layers. The second problem is "the specialization of higher layer neurons to their original task at the expense of performance on the target task", this means that the higher layers (i.e. final layers) are more adapted to the original task than the bottom layers, which generalize more easily to new datasets, therefore there are cases where retraining these last layers can be efficient. The authors also confirm that transfer learning is more effective when the base network is trained on a more similar task.

In (Oquab, Bottou, Laptev, & Sivic, 2014) an AlexNet from (Krizhevsky et al., 2012), trained on the ImageNet dataset, is used as the base model for the transfer learning task. The authors propose replacing the last fully-connected layer by two new fully-connected layers that receive as input the output of the penultimate layer of the AlexNet. The first layer of the new layers has size of 2048 and the second layer has a size equal to the number of classes of the new target classification, which in this case are 20 classes. The results show that a transfer learning procedure is effective when using knowledge obtained from a large dataset in a smaller dataset.

A technique to accelerate the transfer of knowledge of one neural network to another was proposed in (Chen, Goodfellow, & Shlens, 2015). This technique it is based on "function-preserving transformations of neural networks" and allows a newer model to contain all the knowledge of an older model and be trained to improve its performance.

2.1.1.2 Libraries

There are many deep learning libraries that provide an easier implementation, training and use of CNNs. In this section, we review some of these libraries.

A review of several deep learning toolkits and libraries was performed by (Erickson, Korfiatis, Akkus, Kline, & Philbrick, 2017). The first reviewed library is Caffe, developed by Berkeley Vision and Learning Center. It is modular and fast and supports multiple GPUs. There is also a website where Caffe models can be download as well as network weights. One of the disadvantages that the authors of this review point out is that tuning the hyperparameters is a tedious process.

Another library reviewed is TensorFlow, developed by Google. Like Caffe, TensorFlow also supports multiple GPUs. It provides tools for tuning and monitoring performance like TensorBoard.

Theano is another deep learning library, written in python, which has improved performance because of the efficient code base of numpy. It is good to build networks but challenging to create complete solutions.

Keras is a deep learning library written in python that can use either Theano or TensorFlow as backend. It is easier to build and read complete solutions. It is well documented and pre-trained models of common architectures are provided. According to the Keras website (*Why use Keras?*, 2018), Keras is the second most used deep learning libraries behind TensorFlow. A speed and accuracy comparison between Theano with Keras, Torch, Caffe, TensorFlow and Deeplearning4J was performed by (Kovalev, Kalinovsky, & Kovalev, 2016). The data used in this comparison was from the MNIST database, which is a large dataset of handwritten digits used for training various image classification approaches. The results show that, when increasing the number of neurons, Caffe and Deeplearning4J drop in the accuracy, TensorFlow and Torch increase the accuracy, and Theano with Keras is stable with different number of neurons. In terms of speed, the libraries can be ranked as follows: Theano with Keras, TensorFlow, Caffe, Torch, Deeplearning4J.

2.1.2 Data Augmentation

It is a common mistake to think that a good model equals good results and if you are aiming to improve the results you should consider improving the model.

Sometimes this can be true, but most times improving the data is better than improving the model. In this section, we provide a view to some of the techniques used to improve the data.

Two of the major problems with low quality data are: not enough data and unbalanced data. Both problems can cause *overfitting* of the model, but both can be fixed via data augmentation.

A description of the best practices when using CNNs is provided in (Simard, Steinkraus, & Platt, 2003) and the authors conclude that the most important practice is to have a dataset as large as possible. By doing augmentations based on distortions of an original image, the model can achieve better results.

The most common type of data augmentations in image classification problems, and proven to be successful, are techniques such as cropping, rotating and flipping the images. In (Perez & Wang, 2017) show this to be true and propose a method that allows the model to learn which are the best augmentations to perform and achieve better results. This method consists of two networks, one for augmentation and another for classification. The augmentation network creates an augmentation image from two input images of the same class and a loss is calculated to check how similar the image should be to the original images. The classification network is a simple CNN. The authors were not aiming for the best classifier, the goal was to see how data augmentation affects the results of the model.

2.2 Incremental Learning

Many processes and applications need to gain knowledge over new information over time, more specifically, some classification problems require learning new classes progressively. The problem is that most of the classification algorithms cannot perform this task. In this section we take a look at the work performed in incremental learning.

Incremental learning it is not a recent research topic, some of the work performed in this area can date back to the end of the eighties. In (Utgoff, 1989), an algorithm is proposed for training decision trees incrementally. The proposed algorithm is capable to build a decision tree identical to the one present in (Quinlan, 1986), which is non-incremental, but built in an incremental fashion.

Neural networks have proven to be an efficient machine learning classification algorithm. Taking this into account many researchers have tried to implement incremental learning on this type of algorithms. In (Carpenter, Grossberg, Markuzon, Reynolds, & Rosen, 1992), a neural network for incremental learning of analog multidimensional maps is introduced, later in (Polikar, Upda, Upda, & Honavar, 2001), a neural network called Learn++ is introduced. This neural network is capable of learning from new data, including examples of previously unseen classes. Like the neural network proposed in (Carpenter et al., 1992), Learn++ does not need to access data used in previous training session, it can learn using only new data. It does this and, simultaneously, does not forget previously gained knowledge.

In (Fuangkhon & Tanprasert, 2009) an "alternative algorithm for integrating the existing knowledge of a supervised learning neural network with the new training data" is presented. The algorithm uses a counter preserving algorithm to increase the classification accuracy, while maintaining old knowledge as the neural network is retrained with new data. Unlike the previous reviewed works, this approach uses some data used in previous training sessions to help maintain the older knowledge.

One of the reasons incremental learning is a challenging task for neural networks is their tendency to suffer significant loss of prior knowledge as the network tries to learn from new data. This is called *catastrophic forgetting*.

Trying to solve the catastrophic forgetting problem, the authors in (Kirkpatrick et al., 2017) present an approach that uses an algorithm called elastic weight consolidation (EWC) and show that is possible to overcome the catastrophic forgetting problem. EWC allows learning to slow down on weights based on how important they are to previous tasks.

In (Rebuffi, Kolesnikov, Sperl, & Lampert, 2017) the authors showed a new training strategy, that allows incremental learning, called iCaRL. It does not require much data as new classes are added incrementally to the model. ICaRL is made up by three components, a nearest-mean-of-exemplars classifier, a herding-based step for prioritized exemplar selection and a representation learning step that uses distillation to avoid catastrophic forgetting. This method achieved satisfactory results, compared to other techniques, when a significantly large number of classes is added to the original model.

A new training methodology that allows CNNs to learn new classes from new unseen data is proposed in (Sarwar, Ankit, & Roy, 2017). This methodology uses principals of transfer learning by updating a network for a set of new classes using the initial part of the base network. It consists of a mechanism able to identify how many of the base network layers weights and parameters can be shared when adding new classes. When training the model for new classes, new layers are created which become a branch of the existing network.

Mondrian forests are a type of random forests that can be used both for classification (Lakshminarayanan, Roy, & Teh, 2014) and regression (Lakshminarayanan, Roy, & Teh, 2016). They consist of an ensemble of decision trees based on Mondrian processes that can be trained incrementally, called Mondrian trees.

Most of the decision trees, that also support incremental learning, usually perform splits on the leaf nodes when being trained incrementally. Mondrian trees also perform splits on leaf nodes when updating their knowledge with new data, however, Mondrian trees also perform two more operations. Therefore, there are three different ways to update Mondrian trees:

- A new split "above" an existing slip.
- Extension of an existing split.
- Split of a leaf node into two new nodes.

In a training session with new data the Mondrian tree is updated, and an algorithm decides which operation of three mention above is used for each node. This decision is made considering the split cost of each node.

To perform a prediction of a certain data input, each Mondrian tree contained in the forest determines the probability of the input belonging to every one of the existing classes. After the average of the trees predictions is calculate and the class with the highest probability corresponds to the predicting class.

Although Mondrian forests can learn from new unseen data the authors of (Lakshminarayanan et al., 2014) do not experiment adding new classes to the model. In (Narr, Triebel, & Cremers, 2016) an extension of the Mondrian forest algorithm for classification of images is presented. This approach can learn new unseen classes. Some alterations to the original implementation were performed.

In the original implementation of the Mondrian forest when all the labels of a certain node are the same, no split is performed for that node. This "pause" operation, as the authors call it in their work, allows the comparison between Mondrian and other random forest implementations. However, in (Narr et al., 2016), the authors chose to remove this operation and instead define a threshold to limit the maximum number of samples with the same label that a leaf node can have.

2.3 Quality Control

Quality control using machine learning techniques has been a hot research topic for a few years. Earlier work performed in this area used Fourier analysis to detect defects in fabric (Chan & Pang, 2000). Using white fabric for the experiences, the system was able to detect four different types of defect by applying Fourier transforms to the images and analyzing the frequency spectrum.

In (Kumar & Pang, 2002) the frequency spectrum is also analyzed to detect defects but instead of using Fourier analysis, the authors used Gabor filters. Two approaches are presented, a supervised and an unsupervised defect detection. The difference between the two is that in the supervised approach, information about the orientation and the size of the defects is given. This information makes spectrum-sampling of the spatial-frequency plane unnecessary, allowing the system to work with just one Gabor filter. As for the unsupervised approach there is a need to use multiple filters so that the system can get information about the orientation and size of the defects.

The same authors then proposed a different approach for this problem using a feed forward neural network. This neural network was used to classify feature vectors of pixels extracted from images (Kumar, 2003). Despite proposing a feed forward neural network for defect detection, due to the computational cost of this type of networks at the time of the work, the authors also propose a linear neural network with a lower computational cost and instead of classifying 2-D images the input was reduce to 1-D vectors.

Another approach of detecting defects is to calculate and evaluate the degree of similarity between an image with defect and an image with no defect. In (Tsai & Lin, 2003) this approach, a normalized cross correlation to calculate the degree of similarity is utilized. A sliding window is used to process the image, but it size affects the computational efficiency and the effectiveness of the detection. Taking this into account the authors proposed a sum-table scheme that allows the calculation of the normalized cross correlation to be invariant to the window size.

More recently in (Çelik, Dülger, & Topalbekiroğlu, 2014) it is proposed a method for defect detection as well as defect classification. For the defect detection it was used a wavelet transform, as for the classification task it was used a feed forward neural network. The system was able to detect effectively five types of defect: warp lacking, weft lacking, soiled yarn, hole and yarn flow. Figure 2.4 shows examples of these types of defect.

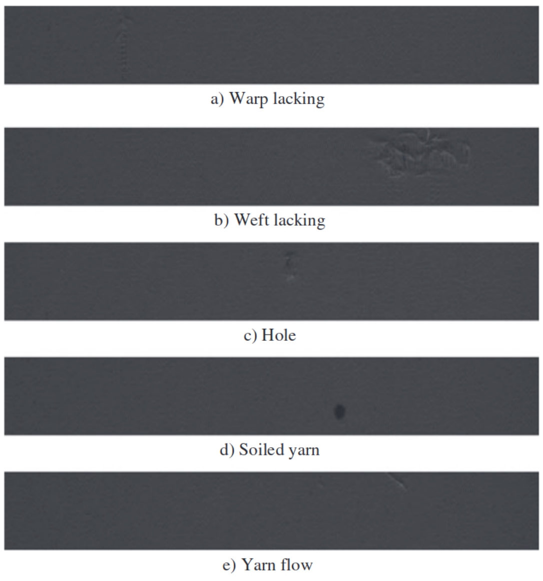

FIGURE 2.4: Fabric defect types. Source: (Çelik et al., 2014)

In (Li et al., 2014) a mobile phone cover glass defect detection is presented, and even though it is for glass defects and not fabric defects like the previous reviewed works, the principles are the same. The propose system consists in a defect inspection system based on the Principal Component Analysis (PCA) and is capable of identify five cover glass defects.

Works such as (Weimer, Scholz-Reiter, & Shpitalni, 2016) and (Wang, Chen, Qiao, & Snoussi, 2018) have used CNN to perform quality control and defect detection tasks. In the first work, four different CNN architectures are proposed and trained to identify six types of defects. Each architecture had a different number of convolutional layers with the results showing that, in this case, adding layers increases the detection accuracy. In the second work a CNN able to detect six types of defects is also proposed. The architecture of this CNN consists of 11 layers, five of which are convolutional layers and two pooling layers.

Additional work regarding the topic of quality control and defect detection can be found in (Kumar, 2008) and (Ngan, Pang, & Yung, 2011).

Chapter 3

Quality Control System for Clothing Manufacturing

In this chapter, we describe the quality control system for clothing manufacturing (QCSCM) proposed in the first chapter as the objective of the present work. The purpose of the QCSCM is to detect defects on clothing items. This is achieved by using a **classification model** supporting **incremental learning**. This classification model can, however, be easily adapted to other contexts.

The present work was developed to be used by a real clothing factory and was made under the collaboration ISCTE-IUL/INOV INESC Inovação. INOV wants to create a commercial system to help manufacturing companies in their quality control processes, in this case oriented to clothes. Based on a defined client, INOV defined a set of requirements that the system should fulfill based on training sets and a machine learning approach. These requirements are as follows:

- **A system capable of detecting defects on clothing items using pictures.** The system outputs a binary classification, **defect** or **no defect**, based on the classification of the clothing items components.
- **A mobile application to take the pictures of the clothing items and be used by the quality control officers to perform their quality control tasks**. The system is fed by the quality control officer using the mobile application.

- **Increase the speed of the quality control processes and the percentage of detected defects.** For the system to be useful, it should improve the performance of the quality control processes.
- **The ability of the system to learn from new data, as new components of clothing items are created.** The system must learn new classes maintaining its previous knowledge. The quality control officer creates the new data using the mobile application and feed the system in a collaborative way.

INOV also defined the types of defect this system aims to detect on a clothing item. A clothing item is made up by a set of components, such as buttons, pockets, stamps, etc. Therefore, a defect can be one of two types:

- Wrong component.
- Missing component.

Figure 3.1 shows an example for each type of defect. The one on the left corresponds to a shirt with a missing button (**missing component**) and the one on the right shows a t-shirt with a yellow pocket instead of a pink pocket (**wrong component**).

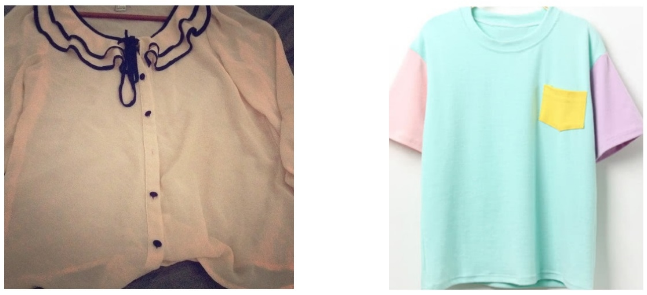

FIGURE 3.1: Examples of the two types of defects. Missing component on the left, wrong components on the right.

Considering the requirements and the types of defect, the QCSCM architecture was defined in Figure 3.2. Using a **client-server model approach**, the QCSCM consists of a **mobile application and a server**, we called Defect Detection Server (DD Server). The mobile application is used to take pictures of the clothing items and the DD Server is responsible for processing and storing the pictures, detect the defects making use of the classification model and finally, register the defects. To improve the QCSCM performance a **user feedback** approach was also defined.

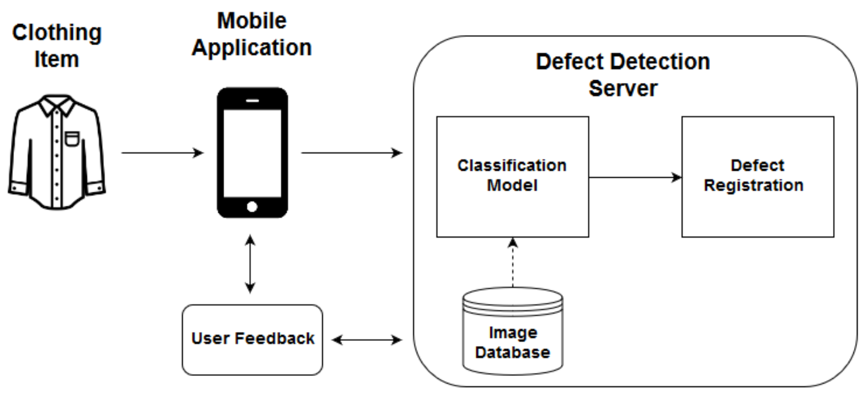

FIGURE 3.2: QCSCM general architecture

The responsibility of the quality control in the factory lies with a group of factory workers called **quality control officers**. The function of the quality control officers is to detect defects on the clothing items, register them and decide whether to send the clothing item back to the manufacturing process, remove the clothing item from production, or continue to the next production step. A clothing item is sent back to the manufacturing process if a repairable defect is detected and is removed from production if an unrepairable defect is detected.

To execute their function, the quality control officers use the mobile application to take pictures of the clothing items and create **bounding boxes** around the components that compose a clothing item. This information is sent to the DD Server that **crops the content of the bounding boxes to create the images of the components**. These images of the components are classified by the classification model and the results are compared with the **product data sheet** to see if there is a defect or not. Finally, the classifications of components are sent back to the mobile application being used by the quality control officer.

A **product data sheet** is information associated to each model produced by the clothing factory. The product data sheets are defined by the clothing factory every time a new clothing item model is created. The information present in the product data sheet information consists of a list of specifications and components that compose a clothing item. Its structure is as follows:

- Model.

- Size.
- Color.
- Fabric.
- List of components.

The QCSCM focuses on the list of components specification. All the clothing items have an identifier corresponding to a product data sheet, so when a clothing item is going through a quality control process the system can know which components it has to identify.

As in any other manufacturing business, in a clothing manufacturing business new products and components can be created. Therefore, the QCSCM uses a classification model which has the ability of learning new classes incrementally. The responsibility of creating images of the components to train the classification model also relies on the quality control officers. To create images of the new classes or even more images of old classes, the quality control officer can choose an option in the mobile application to use the pictures it took to train the classification model instead of using the pictures for the defect detection process. The quality control officer can also create more images by confirming or correcting the classifications it received from the DD Server, this is the **user feedback** feature. In Figure 3.3 we defined a use case diagram that explains the actions the quality control officer performs using the mobile application.

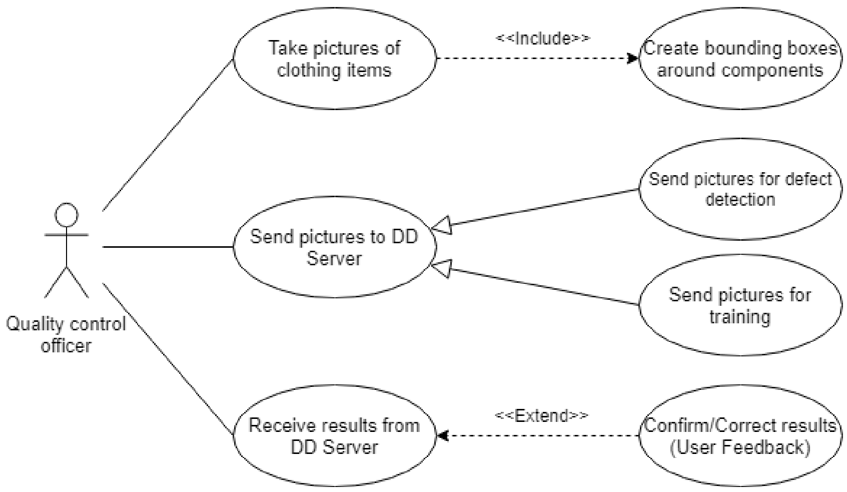
Take pictures of clothing items
<<Include>>
Create bounding boxes around components
Send pictures for defect detection
Send pictures to DD Server
Send pictures for training
Quality control officer
Receive results from DD Server
<<Extend>>
Confirm/Correct results (User Feedback)

FIGURE 3.3: Use case diagram of the quality control officer actions using the mobile application

3.1 Defect Detection Server

The first main component of the QCSCM is the DD Server responsible for feeding the classification model with images of the clothing items components. The DD Server must perform the following tasks:

1. **Pre-process the images of the components it receives from the mobile application used by the quality control officers**. This task of pre-processing the images consists of cropping the bounding boxes of the pictures taken by the quality control officers creating images of the components. These images of the components are then resized and, in case of training, new images are created using data augmentation techniques. The pre-processing task is necessary so that the classification model can perform its tasks.

2. **Predict the classes of these components**. In this second task, the classification model present in the DD server predicts the classes of the components it received from the quality control officers.

3. **Compare the results with the product data sheet and save the results**. After the classifications are made the DD server performs the third task of comparing the results with the product data sheet. If the identified components match with the ones present on the product data sheet it means no defect was detected and nothing needs to be registered. If they do not match, it means a defect has been detected and the DD Server performs the **defect registration**.

4. **Store pictures of the components and train the classification model with new data**. This fourth and final task is only performed if a quality control officer selected the option of using the pictures to train the classification model. The DD Server after cropping the bounding boxes of the pictures taken by the quality control officers, saves the content of the bounding boxes (images of the components) along with the corresponding labels in a database. If enough images of the components are stored in the database, the training of the classification model is performed.

3.1.1 Image Database

When a quality control officer sends pictures of clothing items with bounding boxes around the components it has to select the option of the task to execute. If the selected option is to use the pictures to train the classification model, the images of the components of the clothing items need to be stored. In this section we describe the image database represented in Figure 3.2 as a module of the DD Server.

After the pictures of the clothing items are processed and the images of the components are created, the DD Server saves the images according to their classes. Each class has an associated directory where all images corresponding to that class are stored. The names of the directories serve as labels for the images when the classification model is trained.

This image database allows the creation of the dataset that is used to train the classification model. Every time the classification model needs to be trained, the DD Server loads the images and labels from the image database and feeds them to the classification model.

The image database also contains a list of the classes and the number of new images available from each class. This list is used to check if there are enough images to train the classification model and it is also sent to the quality control officers when they want to label the components of the clothing items using the mobile application.

3.1.2 Defect Registration

The **defect registration** is represented in Figure 3.2 as a module of the DD Server. It is performed after the classification model classifies the components that are sent to the DD Server and after the results of the classification are compared with the product data sheet to check if there are defects. In case of a positive defect detection, the type of the defect, missing component or wrong component, also needs to be registered. For example, let's assume we have a clothing item that is supposed to have three black buttons and one silver zipper, but the classification model returns two black buttons and one silver zipper. In this case the DD Server would register the defect as **missing component** along with the components that are missing, in this case a black button.

Another example using the same clothing item, the classification model returns three black buttons and a golden zipper. In this case the DD Server would register the defect as **wrong component** and register the misplaced component, in this case a golden zipper instead of a silver zipper.

Apart from the **image database** and the **defect registration** the other main module of the DD Server is the **classification model**. However, due to its importance we decided to describe the classification model in a separated section.

3.2 Mobile Application

The second main component of the QCSCM is the mobile application, developed using Android Studio. It is supported by the Android versions 4.1. and above. The quality control officers use this mobile application to take pictures and create bounding boxes around the components that make up the clothing items. The mobile application consists of a simple user interface, that displays the pictures taken by the quality control officer using the mobile phone camera. It allows the quality control officers to choose whether to use the pictures to detect defects or to use them to train the classification model. The reason of using a mobile application to take pictures instead of a fixed camera is because this way allows the quality control officers to walk around the factory and take pictures of the clothing items in different production steps.

To create the bounding boxes all the quality control officer must do, is drag a finger over the picture and surround the component. By doing this the application stores four coordinates of the image per bounding box. This set of four coordinates consists of two x coordinates (width) and two y coordinates (height). When creating the bounding box, the quality control officer can choose, through a check box, if he wants to classify the component or if he wants to create new data for training. This check box will determinate which process will be executed by the DD Server, the defect detection process or the training process.

For the mobile application to send the pictures to the DD Server, the pictures need to be converted to bytes as well as information about the bounding boxes. After receiving the bytes from the mobile application, the DD Server converts them back to the original format.

Regardless of whether the quality control officer chooses to use the pictures it took to train the classification model or to perform defect detection, they must always create bounding boxes around the relevant components present in the pictures. Figure 3.4 shows an example of a picture of a shirt and the bounding boxes around the components that will be classified.

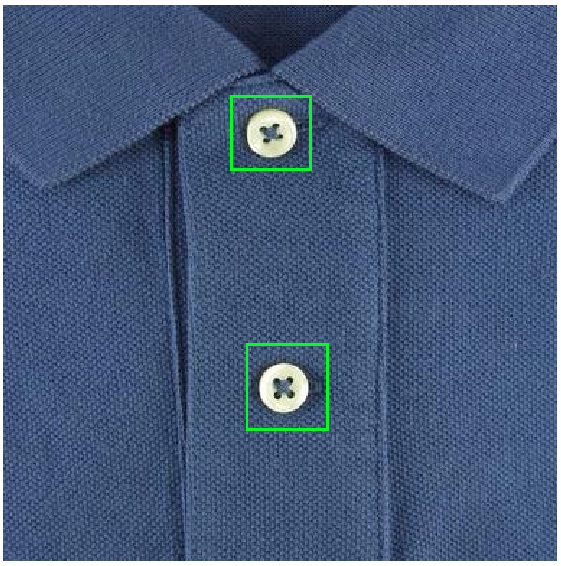

FIGURE 3.4: Bounding boxes example

During the creation of new data to train the classification model, after drawing bounding boxes around the relevant components in the picture, the quality control officers must label each component with the corresponding classes. The classes can be chosen from a list of existing classes or, if the object consists of class not present in the classification model, the quality control officers can create a new class that will be added to the list of existing classes.

During the defect detection process, after receiving the pictures taken by the quality control officers, the DD server sends back the results of the classification model – classified components – so that the quality control officers can give feedback on the classifications made. This interaction between the DD Server and the mobile application – **user feedback** – allows the quality control officer to correct wrong classifications made by the classification model of the DD Server and confirm the correct ones.

The **user feedback** feature is easy to execute, after performing a defect detection process using the mobile application and receiving the results from the DD Server, the quality control officers can perform the necessary corrections by clicking on the bounding boxes with the wrong predictions and select the correct label from a list of the existing classes.

After the corrections are made, the quality control officer sends the information again to the DD Server and new images are created to train the classification model. Although in an initial stage of the QCSCM life-cycle it is useful that the quality control officer performs this feature, it is not mandatory.

Using the feedback from the quality control officers, the DD Server corrects the registered defect and creates new training data that will be used in future training sessions. So, basically the **user Feedback** feature is mainly another way to trigger the training process of the DD Server. Its implementation consists of allowing the quality control officers to change the wrong labels of the bounding boxes by clicking on them via the mobile application or confirming the results using an application button that will send a Boolean value set to true back to the server and initializes a new training process.

This user feature contributes to better training and subsequently better performance of the classification model present in the DD Server. This is only possible thanks to the ability of the classification model to learn incrementally.

Figure 3.5 shows all the steps a quality control officer goes through during the defect detection process using the mobile application, with a last step being the execution of the user feedback.

3.3 Classification Model

In this section, we propose a classification model for images that can learn incrementally as new classes are created and be used in a quality control system to perform defect detection.

The proposed classification model is divided in a feature extraction model and classifier with incremental learning abilities, as seen in Figure 3.6. Although in this work we used the classification model to classify components of clothing items, it can be adapted to other quality control environments.

The feature extraction model consists of a pre-trained CNN that extracts important features from the content of the images. After the extraction, the

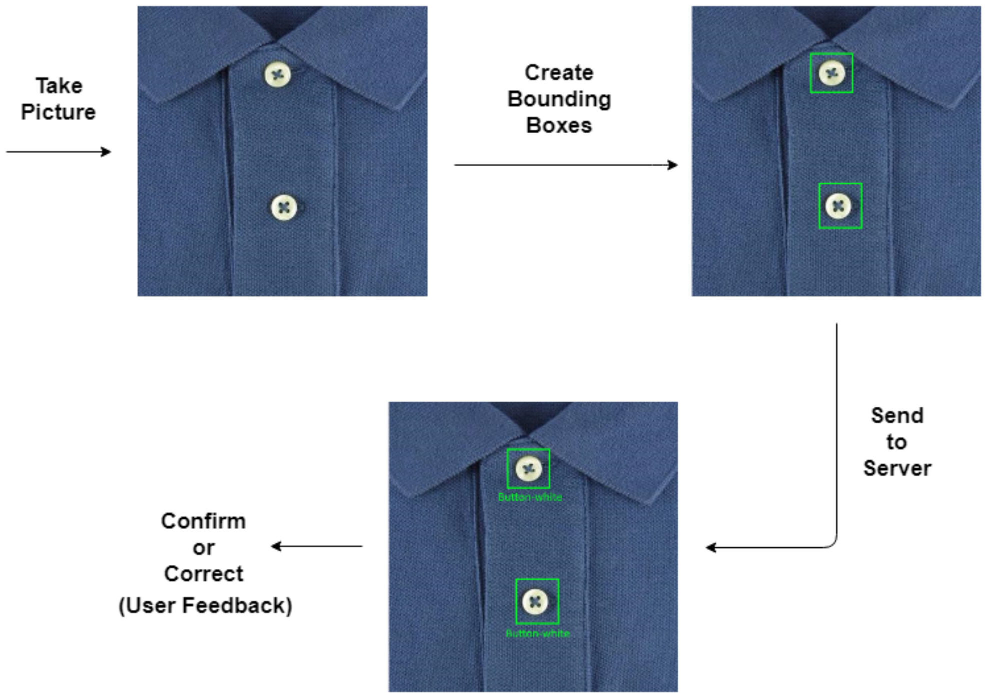

FIGURE 3.5: Quality control officer process of defect detection using mobile application

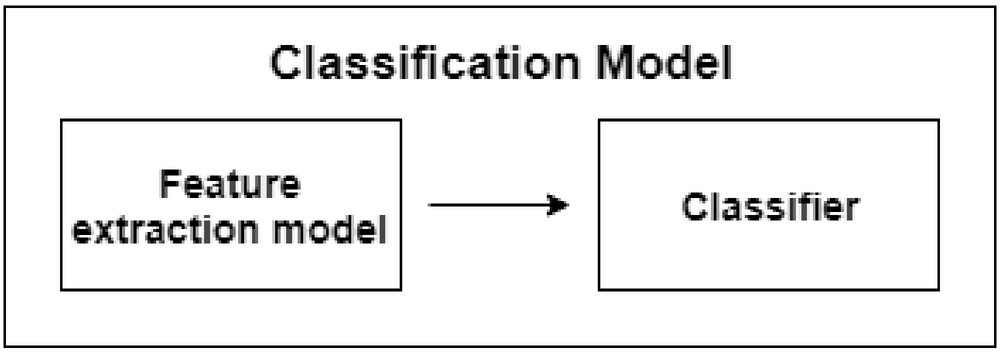

FIGURE 3.6: Classification model architecture

images are classified by the classifier using the extracted features and modified version of the Mondrian forest algorithm that supports incremental learning (Lakshminarayanan et al., 2014). We chose this architecture for the classification model, because by using the principles of **transfer learning**, we can combine the benefits of using a CNN to extract relevant information from an image with the ability of Mondrian forest to learn incrementally.

3.3.1 Feature extraction model

The idea of using a feature extraction model in the classification model was to make sure that the classifier only needs to process and classify relevant information and to reshape the input of the classification model from a three-dimensional array (an image) to a one-dimensional array that can be fed to the classifier. We chose to use a CNN as the feature extraction model because of the recent state-of-the-art results of this type of neural networks when it comes to image classification problems. Figure 3.7 shows a simple CNN architecture and how it extracts relevant features from an input image and converts them into a feature array.

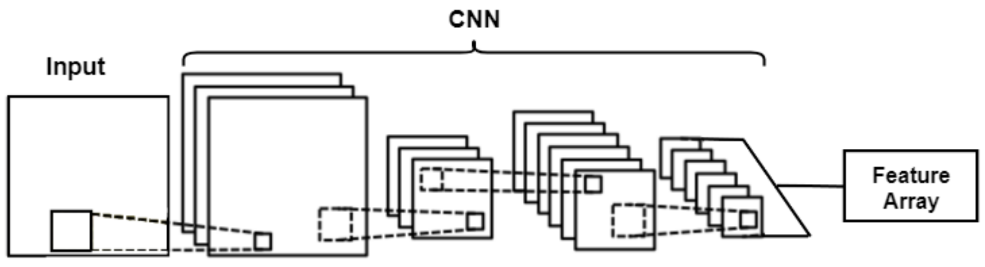

FIGURE 3.7: CNN architecture and feature extraction. Adapted from: (Britz, 2015)

As we seen in chapter 2, there are many CNN architectures and these architectures can be trained in very large datasets to create knowledge and know which are the relevant features that can be extracted from an image. This knowledge consists of using the best weights and biases for each layer of a CNN and it can be adapted to similar image classification problems using transfer learning.

3.3.2 Classifier

The function of the classifier is to classify the images of the clotting items using the features extracted from the feature extraction model. As any other classification algorithm, the classifier present in the classification model needs to be trained with data relative to the classes it wants to classify. However, our classifier must be able to learn incrementally new classes and gain knowledge from unseen data.

Incremental learning is the ability of an algorithm to gain knowledge from new data without losing previous knowledge from old data. It can lead to an increase of classes by training the model with data of new unseen classes, or an increase of knowledge of the old classes by training the model with new data from old classes. Figure 3.8 shows a representation of how a model can learn incrementally.

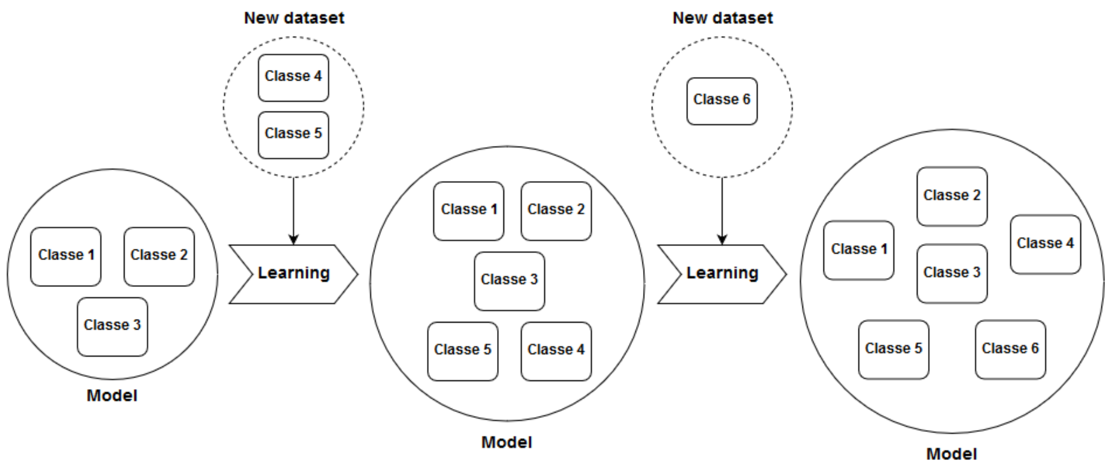

FIGURE 3.8: Incremental learning.

As said before, the classifier present in the classification model consists of a Mondrian forest presented in (Lakshminarayanan et al., 2014). A Mondrian forest is a type of random forest that can learn incrementally. The input of the Mondrian forest is a one-dimensional array, therefore, it is able to train with the feature arrays extracted using the feature extraction model. In the next chapter we detail how we developed the classification model and how our classifier (Mondrian forest) behaves when classifying the feature arrays extracted using different CNN architectures.

Chapter 4

Development of QCSCM

In this chapter, we present the tools and techniques used to develop the classification model and the QCSCM as well as detailed information on the steps we took during the development.

4.1 Classification Model

As seen in Figure 3.2, the classification model, described in section 3.3, is a component of the DD Server of the QCSCM. However, due to its importance and the work we had to develop it, we describe its development in a separated section first. This classification model was developed to classify objects that correspond to the components of the clothing items and would then be implemented in the DD Server of the QCSCM.

As shown in Figure 3.6 in the previous chapter, the classification model is made up by a feature extraction model and a classifier. The feature extraction model consists of a pre-trained CNN without the last layers (fully-connected layers), which are normally use for the classification task. As for the classifier it consists of a modified version of the Mondrian forest algorithm. By doing this, we follow the principle of transfer learning, but instead of adding new layers and trained them with new data, we use the Mondrian forest algorithm which is capable of learning incrementally.

4.1.1 Feature Extraction Model

As defined in the previous chapter, to extract relevant features from the input images we use a pre-trained CNN. The Keras Library provides many CNN architectures already pre-trained on the ImageNet dataset that can be adapted to this problem.

The ImageNet Dataset (Deng et al., 2009) is a very large image database containing more than one million images of objects divided in 1000 classes. The knowledge gained by a CNN when trained with this dataset can be easily adapted to similar image classification problems.

In order to set some baseline results and due to the lack of real images of components of clothing items, we used the Cifar-10 dataset (Krizhevsky & Hinton, 2009) to perform some experiments and check if the classification model can perform well in an image classification problem. The Cifar-10 dataset consists of 60000 images in 10 classes, with 6000 per class. Of these images, 50000 are used for training and 10000 are used for test. Each image consists in a 32x32 color image. The 10 classes are the following: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck.

To use one of the CNN architectures in Keras as a feature extractor, all we have to do is load the desired CNN architecture with the weights obtained during the ImageNet training, remove the last fully connected layers used for classification and process the input images through the network to extract the features. Finally, we save these features in arrays that will serve as the input of the classifier.

4.1.1.1 Why extract features?

Why should we use a CNN to extract features of the images of the components of the clothing items and then classify these features instead of classifying directly the images? In this section, we answer this question and show the importance of feature extraction.

If we did not used a CNN as features extractor, the classifier would use the raw images as input instead of a feature array. For us as humans it is easy to look at an image and understand what it represents, the same cannot be said when we look at a feature array extracted from a CNN.

Figure 4.1 shows a raw image of a clothing item component on the right and its corresponding feature array on the left extracted using InceptionResnet. We converted the 1536 size array to an 32x48 matrix for easier interpretation.

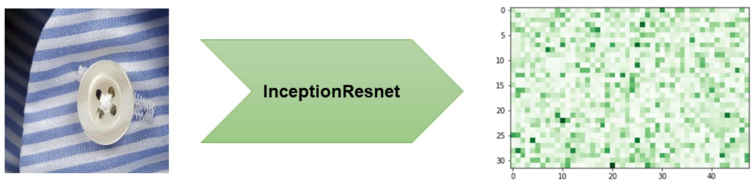

Figure 4.1: Representation of feature extracted using InceptionResnet

If we look at the representation of the extracted features in Figure 4.1 it is hard to understand what it represents and difficult to extract some useful information, just by looking at it. To compare how the classifier, Mondrian forest, performed on classifying the raw images versus classifying the features extracted from the feature's extractor, we performed an experiment using python to calculate the classification test accuracy of the two methods. The accuracy is a metric for evaluating classification models and measures the fraction of the number of correct predictions (Nc) over the total number of predictions (Np) (*Classification: Accuracy*, 2018):

$$accuracy = \frac{Nc}{Np}. \tag{4.1}$$

Table 4.1 shows the results of this experiment on the Cifar-10 dataset and, as we can see, the classification of CNN features is much better than the classification images. This can be explained for two reasons, the first one being the fact that our classifier receives as input a one-dimensional array, therefore, to feed the images to the classifier they had to be converted from a three-dimensional array to a one-dimensional array. This is not efficient because this way the image loses the relationships between pixels. On the other hand, a feature array extracted from a CNN is already a one-dimensional array.

Raw images classification vs CNN features classification		
Number of classes	Raw images	InceptionResnet features
10	0.38	0.85

Table 4.1: Raw images classification vs CNN features classification

The second reason for the classifier performing better when classifying features is because the layers of the CNN are trained to extract relevant information from the images and pass this information through the network. This results in a feature array containing mainly useful information, allowing a better classification.

4.1.1.2 CNN Architectures

To choose which CNN to use in the final version of the classification model, we performed some experiments on some of the architectures provided by the Keras library. The chosen architectures were: VGG16, MobileNet-V1, Inception-V3, ResNet50 and InceptionResnet-V2.

The VGG16 and the MobileNet architectures were chosen because the feature arrays they output are of size 512 and 1024, respectively. Therefore, the computational cost and time of training our classifier with these features is smaller when compared to the feature arrays of size 2048 of the ResNet and the Inception architectures, and the feature array of size 1536 of the InceptionResNet. These last three architectures were chosen because of their state-of-the-art results.

We created a python script to train the classifier on features extracted from the Cifar-10 dataset using each of the selected CNN architectures in an incremental fashion, first we trained it with 5 classes and the we added classes progressively until the classifier was trained for all 10 classes of the dataset. The number of Mondrian trees of Mondrian forest was set to 100. We used this number of trees because it is a common value used in decision forests (Lakshminarayanan et al., 2014). Table 4.2 shows a comparison of the classification accuracies for each set of features using our classifier.

Comparison of CNNs					
Number of classes	Inception	Resnet	InceptionResnet	MobileNet	VGG16
5	0.85	0.86	0.91	0.79	0.77
6	0.80	0.81	0.87	0.71	0.69
7	0.77	0.79	0.85	0.68	0.67
8	0.75	0.77	0.84	0.67	0.64
9	0.74	0.76	0.83	0.65	0.62
10	0.72	0.76	0.83	0.63	0.60

Table 4.2: Comparison of CNNs features classified with Mondrian forest

As the Table 4.2 and Figure 4.2 show, for all CNN architectures, the accuracy decreases when new classes are added. The InceptionResnet shows the best results, followed by the Resnet and the Inception. Furthermore, the classifier trains faster on the InceptionResnet features than on the Resnet or Inception features, this is because the InceptionResnet returns a feature array of size 1536, which is smaller than the 2048 size array of both the Resnet and Inception.

Although the training of the classifier with the features of the VGG16 and MobileNet was significantly faster than the training with the InceptionResNet features, the accuracies were much worse. Taking these results into account we chose to use the InceptionResnet CNN as our feature extraction model in the following experiments.

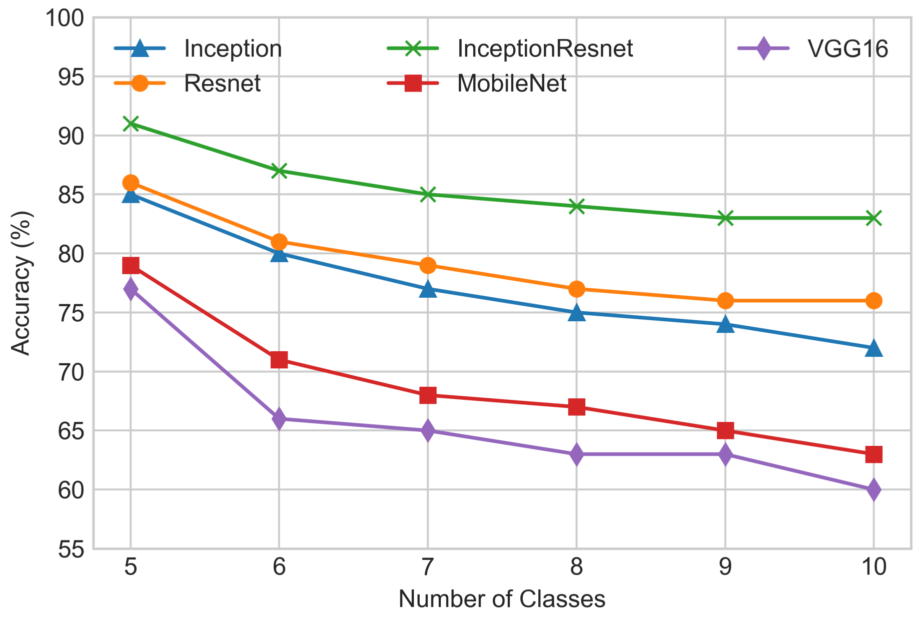

FIGURE 4.2: Comparison of CNNs features classified with Mondrian forest - Graph

4.1.2 Classifier

For the classifier that classifies the images using the features extracted from the feature extraction model, we used the Mondrian forest algorithm presented in (Lakshminarayanan et al., 2014). Although the original implementation of the Mondrian forest[1]can train with new data maintaining the previous knowledge, it does not support the capability of updating the number of classes, thus some modifications had to be implemented.

[1]`https://github.com/balajiln/mondrianforest`

In the original implementation of the Mondrian forest when initializing the model, a series of data related parameters must be defined, such as, the number of classes of the data, the training and test data and its corresponding labels. In the implementation developed in the present work, these parameters are also defined, but after each training session, the number of classes used in that session is saved in the model.

When new data arrives for a new training session the number of classes of the data is compared to the number of classes in the model and if the number differs, the number of classes in the model is updated. By updating the number of classes, it is possible to update each node information of each Mondrian tree on the new data, which consists mostly of arrays of size equal to the number of total classes. This way, every time the model is trained on new classes it can easily update the total number of classes.

4.1.2.1 Incremental Learning

To see how the classifier performed after the modifications we made to the original implementation of the Mondrian forest, we experimented training the classifier incrementally with new classes and training the classifier with new classes from scratch. After the experiments we compared the accuracies of both training methods in Table 4.3.

Full training vs Incremental training		
Number of classes	Full training	Incremental training
5	0.91	0.91
6	0.88	0.87
7	0.86	0.85
8	0.86	0.84
9	0.86	0.83
10	0.85	0.83

TABLE 4.3: Comparison of classification model accuracies trained from scratch and trained incrementally

To perform the experiments that resulted in Table 4.3, we created two python scripts. The first one, *Full training*, the classifier was first trained from scratch with five classes, then trained from scratch with six classes and then again until was trained with all ten classes. No incremental learning was used. In the second script, *Incremental training*, the classifier was first trained with five classes from scratch and then one class was added incrementally to the classifier using only data from the new class, until all ten classes were learned. In both python scripts we used the features extracted from the Cifar-10 dataset using the feature extractor.

As we can see in Figure 4.3 , the difference between the two training methods is not big, with just a small drop, of around 1% to 2%, in accuracy when trained incrementally compared to training with all classes from scratch. These results show that the classifier can be trained incrementally in a satisfactory way, which is important for the QCSCM.

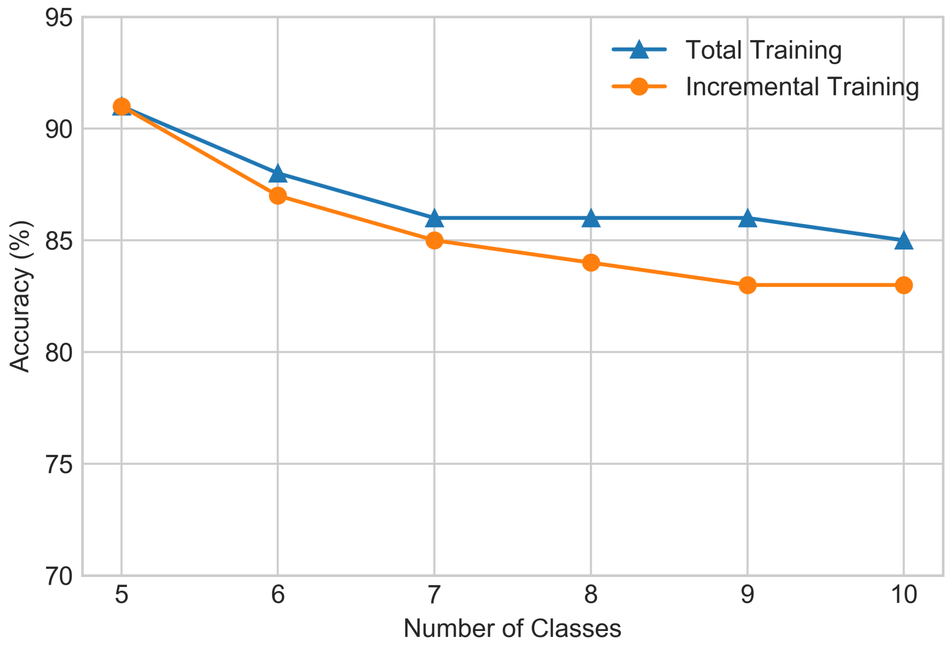

FIGURE 4.3: Full training vs Incremental training - Graph

4.1.2.2 Mondrian Forest Settings

The Mondrian forest algorithm developed in (Lakshminarayanan et al., 2014) has a series of parameters that can be adjusted. In this section, we present results produced using different configurations of these parameters.

A random forest is basically a set of decision trees and, in the case of the Mondrian forest, a set of Mondrian trees. Therefore, one of the main parameters of this algorithm is the number of trees use to make up the forest.

We experimented the number of trees of the Mondrian forest by creating a python script to train the Mondrian forest with a different number of trees. We used 10, 25, 50, 100, and 150 trees to create six different size Mondrian forest and trained these forests with all ten classes of the Cifar-10 dataset. The original implementation of the Mondrian forest allows the calculation of different forest related metrics, such as, the number of leaves, the number of non-leaves and the trees depth. In Table 4.4, we show the different accuracies of the different size forests, as well as the forest related metrics.

Number of trees in Mondrian forest Comparison				
Number of trees	Accuracy	Number of leaves	Number of non-leaves	Tree depth
10	0.61	215	214	13
25	0.80	1741	1740	21
50	0.84	1705	1704	22
100	0.85	1724	1723	21
150	0.85	1741	1740	22

TABLE 4.4: Number of trees in Mondrian forest comparison

In Figure 4.4, we can see that the Mondrian forest increases the accuracy significantly with the increase of number of trees from 10 trees to 25 trees, but then, with more than 50 trees, the increase of accuracy starts to stagnate. A Mondrian forest with 100 Mondrian trees achieves the same accuracy of a Mondrian forest with 150 Mondrian trees.

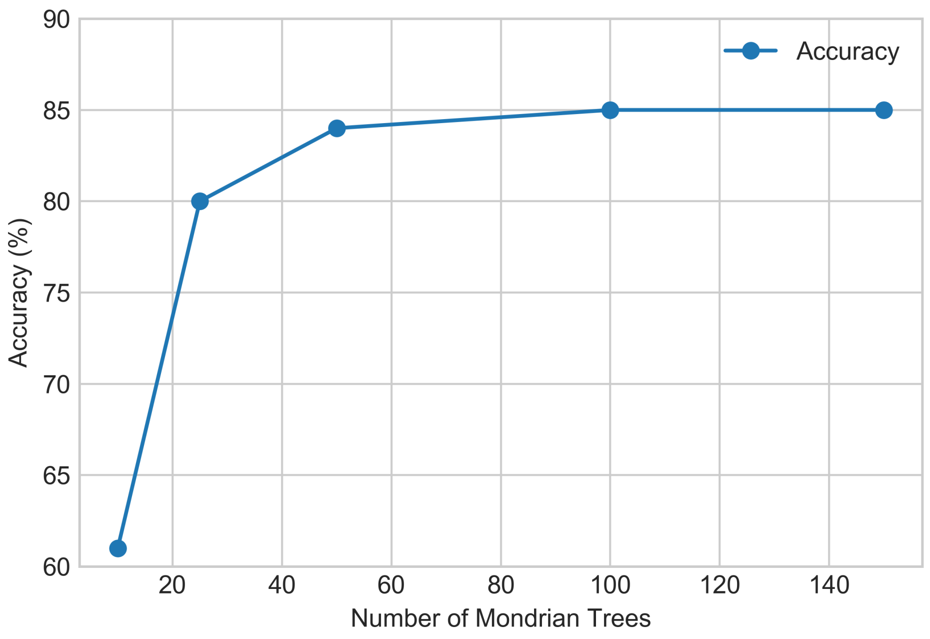

FIGURE 4.4: Number of trees in Mondrian forest comparison - Graph

As for the average number of leaves, non-leaves nodes and tree depth the results do not vary much with the increase of number of trees, except for a Mondrian forest with 10 trees where these metrics are smaller.

The other parameters that can be set for the Mondrian forest did not show significant changes in the results, therefore, we have decided to use the default ones. These parameters included the **budget** for the Mondrian tree, the **discount factor** and the **bagging** (Lakshminarayanan et al., 2014).

4.2 Defect Detection Server

To develop the DD Server, described in section 3.1, we chose to use python programming language, because it is one of the most used languages in machine learning and due to the fact, it supports many machine learning and deep learning libraries, like the ones we used to develop our classification model.

The DD Server consists of a socket with an IP address and a port and handles multiple clients with the use of threads, one for each client. In this case a client consists of the mobile application used by a quality control officer. When a quality control officer connects to the DD Server, the DD Server sends back to the quality control officer the classification of one or multiple components or the confirmation that the images it received were saved for future training sessions, depending on the process the quality control officer triggers.

During the development of the DD Server we defined two main processes that it must execute, the **defect detection process** and the **training process**. The defect detection process is where the server classifies the images of the components of the clothing items, compares the predictions with the product data sheet and registers the defects. The training process consists of saving images of the components in the image database and train the model when enough images are available. However, before any of these processes is executed, the DD Server must perform **data pre-processing** on the pictures it receives, we describe this data pre-processing step in the next section.

4.2.1 Data Pre-processing

To feed the classification model present in the DD Server, some data pre-processing must be done regardless of whether the quality control officer has started the training process or the process of defect detection. The content of the bounding boxes, which represents an image of a component of a clothing item, is cropped from the clothing item picture and resized to a certain width and height using **OpenCV** library for python. This width and height depend on the CNN architecture used for the feature extraction model, the image must be resized to fit the input layer of the CNN. In Figure 4.5 we can see a representation of this process. https://www.overleaf.com/project/5bd8785305a9f82109075d29 We used the OpenCV library because it can be used in python, which is the programming language we used to develop the DD Server, but also because it is an open source library the simplifies the processing of images and it was design for real-time applications [2].

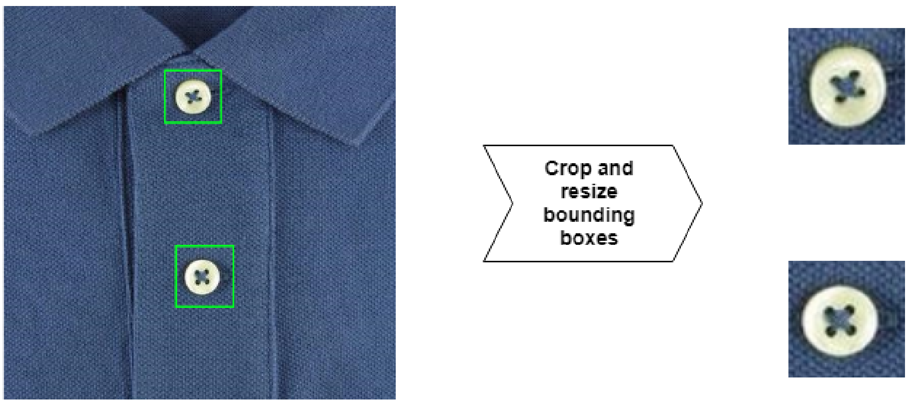

FIGURE 4.5: Data pre-processing - Crop and resize

After data pre-processing, what happens to the images depends on the process that is being executed, as we can see in the next sections.

4.2.2 Defect Detection Process

In the defect detection process, after the pre-processing of images is completed, these images are sent to the classification model. In the classification model the features are extracted from the images and then classified. We called the time it

[2] `https://opencv.org/`

takes to read the images, extract the features and predict a class, **classification time**. This classification time is evaluated in the next chapter.

After the classification is completed, the DD Server accesses the database containing the product data sheets, selects the one corresponding to the clothing item that is going through the process and compares the predicted classes to the ones present in the list of components of the product data sheet, if a defect is detected it is registered by the DD Server. Finally, the results are sent to the quality control officer. Figure 4.6 shows a representation of the defect detection process.

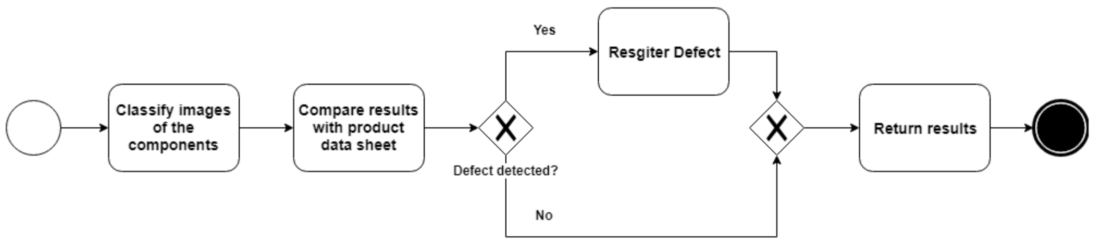
Classify images of the components
Compare results with product data sheet
Defect detected?
Yes
No
Resgiter Defect
Return results

FIGURE 4.6: Defect detection process

4.2.3 Training Process

In the training process, after the pre-processing of images is completed, the images of the components are saved in a directory and augmentations of these images are made to create more data. If there are enough new images and the number of images per class is about the same, these images are loaded, the features are extracted and the classifier within the classification model is trained, if not the DD Server waits for more images.

To train the classifier, the DD Server needs to load it, train it and then save it. We call the time it takes to load the images, extract the features, load the classifier, train the classifier and save the classifier, **training time**. This training time is also evaluated in the next chapter.

To determine if the number of new images is enough to train the classification model, the DD Server uses a threshold. This threshold is defined by the quality control officer responsible for the supervision of the QCSCM training and performance. Every time the DD Server saves new images it also saves the number new images per class. If the sum of these numbers is higher than the threshold the classifier is trained, and the number of new images is set to zero. Figure 4.7 shows a representation of the training process.

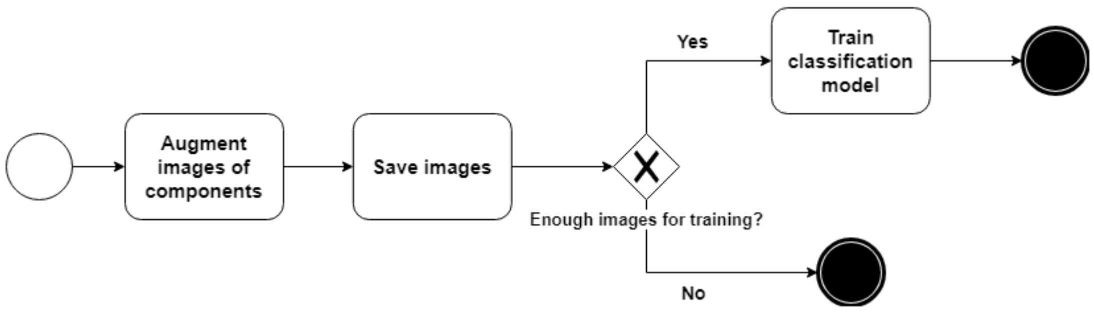
Augment images of components
Save images
Enough images for training?
Yes
Train classification model
No

FIGURE 4.7: Training process

As we said, in order to create more data, we perform some **data augmentation** to the images the DD Server saves for training. This data augmentation process includes augmentations such as, vertical flip, horizontal flip, rotations, noise, blur, etc. In Figure 4.8, we can see some of the augmentations made to an image received by the server.

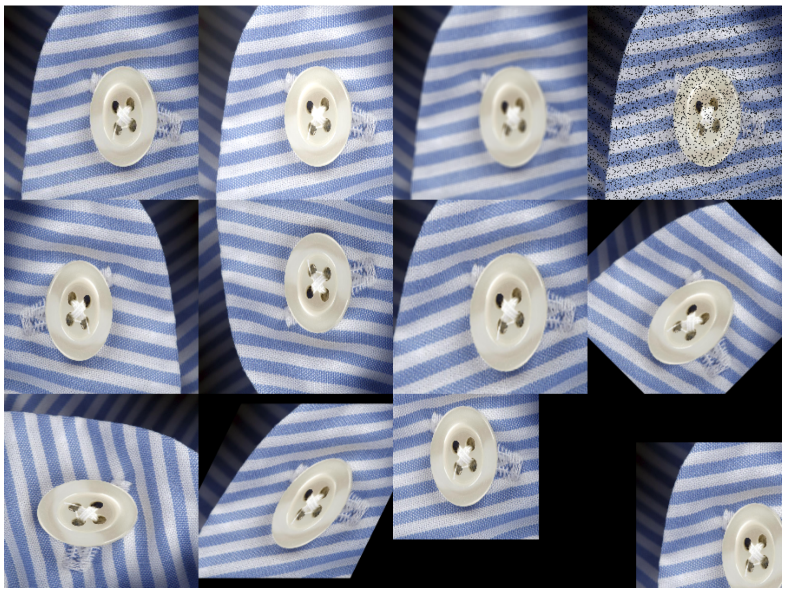

FIGURE 4.8: Image augmentations performed

Using the developed mobile application, we describe in the next section, we took pictures of clothing items and draw bounding boxes of the components that compose the clothing item. The images of the components were stored in the image database of the DD Server creating a custom dataset with a total of 18 classes. These classes are the following: zipper-white, zipper-silver, zipper-black, button-grey, button-black, button-bronze, button-white, button-yellow, button-blue, button-red, belt buckle-gold, belt buckle-silver, belt buckle-black, pocket-yellow, pocket-red, stamp1, stamp2, stamp3. Notice that for example, a black button does not belong to the same class as a blue button. Only components with the same exact characteristics belong to the same class.

Some experiments were made to see how the classification model performed on this dataset as well as which were the best augmentations to perform. First, we trained and tested our classification model on the custom dataset without perform any single augmentation and achieved an accuracy of 74%.

Trying to improve these first results, 11 augmentations were made to each image of the dataset. These augmentations were as follows: add, blur, dropout, lateral flip, up flip, perspective, 45º rotation, 90º rotation, shear, shift up and left, shift down and right. After this process each class contained 144 images.

We trained and tested the classification model using the now augmented dataset and achieved an accuracy of 74% as well. This means the augmentations that were used did not improved the training of the classification model.

After some digging of the results we achieved in the previous experiment, we noticed that the wrong predicted images were the ones where the augmentation created black borders our dots in the images, such as, dropout, shear, shift and 45º rotation. Figure 4.9 shows these augmentations.

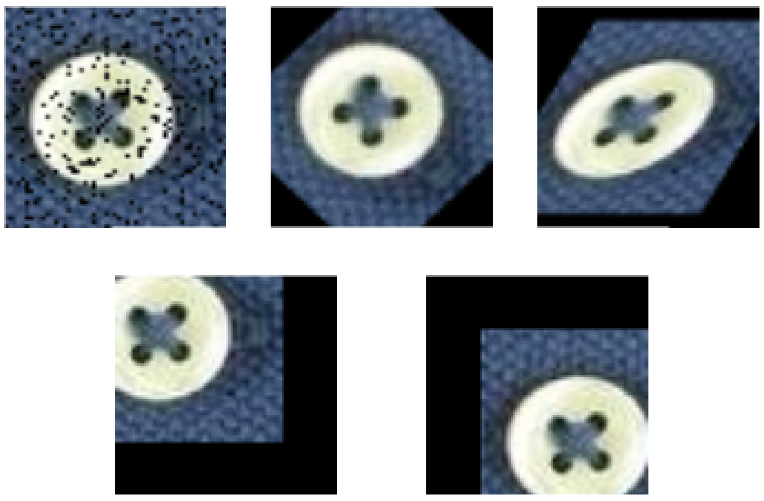

FIGURE 4.9: Bad image augmentation techniques

Taking this into account, these augmentations were removed, and new ones were added, such as, -90º rotation, 180º rotation and subtraction. Then, we trained and tested the classification model on the newly augmented dataset and the accuracy improved from 74% to 86%. After this process each class contained around 108 images. In both experiments the augmentations were performed using *imgaug* library for python. In Table 4.5 a comparison between the augmentations used is presented.

Data augmentation accuracies comparison (%)		
No augmentations	First augmentations	Second augmentations
0.74	0.74	0.86

Table 4.5: Data augmentation accuracies comparison(%)

Chapter 5

Classification Model and QCSCM Evaluation

In this chapter we evaluate our proposed classification model used in the QCSCM, ensuring that it fulfills the propose of the present work. The two main topics of the evaluation are as follows:

- **Speed**. This evaluation topic focuses on the time the classification model takes to predict a class of a component and consequently the time the QC-SCM takes to detect a defect. It also focuses on the time it takes for the classification model to train with need data. The shorter the time, the higher the speed.
- **Performance**. The second topic evaluates the performance of the classification model when classifying the images of the clothing items. In other words, we measured some classification metrics to evaluate the classifications made by the classification model.

At the time of the writing of this work, the QCSCM was not yet implemented in the real clothing factory environment. That is why, in this section we focus on evaluating the classification model of the QCSCM and in a **simulated environment**. The evaluation of the system as a whole will be made as soon as the implementation is completed, which was set to be made by INOV INESC Inovação at its clothing manufacturing client in January of 2019. This evaluation includes a satisfaction survey of the quality control officers to understand if the mobile application is user-friendly and if the QCSCM can indeed be useful in their jobs.

In following evaluation and experiments we put ourselves in the position of the quality control officers and used the developed system, more precisely the mobile application to take pictures of clothing items and create bounding boxes of the components. The pictures were sent to the DD Server that stored the images of the components along with the labels in the **image database creating a custom dataset**. This dataset consists of around 2100 images divided in 18 classes listed in the previous chapter. These images include images generated with data augmentation techniques.

To evaluate the classification model, the dataset had to be divided in a training set and a test set. The training set is about 80% of the total dataset and the test set is the remaining 20%. This is a common split for datasets in machine learning problems.

5.1 Classification Model Speed

In this section we focus on how fast the classification model is. The following experiments were performed using a Nvidia GeForce GTX 1080 Ti GPU, 16GB of RAM and an Intel Core i5-7600K CPU.

5.1.1 Training time

The training time is, as the name suggests, the time it takes to train the classification model. Evaluating this time is important because the QCSCM needs to be quickly updated as new classes of components are created. If it takes too much time to train the classification model, the quality control officers will not be able to use the QCSCM for the new components.

The training process of the classification model can be divided in the following sub-processes, as shown in section 4.2.3:

- Load images.
- Extract features.
- Load classifier.

- Train classifier.
- Save classifier.

Each of these sub-processes has an associated time and the sum of these times represents the total training time. Table 5.1 shows these training times measured in seconds (s).

Training times with increase of classes							
Number of classes	Number of images	Load images	Extract features	Load classifier	Train classifier	Save classifier	Total training
3	250	0.94s	13.39s	-	2.99s	8.72s	26.04s
6	496	1.77s	21.44s	6.45s	5.36s	14.91s	49.93s
9	750	2.7s	32.28s	7.34s	6.89s	17.19s	66.41s
12	996	3.79s	43.18s	8.49s	8.02s	17.33s	80.81s
15	1246	5.09s	53.87s	8.6s	9.42s	17.24s	94.24s
18	1500	6.41s	65.24s	8.35s	10.66s	19.01s	109.68s

TABLE 5.1: Training times with increase of classes.

To measure the training times presented in Table 5.1 we created a python script that performed six training sessions and calculated the time of each sub-process of training using the *time* package. For each of the six training sessions we added three new classes, training the classification model incrementally. The first training session used three classes and the sixth and last training session used all 18 classes of the created dataset.

As expected, with the increase of the number of classes, and therefore an increase of the number of images, the training times increase as well. The increase of number of classes and number of images is linear, with the classes increasing in three at each training and the number of images increasing in around 250 images. Figure 5.1 show us that the total training time also increases in a linear way. Time increases between 14 to 16 seconds after each increase in the number of classes, except for the first increase, where the time increased about 24 seconds. This is related to the fact that in the first training a new classifier is created from scratch and no time is spent on loading the classifier since there are no older classes.

If we take a closer look to the time of each sub-process, we can see that the sub-process of extracting features is the one with the biggest increases in time when new classes are added, and more images are loaded. The other sub-processes also increase in time but in a much softer way. Figure 5.2 presents the time increase for each training sub-process after each increase in the number of classes.

After testing the training of the classification model with an increase of classes, we decided to test classification model with all the classes trained from scratch,

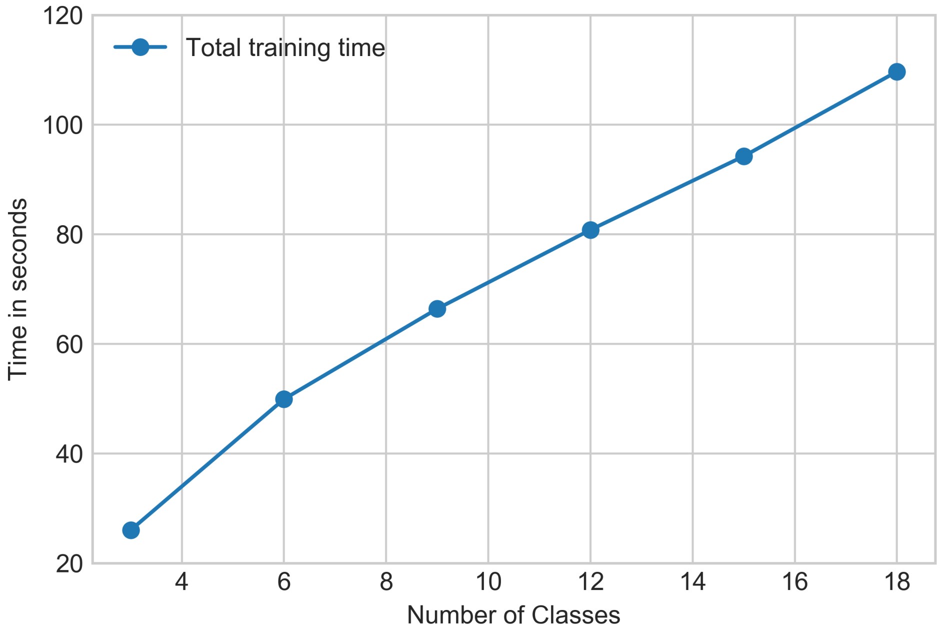

Figure 5.1: Training times with increase of classes - Graph.

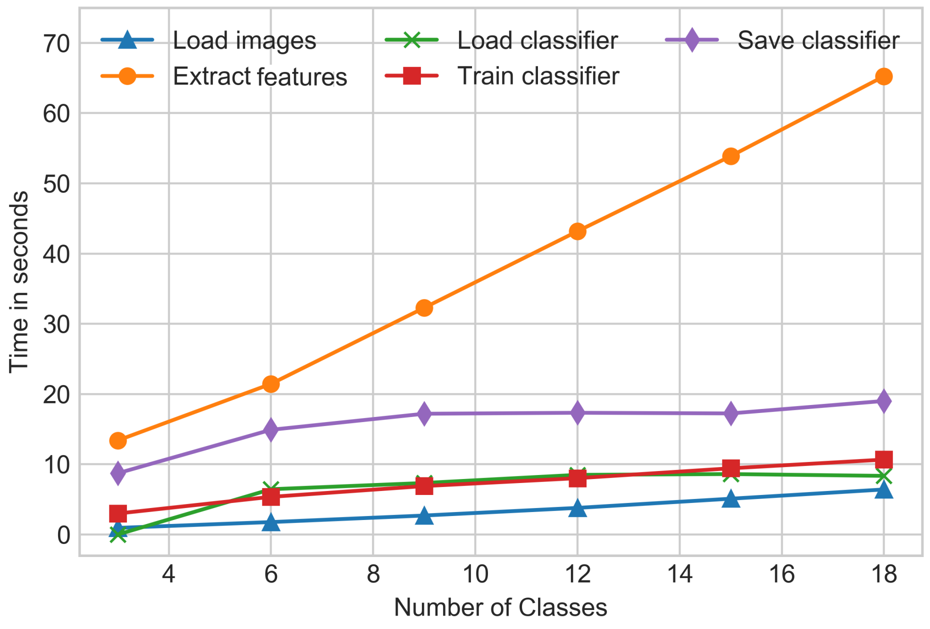

Figure 5.2: Training sub-processes times with increase of classes - Graph.

just increasing the number of images, and see if the training times differ. In other words, see if the number of classes influences the training time or if it is just the number of images.

To perform this experiment, we created another python script that also perform five training sessions and measured the training times using the *time* package. For each training session the classification model was trained with all 18 classes of the created dataset using around 250 new images per training session. Table 5.2 presents the results of this experiment.

Training times with increase of number of images						
Number of images	Load images	Extract features	Load classifier	Train classifier	Save classifier	Total training
243	3.94s	13.09s	-	3.13s	10.34s	30.51s
495	4.35s	21.07s	7.58s	6.79s	22.08s	61.86s
740	4.91s	31.58s	9.01s	8.22s	23.48s	77.18s
991	5.38s	42.3s	8.76s	9.31s	23.83s	89.58s
1243	5.88s	53.48s	9.08s	10.7s	23.63s	102.77s
1500	6.43s	64.91s	8.82s	12.08s	23.36s	115.61s

TABLE 5.2: Training times with increase of number of images.

As the table shows, it is possible to see that the training times are slightly bigger when training with all classes. In Figure 5.3 is possible to compare the results of both experiments.

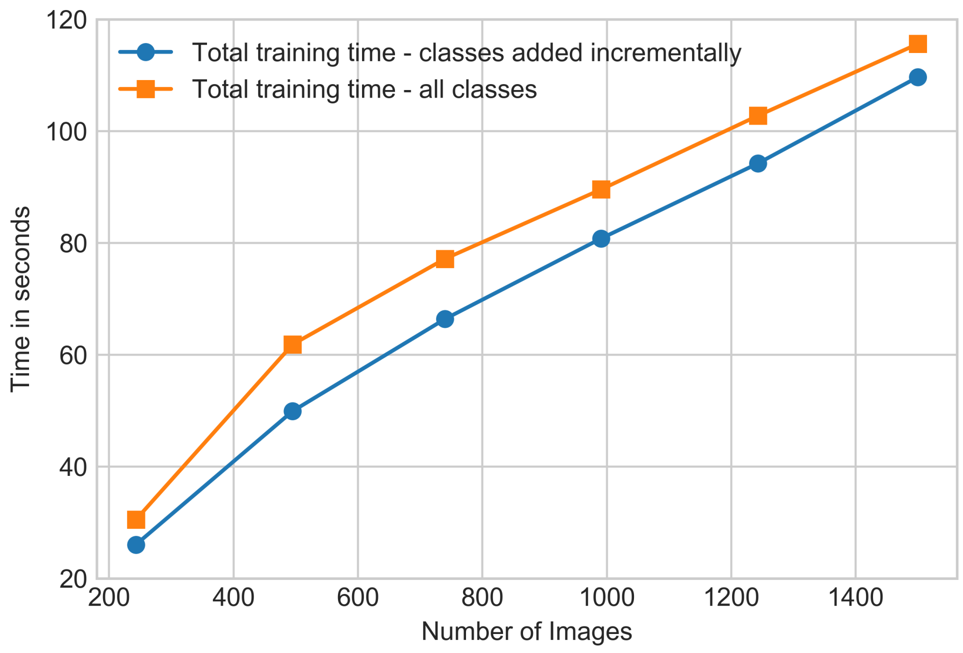

FIGURE 5.3: Training times comparison - Graph.

In Figure 5.4, we can see a more detailed view of the two experiments. By comparing the sub-processes times, it is possible to see that the times are similar in the *extract features* and *load classifier* sub-processes, but differ in the remaining ones.

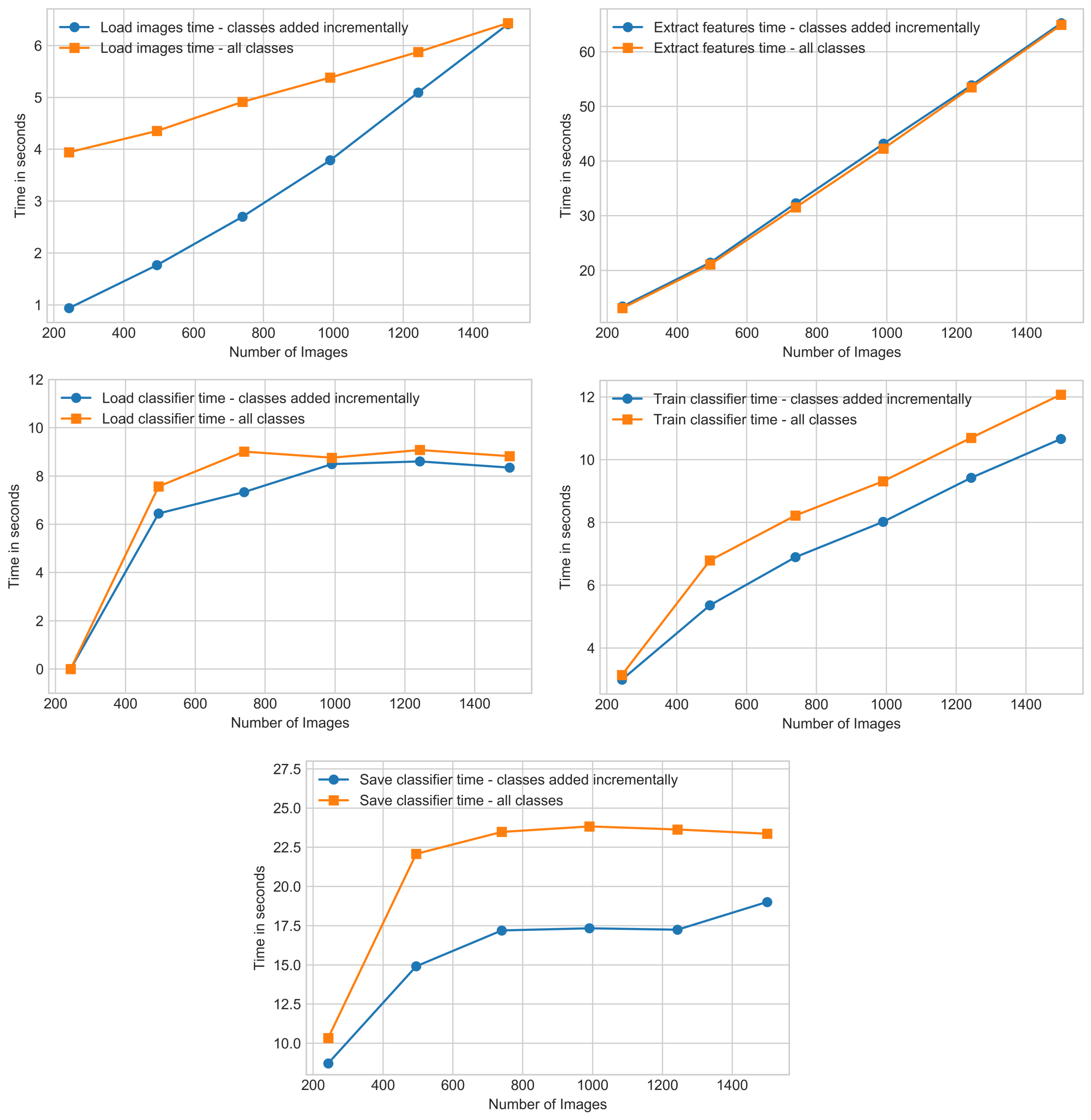

FIGURE 5.4: Training sub-processes times comparison - Graph (Row 1 - Load images, Extract features; Row 2 - Load classifier, Train classifier; Row 3 - Save classifier.

The difference of the *load images* sub-process times can be explained by the fact that the images are stored in a directory for each class. Therefore, as the number of classes increases, the number of directories that need to be access increases as well. In the final training instance of each experiment, the number of images and classes are the same for both experiments, thus the time it takes to execute this sub-process is the same.

In the *train classifier* and *save classifier* sub-processes the times of the second experiment, the one on which the classifier was trained in every training instance with all the 18 classes, are bigger than the times of the first experiment. This is related to the size of the classifier, as the number of classes increases the number, the classifier size increases as well.

After these experiments we can conclude that the number of classes does not have a significant impact in the training times of the classification model. Although every time we added classes the training time increased, it was not due to the fact the number of classes increased, but due to the increase of the number of images.

In both cases the training time in the final training instances was bellow two minutes, which is a satisfactory result. Most of the images classification algorithms take much more time to training.

By having a classification model able learn new information fast, the quality control officers and the quality control processes can adapt rapidly to new components of clothing items and detect new defects. On top of that, the training of the classification model can be performed outside of working hours, in such a way it does not interfere with the clothing manufacturing works.

5.1.2 Classification time

The classification time corresponds to the time during which a quality control officer sends a picture with components of clothing items that need to be classified and the server outputs the results. This classification process be divided in sub-processes as follows, as shown in section 4.2.2:

- Read image.
- Extract features.
- Predict classes.

The *read image* sub-process consists of receiving the image from the mobile application and convert it to the correct format, then the bounding boxes contain within the image are cropped and resized creating new images that will be send to the classification model for the *extract features* and *predict classes* sub-processes.

With the classification model trained with all 18 classes, we as quality control officers took some pictures with the mobile application, created bounding boxes around the components we wanted to classify and sent the information to the server in order to calculate the time of the classification and its sub-processes. Table 5.3 presents the times calculated during these experiments, where we created one up to five bounding boxes per picture. Each time was calculated after each sub-process of classification was completed using the *time* python package.

Classification Times				
Bounding boxes	Read image	Extract features	Predict classes	Total time
1	0.186s	0.04s	0.103s	0.329s
2	0.181s	0.08s	0.217s	0.479s
3	0.205s	0.121s	0.299s	0.624s
4	0.205s	0.156s	0.379s	0.74s
5	0.182s	0.196s	0.454s	0.832s

TABLE 5.3: Classification times.

In information retrieved from the clothing factory for which the classification model and the QCSCM were developed, we know that it takes on average around 18 seconds for a quality control officer to detect a defect. Therefore, for the QCSCM to be useful for the factory it must perform at least within the same values of time.

As we can see in Figure 5.5, in all cases of the experiment, it takes less than a second for the QCSCM to read the picture and identify the classes of the components. As expected, it takes longer to predict five classes than one class, but except for the *read image* time, which is similar in all cases, the increase of the *extract features* time and the *predict classes* time is linear. This makes it easier to estimate the time it takes to identify n components.

Although the results of this experiment seem promising, it is hard to tell if the defect detection process speed is improved with the use of the classification model and QCSCM. The time it takes for a quality control officer to take a picture or more of the clothing item was not considered, because the system was not yet implemented in the factory. Therefore, it is not possible to compare these results with the time it takes a quality control officer to detect a defect.

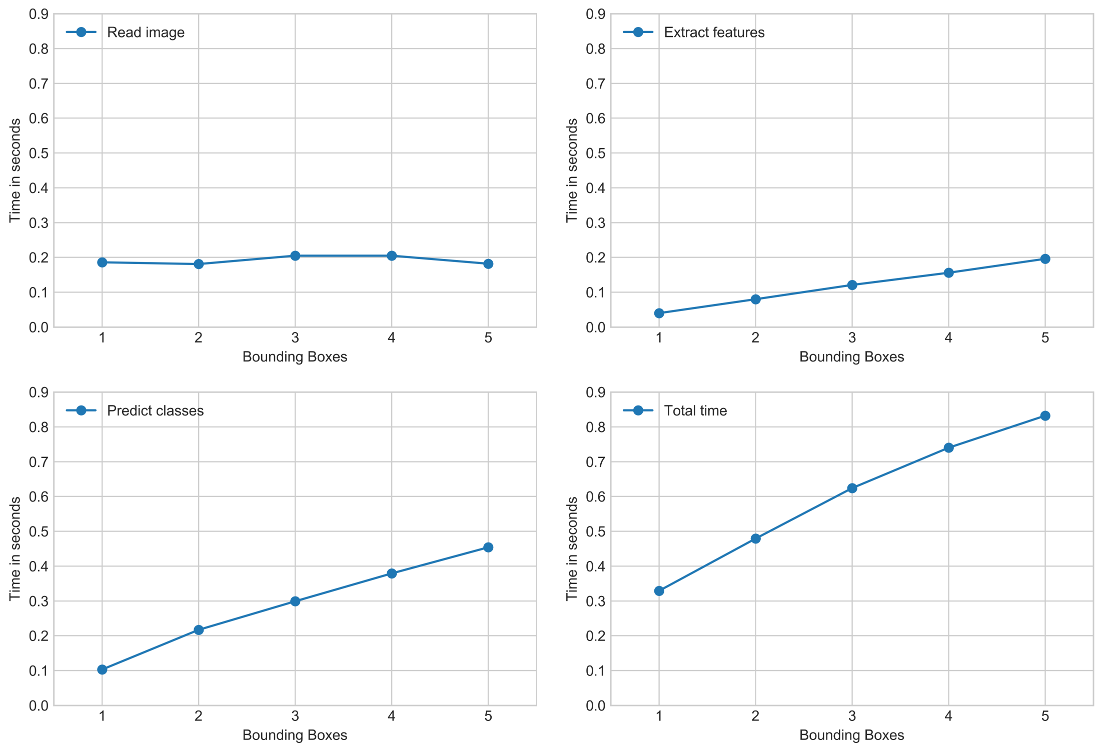

FIGURE 5.5: Classification Times - Graph (Row 1 - Read images, Extract features; Row 2 - Predict classes, Total time).

5.2 Classification Model Performance

The classification model must be an efficient tool in order to be a valid option for the QCSCM and for the quality control officers in their quality control processes. To measure how efficient the classification model is, we calculated some classification metrics using the custom dataset. We evaluated the incremental learning performance of the classification model and its overall performance when trained with all classes of the dataset.

5.2.1 Incremental Learning

To evaluate the incremental learning capability of the classification model two experiments were performed, the metric used to evaluate these experiments was the accuracy. To calculate the accuracy, we used the equation 4.1 described in the previous chapter.

The first experiment aimed to evaluate how the classification model performed when new classes were added to the system. In Table 5.4 is possible to see the results of this first experiment.

Incremental training with new classes	
Number of classes	Accuracy
3	0.91
6	0.95
9	0.92
12	0.91
15	0.91
18	0.95

TABLE 5.4: Incremental training with new classes.

We created a python script that trained the classification model using the training set of the dataset created. First, the classification model was trained with three classes and then three classes were added incrementally until it was trained with all the 18 classes. To measure the accuracy, we used the test set of the dataset. As we can see in Table 5.4 and more easily in Figure 5.6, the increasing number of classes does not have a significant effect on the accuracy of the classification model, averaging between 91% and 95%. Taking these results in to account we can conclude that the classification model was capable of learn new classes of the created dataset without significant drops of accuracy.

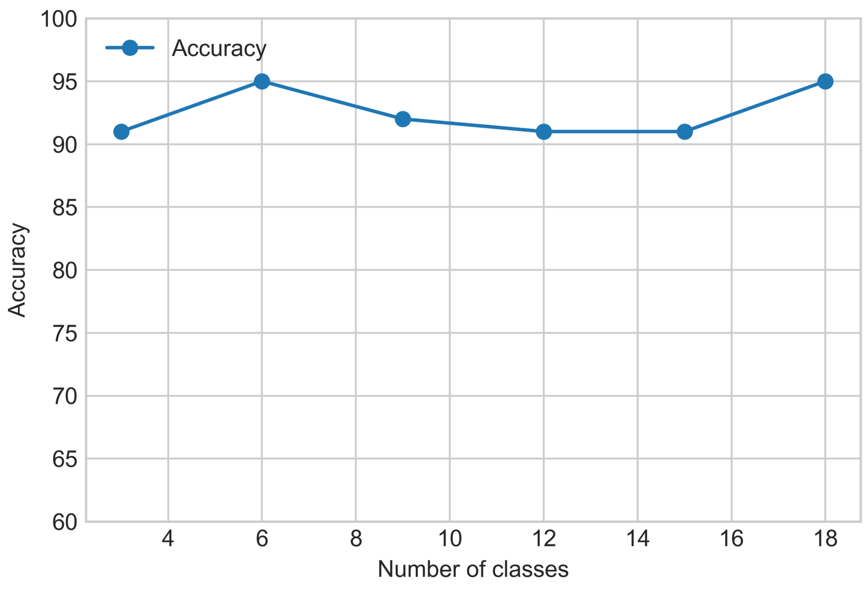

FIGURE 5.6: Incremental training with new classes - Graph.

Contrary to what one would expect, the accuracy did not decrease with the increase of the number of classes. This can be explained by the differences and similarities of the classes used in each training session. The first classes used were more similar between each other, therefore, the classification model found it more difficult to distinguish the classes. The last classes to be used to train the classification model were more different, thus, not decreasing the accuracy.

In the second experiment aimed to check if the classification model can improve its knowledge in existing classes. We created another python script to train the classification model with a fixed number of classes, then it was retrained using new and more data of the same classes, in a total of five training sessions. The experiment was performed using three, ten and 18 classes. Table 5.5 shows the results of this experiment. The column *Number of Images* represents the number of images that were used to train the classification model at each training step.

Incremental training with new data					
3 classes		10 classes		18 classes	
Number of images	Accuracy	Number of images	Accuracy	Number of images	Accuracy
50	0.67	167	0.79	300	0.88
50	0.78	167	0.88	300	0.91
50	0.81	167	0.91	300	0.94
50	0.84	167	0.94	300	0.95
50	0.91	164	0.96	300	0.96

TABLE 5.5: Incremental training with new data.

As we can see in Figure 5.7, in all cases the classification model improves its accuracy when trained with new unseen data. We can conclude that the classification model is capable of improve its knowledge when trained with new data of old classes. Therefore, the user feedback feature of the QCSCM is important, as it helps the quality control officers create new data that will be used in future training sessions and, consequently, improve the classification model performance.

Again, the higher accuracy of the classification model on the higher number of classes can be explained by the differences and similarities of the classes used in the train with three classes, ten classes and 18 classes. When we trained the model with 3 classes those classes were all zippers, zipper-white, zipper-silver and zipper-black, which are more similar then the rest of the 18 classes, thus the lower accuracy. Another factor can be associated to the number of images used in the different training sessions. The model when trained with 18 classes used more images then when trained with three classes. However, what is important to

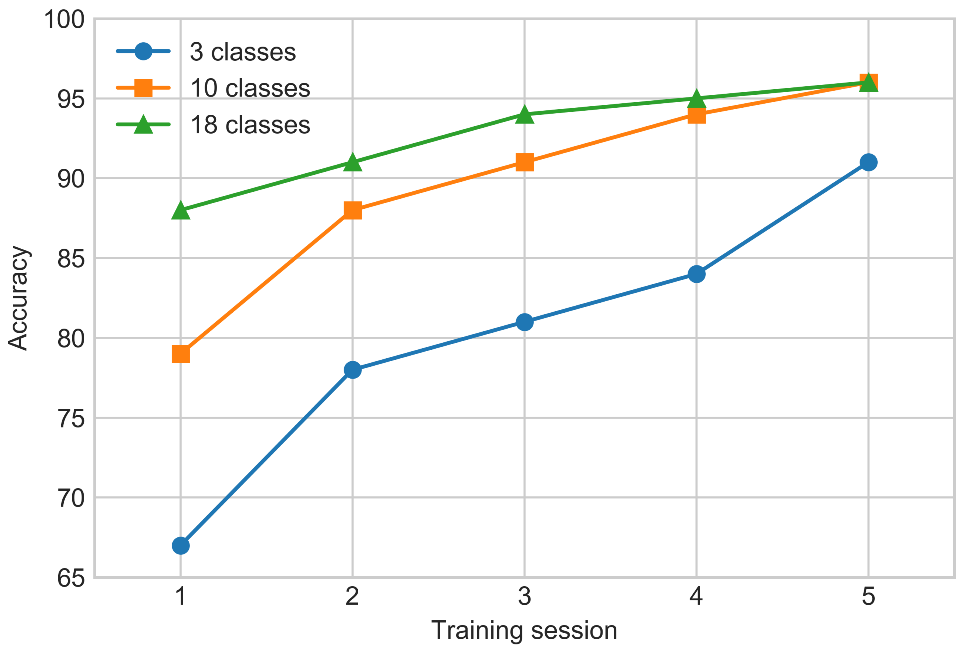

FIGURE 5.7: Incremental training with new data - Graph.

conclude is that, no matter the number of classes, the classification model increase its accuracy when trained with new data from old classes.

5.2.2 Classification Performance

In the previous section, we used accuracy as the metric to evaluate the incremental learning abilities of the classification model. The results were promising, but the use of this metric can be misleading sometimes. In this section, we evaluate the performance of the classification model using more metrics. The classification model was trained with all 18 classes of the dataset we created.

Using the training set of the dataset and all 18 classes we created a python script to train the classification model and then evaluated the model using the test set. In Table 5.6 we can see a confusion matrix describing the performance of the classification model on the test set.

The diagonal elements of the confusion matrix show the correct classifications (true positives (TP)), the columns excluding the diagonal elements show the false positives (FP) and the rows excluding the diagonal elements show the false negatives (FN). This confusion matrix was created using the *scikit-learn* library.

Confusion matrix																		
Class	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
1	19	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	2	18	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	22	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	14	0	0	5	1	1	0	0	0	0	0	0	0	0	0
5	0	0	0	0	20	0	0	0	1	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	21	0	0	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	22	0	0	0	0	0	0	0	0	0	0	0
8	0	0	0	1	0	0	0	20	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	22	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	21	0	0	0	0	0	0	0	0
11	0	0	0	0	0	0	0	0	0	0	21	0	0	0	0	0	0	0
12	0	0	0	0	0	0	0	0	0	0	0	21	0	0	0	0	0	0
13	0	0	0	0	0	0	0	0	0	0	0	1	21	0	0	0	0	0
14	0	0	0	0	0	0	0	0	0	0	0	0	0	21	0	0	0	0
15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	21	0	0	0
16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	21	0	0
17	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	22	0
18	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	22

TABLE 5.6: Confusion matrix - Class legend: 1. zipper-white; 2. zipper-silver; 3. zipper-black; 4. button-grey; 5. button-black; 6. button-bronze; 7. button-white; 8. button-yellow; 9. button-blue, 10. button-red; 11. belt buckle-gold; 12. belt buckle-silver; 13. belt buckle-black; 14. pocket-yellow; 15. pocket-red; 16. stamp1; 17. stamp2; 18. stamp3

With the help of the confusion matrix it is possible to calculate the precision, the recall and the F1-score. The precision evaluates how accurate are the predictions, in other words it calculates proportion of positive classification that were correct using the following equation:

$$precision = \frac{TP}{TP + FP}. \tag{5.1}$$

The recall measures how good the model finds all the positives. It calculates the proportion of positives correctly classified using the following equation:

$$recall = \frac{TP}{TP + FN}. \tag{5.2}$$

Finally, the F1-score is the harmonic mean of the precision and the recall. It is calculated using the following equation:

$$F1 = 2 * \frac{precision * recall}{precision + recall}. \tag{5.3}$$

These metrics allow a better interpretation of the classification model performance. To calculate these metrics, we use the *scikit-learn* library and the information shown in the confusion matrix. The results of these calculations are present in Table 5.7. As shown in this table, the metrics are high across all classes except for class number four, which has a lower recall, and class number seven, which has a lower precision.

Class	Precision	Recall	F1-Score
1	0.90	0.90	0.90
2	0.90	0.86	0.88
3	0.96	1.00	0.98
4	0.93	0.67	0.78
5	1.00	0.95	0.98
6	1.00	1.00	1.00
7	0.81	1.00	0.90
8	0.95	0.95	0.95
9	0.92	1.00	0.96
10	1.00	1.00	1.00
11	1.00	1.00	1.00
12	0.95	1.00	0.98
13	1.00	0.95	0.98
14	1.00	1.00	1.00
15	1.00	1.00	1.00
16	1.00	1.00	1.00
17	1.00	1.00	1.00
18	1.00	1.00	1.00

TABLE 5.7: Precision, recall and F1-score

In the case of class number four, which is button-grey, the high precision and low recall implies that the classification model does not classifies many things as button-grey, missing a lot of them. However, when it classifies an object as button-grey it is very precise.

As for the case of class number seven, button-white, the high recall but lower precision implies that the classification model correctly classifies a significant proportion or even all the white buttons as button-white. However, it also incorrectly classifies other classes as button-white.

These results also confirm our explanation for the classification model having a higher accuracy when trained with all 18 classes then when trained with just the first three classes of the dataset. The first three classes have a lower precision compared to the final classes.

The confusion between the white button and the grey button classes, can also be explain by the light conditions. If the light condition are not consistent a white button can indeed look very similar to a grey button.

Since the number images per class is quite balanced, we can average the results of each class and get the overall precision, recall and F1-score. This is called macro-averaging. The overall metrics, converted to percentages, along with the accuracy of the classification model is presented in Table 5.8.

Evaluation metrics			
Accuracy	Precision	Recall	F1-Score
96.09%	96.29%	96.04%	95.96%

Table 5.8: Evaluation metrics

The results achieved on the evaluation metrics show us that the classification model performs well on the dataset we created in a simulated environment. However, this dataset and environment represent just a fraction of the larger real environment of the clothing factory.

5.3 QCSCM Simulation

To further evaluate the classification model and to test the QCSCM, we experimented the QCSCM by taking some pictures of clothing items. Some of these pictures are presented here, where we can see how the QCSCM performed on them.

To take these pictures, we installed the developed mobile application in three mobile devices from INOV collaborators and **created a simulated environment** over a period of one week. The three installed mobile applications allowed us to put ourselves in the role of quality control officers.

By installing the mobile applications in multiple devices in the simulated environment we created, we were capable creating more images to be used by the QCSCM in a **collaborative way**. All of the installed mobile applications were capable of connecting to the DD Server allowing a faster creation of images and subsequently a better training of the classification model.

In Figure 5.8 it is possible to see some examples of correct classifications. On the left, a picture of a shirt sleeve with a bounding box around a component correctly labeled as button-white. On the middle, a picture of part of a belt with its buckle surrounded with a bounding box correctly classified as silver belt buckle. On the right, a picture of a polo shirt with two bounding boxes correctly classified as white buttons.

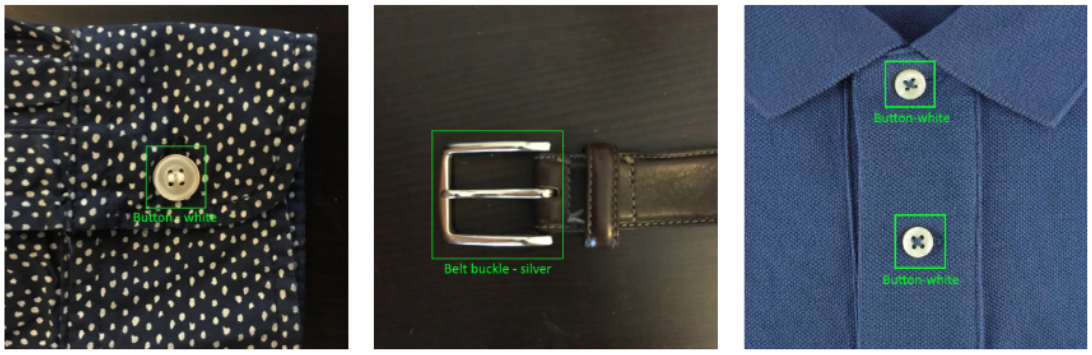

FIGURE 5.8: Examples of correct classifications

As the Figure 5.8 also shows, the QCSCM can use the classification model to classify more than one component at a time. The picture on the right has two bounding boxes correctly classified.

In the real quality control environment, the quality control officers when receiving results such as the ones present in the figures above, could confirm the results and create new images for training with them. As for the DD Server, it would register a defect in case of one being detected.

As seen in previously the classification model is not 100% accurate, sometimes it makes wrong classifications of clothing items components. Figure 5.9 shows some of these cases. On the left, we can see a silver zipper mistakenly classified as a white zipper. On the right, it is possible to see four bronze buttons, three of them correctly classified but one incorrectly classified as a black button.

Some important information can be retrieved from these examples of incorrect classifications. In these examples the classification incorrectly classified the components, however the main characteristic of the components was correctly classified. In the case of the silver zipper, the component was correctly classified as a zipper, but the color was incorrect. The same for the buttons example, all of them were classified as buttons, but in one of them the color was incorrect. This suggests that some class hierarchy and multi-label classification could improve

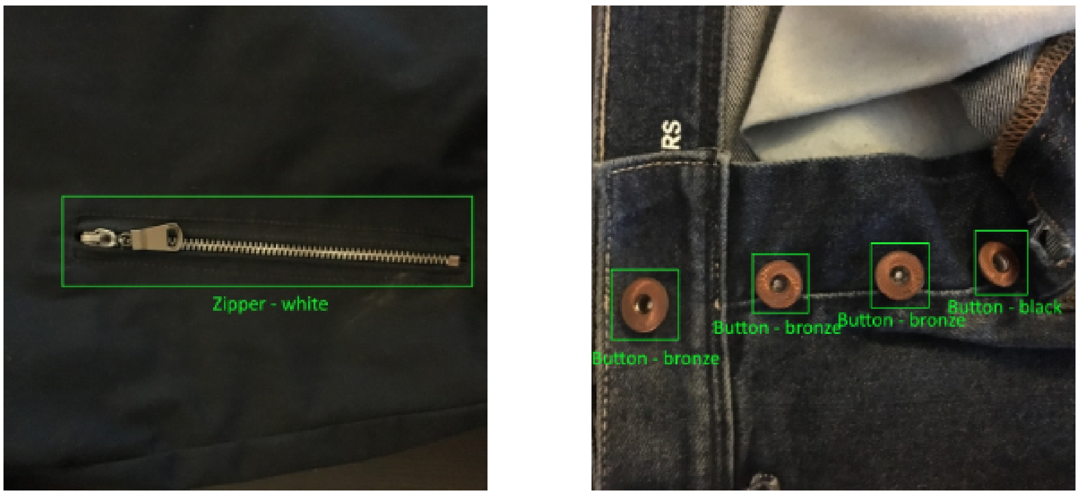

FIGURE 5.9: Examples of incorrect classifications

the performance of the classification model, since the are many components that shared some characteristics.

As said before, when the quality control officer receives incorrect results, he should make use of the user feedback feature of the QCSCM and correct wrong predictions made by the classification model. This will help the classification model improve its accuracy, as we saw in the previous section.

The pictures and results presented here, represent a simulated environment similar to the one present on the clothing factory. In terms of the QCSCM usability, we were able to take pictures using the developed mobile application in multiple mobile devices and send the information to the DD Server. Then, the DD Server used the classification model to classify the components present in these pictures and send back the results to the mobile application. Using also the developed mobile application, we created the dataset used to train the classification model. As for the classifications made by the classification model, it is hard to tell if the results would be similar in the real environment because of multiple factors, such as, light conditions, mobile phone camera specifications and configurations, etc. However, it was possible to verify that the classification model can correctly classify the components we created and can improve its knowledge as more images are created for training.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

The goal of the present work was to develop a system, that makes use of an image classification model capable of learning new classes incrementally and increase its knowledge, to help the quality control officers of a clothing factory in their quality control processes. Using a mobile application used by the factory workers combined with a server containing a classification model created using machine learning algorithms, we proposed the QCSCM. This system can classify objects that compose clothing items, checking if the identified objects correspond to the ones used to produce a certain clothing item.

The main topics that the present work addresses are image classification and incremental learning. Therefore, we focused our literature review in these two topics. This review revealed that deep learning achieves the best results in the field of image classification and that incremental learning is an area still in need of further research. To combine these two topics of research in our final system, we also looked for transfer learning techniques. This way we could create a classification model using a deep learning neural network that could extract important features from the images and a classifier with incremental learning capabilities, where new classes can be added over time.

After completing the research and defining the classification model and the QCSCM architecture, we decided to use a CNN as feature extractor of the classification model and Mondrian forest as the classifier. During the development of the classification model some experiments were perform using different CNN architectures, to check which of these produced the better results and therefore, be used in the final version of the QCSCM. The results of these experiments have shown that the InceptionResnet architecture was the one to use in the classification model. The Mondrian forest algorithm implementation was slightly modified so that new classes could be added over time.

A mobile application was created, with the propose of being used by the quality control officers to take pictures of the clothing items and then draw bounding boxes around the relevant components of the clothing items creating images of the components. These images could be used to train the classification model or to perform the defect detection. Using the mobile application and the incremental learning capabilities of the classification model, an important feature of the QCSCM, the User Feedback, was developed. This feature allows the quality control officer to correct and confirm the results of the classification model and create new data that will be use in future training sessions.

To evaluate the classification model, we focused on two topics, the speed of the model and the performance of the model. The classification model was created to be used in a quality control system to improve a quality control process. In this type of processes time resources are important, therefore, the classification model should perform close to real-time. The training times and the classification times were the metrics we used to evaluate the speed. For the performance evaluation we focused on classification metrics of the classification model and how it performed when trained in an incremental fashion.

In both evaluation topics, the experiments were made using a custom dataset in a simulated environment, where we put ourselves in the role of a quality control officer to create data and classify some clothing item components. The results achieved although promising, where difficult to compare to the real-life environment of the factory, due to different conditions, lack of data of real clothing items and feedback of the quality control officers.

In the first chapter of the present work we asked the following question:

- How to develop a system capable of identifying defects and gain more knowledge over time in a robust and efficient way using machine learning algorithms to improve the quality control process of a clothing?

The system we developed, the QCSCM, makes use of machine learning algorithms, is capable of identifying defects and its classification model can gain knowledge over time as new data is created. However, the system can still be improved, the identification part of the problem (creation of bounding boxes of the components) needs a big input from the quality control officers, it is not completely automated. In the next section we discuss how can this be improved.

6.2 Future Work

As said before, although we achieved some satisfactory results in the experiments we have made, it is hard to tell if the QCSCM would perform well in a real-life environment. Therefore, one of the next steps to make is to implement the system in the factory so that some quality control officers can evaluate it and so that pictures of real clothing items and its components can be created.

In terms of improving the QCSCM functionality, we will focus of implementing object detection in the system. The quality control officers must create bounding boxes of the components they want to identify. This process can be slow if there are several components on a clothing item. By implementing object detection this process can be automated, simplifying the job of the quality control officers to just taking pictures.

In the current architecture of the classification model, each different component of a clothing item corresponds to a different class. If the number of classes increases exponentially this can lead to some drops in accuracy. Also, some classes of object can be more difficult to classify then others. Taking this into account, a focus to create a class hierarchy and multi-label classification will be made. For example, the current classification model classifies a black button and a blue button as two different classes. In the future we will develop a classification model that first classifies the more generic class, such as button, zipper, pockets, and then classifies its characteristics, for example, color, size, etc.

Bibliography

Anthony, G., Greg, H., & Tshilidzi, M. (2007). Classification of images using support vector machines. *arXiv preprint arXiv:0709.3967*.

Britz, D. (2015). *The top 10 ai and machine learning use cases everyone should know about.* Retrieved from `http://www.wildml.com/2015/11/understanding-convolutional-neural-networks-for-nlp/` (Accessed: 2018-10-07)

Carpenter, G. A., Grossberg, S., Markuzon, N., Reynolds, J. H., & Rosen, D. B. (1992). Fuzzy artmap: A neural network architecture for incremental supervised learning of analog multidimensional maps. *IEEE Transactions on neural networks*, *3*(5), 698–713.

Çelik, H., Dülger, L., & Topalbekiroğlu, M. (2014). Development of a machine vision system: real-time fabric defect detection and classification with neural networks. *The Journal of The Textile Institute*, *105*(6), 575–585.

Chan, C.-h., & Pang, G. K. (2000). Fabric defect detection by fourier analysis. *IEEE transactions on Industry Applications*, *36*(5), 1267–1276.

Chapelle, O., Haffner, P., & Vapnik, V. N. (1999). Support vector machines for histogram-based image classification. *IEEE transactions on Neural Networks*, *10*(5), 1055–1064.

Chen, T., Goodfellow, I., & Shlens, J. (2015). Net2net: Accelerating learning via knowledge transfer. *arXiv preprint arXiv:1511.05641*.

Classification: Accuracy. (2018). Retrieved from `https://developers.google.com/machine-learning/crash-course/classification/accuracy` (Accessed: 2018-10-04)

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., & Fei-Fei, L. (2009). Imagenet: A large-scale hierarchical image database. In *Computer vision and pattern recognition, 2009. cvpr 2009. ieee conference on* (pp. 248–255).

Erickson, B. J., Korfiatis, P., Akkus, Z., Kline, T., & Philbrick, K. (2017). Toolkits and libraries for deep learning. *Journal of digital imaging*, *30*(4), 400–405.

Fuangkhon, P., & Tanprasert, T. (2009). An adaptive learning algorithm for supervised neural network with contour preserving classification. In *International conference on artificial intelligence and computational intelligence* (pp. 389–398).

Haralick, R. M., Shanmugam, K., & Dinstein, I. (1973). Textural features for image classification. *IEEE Transactions on systems, man, and cybernetics*, *3*(6), 610–621.

He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the ieee conference on computer vision and pattern recognition* (pp. 770–778).

Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., … Adam, H. (2017). Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*.

Ioffe, S., & Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*.

Kim, J., Kim, B.-S., & Savarese, S. (2012). Comparing image classification methods: K-nearest-neighbor and support-vector-machines. *Ann Arbor*, *1001*, 48109–2122.

Kirkpatrick, J., Pascanu, R., Rabinowitz, N., Veness, J., Desjardins, G., Rusu, A. A., … Hadsell, R. (2017). Overcoming catastrophic forgetting in neural networks. *Proceedings of the national academy of sciences*, 201611835.

Kovalev, V., Kalinovsky, A., & Kovalev, S. (2016). Deep learning with theano, torch, caffe, tensorflow, and deeplearning4j: Which one is the best in speed and accuracy?

Krizhevsky, A., & Hinton, G. (2009). *Learning multiple layers of features from tiny images* (Tech. Rep.). University of Toronto.

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems* (pp. 1097–1105).

Kumar, A. (2003). Neural network based detection of local textile defects. *Pattern Recognition*, *36*(7), 1645–1659.

Kumar, A. (2008). Computer-vision-based fabric defect detection: A survey. *IEEE transactions on industrial electronics*, *55*(1), 348–363.

Kumar, A., & Pang, G. K. (2002). Defect detection in textured materials using gabor filters. *IEEE Transactions on industry applications*, *38*(2), 425–440.

Lakshminarayanan, B., Roy, D. M., & Teh, Y. W. (2014). Mondrian forests: Efficient online random forests. In *Advances in neural information processing systems* (pp. 3140–3148).

Lakshminarayanan, B., Roy, D. M., & Teh, Y. W. (2016). Mondrian forests for large-scale regression when uncertainty matters. In *Artificial intelligence and statistics* (pp. 1478–1487).

LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, *86*(11), 2278–2324.

Li, D., Liang, L.-Q., & Zhang, W.-J. (2014). Defect inspection and extraction of the mobile phone cover glass based on the principal components analysis. *The International Journal of Advanced Manufacturing Technology*, *73*(9-12), 1605–1614.

Lu, D., & Weng, Q. (2007). A survey of image classification methods and techniques for improving classification performance. *International journal of Remote sensing*, *28*(5), 823–870.

Marr, B. (2016). *The top 10 ai and machine learning use cases everyone should know about.* Retrieved from `https://www.forbes.com/sites/bernardmarr/2016/09/30/what-are-the-top-10-use-cases-for-machine-learning-and-ai/#57cf2eb794c9` (Accessed: 2018-01-29)

Narr, A., Triebel, R., & Cremers, D. (2016). Stream-based active learning for efficient and adaptive classification of 3d objects. In *Robotics and automation (icra), 2016 ieee international conference on* (pp. 227–233).

Ngan, H. Y., Pang, G. K., & Yung, N. H. (2011). Automated fabric defect detection—a review. *Image and Vision Computing*, *29*(7), 442–458.

Nielsen, M. A. (2015). *Neural networks and deep learning.* Determination Press.

Oquab, M., Bottou, L., Laptev, I., & Sivic, J. (2014). Learning and transferring mid-level image representations using convolutional neural networks. In *Proceedings of the ieee conference on computer vision and pattern recognition* (pp. 1717–1724).

Perez, L., & Wang, J. (2017). The effectiveness of data augmentation in image classification using deep learning. *arXiv preprint arXiv:1712.04621*.

Polikar, R., Upda, L., Upda, S. S., & Honavar, V. (2001). Learn++: An incremental learning algorithm for supervised neural networks. *IEEE transactions* *on systems, man, and cybernetics, part C (applications and reviews)*, *31*(4), 497–508.

Pratt, L. Y. (1993). Discriminability-based transfer between neural networks. In *Advances in neural information processing systems* (pp. 204–211).

Pratt, L. Y., Mostow, J., Kamm, C. A., & Kamm, A. A. (1991). Direct transfer of learned information among neural networks. In *Aaai* (Vol. 91, pp. 584–589).

Quinlan, J. R. (1986). Induction of decision trees. *Machine learning*, *1*(1), 81–106.

Rebuffi, S.-A., Kolesnikov, A., Sperl, G., & Lampert, C. H. (2017). icarl: Incremental classifier and representation learning. In *Proc. cvpr.*

Sarwar, S. S., Ankit, A., & Roy, K. (2017). Incremental learning in deep convolutional neural networks using partial network sharing. *arXiv preprint arXiv:1712.02719*.

Simard, P. Y., Steinkraus, D., & Platt, J. C. (2003). Best practices for convolutional neural networks applied to visual document analysis. In *null* (p. 958).

Simonyan, K., & Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.

Szegedy, C., Ioffe, S., Vanhoucke, V., & Alemi, A. A. (2017). Inception-v4, inception-resnet and the impact of residual connections on learning. In *Aaai* (Vol. 4, p. 12).

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., ... Rabinovich, A. (2015). Going deeper with convolutions. In *Proceedings of the ieee conference on computer vision and pattern recognition* (pp. 1–9).

Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., & Wojna, Z. (2016). Rethinking the inception architecture for computer vision. In *Proceedings of the ieee conference on computer vision and pattern recognition* (pp. 2818–2826).

Szeliski, R. (2010). *Computer vision: algorithms and applications*. Springer Science & Business Media.

Tsai, D.-M., & Lin, C.-T. (2003). Fast normalized cross correlation for defect detection. *Pattern Recognition Letters*, *24*(15), 2625–2631.

Utgoff, P. E. (1989). Incremental induction of decision trees. *Machine learning*, *4*(2), 161–186.

Wang, T., Chen, Y., Qiao, M., & Snoussi, H. (2018). A fast and robust convolutional neural network-based defect detection model in product quality control. *International Journal of Advanced Manufacturing Technology*, *94*(9-12), 3465–3471.

Weimer, D., Scholz-Reiter, B., & Shpitalni, M. (2016). Design of deep convolutional neural network architectures for automated feature extraction in industrial inspection. *CIRP Annals*, *65*(1), 417–420.

Why use keras? (2018). Retrieved from `https://keras.io/why-use-keras/` (Accessed: 2018-10-06)

Wu, J. (2017). Introduction to convolutional neural networks. *National Key Lab for Novel Software Technology. Nanjing University. China*.

Yosinski, J., Clune, J., Bengio, Y., & Lipson, H. (2014). How transferable are features in deep neural networks? In *Advances in neural information processing systems* (pp. 3320–3328).