POLITECNICO DI TORINO

Master's Degree in Electronic Engineering

Embedded Systems Curriculum

Master's Degree Thesis

A multiclass neural network model for contaminant detection in hazelnut-cocoa spread jars

Supervisors

Prof. Mario R. CASU

Prof. Luca CARLONI

PhD Giuseppe DI GUGLIELMO

Prof. Christian PILATO

Candidate

Bernardita STITIC

April 2020

Abstract

This project aims to develop a Machine Learning algorithm for detecting six different types of contaminants in hazelnut-cocoa spread jars. In particular, work began by analyzing an already existing binary, 2 hidden layer MLP which presented an error ratio of 51% for one of the contaminants: a triangle shaped plastic. New MLP architectures were developed, trained and validated using a multiclass approach instead after careful examination of the available samples and previous preprocessing methods. After optimization procedures were finished using exhaustive grid search and stratified cross-validation, the best candidate turned out to be a 1 hidden layer MLP architecture. This model managed to reduce the error rate for the problematic contaminant from 51% to 7.5% in the worst case. This was calculated after analyzing all relevant confusion matrices during the testing phase. These were additionally reduced to a 2x2 dimension to make results comparable to the binary case. Finally, the model was deployed to hardware, with a ZCU106 as the target board. Results from HLS already showed the potential for a reduced maximum latency of 0.000349 ms with an estimated clock period of 1.312 ns which were both smaller than the Binary Classifier's final latency of 3 ms and clock of 100 MHz.

Keywords: MLP, Neural Networks, Machine Learning, FPGA, HLS, quality assurance, contaminant detection.

Acknowledgements

To Mario, Luca and Giuseppe,
For this opportunity, and especially for your constant support, all your dedication,
time, feedback and every single thing you've taught me. Thank you.

To Christian,
Thank you for being my PoliMi cosupervisor and taking
the time to review this thesis project.

To my family and all my loved ones, in different parts of the world,
Thank you for all your love, care and support,
and for always being there no matter where I go.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

This thesis focuses on the problem of quality assurance. In particular, in contaminant detection in hazelnut-cocoa spread jars. This topic is relevant in today's food industry for several reasons.

Firstly, in today's world, a company is largely affected by its public image and hence the connotation its name is given by the general public. Consumer reviews have become increasingly relevant toward this end.

In a highly technological world affected so strongly by social media and rapid means of communication, the perception of a group of consumers about an industry has the potential to destroy its companies or the industry itself. As a direct consequence, it's in the best interest of each company to maintain a reputable, verifiable image before the local or international community.

Secondly, there is the crucial role of social responsibility that the food industry inherently holds. In particular, from a moral perspective, a food company should protect the consumer's health by committing to high quality standards of food packaging processes. Companies should permanently review their methods, update them and verify that healthy, good quality food is being produced. Contaminants, evidently, go against this goal.

In fact, contaminants such as glass, for instance, will most likely produce internal injuries in, for example, the consumer's digestive tract. Other types of contaminants might cause allergic reactions [1] in certain people. Naturally, angered consumers might also try to take legal action against a company and ask for monetary compensation. In turn, this will put the company's product quality and reputation into question.

Ultimately, not guaranteeing high quality foods will also lead to expensive strategies taken by companies to convince the market to trust in their products. Likewise, organizations with the authority to issue certifications will require companies to show evidence of minimum standards. These certificates will allow companies to continue growing and improving their reputation, attracting more consumers in the process.

As it's possible to deduce, food quality in packaged foods, such as hazelnut-cocoa spread jars which are so popular, is vital to a company's survival and the public's health as well. From a broader perspective, minimizing the amount of contaminants in food production will bring about social, economical, and health benefits in comparison to the opposite case.

Therefore, companies are always looking for improvements for their quality assurance processes and investing in new, innovative methods. This is the point where this thesis found an opportunity to develop a multiclass neural network classifier (a Machine Learning algorithm) to predict the type of contaminant in a jar. The work began after carefully studying a neural network model, known as the Binary Classifier, which was developed by a student from Politecnico di Torino for his master's thesis [1].

In this case, the model was trained for binary classification tasks to predict whether a jar in static conditions had a contaminant or not. Furthermore, the model was trained using real samples obtained with a Microwave Imaging System (MWS) prototype developed at Politecnico di Torino.

Microwave Imaging (MWI) is an innovative technique which uses low power electromagnetic waves in the microwave range. These are used to detect contaminants that other techniques usually chosen in industry, like X-rays or metal detectors, cannot detect [1]. An example of the contaminants that might be missed include glass fragments, which were detected by the multiclass neural network developed as well as the Binary Classifier.

Usually, MWI requires very computationally expensive programs to reconstruct an image of the dielectric properties of a given volume [1]. This is where Machine Learning comes in: in comparison, it is computationally faster and cheaper. Due to this, it can also work in real-time environments to detect contaminants in combination with MWI as proven by [1].

Moreover, the multiclass neural network model solves some of the problems presented by the Binary Classifier. Specifically:

- It was able to reduce an error of 51.220% achieved by the Binary Classifier when attempting to detect a triangular piece of plastic [1] to just 7.5% in the worst multiclass test case.
- HLS results showed a minimum worst case latency of 0.000349 ms with an estimated clock period of 1.312 ns. Instead, the Binary Classifier used a 100 MHz clock, with an estimated latency of around 3 ms [1].
- It was able to simplify the neural network model architecture of the Binary Classifier from 2 hidden layers to just one hidden layer with less neurons in total.

In addition, the multiclass neural network is also able to predict what specific type of contaminant is in a jar. This might help a company better understand where its quality control processes might be facing problems.

Now, regarding the structure of the thesis, this text is subdivided into seven chapters. The first one is this introduction. Afterward, the next chapters follow:

- Chapter 2, "Background", which covers the main theoretical aspects needed to develop the project. This includes Machine Learning and hardware related topics.
- Chapter 3, "Related works", where the Binary Classifier is further discussed.
- Chapter 4, "Methodology", where the steps and choices taken during project development are explained.
- Chapter 5, "Machine Learning results", where all the results achieved during the Machine Learning model development phase are analyzed.
- Chapter 6, "Hardware implementation", where the hardware related results are reported.
- Chapter 7, "Conclusions", where the main conclusions of the project are presented.

Chapter 2

Background

This chapter aims to give a general understanding of Machine Learning and the main techniques used to develop the multiclass neural network model. Specifically, topics about Machine Learning, classification tasks, the Neural Networks algorithm, Keras, TensorFlow and Scikit-Learn will be discussed, as well as some more details. Furthermore, there will be a discussion related to the hardware phase of the project at the end too.

2.1 Introduction to Machine Learning

Many in the field of Machine Learning (ML) have given their own definitions and explanations about this field. As a general idea, Machine Learning is the act of programming computers to learn from data, without creating a program in which such a learning act is explicitly described [2]. In other words, ML uses algorithms and techniques that allow computers to learn and generate new information by themselves.

More concretely, engineering oriented definitions describe Machine Learning as being the instance when programs can learn from some experience E, with respect to some task T and some performance measure expressed by a metric P. It is said, then, that such a program effectively learned if its performance on T, as measured by P, improved with E [2].

There are certain problems that are best approached when using Machine Learning. In particular, as explained in [2], these usually are:

- Problems for which already existing solutions require a lot of optimization steps or several rules. Machine Learning can often simplify code and perform better.
- Complex problems for which traditional approaches offer poor solutions. Machine Learning be used to perhaps obtain good results.
- Environments with a lot of fluctuation since Machine Learning can usually adapt to new data.
- Problems where it's required to gain insight about large amounts of data or the nature of the problem itself.

As introduced in Chapter 1, the previous model developed to solve the problem of quality assurance was the Binary Classifier [1]. Quality assurance is indeed a complex problem, for which typical processes that approach it require either different protocols or optimization steps.

In this sense, and in this context, by using Machine Learning researchers can aim for simplification and obtain good results through new algorithms and models. In fact, this was the case of the Binary Classifier, which reached a high level of performance. Therefore, considering all this, the previous list is applicable to the problem of the project, namely the first point.

Until this point, a general definition of Machine Learning has been given, together with situations where it might be appropriate to use it. However, it's also relevant to consider the general workflow of ML since also certain terms involved in this flow will be used later again in this chapter. The following subsection will discuss this topic.

2.1.1 Machine Learning: the general approach

Usually, and in general, a Machine Learning workflow begins by carefully analyzing the problem. There are several points to consider during this initial stage and aspects to describe, including:

- Defining the objective of the problem (often times, in business terms).
- Analyzing already existing solutions to the problem.
- Framing the problem (supervised/unsupervised and other classifications, which will be discussed in the next subsection).
- Establishing an appropriate performance metric that is aligned with the objective.
- Determining the minimum value of the performance metric in order for the solution to be considered adequate enough.
- Listing and verifying assumptions.

The steps just described are part of the box in Figure 2.1.1.1 named "Study the problem".

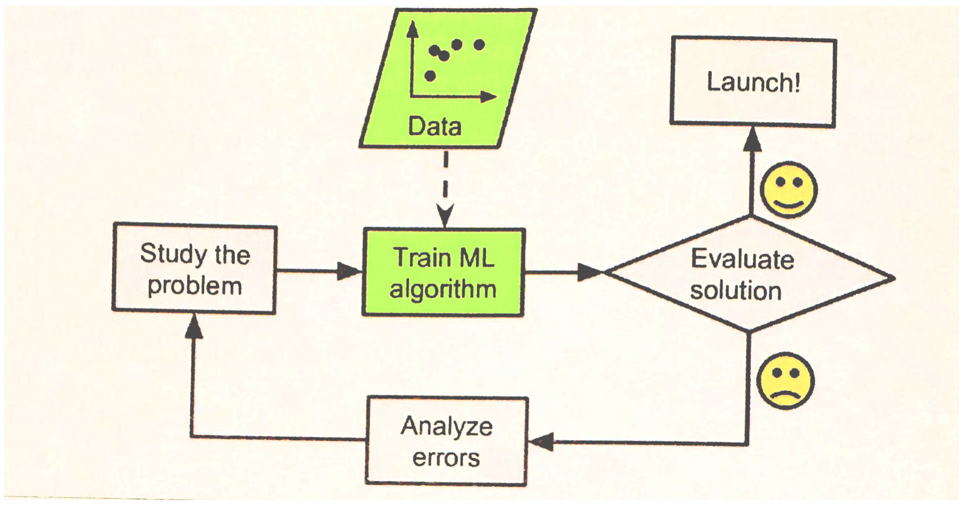


Figure 2.1.1.1: The general Machine Learning approach, adapted from [2].

The next step is to gather data. Here, it's always important to know what and how much data is required, as well as obtaining any authorization that might be needed to gain access to the data. Furthermore, it's also relevant to convert it to a format that can be manipulated easily, like NumPy arrays in the case of this project. Furthermore, some projects might require that sensitive data be protected or even deleted.

At this point, it is useful to start organizing the data. In particular, available samples should be divided into different sets, which are used for different purposes. Although several techniques and set types can be used, the typical sets used for Machine Learning are the training, validation and test sets except in particular situations with unusual characteristics.

The first set mentioned before is used to train the model. For its part, the second is used to validate the results obtained in the training phase. The third and last set, the test set, is usually used at the very end of the workflow to analyze errors. Therefore, it's set aside at the beginning and not used until the error analysis phase of a Machine Learning project is reached.

The third step in the workflow is to explore the data itself. Exploring the available samples includes checking their name, type, percentage of missing values, noisiness and noise type and data distribution type. Sometimes, it might even be necessary to visualize the data and study correlations between attributes (this term will be explained later in this chapter).

This third step is important since it will allow detecting characteristics in the data. This way, it is possible to identify possible transformations or models for the problem. Once all of the previous activities are done, the workflow can move forward to the fourth step, which is preparing the data.

Usually, Machine Learning projects include this fourth phase, where data is said to be preprocessed. There are different transformations that can be applied to the samples to ease the training phase. The relevant transformations which were applied in this project will be discussed later in this chapter.

Everything that has been described until now is included in the box called "data" in Figure 2.1.1.1. Only when all these steps (two to four) are done, the candidate models can start to be trained (fifth step). In order to do so, typically different ML models, from different categories, are trained quickly using default values for their parameters. The term *training* will be further explained in the next subsection.

Continuing now, this last task is included in the box called "Train ML algorithm" in Figure 2.1.1.1. Moreover, there are different techniques that can be used to measure their performance to compare them. One of them is cross-validation (CV), which will be detailed later on in this chapter since it was used extensively in this project. Another commonly used technique is to evaluate performance using a test set. However, it's often preferred to leave this for the most promising candidates after tuning.

This last activity, where performance is compared, is included in the box named "evaluate solution" in Figure 2.1.1.1. It's also required at this point to analyze not only the obtained numerical values for the metrics used to evaluate performance but also to investigate about the errors each candidate model is making. This is part of the box called "analyze errors" in Figure 2.1.1.1.

The process of training models and comparing their performance is iterative. In fact, Figure 2.1.1.1 shows a loop to indicate this. Usually, this loop is repeated quickly one or two more times and may require to study/check the problem or data again. Once all this is done, the workflow can progress towards a sixth step that consists in hyperparameter tuning for the candidate models.

There are different techniques used to accomplish this task, which is also iterative. Very often, hyperparameter tuning is performed using cross-validation again in combination with other techniques such as GridSearchCV [3] and RandomizedSearchCV [4]. In certain applications, even candidate data transformations for the preprocessing step may be explored using the aforementioned techniques.

Moreover, since ML projects tend to include numerous hyperparameters, a randomized grid search (RandomizedSearchCV) is often preferred over an extensive, in depth grid search (GridSearchCV). Since both of these techniques were considered for this project, they will be explained in the following subsections.

Toward the end of this sixth step which focuses on optimization, if there are several candidate models for the problem it's possible to try to ensemble them together [2]. Then, once a promising final model is obtained, usually performance is evaluated using a test set at this point. The objective is to estimate the generalization error, which is the error that could be made on new data after the model is launched. This term will be addressed again later in this chapter.

Naturally, the last two steps consist in presenting the solution (where everything should be documented) and launching it. This last part is the box named "launch!" in Figure 2.1.1.1. It's particularly important to list any relevant assumptions or limitations of the developed model. All in all then, as a summary, the main eight steps of a Machine Learning project, in general, are:

- Framing the problem.
- Getting the data.
- Exploring the data.
- Preparing the data (preprocessing).
- Exploring preliminary candidate models.
- Hyperparameter tuning (optimization).
- Presenting the solution.
- Launching the final model.

Next, the following subsection will describe the main types of ML systems. This is a topic that's relevant to consider for this project. This is due to the fact that the multiclass neural network approach used supervised training while developing the candidate models.

2.1.2 Machine Learning systems

There are different types of Machine Learning systems. Furthermore, it's useful to classify them in a broad way so that they can be described according to the following criteria [2]:

- Systems whose training phase included human supervision or not (supervised, unsupervised, semisupervised, Reinforcement Learning).
- Systems with the ability to learn incrementally or not from incoming data streams (online vs batch learning).
- Systems that can work by comparing new data to previously existing data or by detecting patterns in the training samples to build predictive models (instance-based vs model-based learning).

Next, these three points will be explained in a general way to be able to apply them to the multiclass model developed for this project.

2.1.2.1 Training phase based classifications

As mentioned previously, the nature of the training phase can lead to different classifications. In particular, the first step to correctly describe a Machine Learning system based on its training is by identifying whether there was a human supervision component or not during this phase, and to what extent.

Consequently, a first type of ML classification is **supervised learning**. In supervised training, each sample (also called *instance*) in a training set is associated to a specific output value (also known as a *target label*). It is considered that there is human supervision for these case, since the labels for each sample should be provided by someone before training begins.

In other words, the program does not deduce the labels of the training set by itself. Typical supervised learning tasks include:

- Classification tasks: where classifying samples as belonging to a specific *class* is required, like with spam or ham emails.
- Regression tasks: where predicting target numerical values is needed, and where samples have a determined number of *features*, which are also called *predictors*.

In the first case, many instances of the different classes, along with the class label, should be given to the model under training. In the second, samples including both their predictors and labels should be fed to the model.

The term *features* should be explained at this point. In Machine Learning, an *attribute* is a data type (like the brand of a car for instance) [2]. A *feature* can have several meanings but usually means an attribute and its value [2], so for example, when referring to the population of an area, population = 5 million (people). Many use the words attribute and feature interchangeably.

It's worth noting that some regression algorithms can be used for classifications and viceversa. For instance, an algorithm known as Logistic Regression can take samples with its features and output a target numerical value. This value represents the probability that a particular sample belongs or not to a given class. However, discussing this is out of the scope of this thesis.

On the contrary, another classification based on the training phase is **unsupervised learning**, where instances are unlabeled. Basically, the model will find patterns in the data and manage it depending on the patterns found. Typical applications where this is useful include:

- Data division (into groups), like when identifying consumer groups in a particular market or types of users visiting a blog.
- Data visualization, where a 2D or 3D representation of the input data is produced.
- Anomaly detection, like strange credit card transactions or manufacturing defects.
- Association rule learning, where large amounts of data are studied to detect links between different attributes.

Both of these can be combined to produce a third category based on the training phase, which is that of **semisupervised learning** systems. The issue is that labeling data is time-consuming and costly as well, so what usually happens is that labeled instances are not too many in the training set.

In other words, semisupervised learning algorithms can manage data that's partially labeled. Some applications where these algorithms are used include photo-hosting services for social media websites. Algorithms are able to identify faces in pictures using unsupervised learning. Then, once a user tags a person, the supervised learning part of the algorithm will be able to name the same person in different pictures.

Finally, the last classification based on the training phase corresponds to **Reinforcement Learning**. This is a very commonly used type of learning system in Robotics. Specifically, the system, which is called an *agent*, observes the environment and performs certain actions. It can receive rewards or penalties, based on its chosen behavior.

Consequently, the system learns by itself to achieve as many rewards as possible over time, adapting its behavior as needed. Using precise terms, this set of actions is called a *policy*. Specifically, the policy defines the groups of actions the agent should take in a particular situation to maximize rewards.

2.1.2.2 Classifications based on incremental learning abilities

Based on the ability to learn incrementally from input data streams, there are two types of Machine Learning systems. The first one is called *batch learning*. This is the case where a learning system does not have the previously mentioned ability. In consequence, it must be trained using all the available data.

Typically, batch learning is performed offline. In this context, offline means that training is completed before the finalized model is launched. Therefore, during operation, the model produces outputs based on what it learnt before during its training phase. This implies that if a new system is required at some point, for instance if a new class is introduced to the problem, then a new model must be designed to replace the old one.

Fortunately, the entire ML workflow (shown in Figure 2.1.1.1) is usually automated in practice. Despite this, evidently batch learning poses certain limitations. Namely, the time required to produce a new model could not be fast enough when fast changes need to be implemented. Additionally, training using large data sets consumes lots of computing resources.

In cases where there are limited resources for training, the previously mentioned limitation will become a significant obstacle. On the contrary, a learning system with the ability to learn incrementally after being launched is not affected by these limitations to the same extent as batch learning systems. This is the case of **online learning systems**.

In this scenario, the ML model can be trained incrementally by feeding it, sequentially, individual samples or small groups of input data. These groups of data are called *mini batches*. Therefore, learning steps are fast, cheap, and the system can learn during operation.

2.1.2.3 Categories based on comparisons or pattern detection

The last criteria for classifying learning systems depends on how systems generalize since most Machine Learning tasks consist in making predictions. Therefore, it's relevant to consider how effectively a model can predict for (or generalize to) new input data. Basically, there are two different learning systems based on this idea.

The first are **instance-based learning systems**. In this case, the Machine Learning model learns the available examples by heart during training. When new data becomes available, it generalizes and makes predictions by using a similarity measure. Specifically, it uses the measure to compare the new samples to the previously learnt ones (or a subgroup).

The second type corresponds to **model-based learning systems**. These systems create a predictive model out of the samples used during training instead. This way, when new data becomes available, predictions are made using the model developed.

Since the topic of model-based learning has been introduced, it's important to discuss the meaning of the term *model* in Machine Learning. In particular, this word can have three possible meanings, depending on the context [2]:

- A type of ML model (for example, Linear Regression, Logistic Regression, Neural Networks). This is the meaning that has been used until now.
- A fully described model architecture (for instance, a Linear Regression model with seven inputs and one output).
- The final trained model which is ready to start predicting on new data (and thus, with its model type, architecture and all model parameters fully defined).

Moreover, on the previous list the word *parameters* was used. As explained, model-based systems develop models. These models require *parameters* (which should not be confused with *hyperparameters*, a term which will be discussed in a later subsection in this chapter).

During training, a specific learning algorithm will look for values for the model parameters that minimize (or maximize) a specific performance measure. Usually, a function is used, which can be either a utility or a cost function. The first aims to measure how good a model is whereas the second one focuses on how bad the model is instead.

Therefore, in other words, during training, the learning algorithm will feed the training samples to the model with a tentative set of parameters. The algorithm will then measure the output of the utility or cost function using the parameters and samples for each training step. This process is iterative and once it's finished, the parameters that best fit the data (that maximize the utility or minimize the cost) will be obtained.

2.1.2.4 The case of the multiclass neural network approach

After discussing the previous three categories, it's relevant to apply this knowledge to the case of this thesis. As it will be detailed later in Chapter 4, the project used target labels (indexes or hot-encoded labels for exploratory purposes) during its training phases. Therefore, it consists in a supervised learning system.

Moreover, the candidate models (as well as the final model) developed for this thesis used batch learning. All training was completed offline, before deploying the neural network to hardware.

Finally, regarding the last classification, the multiclass neural network is evidently a model-based learning system. This is because training consisted in receiving all input data and obtaining parameter models such that a specific cost function would be minimized. Now, the main obstacles that Machine Learning faces will be explained since this thesis project faced mainly data related challenges.

2.1.3 Limiting factors for Machine Learning models

Problems in Machine Learning models can originate in two possible areas. These are the data or the algorithms themselves. Moreover, both areas have different aspects that, if not managed well, could potentially create limitations for a Machine Learning project. Now, all these topics will be further discussed.

2.1.3.1 Data related challenges

Regarding data, there are four different aspects about it that might affect a Machine Learning algorithm. The first is **data quantity**; in particular, insufficient quantity of samples will probably lead to poor performance.

Most projects about Machine Learning, even very simple ones, require thousands of examples for training. More complex problems, like speech recognition, will need millions of examples to properly train the ML algorithm [2].

A second aspect related to data that should be considered is how representative it is of the entire situation under study. Specifically, training data which is **non representative** will create issues. This is applicable both to instance-based learning systems and model-based systems, like this thesis project.

More specifically, in order for a Machine Learning model to generalize well to new samples, the training data should be representative of the new cases. Therefore, training data should aim to represent the entire situation as a whole, not just a portion of the case study.

For instance, to better explain this, there are models that use Linear Regression algorithms to provide future estimations for countries based on their existing information. This information could be, for example, their GDP per capita [2]. If this attribute is considered then, whatever the prediction is, the training set should include both wealthy, intermediate and poor countries. Consequently, it's more likely the model will generalize well to countries not considered in the training set.

Although this may sound simple in theory, in practice it is much harder. In the case when the data sets are small, there could be *sampling noise*, which means that there will be non representative samples in the set because of randomness. On the other hand, when data sets are bigger, then the sampling methodology used to gather the data might be flawed. In cases like this, the term used to describe this situation is called *sampling bias*.

Continuing now, the third challenge related to data is very obvious. It corresponds to **poor quality data**, which is not the same as non representative data. In particular, while non representative samples do not consider an entire situation under study, data is poor quality when samples have a lot of errors, outliers, and/or noise. A significant amount of time should be spent fixing poor quality data. In particular [2]:

- When there are samples which are clearly outliers, it's worth considering removing them or fixing the errors manually.
- If there are missing features (an error) for some samples, there are several options that can be taken. For instance:
  - The attribute(s) could be ignored.
  - The samples that have the missing feature(s) could be discarded.
  - The missing values could be filled with, for example, average values for that feature.
  - Some data scientists will even try to build a model with the problematic feature and another one without it.

The last and fourth aspect related to data that usually creates problems is when samples have **irrelevant features**. Ideally, there would be no irrelevant features, but in practice there usually are some. Therefore, what's important is to have several important ones and a few irrelevant ones. Usually the activities that can be done to avoid a feature related problem are [2]:

- Selecting the most relevant features among the available ones for the training phase (*feature selection*).

- Combining features into more useful ones (*feature extraction*; dimensionality reduction algorithms like PCA can be helpful).
- Creating new features by gathering new, additional data.

All in all then, to summarize, the four main challenges related to data consist in:

- Data quantity (not enough samples).
- Nonrepresentative data.
- Poor quality data.
- Irrelevant features.

2.1.3.2 Algorithm related challenges

The other area where a Machine Learning project might find difficulties is with algorithms. If the type of algorithm chosen for the case study is not well chosen, then it will be possible to observe cases of *overfitting* or *underfitting.* These are the two main challenges related to algorithms and will be addressed now.

Figure 2.1.3.2.1 shows the three possible cases of how an algorithm may fit to the available training data. On the left, there's the case of underfitting, in the center the case of a good fit, and on the right the case of overfitting. Also, it's important to highlight that the illustrated plots could represent the case of a regression algorithm like Linear Regression. However, underfitting and overfitting are applicable to all types of ML algorithms.

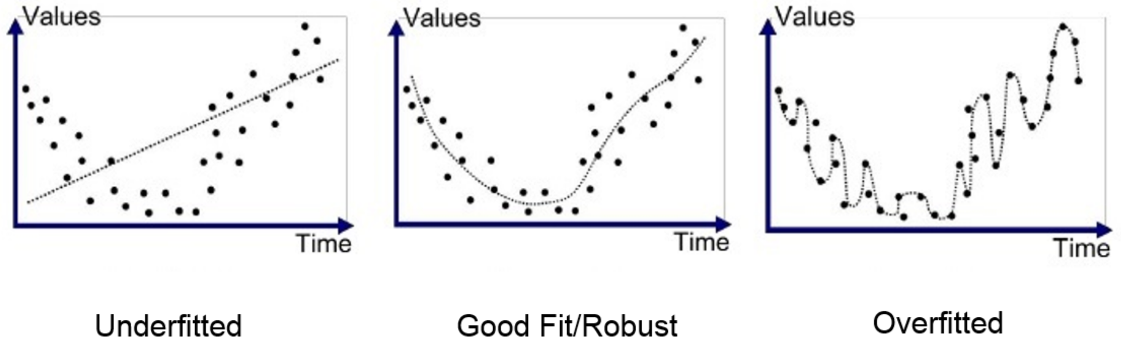


Figure 2.1.3.2.1: Underfitting, normal fitting and overfitting, taken from [5].

Underfitting happens when a model is too simple to learn from the training data appropriately. This could happen if the architecture is too simple, or if a certain feature is being penalized too harshly. This last comment relates to the topic of *regularization* [6], which is out of the scope of this thesis so it will not be discussed in details.

However, to give a general, very simple idea, regularization uses *regularization parameters.* The higher their values, the more they penalize specific *model parameters* that the user chooses. Therefore, the effect in this case is that the relevance, or how much a feature is considered during training, is decreased. If regularization is too high, then it is as if the specific feauture did not exist. The opposite happens when regulatization paramters are too small.

Now, as an example, it's possible to observe in Figure 2.1.3.2.1 that on the left the straight line does not fit the samples appropiately at all. This simple model would be a very bad predictor of new data. Perhaps, for example, a slightly more complex model using a higher polynomial degree would be a better fit if using Linear Regression for a case like this.

In general, then, it's possible to detect underfitting during the training phase since performance will be poor. Some of the metrics used to measure performance will be discussed later in this chapter. Furthermore, it's possible to improve a situation of underfitting by trying the following [2]:

- Using a more complex model, with more parameters.
- Selecting better, more informative features so the model can learn from relevant information.
- Reducing certain constraints on the model such as any regularization parameters being used.

On the other hand, Figure 2.1.3.2.1 on the right shows the opposite case, which is *overfitting*. In a situation like this, the model will fit the training data *too* well. However, the model will perform *notoriously* poorly in comparison on any new available data. In other words, the given ML algorithm will not be able to generalize properly.

Moreover, it's important to define complexity in this context and try to answer the question of when a model would be considered too complex. Precisely, it should be emphasized that it's complexity in relation to the training data that should be taken into account.

For instance, complex models (regarding their architecture) like deep neural networks have the skill to detect very subtle patterns in a given training set. However, if the set is too small it's likely it will introduce sampling noise. Now, if the set is not small, it may be noisy too depending on how data was obtained. Therefore, the ML model will not be able to generalize properly since the patterns detected will be related to noise in these two cases most likely.

Ultimately, overfitting can be defined as a situation where a model is too complex in relation to both the amount and noisiness of the training data [2]. Some strategies to improve this situation are:

- Simplifying the model by: reducing the number of parameters, decreasing the number of features in the data, or by constraining the model (using regularization).
- Increasing the training set.
- Reducing noise by removing outliers or fixing data errors.

Previously, a general idea about regularization was given. Now that both underfitting and overfitting have been explained in more depth, a more specific description of regularization can be described. Therefore, in more precise terms, regularization constrains a model using the regularization parameters mentioned before to simplify the model and to reduce the probability of overfitting.

2.1.3.3 Challenges in the case of the multiclass neural network

In the case of the project developed for this thesis, the main challenges were related to the available data. In particular, data quantity was the main problem. This will be clarified later in Chapter 4 and Chapter 5. Similarly, it will be explained how data was managed.

On the contrary, data quality was considered to be very good, especially since the original sampling method was meticulous, consistent and precise. There are some issues however related to the type of data that was gathered, in particular the homogeneity of such data. This topic will be addressed in Chapter 4 and Chapter 5 as well.

Similarly, because of the types of contaminants that were considered, samples were deemed as representative. Finally, regarding the features, it might have been possible to consider some as irrelevant. Fortunately, the available data had already been processed to some extent, so irrelevant features had already been discarded by the time this thesis started.

2.1.4 The importance of validation and testing

Until now, a general definition of Machine Learning has been given together with the main scenarios where ML is useful. Then, the general workflow was described, as well as the type of Machine Learning systems and the main challenges these can face.

Another important aspect about Machine Learning comes right after training (or sometimes, during this phase). In particular, this crucial step is related to validation and testing. Earlier in this chapter, it was briefly mentioned that data is usually divided into different sets. Now, these ideas will be further developed.

It's already been explained that data is usually divided into a training, validation, and test set. Now, usually, the original data is first divided into a training and a test set. Typical proportions used for this task are 80/20 (or 80% for the training set and 20% for the test set) or 90/10 [6]. Moreover, 80/20 is usually the preferred choice but ultimately data splitting will depend on the size of the dataset [2].

Earlier in this chapter, it was also explained that the main aim of the test set is to have an estimation of the *generalization error* (or also known as the *out-of-sample error*). The generalization error is basically the error rate on new cases [2], as said at the beginning. Therefore, it's a prediction of how the model will behave once it's launched.

Now, specifically, it can be added to this that if the generalization error is high but the training error is low, then the model is very likely overfitting the training data. Then, appropriate steps should be taken to improve this situation as explained previously.

For its part, once the training set and test sets are ready, it's possible to go through a second division to obtain the validation set. For this, the original training set is split to create a final training set and the validation set. Validation is actually a key aspect in the methodology of a Machine Learning project. Consequently, it will be described more in depth immediately.

2.1.4.1 Validation phase: the validation set, cross-validation (CV) and the train-dev set

At first, it might appear like using only a training set and test set should be enough. However, there is one crucial problem with this approach. To understand this, let's suppose there were different versions of the same model. To select the best one, the natural way would be to use the test set and measure their performance. Specifically, the one with the lowest generalization error should be the best option.

Nonetheless, it probably would happen that after launching this "best" model, the error rate would be much higher than predicted. Essentially, the problem is that when only using a training with a test set, most likely the model will end up overfitting the test set in the search for the best model version or best model type. Consequently, a different approach is required to avoid this situation which creates uncertainty about future results.

This is where the idea of having a validation set (which is also known as a development set or dev set) is useful. In particular, the purpose of the validation set is to simply evaluate different models to select the best candidate before using a test set, which should be used at the very end.

In this context, a meticulous workflow would train several models on the reduced training set and then select the best with the validation set. Afterward, the best candidate would be trained using the reduced training set and validation set combined. Finally, the generalization error would be estimated using the test set [6], [2].

As expected, there are aspects to consider regarding this technique. In particular, the following should be noted:

- If the validation set is too small, it could happen that a non optimal model might get selected to move forward to the next phase.
- If, on the contrary, the validation set is too big, then the reduced training set will be too small. This will, of course, affect the development of a model.

A way of avoiding these situations as much as possible is by using another validation technique called *cross-validation* (or also know as "CV"). This technique, which will be described immediately, also avoids having to train models on reduced training sets but allows performing validation as well.

Specifically, CV works by dividing the entire training set into different subsets. Usually, the number of divisions is 3 or 5. Then, one subset becomes the validation set, while the other subsets become the training set. Once the model is trained and validated, the process is repeated again until all subsets have acted as validation sets once.

Scikit-Learn has different implementations for CV, which will be fully described later in this chapter, including the concepts of stratified folds and sets. This is due to the fact that CV was extensively used in this thesis to select the best candidate models for the optimization processes.

Considering everything described until now, the one drawback regarding CV is that training time is evidently increased and multiplied by the number of validation sets [2]. Now, even if CV is used, there are still some applications where other techniques might be more useful. Although this is out of the scope of this project, it's worth mentioning a very particular technique for problems with **data mismatch**.

To be more precise, there might be applications where the data gathered for model development might not be fully representative of the data that will be used during the operative phase. Alternatively, the amount of representative samples might be too small in comparison to the rest. This is especially true of mobile apps in which the data used once the app is launched will depend on what the user will gather as input data.

Of course, training data cannot possibly cover all the types of input data a user will feed the app. In cases like these, and always really, it's imperative to remember that **data in the validation and test sets should be representative of the data that's expected to be used in production** [2]. In consequence, most representative samples, if not all, will be assigned to the validation and test sets. Therefore, for these applications, performance on the validation set could turn out to be poor after training the model.

Furthermore, whenever this happens, the reason why becomes unclear. Did the model overfit the training set, or is there a data mismatch between the training and validation sets? One possible technique suggested by Andrew Ng in [2] is to hold out some of the training samples in yet another set called the **train-dev set**.

So, after the model is trained using the training set, it can be evaluated on the train-dev set. If performance is good enough, it can be deduced that the model is not overfitting the training set. Otherwise, then most likely the candidate model is overfitting the training set. Therefore, the strategies indicated before in this chapter for this case should be considered.

Finally, if the evaluation using the training and train-dev sets goes well, then the potential model can be evaluated using the validation set. If at this step performance is poor, then the likely answer is that there was, indeed, a significant degree of data mismatch between the training and validation sets. It could be possible to improve this situation by preprocessing the available data to make it more representative of the one to be used during production.

2.1.5 The concepts of parameters, hyperparameters and algorithms

Before closing this introductory section on Machine Learning, it's very important to make some key distinctions regarding some terms that will be used throughout this thesis. Specifically, the terms are those of: parameters, hyperparameters, and algorithms.

The concept of parameters (model parameters) has already been explained earlier in this chapter. In particular, it refers to the parameters that are optimized during training to minimize a cost function (or maximize a utility function in some cases).

On the contrary, *hyperparameters* are (non model) parameters which are part of a learning algorithm but not of the model itself [2]. Examples in this category are regularization parameters, which are used to penalize model parameters during the training phase. Also, in this thesis, parameters that are not strictly part of a learning algorithm, and not part of an ML model either, will also be considered to be hyperparameters.

The previous point will be more clear once specific examples of hyperparameters for neural networks are given toward the end of this chapter. Finally, the last term to clarify is that of *algorithms*. For this thesis, Machine Learning algorithms will refer to the type of algorithm used. For instance, Linear Regression or Neural Networks. Similarly, the term *model* can refer to these but also to the other two definitions described earlier depending on the context.

However, the term *learning algorithm* will not refer to the previous description. In particular, it will refer to the algorithm used to minimize the cost function (or maximize a utility function). Examples of these algorithms include Gradient Descent or Stochastic Gradient Descent [6], [7]. Later in this chapter the optimizers that were used for this project will be described and, only then, will these algorithms be addressed to a relevant extent.

2.2 Classification tasks

As mentioned earlier, together with regression, classification is one of the most common tasks performed by supervised learning systems. This thesis focused, in particular, on the development of a multiclass neural network to classify different contaminants. Therefore, after the previous introduction, now relevant terms related to classification in ML will be discussed.

2.2.1 Performance metrics

The relevant terms to discuss consist mainly in performance metrics. Of course, it's possible to use cross-validation (CV) to evaluate candidate models as explained before. In particular, the average accuracy for each iteration of CV can be obtained. Now, it should be noted here that "accuracy" means *classification accuracy*.

Classification accuracy A can be mathematically defined as [8]:

$$A = \frac{CP}{TP}$$

Where "CP" stands for (the number of) "correct predictions" and "TP" for (the number of) "total predictions". However, while accuracy might seem promising for these tasks at first, it should only be used as an initial estimation or complementary metric. In other words, it should not be used as the decisive, most descriptive metric for measuring classification performance.

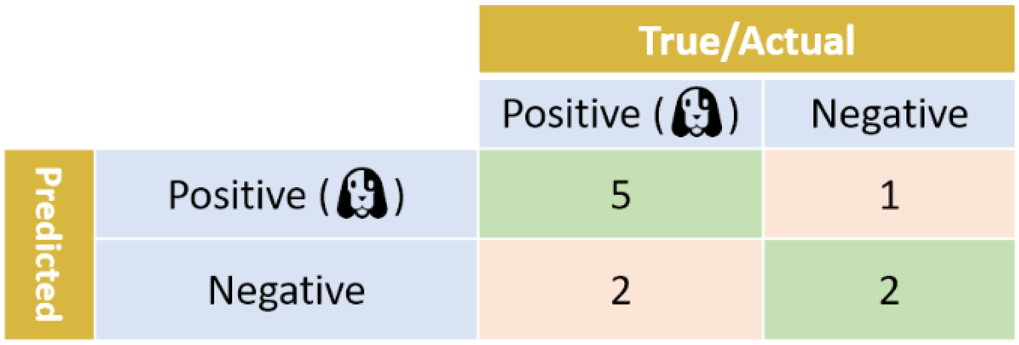
The previous is said because sometimes accuracy results for these tasks might be misleading. This is especially true when working with **skewed classes**. These are classes in which the amount of samples of one class are notably larger in comparison to the samples belonging to the other classes.

Therefore, in cases like these, the *classifier* (or ML classification algorithm) could potentially learn to classify most input data samples as belonging to the larger class. Most likely, accuracy during training will be large; this could also happen during CV, a separate validation phase or the final test set phase if the classes are still skewed.

However, despite the possibility of obtaining these seemingly good results, the samples not belonging to the dominating class will probably be classified incorrectly instead. In consequence, the need arises for more detailed metrics. For classifiers, then, more descriptive ways of measuring performance quality have been developed. These will be discussed now.

2.2.1.1 The confusion matrix

The simplest way to understand the confusion matrix is to begin with the case of binary classification (only two possible classes). In this case, the confusion matrix would be a 2x2 matrix. Rows represent the predictions of the classifier (or predicted classes), and columns the actual classes. An example of a confusion matrix is shown in Figure 2.2.1.1.1.



		True/Actual	
		Positive (🐶)	Negative
Predicted	Positive (🐶)	5	1
	Negative	2	2

Figure 2.2.1.1.1: Example of a confusion matrix, taken from [9].

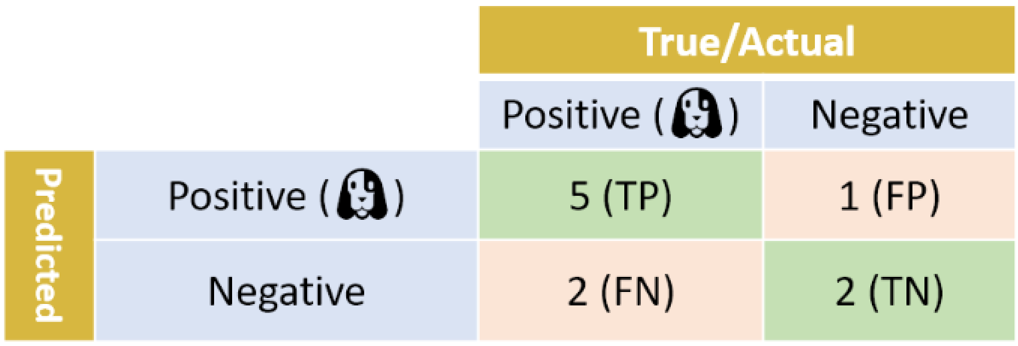
In binary classification, the two classes are usually called "Positive" and "Negative". In the example in Figure 2.2.1.1.1, which was taken from [9], the problem was about detecting dogs in pictures. The dataset consisted in different images. In consequence, the positive class in this case was a dog while the negative class was all the other pictures.

Basically, the classifier will predict whether a given sample image is a Positive (there is a dog) or a Negative (there is something else in the photo). A perfect classifier would only have entries in the diagonal cells of the confusion matrix. However, classifiers in reality are not error free, so the other cells in the matrix will get filled.

In particular, in binary classification, there will be two types of mistakes. One will be the Positives classified as Negatives (*False Negatives or FN*) and the Negatives classified as Positives (*False Positives or FP*). In Figure 2.2.1.1.1, for instance, 2 images were False Negatives, and 1 sample corresponded to a False Positive.

On the other hand, correctly classified Positives are known as *True Positives or TP*. Similarly, correctly classified Negatives are called *True Negatives or TN*. In the example in Figure 2.2.1.1.1, the TP correspond to 5 images and the TN to 2 photos.

Furthermore, in order to visualize these concepts better for this example, Figure 2.2.1.1.2 shows all the categories next to the cell entries. Again, this figure was taken from [9].



		True/Actual	
		Positive (🐶)	Negative
Predicted	Positive (🐶)	5 (TP)	1 (FP)
	Negative	2 (FN)	2 (TN)

Figure 2.2.1.1.2: TP, TN, FP and FN entries in the dog example, taken from [9].

Furthermore, there is more information that can be extracted from a confusion matrix. In particular, there is special interest in obtaining performance metrics that will be able to describe exactly how *good* a specific classifier can perform. Of course, this good or bad performance is quantified with respect to the quality of the predictions. Now, these metrics will be explained.

2.2.1.2 Precision and Recall

When analyzing the results obtained from a classifier more deeply, a natural question to ask is: what proportion of what was classified as Positive (the class of interest in many cases), is *truly* Positive? In the previous example, this question would be: out of all the pictures that were predicted to be dog photos, which ones did actually have a dog?

The performance metric known as **Precision** attempts to answer this question. Specifically, Precision (P) is mathematically defined as:

$$P = \frac{TP}{TP + FP}$$

In this example, since TP $=$ 5 and FP $=$ 1,

$$P = \frac{5}{6} = 0.833$$

Therefore, $P = 83.3\%$. In other words, out of all the pictures that were predicted to be dog photos, the classifier was correct regarding 83.3% of the images it predicted as Positives. Depending on the application, this result could be considered quite reliable. Additionally, it's relevant to note that, for Precision, the denominator of the formula is actually the sum of the **first row** of the confusion matrix.

Another question that is possible to ask, regarding a classifier's performance, is: what proportion of the Positive samples was correctly classified as Positive? In the example, the question would be: out of all the dog pictures, which ones are correctly classified as such?

The metric that can answer this question is **Recall**. Recall R is mathematically defined as:

$$R = \frac{TP}{TP + FN}$$

In this example, TP = 5 and FN = 2, so,

$$P = \frac{5}{7} = 0.714$$

Therefore, out of the *actual* Positive input samples, 71.4% were correctly classified as Positive. In consequence, only this proportion of the dog pictures were classified as dog photos, in the example. It should be noted that, contrary to Precision, the denominator of Recall is the sum of the **first column** of the confusion matrix.

Furthermore, remembering now the definition of Accuracy given before, it's also possible to redefine it in terms of the new concepts: TP, TN, FP, FN. Accuracy is a broader term, so the question it attempts to answer for classification tasks is: what proportion of the input samples, both Positive and Negative, were classified correctly?

In the dog example, the question would be: what proportion of pictures, both Positive and Negative, were predicted to belong to the right class? Considering this, it's possible to redefine Accuracy A as:

$$A = \frac{TP + TN}{TP + TN + FN + FP}$$

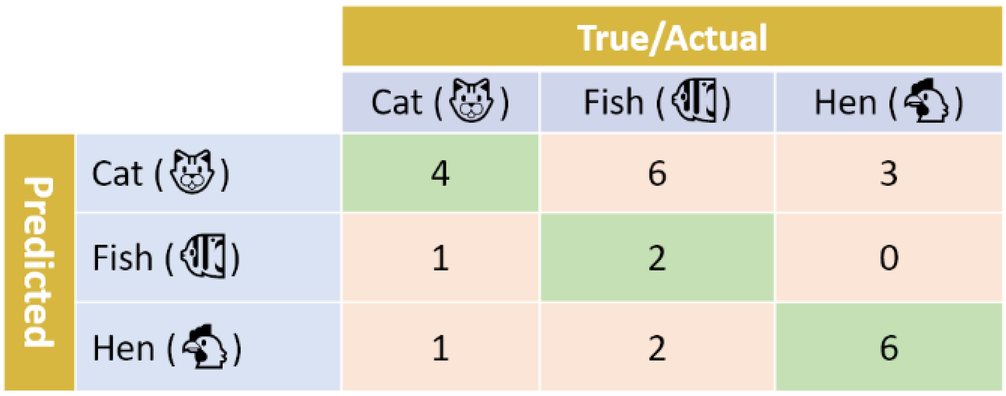
In this case,

$$A = \frac{7}{10} = 0.7$$

Therefore, 70% of the pictures were correctly classified. At this point, another natural question to ask is whether Precision or Recall is more important. The answer is that it will depend on the particular application [9]. Moreover, there is generally a trade-off between Precision and Recall. This is due to the fact that results will be influenced by thresholds defined for a classifier to help it determine whether a sample should be classified as Positive or Negative [2], [6], [9].

2.2.1.3 Precision and Recall for multiclass models

Having described the previous performance metrics for the binary case, the next topic to discuss is how these are applicable to a multiclass case. Firstly, a confusion matrix for a multiclass classifier will be an NxN matrix, where N is the number of classes. The same author [9] that created the dog example, expanded it to create a confusion matrix for a multiclass classifier. This matrix can be found in Figure 2.2.1.3.1.



		True/Actual		
		Cat (🐱)	Fish (🐟)	Hen (🐔)
Predicted	Cat (🐱)	4	6	3
	Fish (🐟)	1	2	0
	Hen (🐔)	1	2	6

Figure 2.2.1.3.1: Multiclass confusion matrix example, taken from [9].

In Figure 2.2.1.3.1, it's possible to see that the confusion matrix is now a 3x3 matrix. In fact, there are three classes: cat, fish and hen, since the sample pictures are now about these three animals. Precision and Recall can be calculated once again, per class, by following the same logic.

In fact, when calculating Precision and Recall for a particular class, it's possible to "collapse" the multiclass confusion matrix into a binary confusion matrix. In other words, it's possible to pretend, for a second, that the class of interest is the "Positive" class and the other classes are one big "Negative" class.

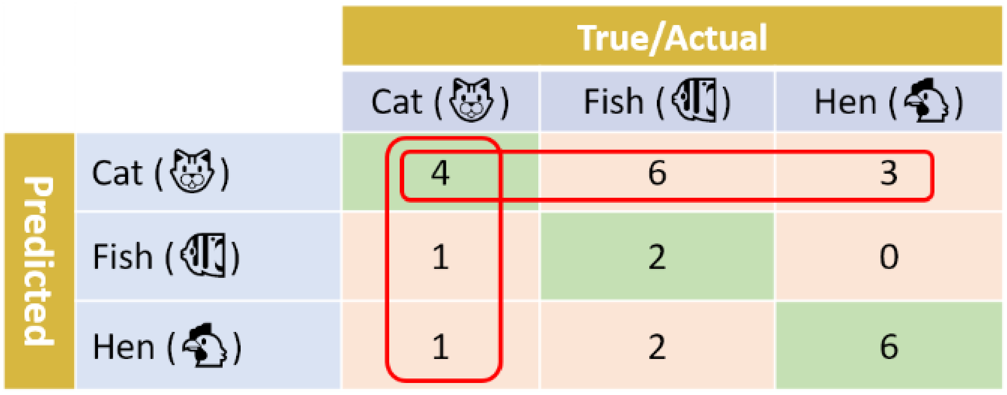
Following this interpretation, for example, Precision for the cat class will be the TP cats (4), divided by all the entries that were predicted to be cats (4+6+3). Therefore, $P = 0.308$ or 30.8% for the cat class. This means that out of all the pictures that are classified as cats, only 30.8% of these are actually cats.

Similarly, Recall for the cat class would be the TP cats (4), divided by the total amount of cat photos (4+1+1). Consequently, Recall for the cats would be $R = 0.667$ or 66.7%. This means that, out of all the photos that have a cat, 66.7% of them were classified as cat pictures.

It's possible to note that there's a similar pattern to the one described before for the binary case. In fact, for Precision, the denominator is still the sum of a row. In particular, it's the sum of the row associated to the cat class. Similarly, for Recall, the denominator is associated to the sum of the elements in the column where the cat class is found.

In consequence, it's **possible to generalize the previous description to all classes**. Precision for a class will be, therefore, the TP for that class divided by the sum of the entries in the row where that class is in the matrix. Similarly, Recall will be the TP for a specific class divided by the sum of the cells in the column where that class is in the confusion matrix.

To better visualize this, Figure 2.2.1.3.2 illustrates the previous explanation for the case of the cat pictures. Once again, the confusion matrix was taken from [9].



		True/Actual		
		Cat (🐱)	Fish (🐟)	Hen (🐔)
Predicted	Cat (🐱)	4	6	3
	Fish (🐟)	1	2	0
	Hen (🐔)	1	2	6

Figure 2.2.1.3.2: Calculating Precision and Recall for the cat pictures, taken from [9].

2.2.1.4 The confusion matrix in Scikit-Learn

Since Scikit-Learn was used extensively for project development, there is one **crucial** detail that should be strongly emphasized at this point. Scikit-Learn can work with confusion matrices for classification tasks. However, the matrices it works with are **transposed** [10], [9].

In other words, rows will represent the *actual* classes and columns the *predicted* classes. Precision, Recall and Accuracy metrics can still be calculated, of course. Nevertheless, when calculating the denominator of Precision, for instance, this will be the sum of the entries of a column, not a row. Similarly, for Recall the calculation will also be reversed. Consequently, the denominator will be the sum of the cells in a given row, not a column.

Moreover, Scikit also works with *normalized* confusion matrices. The concept is very simple: divide each entry in a row by the sum of the elements in that row. In other words, the normalized confusion matrix in Scikit normalizes each entry in a row with respect to the total amount of samples for the class associated to that row. Figure 2.2.1.4.1 and Figure 2.2.1.4.2 are examples of this, both taken from [10].

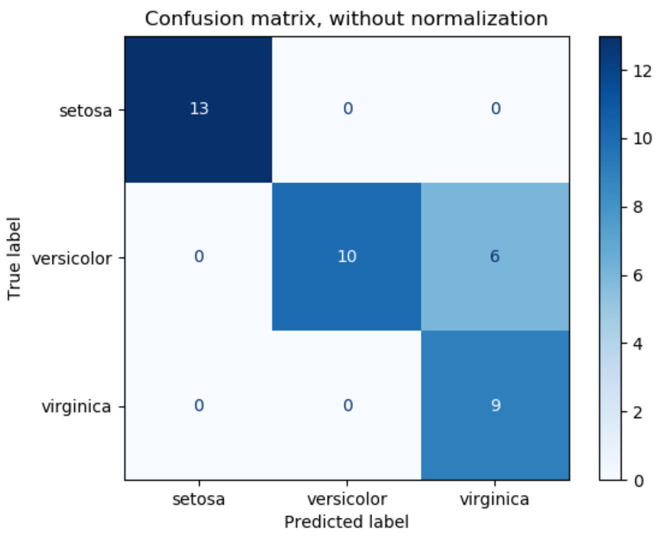


Figure 2.2.1.4.1: Confusion matrix (Scikit-Learn version) for three classes of flowers, taken from [10].

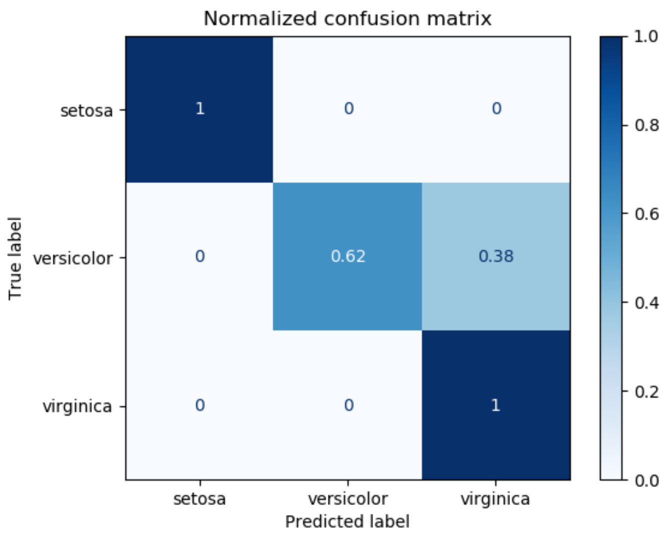


Figure 2.2.1.4.2: Normalized confusion matrix (Scikit version) for three classes of flowers, taken from [10].

In fact, if Figure 2.2.1.4.1 is considered, it's possible to calculate that the amount of setosa flowers in the dataset is 13. For versicolor flowers, there are 16 samples, and for the virginica flowers there are 9 flower samples. If then each entry in the first row in Figure 2.2.1.4.1 is divided by 13, the entries in the first row of the matrix in Figure 2.2.1.4.2 will be obtained. The same steps should be repeated for the other two rows.

2.2.1.5 The F1-score

After having discussed all the previous topics, there is one final metric that is worth mentioning since it's quite popular for classification tasks. This last metric corresponds to the *F1-score*, which combines Precision and Recall into one metric.

In the case of binary classification, the F1-score is defined as follows:

$$F1 - score = \frac{2 * P * R}{P + R}$$

It should be mentioned that the F1-score is significantly affected by low values. In particular, it gives a larger weight to lower values due to its mathematical expression [11] and the range of Precision and Recall, which goes from 0 to 1.

Now, the F1-score is typically used since it is able to combine two metrics into one as mentioned earlier. Usually, higher F1-scores point to a better performing classifier. However, it should be used carefully, in both binary and multiclass cases.

Furthermore, in the case of multiclass classifiers, both the measurement and definition of the F1-score become more complex. In fact, there are various types of multiclass F1-scores. In consequence, to calculate these multiclass F1-scores, new definitions for Precision and Recall have to be made sometimes.

To give a general idea, three types of multiclass F1-scores are **the Macro-F1, the Weighted-F1, and the Micro-F1** [11]. The first one is perhaps the easiest to understand since it's a simple average. More specifically, it requires to calculate the F1-score for each class, add the results and then divide the total by the number of classes.

So, for example, earlier in this chapter, Figure 2.2.1.3.2 showed how to calculate Precision and Recall for the cat pictures. Therefore, after having the Precision and Recall values for this class, the F1-score for the cat class would need to be calculated using the formula just described. This would have to be repeated for each class (fish and hen). Then, all three F1-scores should be added and divided by 3.

For its part, Weighted-F1 is a variation of Macro-F1. In the case of Weighted-F1, each F1-score (for each class) is multiplied by the total amount of samples in that class, to give it a proportional weight. Then, these products are added and divided by the total number of classes.

Moreover, similar, new variations of Precision and Recall can be made. This way, it's possible to calculate Macro-Precision, Macro-Recall, Weighted-Precision and Weighted-Recall. These all follow the same logic as the Macro and Weighted-F1 scores.

The last type of F1-score, Micro-F1, requires to first calculate the Micro-Precision and Micro-Recall values. These metrics consider the entire set of samples (and therefore, all classes together). Both Micro-Precision and Micro-Recall have the same value, since both in the end are calculated as the sum of the diagonal of the confusion matrix divided by the sum of all the remaining cells.

However, this value is also a multiclass classifier's accuracy. Then, regarding the Micro-F1 score, the formula presented at the beginning for the binary F1-score has to be used. Ultimately, the result will be that the Micro-F1 score will be the same as the Micro-Precision and the Micro-Recall values, which are both the same too.

As a last point of discussion, it should be clarified why, even if there are different types of F1-scores for multiclass classifiers, these should be used carefully. The reason why is that, depending on the application, Precision or Recall will be more important. Therefore, when calculating an F1-score, either Precision or Recall should be given a larger weight depending on the situation.

Nevertheless, for all types of F1-scores, Precision and Recall are treated equally in a mathematical sense [11]. Even in the case of the Weighted-F1, the values that are used are the F1-scores for each class. Each one of them is calculated using the same weight for Precision and Recall. Ultimately, then, the F1-score should be used for simple high-level comparisons and not as a fully descriptive performance metric.

2.3 Neural Networks

Neural Networks (NNs), also known as Artificial Neural Networks (ANNs), are a Machine Learning model inspired by biological neurons [2]. NNs, which are at the core of Deep Learning, have the advantage of being powerful, versatile, and, most importantly, scalable. These characteristics make them good candidates for solving complex Machine Learning problems.

In this thesis, the specific types of NNs used to develop the multiclass model were Multilayer Perceptrons (MLPs) and Deep Neural Networks (DNNs). The words "specific types" should be emphasized, since there are many other types of NNs, like Convolutional Neural Networks (CNNs). Therefore, this introductory part will focus on the two types of NNs that were explored, along with some other related topics.

2.3.1 The Multilayer Perceptron (MLP)

To understand the MLP, it's necessary to first understand the *Perceptron*, an ANN architecture. A Perceptron is based on a specific type of artificial neuron called the *threshold logic unit (TLU)* or also *linear threshold unit (LTU)*. A TLU is shown in Figure 2.3.1.1.

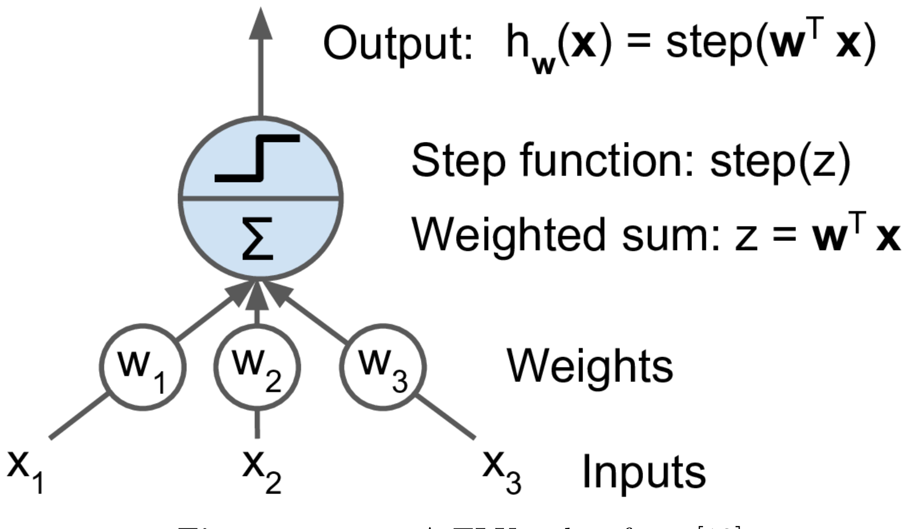

Output: $h_{\mathbf{w}}(\mathbf{x}) = \text{step}(\mathbf{w}^T \mathbf{x})$
Step function: step(z)
Weighted sum: $z = \mathbf{w}^T \mathbf{x}$
Σ
w_1
w_2
w_3
Weights
x_1
x_2
x_3
Inputs

Figure 2.3.1.1: A TLU, taken from [12].

In Figure 2.3.1.1, the specific TLU has three inputs (generally speaking, it could be any amount). TLUs in general have inputs and outputs which are numbers (instead of on/off values like other types of artificial neurons). Furthermore, as indicated in the figure mentioned before, the input connections to the neuron are associated with weights. In this case, w_1, w_2, and w_3, with inputs x_1, x_2, and x_3.

Also, the TLU, as shown in the picture, computes a value z which corresponds to a linear combination of the inputs. In particular, z is such that $z = w_1x_1 + w_2x_2 + w_3x_3$. In dot notation, this is the same as $z = w^T x$, where w is the vector with the input weights and x the input vector. Then, the final output, denoted as h_w, is the result of applying a step function to z, hence $h_w = step(z)$.

It's worth mentioning, before continuing, that both w and x are n-dimensional vectors, where n corresponds to the number of inputs. This number will depend on the specific problem. Now, the next topic to explain is the step function, which is a particular case of an *activation function*.

The activation function of a neuron is basically the function that makes the internal computation visible at the output. A commonly used step function for Perceptrons (which will be described after this topic) is the *Heaviside step function*, which is mathematically expressed as:

$$H(z) = \begin{cases} 0 & \text{if z} < 0 \\ 1 & \text{if z} \geq 0 \end{cases}$$

Sometimes, however, the sign (sgn) function might be used instead [2]:

$$sgn(z) = \begin{cases} -1 & \text{if z} < 0 \\ 0 & \text{if z} = 0 \\ +1 & \text{if z} > 0 \end{cases}$$

Having defined the TLU, it is time now to explain the concept of the Perceptron. Figure 2.3.1.2 shows the Perceptron architecture. As the picture shows, a Perceptron is just a single layer of TLUs, with each TLU connected to all the inputs.

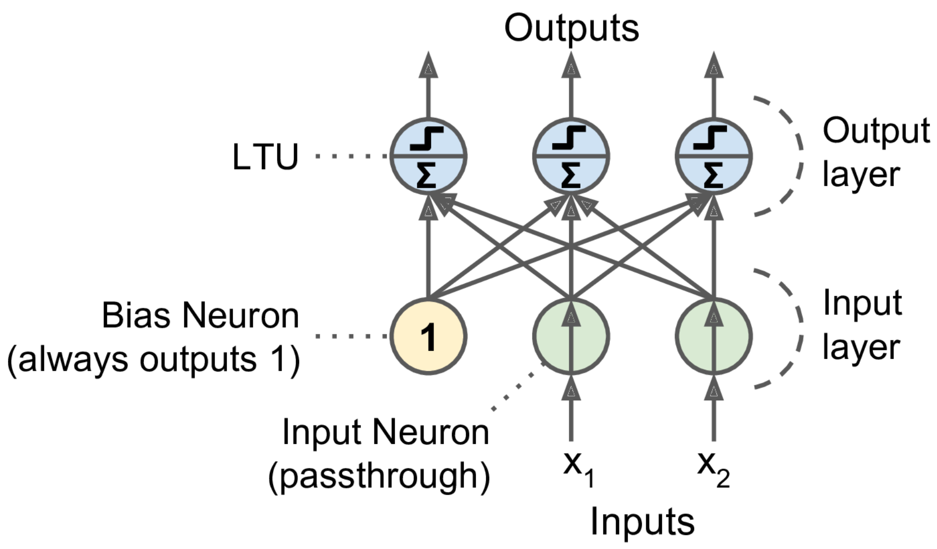


Figure 2.3.1.2: An example of a specific Perceptron architecture, taken from [12].

Furthermore, some more details can be observed in Figure 2.3.1.2. Firstly, the inputs to the Perceptron, x_1 and x_2, are fed to the structure using special neurons called *input neurons* which let values pass through them. These correspond to the green circles in the picture.

Similarly, the yellow circle is also a special type of neuron. It's called a *bias neuron*, which adds an extra bias feature $w_0 = 1$. This special neuron, as the previous description implies, outputs the value 1 permanently.

The example Perceptron in Figure 2.3.1.2 has therefore two layers. The first one is the input layer, composed of the input neurons. The second one is the output layer, which has the three output neurons. It should be noted, also, that the output layer is a *fully connected layer* or *dense layer*, since each neuron is connected to all the outputs of the neurons in the previous layer.

Next, after becoming familiar with the concepts of a TLU and a Perceptron, it's possible to define the *Multilayer Perceptron or MLP*. Figure 2.3.1.3 shows an example of an MLP.

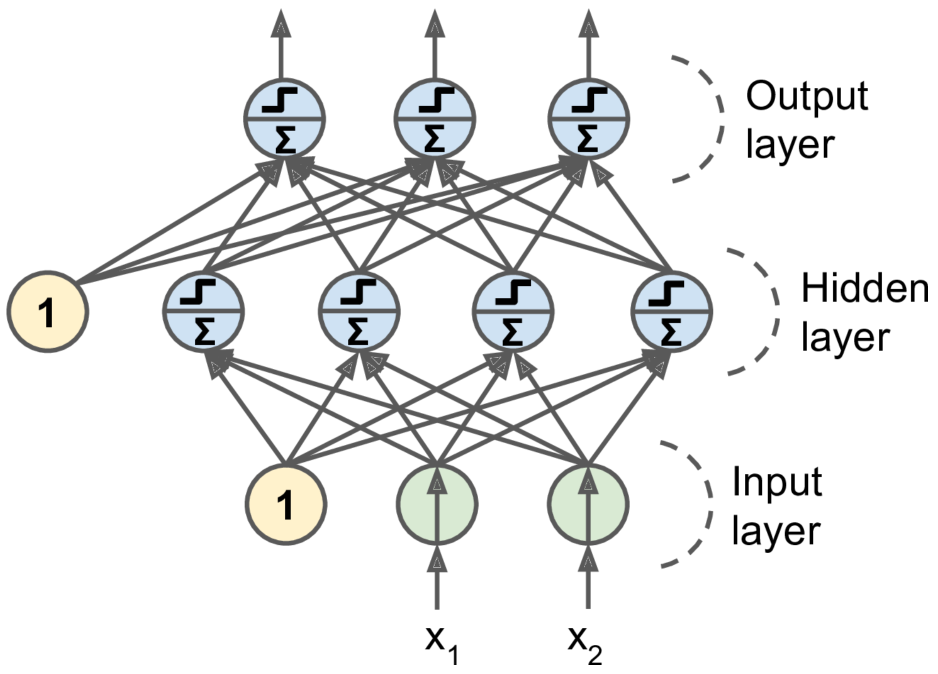


Figure 2.3.1.3: An example of an MLP architecture, taken from [12].

As it's possible to see in the figure, an MLP is basically the combination of layers of TLUs. In general, an MLP is composed of the input layer (which is passthrough), one or more layers of TLUs called the *hidden layers*, and finally the output layer.

Usually, the layers which are close to the input neurons are referred to as *lower layers*, and the ones closer to the output neurons as the *upper layers*. Furthermore, this particular architecture is a *Feedforward Neural Network (FNN)*, because signals flow in one direction from the inputs to the outputs.

The picture also includes all the bias neurons, which are present in all layers except for the output layer. Also, it's possible to observe that the MLP in this case has a fully connected architecture. This does not always happen since it's possible to define a hyperparameter called the *dropout rate*, which will be explained later.

2.3.2 Mathematical representation of neural networks

Now that an initial, simple description of MLPs has been given, it's worth briefly addressing the mathematical representation of MLPs. Ultimately, all neural networks, not only MLPs, are represented using similar mathematical models. Two that are extremely important will be described now.

Mathematically speaking, an NN (or ANN) is simply a lot of matrix-vector multiplications used to express the information in each layer. Different expressions and mathematical symbols are found in literature, however they all represent the same idea.

For instance, there is an equation found in [2] which is used to mathematically describe an MLP for a general number of inputs, samples and neurons in an output layer. In particular, the expression can be used to calculate the outputs of the output neurons in matrix form. This equation is the following:

$$h_{W,b}(X) = \phi(XW + b)$$

Where:

- **X** is the input matrix, with one row per sample and one column per feature.
  - Therefore, its dimension is *mxn*, where m is the number of samples and n is the number of features.
- **W** is the weight matrix. It contains all the connection weights, except for the biases coming from the bias neurons. It has one row per input neuron to the layer, and one column per neuron in the layer.
  - In this case, then, the dimension would be *nxk*, where n is the number of features (or input neurons to the layer) and k is the number of output neurons in the layer.
- Vector **b** is the bias vector for the layer. In other words, it holds the connection weights for the bias neuron (which always outputs 1) connected to the artificial neurons in the layer.
  - It has one bias term for each neuron in the output layer.
- ϕ is the output layer's activation function, which is the same for all the neurons in the layer. In the case of TLUs ϕ is the Heaviside function. However, this topic will be addressed later again in this chapter.

It's relevant to note that when referring to the "output" layer previously, any layer could be considered an output layer (and the layer giving it inputs, an input layer), not necessarily the last layer. This is because the equation can be applied to any consecutive pair of layers in an MLP, or, more generally, neural network.

Therefore, matrix **W** and vector **b** should be defined for each pair of consecutive layers. Consequently, the equation shown before would have to be applied consecutively, until reaching the final output of the neural network. In other words, once the output of a layer is obtained and expressed using $h_{W,b}$, this will be the input to the next layer.

Furthermore, there is still another very good mathematical representation for neural networks that should be described, since it is also more generic in its formulation. It is the one developed by Andrew Ng [6] in his online course on Machine Learning:

$$a^{(j+1)} = g(\Phi^{(j)}a^{(j)})$$

Where:

- $a^{(j)}$ is the activation vector of layer j, where the input layer is $j = 1$ and $a^{(1)} = x$, the input vector.
- $\Phi^{(j)}$ is the weight matrix that performs the mapping from layer j to $j + 1$.
- $a^{(j+1)}$ is the activation vector of layer $j + 1$.
- g is the activation function.

The activation vectors hold the final output of a neuron in a layer. Naturally, the output of a neuron will be passed on to the next layer or not will depending on the activation function selected.

A good way to understand the previous model is to use an example. The same example of the online course will be used [6]. Therefore, suppose there is an MLP, with an input layer with 3 neurons ($j = 1$), one hidden layer with 3 neurons ($j = 2$), and an output layer with only 1 output neuron ($j = 3$).

Then, for this case, the following system of equations can be obtained for the first two layers, based on the previous model:

$$a_1^{(2)} = g(\Phi_{10}^{(1)} x_0 + \Phi_{11}^{(1)} x_1 + \Phi_{12}^{(1)} x_2 + \Phi_{13}^{(1)} x_3)$$

$$a_2^{(2)} = g(\Phi_{20}^{(1)} x_0 + \Phi_{21}^{(1)} x_1 + \Phi_{22}^{(1)} x_2 + \Phi_{23}^{(1)} x_3)$$

$$a_3^{(2)} = g(\Phi_{30}^{(1)} x_0 + \Phi_{31}^{(1)} x_1 + \Phi_{32}^{(1)} x_2 + \Phi_{33}^{(1)} x_3)$$

Where:

- x_k is the $k-$th element of the input vector x.
- $\Phi_{ik}^{(1)}$ is the element of $\Phi^{(j=1)}$ that acts as the weight for the $k-$th element of x. The multiplication between $\Phi_{ik}^{(1)}$ and x_k contributes toward the output value $a_i^{(j=2)}$, which is the $i-$th element of the activation vector $a^{(j=2)}$.
  - Note that $a_i^{(j=2)}$ is the output of the $i-$th neuron in the next layer, $j = 2$.
  - Also, note that x_0 represents the bias neuron in the first layer ($j = 1$).

Then, a similar equation can be produced for the final output, $a_1^{(3)}$:

$$a_1^{(3)} = g(\Phi_{10}^{(2)} a_0^{(2)} + \Phi_{11}^{(2)} a_1^{(2)} + \Phi_{12}^{(2)} a_2^{(2)} + \Phi_{13}^{(2)} a_3^{(2)})$$

The notation is the same. A small, yet important detail should be emphasized now. When calculating the elements of the activation vector for the next layer (layer $j + 1$), the element with index 0 ($a_0^{(j+1)}$) is never calculated. Instead, it's hardcoded into $a^{(j+1)}$ since $a_0^{(j+1)} = 1$.

The last comment regarding this approach is the dimension of the matrix Φ^j and vectors. Contrary to the previous model found in [2], this new weight matrix includes weight terms for the bias neuron and multiplies the activation vector (or input vector) by the right, not the left. As a consequence, no extra bias vector is needed.

Still, for the very same reason, the dimensions of Φ^j are different from the previous case. In particular, if layer j has s_j elements (excluding the bias), and layer $j + 1$ has s_{j+1} neurons (excluding the bias too), then the dimension of Φ^j will be (s_{j+1}) x $(s_j + 1)$. As a consequence, the dimensions of the vectors are $(s_j + 1)$ x (1).

2.3.3 Learning algorithms for MLPs

Although this topic is out of the scope of this thesis, it will be discussed in a general way. This is because some terms related to the training algorithms for MLPs were used during project development and will appear in later chapters.

In order for MLPs to be trainable, the activation function of the TLUs must be changed. Although today there are several activation functions, early researchers replaced the step function with the *sigmoid function*, which is the following:

$$\sigma(z) = \frac{1}{1 + exp(-z)}$$

This was key, since the training algorithm for MLPs uses the gradients of the activation function. Now, the name of the training (or learning) algorithm generally used for MLPs is called the *backpropagation training algorithm*.

Backpropagation uses another widely used technique in Machine Learning, called Gradient Descent. Backpropagation uses this technique in combination with yet another technique to calculate gradients efficiently [2]. Essentially, there are two passes through the neural network (or MLP), one forward and one backward. The backpropagation algorithm is capable of then calculating the gradient of the network's error with respect to all model parameters.

Therefore, what this training algorithm can do is that it can find how each weight and bias value should be optimized to reduce the error of the NN. Once the gradients are obtained, a step of Gradient Descent is performed. Now, a list of steps that the algorithm follows will be given, highlighting relevant terms that were used during the development of this thesis:

- It handles one *mini-batch* at a time. A mini-batch is a subset of the total samples, for example, 32 samples out of 256 in total. This term is also referred to as the *batch size* in Scikit.
- The backpropagation algorithm also goes through the entire training set multiple times. An *epoch* is when the algorithm goes through all the samples, and then begins the process again to start another epoch (unless it's the last one).
- The *forward pass* is performed: a mini-batch is passed to the input layer, which passes the mini-batch to the first hidden layer. Backpropagation then computes the output for each neuron in that hidden layer, and passes these results to the next layer. This is repeated until the last layer is reached.
    - All intermediate results are saved.
- Backpropagation then measures the output error by using a cost function. In particular, the loss function will compare the target and the actual outputs and return some error metric.
- Then, backpropagation will compute how much each network output contributed to the error. This is performed by applying the chain rule, so it's a fast step.

- The algorithm will then measure how much of these error contributions came from the previous layer. This is once again done using the chain rule, following the connections between the current layer and the previous one. This is done until the input layer is reached.
  - These reverse passes measure the error gradient of all connections.
- The last step is performing a step of Gradient Descent to update the weights and biases.

In summary, for each sample, the backpropagation algorithm makes a prediction first (forward pass). Then, it measures the error. Afterward, it goes back through all the layers, measuring the error contribution from each connection. Finally, it performs a step of Gradient Descent [2].

For training, it should be considered that all connections (the weights) should be initialized randomly, otherwise **training will fail**. If, on the contrary, weights and biases are initialized to zero, all neurons will be identical so the algorithm will affect them in the same way. Thus, they will continue to remain identical [2]. Therefore, what's important is to break this symmetry and produce a diverse set of neurons.

Finally, just to give a general idea about it, the algorithm for Gradient Descent is presented next. This formula was taken again from Andrew Ng's online class [6]:

Repeat {

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

}

With $j = 0, ..., n$; n the number of features per sample and where:

- θ_j represents the j-th model parameter, associated to the $j - th$ feature of an input sample.
- m is the number of samples in the dataset.
- h_θ is called the *hypothesis*, the function which takes as input a data sample and outputs a prediction. Evidently, the specific form of the hypothesis varies per ML algorithm.
- $x^{(i)}$ is the i-th sample.
- $x_j^{(i)}$ is the $j - th$ feature of the $i - th$ sample, with $x_0^{(i)} = 1$.
- $y^{(i)}$ is the corresponding target label / value.
- The term $\frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$ represents the partial derivative of the cost function $J(\theta)$ with respect to parameter θ_j.
- α is the **learning rate**.

This algorithm should be performed separately for each θ_j in parallel. Furthermore, when updating the value of θ_j, this should be done simultaneously for all parameters. Also, the algorithm should be repeated until Gradient Descent converges to a solution.

To be more precise, after one iteration is completed, Gradient Descent will update the value of the model parameters. Then, it will combine them with the input samples in the hypothesis to obtain a prediction.

After this, the algorithm will measure the error with respect to the true target using a cost function and make a decision about whether or not to perform another iteration. Gradient Descent could update the parameters again and repeat the overall process until finding a solution that hopefully leads to the lowest cost.

Particular attention should be paid to the learning rate. This is a very important hyperparameter, which is used when training MLPs. As a hyperparameter, it can be also optimized. Ultimately, α will influence the speed at which the NN will converge to a solution (if its value is appropiate).

In fact, if the learning rate is too small, Gradient Descent can be slow and take too long to converge [6]. On the contrary, if it's too large, Gradient Descent might oveshoot the minimum; it may diverge or fail to converge [6]. Different values of α will also most likely lead to different solutions.

2.3.4 MLPs for classification tasks

MLPs can be used for different activities, such as regression and classification. With respect to the last case, MLPs are useful for [2]:

- **Binary classification**: a single output neuron is needed, which outputs a value between 0 and 1. This value is interpreted as the probability that a given sample belongs or not to the positive class (class '1').
- **Multilabel binary classification**: in which the amount of output neurons will match the number of positive classes. Multilabel binary classification occurs when, for instance, one sample can belong to two different classes simultaneously.
  - The outputs are, once again, values between 0 and 1 which are interpreted as the probability that a sample belongs to each class, separately.
  - For the very same reason, the probabilities at the output will not add up to 1.
- **Multiclass classification**: this is the case when there are multiple output classes, and each sample can belong to only one. In this case, one output neuron is required for each class in the output layer.
  - In this case, classes will be exclusive. There is a special activation function which ensures that the probabilities at the output will add up to 1 called softmax. More information will be given later.

The last type is the case of this thesis project: multiclass classification. Later in this chapter, when some details about Keras, TensorFlow and Scikit are given, specific details to implement multiclass classification will be described.

2.3.5 Hyperparameters to optimize in neural networks

For neural networks, as well as MLPs, the following hyperparameters can be tuned to improve performance:

- Number of hidden layers (the amount of layers between the input and output ones).
- Number of neurons in each hidden layer.
- Learning rate α.
- Optimizer (this will be described further in the next section).
- Batch size (the mini-batches mentioned earlier when describing backpropagation).
- Activation function for the hidden layers (the one for the output layer depends on the task).
- Number of iterations (training iterations; however, callbacks, a topic which will be described later, can be used to stop training after no progress is found).
- Dropout rate (portion of neurons ignored) and regularization parameters (if used).

2.3.6 Deep Neural Networks (DNNs) in this project

This topic was created only to highlight DNNs since the term was used for some model versions developed for this thesis. There is no exact definition and it varies over time, but, in general, a Deep Neural Network is a type of neural network that contains lots of layers [2].

Even though today DNNs are understood as deep stacks of layers, sometimes the term is used vaguely to refer to, even, NNs of little depth. For instance, there are researchers who work with architectures with three hidden layers in their studies of "deep" networks [13].

Even more so, in the past (1990s), ANNs with more than two hidden layers were already considered to be "deep" enough [2]. Today, it's possible to find "ANNs" with dozens or hundreds of hidden layers, so, again, the definition is not clear [2].

In this thesis, and to refer only to the work developed for it, MLPs with three hidden layers will be considered to be "DNNs". This is because three hidden layers add a higher degree of complexity in comparison to two or one. Also, these models were much bigger than the final model, which only had 1 hidden layer.

However, in reality, these models are just MLPs. Furthermore, as shown already, the definition of a DNN is vaguely used and depends on the context. For this very same reason, the term was explained here.

2.4 Keras, TensorFlow and Scikit related topics

This section aims to cover some relevant topics related to the Keras API, TensorFlow (which implements it) and Scikit-Learn (the Python library for Machine Learning). These topics are mostly related to specific details that should be considered when developing the multiclass neural network for this thesis.

2.4.1 Keras optimizers

Earlier in the section on neural networks, Gradient Descent was briefly (and very simply) described. This is just one example of a learning / training algorithm, since there are many others. It is, however, still a widely used learning algorithm, from which other algorithms have been developed such as Stochastic Gradient Descent (SGD).

For its part, Keras offers several options of *optimizers*, which are the agents that perform the steps of a given learning algorithm. All optimizers work in the same way: they update the model parameters, evaluate a loss function, and decide whether to update the parameters again. This process is repeated until a solution is reached.

The difference lies in, evidently, *how* this process is performed and the techniques used. For this reason, Keras [7] offers the following optimizers:

- SGD
- RMSprop
- Adagrad
- Adadelta (variation of Adagrad)
- Adam
- Adamax (variation of Adam)
- Nadam (variation of Adam)

Each one of these optimizers takes parameters as inputs during compile time. Some are optimizer specific, while others are common to all. For instance, all of them take as input a learning rate, which was optimized during project development for the multiclass model.

Moreover, for neural networks, usually the SGD and Adam optimizers are selected [2]. In this thesis project, these optimizers were explored in addition to a third one that was considered during the optimization process. This third option was the Adagrad optimizer. More specific details will be given later.

2.4.2 Activation and cost functions for NN classifiers

As described earlier, there are three main types of classification tasks that can be performed by neural networks (MLPs). For each one of them, the activation of the output layer and the loss (cost) function to use will be different. Table 2.4.2.1, which is adapted from [2], shows what options to choose for each case.

Table 2.4.2.1: NN (MLP) details for each classification task type, adapted from [2]

Hyperparameter	Binary cl.	Multilabel Binary cl.	Multiclass cl.
Output neurons	1	1 per label	1 per class
Output layer activation	Logistic	Logistic	Softmax
Loss function	Cross entropy	Cross Entropy	Cross Entropy

Notes:

- "cl." above refers to "classification".
- The logistic function is the sigmoid function described earlier.
- The softmax function is the one explained before that ensures all output probabilities will add up to 1.
- The specific type of cross entropy loss function will depend on data encoding techniques. This topic will be addressed later.
  - For now, it should be said that, in the case of binary classification, the specific type of cross entropy function to use is *binary crossentropy*.

Now, another relevant topic to highlight at this point is the activation function of the hidden layers in an MLP. There are different options, including the sigmoid (logistic), Relu and tanh functions. From these, for MLPs Relu usually gives good results.

In order to visualize the functions just mentioned, Figure 2.4.2.1 shows the most popular activation functions that are typically chosen for neural networks [2]. All of these (except for the step) can be implemented using Keras and TensorFlow.

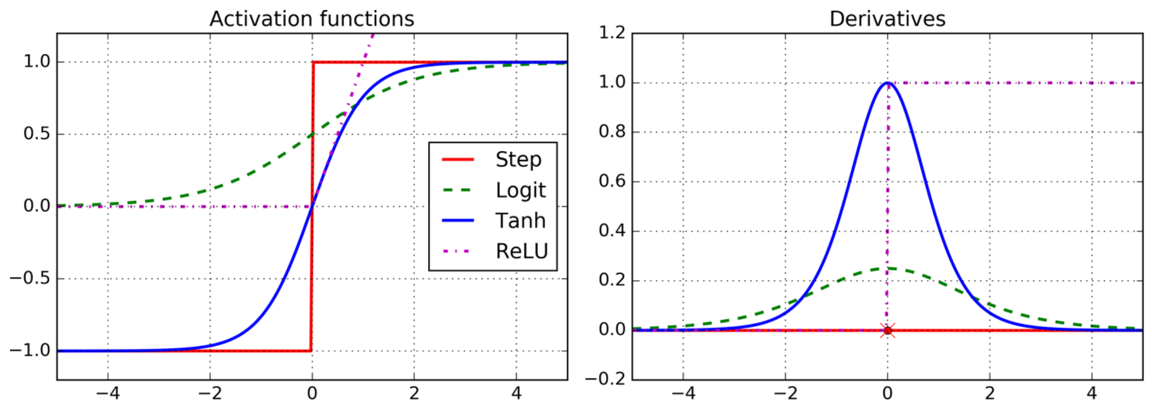


Figure 2.4.2.1: Popular activation functions for neural networks, taken from [2].

Of course, the functions illustrated in Figure 2.4.2.1 are just four out of many. In fact, there are other options like the Linear function, the Leaky Relu, Softplus, and the Gaussian. Additional information regarding these can be found in the Appendices.

2.4.3 Cross-validation in Scikit-Learn

The concept of cross-validation was introduced earlier in this chapter. Now, it's relevant to highlight how Scikit performs cross-validation specifically, what options it provides and which ones were chosen for this thesis.

For CV, there are different Scikit functions available. There are some that perform CV automatically and that take the number of folds as an input parameter. Examples of these include **cross_val_score**.

This helper function, specifically, evaluates a certain "score" using CV [14]. It takes as input an estimator object (the predictor) and a training set, among other parameters. Then, after being called, it returns an array of numbers, where each one is the average result obtained for each fold.

These results correspond to the "score" method of the estimator object [15]. Usually, it corresponds to the accuracy metric. It should be clarified that Scikit calls the different metrics by the name "score". Additionally, a score type can be changed, but there's a limited amount of metrics that can be measured.

In particular, a missing metric is the validation accuracy. Naturally, this limits the degree of control of the cross-validation process and the information that can be extracted. For this reason, Scikit also has the option to perform CV manually.

Therefore, by using Scikit's **KFold**, it's possible to obtain indices for the training and validation subsets. It will take as input the training set and return the indices for every iteration. Figure 2.4.3.1 shows an illustration developed by Scikit to show how KFold works [15] and to emphasize it is not affected by classes or groups.

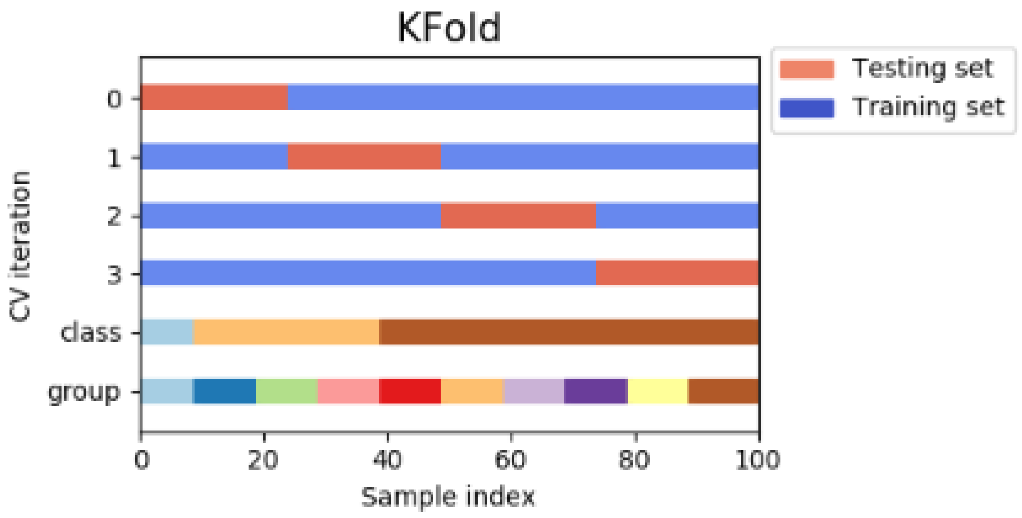


Figure 2.4.3.1: Illustration of Scikit's KFold, taken from [15].

In Figure 2.4.3.1, it should be noted that "testing set" refers to the set used for validation during CV, meaning the validation set. Also, in the picture it's possible to see how all folds (the sub divisions of the original set) eventually become validation sets once.

Consequently, when referring to KFold later, it will be called "manual KFold" in this text to point to the fact that this CV is performed manually by the user and not with a helper function. Moreover, there is still another type of manual CV that should be mentioned.

This second type of CV that Scikit provides is **Stratified CV**. This means that each subset (or fold) of the original dataset will contain, approximately, if not perfectly, the same proportion of samples of each target class like in the training set that is given as input for CV [15], [16].

Then, to perform cross-validation as just described, Scikit provides the user with **StratifiedKFold**, which works just like KFold. Similarly to this last case, later in this thesis StratifiedKFold will also be referred at times as "manual" StratifiedKFold. Now, Figure 2.4.3.2 illustrates how StratifiedKFold works.

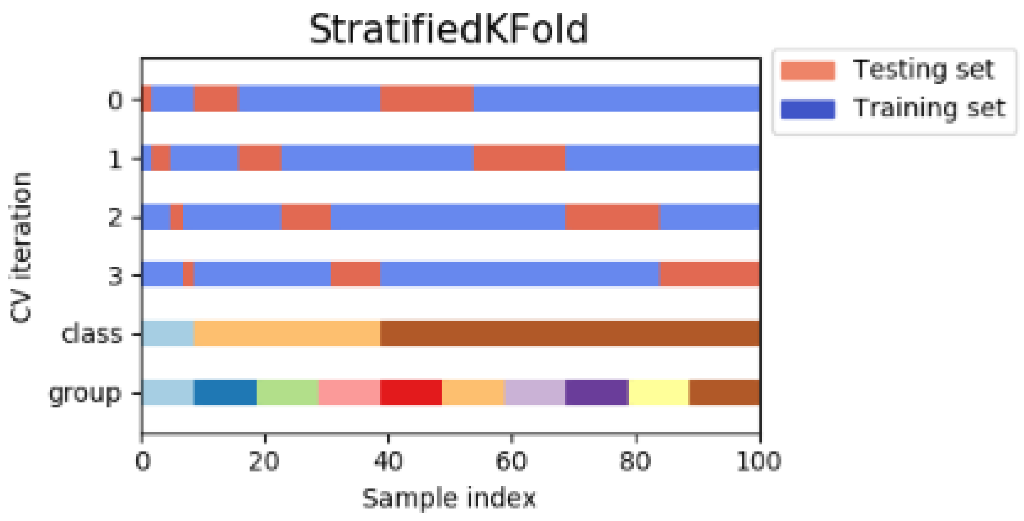


Figure 2.4.3.2: Visualization of Scikit's StratifiedKFold, taken from [15].

Just like for the previous figure, in Figure 2.4.3.2 the word "testing set" is used to refer to the validation set. Moreover, it is clear in the picture how the validation set is made using different proportions, most likely to match the class proportions from the original training set.

2.4.4 Hyperparameter tuning: searching with grids in Scikit-Learn

For hyperparameter tuning, which will be described in depth in Chapter 4 and Chapter 5, Scikit offers different options. These consist in, basically, using grids. A grid in this context is a collection of arrays. Each array is associated to a hyperparameter, whose name is already defined by Scikit.

Alternatively, if constructing the MLP using a builder function, it's possible for the user to assign a name to input hyperparameters to that function. This way, these can be later considered during the search.

Moreover, each array of the grid will hold several values to test. It's at this point, when selecting what values for each hyperparameter to try, when the grid techniques developed by Scikit differ.

In particular, there are two main techniques, which were both explored during the project development phase for this thesis. These are:

- **GridSearchCV**: this corresponds to an exhaustive search, which means that all the possible combinations of hyperparameter values that belong to the grid are tested.
- **RandomizedSearchCV**: in this case, the combinations selected are random. Therefore, an input parameter to RandomizedSearchCV is the number of iterations to control how many combinations will be explored.

In both cases, an object is returned, which has specific attributes. After finishing the search process, it will be possible to obtain the results by studying these attributes. In particular, .best_ params_ and .best_ score_ are very useful. These return the best combination and its score, respectively.

Finally, it should be noted that for GridSearchCV and RandomizedSearchCV, there is a cross-validation process involved. In fact, one of the input parameters to initialize the search is "cv", which is used to indicate the number of folds. In other words, each combination is tested by performing cross-validation on the training set and saving its score, which is, usually, the accuracy by default.

2.4.5 Data preprocessing techniques using Scikit

Until now, topics related to ML algorithms or models have been discussed. Equally important, as emphasized before, are topics related to data. Therefore, now the two main techniques explored for data preprocessing and how these are implemented by Scikit-Learn will be explained.

The first one corresponds to *standardization*. To perform this data transformation, Scikit provides a "scaler" object which is created using **StandardScaler** [17]. Specifically, what this process does is that it performs the following [17]:

$$z_k^{(i)} = \frac{x_k^{(i)} - \mu_k}{s_k}$$

With $k = 1, .., n$, with n the number of features and where:

- $z_k^{(i)}$ is a transformed *feature*. In particular, it's the transformed $k-$th feature of sample i, with $z^{(i)}$ the new, transformed sample.
- $x_k^{(i)}$ is the $k-$th feature of sample i, $x^{(i)}$.
- μ_k is the average of the $k-$th feature (calculated using the training samples).
- s_k is the standard deviation related to the $k-$th feature (calculated using the training set).

It should be noted, above, that the notation developed by Andrew Ng was used once more [6].

Continuing now, what StandardScaler does is that it standardizes features. Furthermore, as indicated before and explained in [17], the actions of centering and scaling happen **independently** on each feature of the data by calculating the required statistics with the training set.

Additionally, the scaler object can be saved and reused later. This is because this object will store the required statistics which were calculated with the training set. Specifically, the "fit" method should be used with the training set as an input. Then, the "transform" method should be used with the training set and any validation / test sets.

Standardization is very commonly used in Machine Learning, and was extensively used in this thesis project. Very often it is, in fact, a requirement, since models might not perform adequately when features are not normally distributed [17] (ie. Gaussian distribution with a mean equal to zero and unit standard deviation).

The second technique that was explored was min-max scaling (or normalization). It was explored since sometimes neural networks perform better when inputs fall in a range between 0 and 1 [2]. What Scikit provides to accomplish this is the transformer called **MinMaxScaler**. On the contrary, using the transformer StandardScaler will not guarantee the standardized features will fall in the previous range.

In particular, what MinMaxScaler does is return an object like StandardScaler. This object, in turn, transforms each feature, **individually**, by shifting and rescaling it to a given range [18], which is 0-1.

Specifically, this is done by subtracting the minimum value from the feature and then dividing this result by a subtraction. This subtraction is the maximum value minus the minimum value.

Also, just like before, the transformation must be first applied using the test set, and then applied to all available sets. It is possible to save the object with the statistics and reuse it later too.

As a final comment to make a distinction between both techniques, there is a difference regarding how affected they are by outliers in the dataset. In fact, standardization is less affected by outliers than normalization. This is because of how the transformations are mathematically performed.

2.4.6 Hot-encoded vectors with Keras and their effect on the loss function

A data encoding technique that should be addressed is that of *hot-encoding* data. More precisely, in this thesis, the option of hot-encoding the target indexes was explored. However, it is possible to hot-encode datasets as well.

It should be explained, at this point, that each of these target indexes represented a specific class. Since there were seven classes, indexes ranged from 0 to 6, one for each class. Hence, the target labels were, simply, integer values.

Then, hot-encoding in the context of this thesis (a multiclass model) occurred when a target index was encoded in vector form. Assuming there are N exclusive classes to which a training sample can belong to, a target label will be transformed into an N-dimensional vector where each element will represent a class.

As a consequence, for this thesis, the vectors were 7-dimensional. Keras, as implemented by TensorFlow, can automatically organize the elements of the new vectors and associate each of them to a specific class. Consequently, all entries will be 0 except for one, which will be 1. This 1 is used to associate the old integer value to the class represented by that element in the hot-encoded vector.

Now, when using hot-encoded vectors in classification tasks, the loss function (crossentropy) will have to be modified when using Keras and TensorFlow. Specifically:

- When using hot-encoded target labels for multiclass models, the function *categorical_crossentropy* should be used.
- When developing a multiclass model, and if target indexes are used, the loss function that should be chosen is *sparse_categorical_crossentropy.*
- For binary classifiers, the loss function to use instead is *binary_crossentropy.*

2.4.7 Other details

There are, still, many details regarding the development of the project using Keras, TensorFlow and Scikit-Learn. In particular, certain functions and methods used, packages and callbacks. However, all these specifications, and the choices made, will be explained and justified at approppiate times in Chapter 4 and Chapter 5.

Perhaps, for now, the following should be briefly mentioned:

- Keras NN models have several methods [19].
  - .compile compiles the model; here, the optimizer and loss functions need to be specified.
  - .evaluate returns the loss value and the metrics obtained during test mode.
    - These results were saved with the names of "best_test_loss" and "best_test_accuracy".
    - These will be reported in Chapter 5.
  - .fit trains the model; the number of epochs can be specified.
  - .predict generates real predictions based on the input samples, post training.
- Scikit estimator objects also have methods. One is .score, which returns a "score" value which is usually the accuracy metric by default on a given dataset.
- There's a function called cross_val_predict, by Scikit. It generates cross-validated predictions [20].
  - Although it works with CV, Scikit makes it so a prediction will be made once for each sample, which will in turn belong to only one validation set.
- There's a function called train_test_split, by Scikit. It allows to divide one set into two subsets, while specifying the proportions for each one.

Chapter 3

Related works

This chapter will focus on describing a project which is strongly related to the topic of contaminant detection in hazelnut-cocoa spread jars. Specifically, this project was developed by another student from Electronic Engineering at Politecnico di Torino for his master's thesis [1].

The previously mentioned work focused on the problem of contaminant detection within the broader area of quality assurance. Toward this purpose, different classifiers were developed. Only one, in the end, was deployed to hardware, and will be referred to as the *Binary Classifier* throughout this thesis. Now, this project will be briefly described.

3.1 Microwave Imaging (MWI) Technology for food contaminant detection

The goal of the thesis project mentioned before was to develop Machine Learning (ML) algorithms, specifically classifiers, for detecting contaminants in hazelnut-cocoa spread jars on conveyor belts. These algorithms were intended to be applied to a Microwave Sensing System (MWS), which was a prototype developed at Politecnico di Torino.

The project was divided into two main data phases. In the first, synthetic tomographic images were studied. In the second, real data was obtained using the MWS prototype for different types of contaminants. Furthermore, for both types of data, two different ML classification algorithms were developed and tested. These corresponded to Support Vector Machines (SVMs) and Neural Networks (NNs).

Regarding the first dataset, the total number of samples was 1800 images. In this case, a 10-folds CV accuracy of 99.167% was reached using the SVM developed, and a 5-folds CV accuracy of 99.167% was obtained for the MLP developed [1]. The errors calculated when evaluating the algorithms using a test set of 360 samples corresponded to 0.278% and 1.111% respectively [1]. Therefore, it was considered that this was enough proof to reinforce the idea of applying Machine Learning to the foreign body detection problem with MWI.

Also, given that these results looked promising, the project moved forward to the second dataset. For this part, as already mentioned, real samples were obtained using the prototype at Politecnico di Torino. The specific steps taken to gather the data will be described in detail in Chapter 4. However, a general idea can be given at this point.

In particular, the real dataset consisted of 2400 samples, with six different types of contaminants: a cap shaped plastic, a glass fragment, a small plastic sphere, a metal sphere, a triangular plastic piece and a big, pink, plastic sphere. There were 200 samples for each contaminant, resulting in a total of 1200 samples. Moreover, the remaining 1200 samples were about non contaminated jars.

The general idea is that a jar filled with oil, for reasons that will be explained in Chapter 4, would be put underneath an arc of six antennas. This way, using a Vector Network Analyzer (VNA) and a controller which would monitor the VNA and a switching matrix, a single sample would be acquired. In precise terms, each sample consisted in a 6x6 scattering matrix, which would be then processed to obtain an array with 15 features per sample.

After preprocessing the dataset with standardization and training the classifiers, the results obtained were the following: 10-folds CV accuracy of 95.052% for the SVM and a 5-folds CV accuracy of 95.833% for the MLP developed [1]. The error obtained had a value of 6.04% when using a test set with 480 samples [1].

It should be noted at this point that, of course, an optimization process was performed to obtain the final SVM and MLP classifiers. In particular, the classifier of interest for the multiclass neural network is the MLP classifier. This MLP, which had an architecture consisting of 2 hidden layers, will be recurrently referred to in this text as the *Binary Classifier*, as mentioned at the beginning of this chapter. This term is used since the MLP was trained for a binary classification task (contaminated or non contaminated jars).

Now, returning to the SVM and MLP classifiers for the real dataset case, only one was chosen for hardware implementation. Since both showed the same performance, the MLP was chosen for being considered to have the potential for faster performance in hardware [1]. A step of HLS was performed with Vivado HLS which then allowed to synthesize the architecture for a target board, which was the Xilinx Zynq-7000 SoC.

In the following two sections, some important results that will be constantly referred to in this thesis will be mentioned. Also, a brief discussion on the challenges left open by this project will be discussed.

3.2 Important results achieved by the Binary Classifier

In addition to the values already reported, there are some important results that should be considered. In particular:

- The misprediction ratio of the triangular plastic piece was equal to 51.220% [1], with 21 of its samples being mispredicted as non contaminated jars. In the test set where this percentage was calculated, there were 41 triangular plastic samples.
- The latency obtained in hardware using the FPGA accelerator was around 3 ms [1], which satisfied the project's requirement of achieving a latency of 100 ms maximum for the conveyor belt.

This information might not seem very useful at the moment. However, these results will be addressed again later in this thesis. Also, they were used as points of reference for the multiclass neural network throughout development.

3.3 Challenges

In the project's conclusion, some of the ideas highlighted were [1]:

- The real dataset should be enlarged to also include new types of contaminants, like wood pieces.
- Try different ML algorithms and validate them dynamically (when the conveyor belt is moving). A specific example given was to use Ensemble Learning (combined models).
- Try different homogeneous food products, like honey or yoghurt.
- Try different heterogeneous food products, like chocolate spreads with hazelnut grains.

For its part, the multiclass neural network model developed for this thesis used the real dataset with which the Binary Classifier was trained and evaluated. Additionally, all project files developed for the binary MLP were received and studied carefully.

However, it is precisely because of this reason that some aspects regarding the Binary Classifier were noticed. A situation left as a further challenge in [1] was the high error ratio concerning the triangular plastic contaminant. After studying the project and the files, it was thought that maybe the problem was not the model itself but, instead, the samples.

The methodology used in [1] is highly meticulous and controlled. For the same reason, the samples that were received were each considered to be high quality and reliable as well. However, there were some aspects to further consider:

- The homogeneity or heterogeneity of the samples obtained for the real dataset as a group.
  - In particular, when taking a sample, the contaminant would be in a specific position in the jar.
  - The amount of samples taken for every position was not the same for all contaminants. This will be detailed in Chapter 4.
  - Therefore, even though each sample is good quality, the experimental conditions for all contaminants are not exactly the same.
- Although cross-validation was performed with balanced sets, the sets were not stratified.

It was based on these observations, mainly the first one about the data itself, that the idea of a multiclass approach was formulated. The initial goal was to study how and if a classifier could detect each contaminant separately and to what extent. This was the start of the multiclass neural network approach, which brought a new perspective and challenges regarding contaminant detection in the jars.

Chapter 4

Methodology

This chapter aims to explain the steps followed throughout this project to give a clear idea of the design, initial training, optimization, selection, validation, testing, analysis and implementation stages. As a direct consequence this will also ease the understanding of the results presented in the next chapter.

The methodology chapter begins with a characterization of the working environment, followed by a section on data management and source samples. Then, the steps taken during project development are presented and explained.

After this, precise descriptions of the Python notebooks developed in the project are given. Finally, some further details are explained like the project file formats, the callbacks used and the Keras wrapper required for training the various models.

4.1 Environment description

The project was developed both on a computer at Columbia University and using a virtual machine. In the case of the virtual machine environment, the OS was Ubuntu, version 18.04.

In both cases, a virtual environment was used with specific libraries and packages as well as a particular API (Keras) and web application (Jupyter). The following subsections will specify the versions used for model development. To clarify, when referring to all these as a group, the word "elements" will be used.

4.1.1 Most important elements

The most important elements are indicated in Table 4.1.1.1. It's possible to find their version and classification.

Table 4.1.1.1: Details of the most relevant elements

Element	Type	Version
Jupyter Nt.	web app.	6.0.2
Keras	API	2.2.4-tf
Matplotlib	library	3.1.2
NumPy	library	1.17.4
Python	language	3.6.9
scikit-learn	library	0.22
TensorBoard	library	2.1.0
TensorFlow	library	2.1.0

Above, it should be noted that "Nt." refers to "Notebook", and that "web app." and "language" obviously refer to "web application" and a programming language. More important that this is the fact that the Keras version includes the abbreviation "tf". This points to the fact that it's TensorFlow specifically that's used to implement the API.

4.1.2 Complete package list

In order to fully characterize the virtual environment it's necessary to indicate all the installed packages. These can be found in Table 4.1.2.1 and Table 4.1.2.2.

Table 4.1.2.1: Package versions (Part I)

Package	Version	Package	Version
absl-py	0.9.0	joblib	0.14.1
jsonschema	3.2.0	jupyter	1.0.0
astor	0.8.1	jupyter-client	5.3.4
attrs	19.3.0	jupyter-console	6.0.0
backcall	0.1.0	jupyter-core	4.6.1
bleach	3.1.0	Keras-Applications	1.0.8
cachetools	4.0.0	Keras-Preprocessing	1.1.0
certifi	2019.11.28	kiwisolver	1.1.0
chardet	3.0.4	Markdown	3.1.1
cycler	0.10.0	MarkupSafe	1.1.1
decorator	4.4.1	matplotlib	3.1.2
defusedxml	0.6.0	mistune	0.8.4
entrypoints	0.3	more-itertools	8.0.2
gast	0.2.2	nbconvert	5.6.1
google-auth	1.10.0	nbformat	4.4.0
google-auth-oauthlib	0.4.1	notebook	6.0.2
google-pasta	0.1.8	numpy	1.17.4
grpcio	1.26.0	oauthlib	3.1.0
h5py	2.10.0	opt-einsum	3.1.0
idna	2.8	pandas	0.25.3
importlib-metadata	1.3.0	pandocfilters	1.4.2
ipykernel	5.1.3	parso	0.5.2
ipython	7.10.2	pexpect	4.7.0
ipython-genutils	0.2.0	pickleshare	0.7.5
ipywidgets	7.5.1	pip	20.0.2
jedi	0.15.2	prometheus-client	0.7.1
Jinja2	2.10.3	prompt-toolkit	2.0.10

Table 4.1.2.2: Package versions (Part II)

Package	Version	Package	Version
protobuf	3.11.2	six	1.13.0
ptyprocess	0.6.0	tensorboard	2.1.0
pyasn1	0.4.8	tensorflow	2.1.0
pyasn1-modules	0.2.8	tensorflow-estimator	2.1.0
Pygments	2.5.2	termcolor	1.1.0
pyparsing	2.4.5	terminado	0.8.3
pyrsistent	0.15.6	testpath	0.4.4
python-dateutil	2.8.1	tornado	6.0.3
pytz	2019.3	traitlets	4.3.3
pyzmq	18.1.1	urllib3	1.25.7
qtconsole	4.6.0	wcwidth	0.1.7
requests	2.22.0	webencodings	0.5.1
requests-oauthlib	1.3.0	Werkzeug	0.16.0
rsa	4.0	wheel	0.33.6
scikit-learn	0.22	widgetsnbextension	3.5.1
scipy	1.4.1	wrapt	1.11.2
Send2Trash	1.5.0	zipp	0.6.0
setuptools	42.0.2		

Evidently, the list is extensive. It's worth mentioning that, when checking for the version of Jupyter, the following packages are displayed:

- ipykernel
- ipython
- ipywidgets
- jupyter-client
- jupyter-core
- nbconvert
- nbformat
- notebook
- qtconsole
- traitlets

As a last comment, it's good to highlight that all packages related to the elements from Table 4.1.1.1 are present in Tables 4.1.2.1 and 4.1.2.2. Some have the exact same name like scikit-learn while others have a related one such as the package: "tensorflow-estimator".

4.1.3 Specific modules used while developing in Jupyter

It's important to mention the specific modules that were used for developing the different model versions. In particular, the Python modules that were used are listed next:

- json
- os
- pickle
- time

In addition to this, various scikit modules were used too. The following list shows them with what was imported from them between parenthesis when applicable.

- datasets
- metrics (confusion_matrix)
- model_selection (cross_val_predict, cross_val_score, GridSearchCV, RandomizedSearchCV, StratifiedKFold, train_test_split)
- preprocessing (MinMaxScaler, StandardScaler)

The last module that was used was pyplot, from matplotlib.

4.1.4 Previously developed files for the Binary Classifier

It's very important to highlight that while developing the multiclass model there were some files that were adapted from the ones made for the Binary Classifier in [1]. These proved to be essential during early development.

The files, which were submitted to Politecnico di Torino in 2019, are:

- custom_functions.py
- libraries.py
- Load Dataset.ipynb
- MLP-NN-BO-2Layers.ipynb
- Preprocess Dataset.ipynb

In the notebooks for the multiclass model, all parts are commented so it's fairly easy to identify which pieces were modified for this project or which ones were used as they originally were.

Moreover, an important aspect should be emphasized which is that both libraries.py and custom_functions.py were loaded to retrieve samples and preprocess them. However, out of the libraries and modules in libraries.py, only the ones that were finally used have been listed before. In the case of custom_functions.py, in the end, no functions were used for the multiclass model.

4.2 Data management

Data preprocessing and sample quality were both taken into consideration for this project. The following subsections will fully describe all related details to these topics. In particular, an important focus of this entire section will be the way in which the available data was manipulated as well as why certain decisions were taken.

4.2.1 Samples used for the multiclass model

The samples used for the multiclass approach are the same ones that were used to train the Binary Classifier [1]. As explained in [1], the total number of samples is 2400, with 1200 for jars with no contaminants and 1200 for jars with contaminants.

There are two main topics to describe at this point. The first topic deals with the contaminant types which are included among the samples. For its part, the second topic focuses on the origin of these samples.

Regarding the first then, there are six types of contaminants. For each type, there are 200 samples, 1200 in total. Specifically, the contaminants are:

- a big, pink, plastic sphere
- a cap shaped piece of plastic
- a glass fragment
- a metal sphere
- a small plastic sphere
- a triangular plastic piece

Just for visualization purposes, the contaminants are illustrated in the following pictures. These are were all taken from [1].

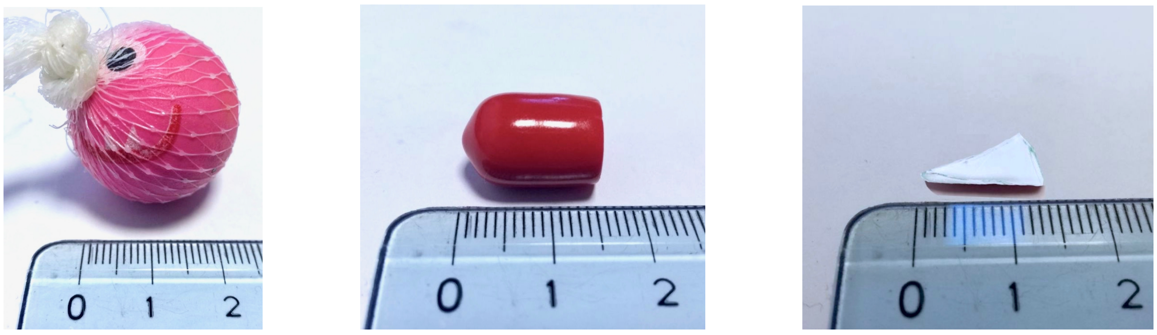


Figure 4.2.1.1: Pink plastic sphere in a plastic net (left), cap shaped plastic (center), glass fragment (right) [1]

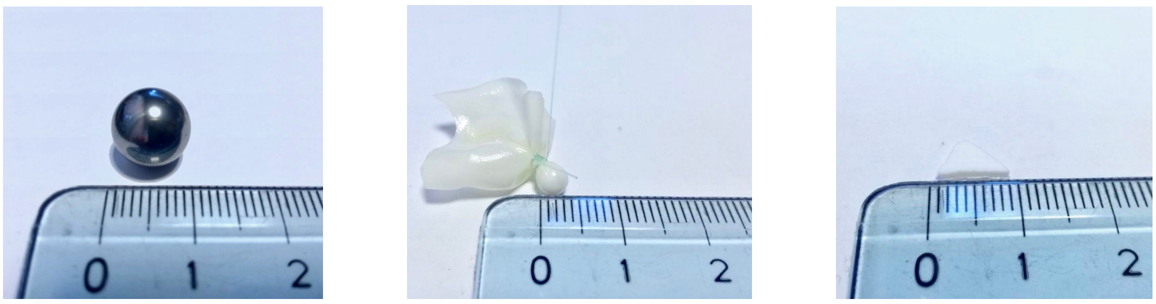


Figure 4.2.1.2: Metal sphere (left), small plastic sphere in a latex glove (center), triangular plastic piece (right) [1]

Regarding the second topic, all the samples (both for contaminated jars and non contaminated jars) were taken by [1] at a laboratory in Politecnico di Torino with the aid of a PhD student with a special prototype. Describing this process in detail is out of the scope of this thesis but a general idea can be given.

Therefore, a simple description of the data acquisition process is the following:

- A contaminated jar (or non contaminated jar) would be put underneath an antenna arch with 6 antennas.
- Then, with a Vector Network Analyzer (VNA) and a controller (a laptop with a Matlab script written at Politecnico), which would monitor the VNA and a switching matrix, a sample would be acquired.
- Therefore, one sample at this stage corresponded to a 6x6 scattering matrix, in a .txt file, since there were six antennas.
- Then, each 6x6 scattering matrix (sample) would be converted into a vector with 30 features. For this, the upper diagonal part of the matrix would be kept. The result, for each sample, would be a new .txt file with 30 values.
  - The previous was done because each matrix was reciprocal and symmetric so only the upper triangular part was required. The main diagonal could be excluded.
  - Consequently, only 15 scattering parameters were really needed but each one was a complex number (with a real and imaginary part). This is why each sample, in the end, became a vector with 30 features.
  - It should be mentioned that each scattering matrix represents the network composed by the antenna arch, the air and the jar under the arch.

For further details, please refer to [1]. Now, the last important aspect regarding sample acquisition is the location of the contaminants in the jar. Out of the six contaminant types, only four had 100 samples taken with the contaminant in the jar at half its height and the remaining 100 on the surface of the oil inside [1]. In the case of the other two contaminants, all 200 samples were taken with the contaminant floating on the surface of the oil.

[1]As explained in [1], safflower oil was used instead of hazelnut-cocoa cream because it's transparent. As a result, contaminant monitoring becomes easier. More important than this, however, is the fact that the oil behaves quite similarly to chocolate spread in the microwave spectrum. In fact, $\varepsilon_{oil} \simeq \varepsilon_{cream} = 2.86$ @ 10 GHz [1].

Furthermore, for three contaminant types, there were variations. For instance, triangular plastics of different sizes were used. Table 4.2.1.1 and Table 4.2.1.2, which were both adapted from [1], contain all the available information.

Table 4.2.1.1: Details of the samples for each contaminant [1]

Contaminant type	Samples at half height	Samples on oil surface
Pink plastic sphere	100	100
Cap shaped plastic	200	-
Glass fragment	100	100
Metal sphere	100	100
Small plastic sphere	100	100
Triangular plastic	200	-

Table 4.2.1.2: Contaminant measurements [1]

Contaminant type	Max. size (mm)	Min. size (mm)
Pink plastic sphere	20	-
Cap shaped plastic	15	9
Glass fragment	13	2
Metal sphere	10	-
Small plastic sphere	3	-
Triangular plastic	8	1

Finally, above, in Table 4.2.1.1, it should be highlighted that the total number of samples for each type of contaminant is 200.

4.2.2 Data preprocessing techniques chosen

During model development, there were two main techniques used for data preprocessing after the samples were loaded. One was standardization and the other one was min-max scaling (which some call normalization although it's not the same transformer of the same name by Scikit-Learn).

Standardization was chosen because the orders of magnitude of the features of the contaminant samples were different. In general, two features in one sample could differ by a factor of x10, x100 or x1000. For instance, sample #188 of the triangular plastic piece had a feature whose value was -0.002533 (feature #1). What's more, other features in the sample had the following values: -0.000627 (feature #2, x10), -0.000032 (feature #27, x100) and -0.000009 (feature #18, x1000).

Therefore, standardization was considered to be necessary to help the optimizer converge during training. For this, the StandardScaler transformer developed by Scikit-Learn was used. It's relevant to mention that feature values could be positive and negative, with one integer digit and six decimal places.

In addition, as mentioned previously in Chapter 2 and in [2], it can be beneficial when training neural networks to limit inputs to a 0-1 range. This can be accomplished by applying a min-max scaling transformation (MinMaxScaler transformer by Scikit [18]).

As a result, after standardizing the features a second preprocessing stage was completed to min-max scale the samples. However, the results were not good; actually performance was worse. Further details on this topic will be given in Chapter 5 in which all the results of the project can be found.

After inspecting the min-max scaled features, it was noticeable that features within one sample became very similar. This could be a reason why performance decreased. Again, more details will be given in Chapter 5.

Ultimately, after exploring the effects of these two different data transformations standardization was chosen. Also, it's a technique that's less sensitive to outliers. Considering this, standardization could potentially aid the training of the multiclass model by improving preprocessed data quality. This was another reason, in addition to the different orders of magnitude, why standardization was strongly preferred.

4.2.3 Data preprocessing steps

To perform data preprocessing for the multiclass model, it's necessary to take certain steps in order. The following is a general work flow that applies to all the notebook files that were developed:

- Declare a path where there must be one folder per class, named like the class, with the original 200 samples in .txt format for the particular class.
- Use the datasets.load_files function (from sklearn.datasets) to load the samples, using a list 'X' for the samples, a numpy.ndarray 'y' for the target indexes and another list 'z' for the class names [21].
- Then, eliminate the commas from each sample in 'X' using the Python function 'split' and save the result as a 2D array 'X' (numpy.ndarray type) to later work with Scikit functions.
- Save the previous information ('X', 'y', 'z') in a .npz file[2] using np.savez (from NumPy).
- The data can optionally be retrieved using the path declared at the beginning using np.load (from NumPy); it's also possible to verify the number of samples per class using NumPy.
- Then, the train_test_split function (from sklearn.model_selection) can be used to create a training set and a test set; the proportions can be verified using Python and NumPy.
  - When using the previous function, it's important to set the right values for the input parameters which are mainly: test_size, train_size, stratify, Shuffle, seed.
  - The first two vary depending on the data splitting approach, the third one be set to 'y' and the fourth to 'True'. The fifth one was set to '0' at this stage. For more information please see [23].
- Next, by using train_test_once more, the original training set can be separated into the final training set and validation set according to the data splittng strategy selected.

[2]A .npz file is a zipped file that saves different files named like the variables they contain. It is not a compressed archive, however. For further details please see [22].

  - In this case, stratify (the parameter) was set to the target indexes belonging to the training set, Shuffle to True and the seed to 42.
- Again, using Python and NumPy the samples for each class in the sets can be verified as well as the percentages due to stratified data splitting, per class.
- The sets at this point can be saved in .npz file format as previously described.
- The last step is to apply the transformer: StandardScaler by Scikit-Learn by generating a scaler object [17].
  - This object takes as input the entire training set and generates the statistics required for standardization.
  - The above is achieved by applying the fit method to the scaler object when taking the training set as input.
  - To apply standardization, it's enough to simply apply the method "transform" to the scaler object by taking each set as input. Therefore, this step is repeated three times for each set.
  - These preprocessed sets can be saved in .npz file format for later use.
  - The scaler object, after performing standardization, can also be saved using the Python's pickle module. The file format in which it can be saved is the .sav file extension.

It's very important to highlight that, since work for the multiclass model began from the Binary Classifier, the previous steps were adapted from [1]. In particular, the files "Load Dataset.ipynb" and "Preprocess Dataset.ipynb" were used as strong references.

4.2.4 Label representation strategies

Given the multiclass nature of the neural network, there were two main techniques for the representation of the outputs (labels): target indexes or hot-encoded labels. Both were explored to see how they could impact model performance, at least after running some initial training.

More details on results and chosen metrics will be given in Chapter 5, as well as the specific way in which this exploration phase was performed. However, after comparing results using the same data splitting techniques and work flow, the results obtained by using hot-encoded labels were marginally different. Ultimately, the target indexes technique was the chosen label representation strategy for the model.

4.2.5 Data splitting: training, validation and test set proportions

In the case of the Binary Classifier, a specific data splitting approach was used. The strategy was to assign 80% of the samples to the training set and the remaining 20% to the test set, both sets stratified. Furthermore, 25% of the initial 80% was set aside for the validation set while the other 75% was assigned to the final training set. Of course, both sets were stratified.

For the Binary Classifier the process described before produced a total of 1440 samples for training, 480 for validation and 480 for testing. In [1], the strategy was shown to be successful with a cross-validation (CV) accuracy of 95.052%. This number is the average of the validation accuracy values over the folds used (5 for that model).

However, for the multiclass model the situation required further consideration. This is because instead of having 1200 samples per class, the multiclass case has 200 samples for each class only. Consequently, the way in which data would be preprocessed and split would be very important.

Nonetheless, the 80/20 approach was used for the multiclass model as a starting point. This is because before developing the neural network with seven classes, different binary models were developed for each contaminant class. The 80/20 proportion was used because of the good results the Binary Classifier achieved. Results for the binary models were promising so it was decided to try using this strategy for the case of the multiclass model.

Also, a second reason for choosing the 80/20 approach as a starting point was that even though the number of samples was not very high, data quality was considered to be satisfactory for each class, independently. Moreover, as it will be shown immediately after this, the size of the training set was good enough to start, with 120 samples per class.

It's important to highlight that all sets were stratified when the data was separated into the three sets. Therefore, for the multiclass case, the following can be said:

- The total number of samples was 1400, 200 samples for each class
- Training set: 80% of 1400 = 1120 samples, 75% of 1120 = 840 samples, 120 samples per class
- Validation set: 80% of 1400 = 1120 samples, 25% of 1120 = 280 samples, 40 samples per class
- Test set: 20% of 1400 = 280 samples, 40 samples per class

Furthermore, there was still the need of having more training samples per class to better train the model. Although the validation and test sets only had 40 samples per class, due to the good quality of each sample more importance was given to the training phase. Potentially, performance could be improved and compared to already existing results.

In consequence, another strategy was developed and used in parallel during early development. Specifically, the second approach was the following, with the sets almost perfectly stratified:

- Total number of samples: 1400, 200 samples per class
- Training set: 85% of 1400 = 1190 samples, 75% of 1190 = 892.5 samples (rounded down to 892 by Python, so truly the 74.96%)
- Validation set: 85% of 1400 = 1190 samples, 25% of 1190 = 297.5 samples (rounded up to 298 by Python, so truly the 25.05%)
- Test set: 15% of 1400 = 210 samples, 30 samples per class

More details on results will be given in Chapter 5. At this point, it's important to emphasize that increasing the number of training samples would have not been very beneficial. Although a 90/10 proportion could potentially increase the training set to 945 samples using the strategy outlined before, the test set would only have 140 samples, with just 20 samples per class.

Such a test set was considered to be too small regardless of individual sample quality. As a result, in the early development stages, the 80/20 and 85/15 approaches were used to compare performance since the compromise between training and test set sizes was considered to be good enough.

Now, toward the end, when reaching the final model, a third decision was made. It's possible to note that by using the 85/15 strategy stratified sets cannot be produced even if the option to stratify is indicated when using Scikit-Learn to perform the splitting. This is because the sizes of the training and validation sets, after being calculated, are not whole numbers as shown before.

Consequently, a third strategy was developed. This last approach, and the 80/20 proportion, were used at the end to test the final optimized model and run error analysis. It was with this new strategy, which changed the proportion for the validation and test sets, that the three resulting sets could be perfectly stratified.

The third approach just described was the following:

- Total number of samples: 1400, 200 samples per class
- Training set: 85% of 1400 = 1190 samples, 80% of 1190 = 952 samples, 136 samples per class
- Validation set: 85% of 1400 = 1190 samples, 20% of 1190 = 238 samples, 34 samples per class
- Test set: 15% of 1400 = 210 samples, 30 samples per class

It should be noted that although the validation and test sets are not equal in size like for the 80/20 approach, their sizes are very similar. Also, using less than 20% leads to a validation set that has less than 200 samples. For this statement, only proportions that are multiples of five are being considered for simplicity.

Further details, like the percentages related to stratification, will be given later in Chapter 5. All in all, it's important to summarize that there were two strategies used during early development for data splitting. These were the first and the second one. Similarly, for the final model, two approaches were considered in parallel as well. However, these corresponded to the first and third strategies.

4.2.6 Proportions used for CV and Stratified CV

At the beginning, when the binary models were developed, manual cross-validation (using KFold by Scikit [24]) was used to verify the results as well as the cross_val_score function [14]. More details will be given later on in Chapter 5.

The important aspect to emphasize at this point is that for performing cross-validation the 80/20 approach was used to generate the training, validation and test sets. Furthermore, when performing this task, whether manually for more control or using the function just mentioned, only the training set was used with 5 folds.

This number was chosen over 3, which is one of the other typical values used for performing cross-validation. The reason for this was to try to maximize the amount of samples for training per fold. Although the validation set per fold was small, once again it should be noted that the sample data for each class was good.

In the case of the multiclass model later on, the option of using functions such as cross_val_score was discarded. This is because at this stage development was focused on finding a preliminary multiclass model which would be later optimized. Therefore, the cross-validation task now required a higher degree of control and precision as well as more options for parameter monitoring.

Performing manual cross-validation using KFold again seemed promising. However, Stratified CV using StratifiedKFold [16] was chosen later on instead to have stratified folds [3]. This way, sets would be more homogeneous and the overall process would be more carefully designed. This could potentially lead to better quality results. Furthermore, due to the reasons explained before when referring to the selection of the number of folds, the decision was to keep the total fold number as five.

Moreover, in the case of the binary models the metric reported after performing cross-validation using KFold was the average validation accuracy per fold. In the case of the multiclass model using Stratified CV, this was still done, in addition to reporting the peak validation accuracy per fold.

Afterwards, the five validation accuracy values were averaged, as well as the five peak validation accuracy results to compare different optimized versions of the multiclass case. This was done repeatedly throughout the optimization and selection processes. More specific details will be given in Chapter 5.

Finally, when performing cross-validation using StratifiedKFold all data splitting strategies were used. For the tuning steps described later in this chapter the data splitting strategies used were the first and the second ones described earlier. The aim was to have a better appreciation of the results, compare the performance with different data splitting approaches and identify possible model versions that might be overfitting the data and/or performing poorly.

However, toward the end a change was made that should be highlighted. When performing validation and error analysis for the final model after all the selected hyperparameters were optimized, the first and third strategies were used to have perfectly stratified folds and better quality results.

4.3 Multiclass model optimization and candidate search

This section aims to explain the overall process of how the final optimized model was developed. The steps taken will be now described. Furthermore, it should be noted that specific results will be given in Chapter 5.

[3]The specific moment when this was decided will be specified later in this chapter.

4.3.1 Preliminary phase: binary models for each type of contaminant

Initially, there was a phase where the work behind the Binary Classifier [1] was carefully studied. In particular, close attention was paid to the way samples were assigned to the two classes and the way in which data was preprocessed. Additionally, the steps taken to develop the Binary Classifier were also analyzed, as well as replicated since the original files were available.

Of course, it was not possible to obtain exactly the same results due to the stochastic nature of the entire process. However, it was possible to achieve a general understanding of how the quality assurance problem was approached. As a result, the idea of a multiclass model was proposed as mentioned in Chapter 3.

Therefore, after finishing this initial phase, this thesis advanced to the next stage which shall be referred to as the "preliminary phase". This preliminary phase consisted in developing six binary classifiers which were six DNNs. Each one of them had the task of identifying whether the sample belonged to the "free" class (non contaminated jar) or a specific contaminant class.

The purpose behind this was to understand to what extent each type of contaminant could be correctly distinguished from the free class given the available samples. Moreover, this way it would be possible, in part, to appreciate the sample quality better for each class. Here, "in part" is mentioned since results are also dependent on the architecture of the DNN, the training phase and several hyperparameters.

Also, another objective of this phase was to try to design an initial architecture which could be later used as a starting point for the multiclass model. This initial architecture was proposed by the project mentor at Columbia University based on past experience. It is shown in Figure **??**.

All six binary classifiers used the same architecture. Now, it's necessary to describe this DNN in more detail. In particular:

- Number of neurons in the input layer: 30, one per feature
- Number of hidden layers: 3
- Number of neurons in each hidden layer: 128
- Activation function of the hidden layers: Relu
- Number of neurons in the output layer: 1
- Activation function of the output layer: Sigmoid
- Optimizer used: Adam
- Learning rate used for the optimizer: Default value (0.001, as listed in [7])
- Loss function used: Binary Crossentropy
- Metric to monitor during training: Accuracy (both training and validation accuracy)

The specific optimizer was chosen above since the Binary Classifier used Adam. Due to this, it was considered as a good starting point for this preliminary analysis. For the very same reason, since this was an initial phase for exploration, the learning rate was not modified.

The results obtained after using the 80/20 data splitting strategy (as mentioned earlier) and manual KFold were good enough for the six models. These results will be presented later in Chapter 5. Consequently, this initial architecture was chosen to start developing a multiclass DNN.
Furthermore, for the multiclass DNN it was necessary to make the following changes:

- Number of neurons in the output layer: 7
- Activation function of the output layer: Softmax
- Loss function used: Sparse Categorical Crossentropy

The results obtained after training and performing an initial cross-validation procedure using KFold were high enough to consider optimizing this model. Now, the following subsections will describe, in a general way, the process of optimizing this initial model and selecting the best candidate.

Moreover, specific results and details will be given in Chapter 5. On the other hand, the various development versions will be described later in this chapter to give brief descriptions of the different versions and techniques used.

It should also be highlighted that before starting the hyperparameter tuning step, it was verified whether adding a min-max scaling step during preprocessing and using hot-encoded target labels achieved better results. As explained before, both of these options were discarded. Also, an initial, exploratory error analysis phase with a confusion matrix using the 80/20 approach was performed too.

4.3.2 First hyperparameter tuning step: optimizing the number of layers and neurons

The first parameters chosen for optimization were the number of hidden layers and the number of neurons per hidden layer. Originally, it was attempted to do this by using the function called RandomizedSearchCV from the model_selection module [4] developed by the Scikit-Learn team.

The reason for this was that since it was a DNN, there are typically many parameters to train. However, the results were, naturally, very random and unpredictable so it was not considered to be a very precise approach to this particular optimization process.

Furthermore, since the DNN itself was not a particularly big neural network, it was possible to consider a more exhaustive method. In consequence, at this point it was preferred to use instead the function called GridSearchCV [3], also developed by Scikit-Learn.

Several candidates were selected at this point (more details, including the search grids and how these were modified with every iteration, will be given in Chapter 5). The important aspect to highlight right now is that once the candidates were selected, these were subjected to a cross-validation process using StratifiedKFold (stratified cross-validation, with 5 folds).

For this, the training sets were generated using both the 80/20 approach and the first 85/15 data splitting approach described earlier (so the first and second strategies). The metrics reported after performing manual stratified cross-validation with StratifiedKFold were the average validation accuracy per fold and the highest peak validation accuracy per fold. Then, the results were averaged.

Once all the calculations were ready, the best candidate was to be chosen by looking at which model scored highest. Also, it was important to consider that a relatively small architecture would make hardware implementation easier later.

Fortunately, the best candidate was a simple model that would not be hard to implement. It was actually an MLP, not a DNN anymore. Specifically, it was an MLP with 1 hidden layer and 160 neurons. This version then continued on to the next step to keep tuning more hyperparameters.

4.3.3 Second optimization step: searching for the best optimizer and learning rate

As previously mentioned, this phase began with the following MLP:

- Number of neurons in the input layer: 30, one per feature
- Number of hidden layers: 1
- Number of neurons in each hidden layer: 160
- Activation function of the hidden layers: Relu
- Number of neurons in the output layer: 7
- Activation function of the output layer: Softmax
- Optimizer used: Adam
- Learning rate used for the optimizer: Default value (0.001, as listed in [7])
- Loss function used: Sparse Categorical Crossentropy
- Metric to monitor during training: Accuracy (both training and validation accuracy)

The next hyperparameters to optimize corresponded to the type of optimizer and the learning rate. Three options were considered for the optimizers, after checking [7]: Adam, SGD and Adagrad.

Afterward, another search process was performed using GridSearchCV, this time for each type of optimizer. Specifically, the architecture would be compiled using one optimizer and then GridSearchCV would be initialized. Then, once the results for that optimizer were ready, the architecture would be compiled again with the next candidate and the process would be repeated.

It's important to highlight at this point that, just like with the previous tuning, the process for one specific optimizer was iterative. The search would start with a grid of learning rates which would then be refined to search for better performing learning rates. However, more specific details will be given later in Chapter 5.

Another relevant aspect to explain now is the fact that to start GridSearchCV the model should be trained first. One can question at this point whether the results from the three different searches would be comparable then. The important detail is that when GridSearchCV starts, the model is retrained and validated using cross-validation several times (three with 3 folds and five with 5 folds).

Therefore, the aim is not to get the final, trained model but instead results that will potentially lead to a good model. This way, it's irrelevant whether the search for a good learning rate for each optimizer is done separately as described or using, for instance, a grid in a similar way to a dictionary. The fact that the model can be trained again before starting the search also becomes unimportant.

Finally, once the results were ready, the next step was once again about performing manual, stratified cross-validation. This was done in exactly the same way as before. In other words, StratifiedKFold (with 5 folds) was used to obtain the average validation accuracy and the average peak validation accuracy over the 5 folds.

Just like before, the aforementioned process was completed in parallel using two data splitting techniques. These were, again, the first (80/20) and the second one (first 85/15 approach). The only difference with the previous optimization step is that, instead of performing stratified-cross validation for every layer value - neurons pair, it was done for each optimizer - learning rate pair.

In the end, the candidate selected (the one that performed best) was the MLP with an Adagrad optimizer and a learning rate of 0.3. Now, the next hyperparameter that could be optimized was the activation function of the hidden layer. However, the Relu function was kept since it typically performs well for MLPs. Also, as it will be shown in the next chapter, the results obtained with the final model described in this paragraph, with the Relu activation function, were satisfactory.

4.3.4 Final optimized model: the 1 hidden layer MLP

This subsection aims to emphasize the specifications of the final, optimized model. The details are the following:

- Number of neurons in the input layer: 30, one per feature
- Number of hidden layers: 1
- Number of neurons in each hidden layer: 160
- Activation function of the hidden layer: Relu
- Number of neurons in the output layer: 7
- Activation function of the output layer: Softmax
- Optimizer used: Adagrad
- Learning rate used for the optimizer: 0.3
- Loss function used: Sparse Categorical Crossentropy
- Metric monitored during training: Accuracy (both training and validation accuracy)

Figure **??** shows the architecture associated to the model just described. The following subsection will discuss how the performance of this final model was evaluated.

4.3.5 Error analysis: methods chosen to evaluate the 1 hidden layer MLP

To evaluate the performance of the final model, two methods were chosen. The first approach consisted in using a test set to estimate the generalization error. This set was created using both the 80/20 and the second 85/15 data splitting strategies described earlier in this chapter to be able to compare results.

The second technique used to evaluate the MLP was the confusion matrix. In particular, since this is a classification problem with seven target classes, the confusion matrix is a 7x7 one where the rows are the actual classes and the columns the predicted ones. Furthermore, it should be noted that a confusion matrix is created based on a particular set.

As a result, in order to be meticulous and achieve good results, six confusion matrices were created. The first three were based on the training, validation and test sets obtained after training the MLP using the 80/20 data splitting approach. It's essential to highlight at this point that the trained model was saved as an object that could later be retrieved.

In fact, to create the remaining three confusion matrices, the model was loaded again. Next, the matrices were created using the training, validation and test sets obtained after using the second 85/15 splitting approach described before in this chapter. However, evidently it was necessary to make these matrices with the final, trained model version to make results comparable.

Then, all six matrices were analyzed and collapsed to the 2x2 binary case (contaminant or no contaminant) to be able to compare the obtained results with those obtained using the Binary Classifier [1]. Additional metrics were calculated, such as precision and recall. All important details and more specific information will be given in Chapter 5, such as the normalized versions of the matrices.

Finally, it's worth mentioning that some preliminary error analysis using confusion matrices was also performed for the initial multiclass DNN, with 3 layers and 128 neurons per layer. The results looked quite good so this was another reason to continue developing the multiclass model. More details will be given later in Chapter 5.

4.4 Summary

Figure **??** summarizes the steps described in this chapter for the optimization process, as well as some preliminary steps such as the exploration of min-max scaling and hot-encoding techniques. It's been put here to better visualize the entire process and aid understanding.

4.5 Python based models: the Jupyter notebooks

The particular environment used to develop code was specified earlier in this chapter. It's relevant to highlight, however, that the various activities described so far were completed using Jupyter notebooks, which use the file format .ipynb.

Next, the following two subsections will clarify what the different notebooks focused on and the specific techniques used. Further details will be given in Chapter 5.

4.5.1 Notebooks created during the preliminary phase

Table 4.5.1.1 shows the notebooks developed for the preliminary phase, when the six binary classifiers for each contaminant type were developed. Just as a reminder, the architecture that was tested at this point was the 3 hidden layer DNN, with 128 neurons in each hidden layer.

Table 4.5.1.1: Notebooks developed for training the six binary classifiers

Contaminant	Model under dev.	Splitting app.	Techniques
Cap shaped plastic	Binary classifier	80/20	3 sets cross_val_score "score" method Manual KFold
Glass fragment	Binary classifier	80/20	3 sets "score" method Manual KFold
Metal sphere	Binary classifier	80/20	3 sets "score" method Manual KFold
Pink sphere	Binary classifier	80/20	3 sets "score" method Manual KFold
Plastic sphere	Binary classifier	80/20	3 sets "score" method Manual KFold
Triangular plastic	Binary classifier	80/20	3 sets "score" method Manual KFold

Notes:

- "Model under dev." and "Splitting app." in Table 4.5.1.1 refer to "Model under development" and "Splitting approach" respectively.
- Samples were preprocessed using the standardization technique only (Standard-Scaler) as explained before.
- The term "binary classifier" in Table 4.5.1.1 should not be confused with the Binary Classifier model developed by [1].
- The "3 sets" term refers to the training, validation and test sets. To evaluate performance on the test set, the "evaluate" method from the Keras API (Model Class) was used [19].
- cross_val_score was performed to get preliminary estimations, with 5 folds (parameter "cv").

- Manual cross-validation using KFold was performed with 5 folds after calling the cross_val_score helper function.
  - The values reported at the end of the process were the five average values for the average validation accuracy per fold.
- The "score" method is a method developed for a Scikit estimator object. The score method returns a number which corresponds to the value of a default evaluation metric; cross_val_score uses that default metric.
  - In this case, it's the accuracy (training accuracy) as explained in [15], [25].
- The names of the files of each notebook are:
  - Cap shaped plastic: "Air surface contaminant model.ipynb"
  - Glass fragment: "Glass fragment contaminant model.ipynb"
  - Metal sphere: "Metal sphere contaminant model.ipynb"
  - Pink plastic sphere: "Pink sphere contaminant model.ipynb"
  - Plastic sphere (small): "Plastic sphere contaminant model.ipynb"
  - Triangular plastic: "Triangular plastic contaminant model.ipynb"

4.5.2 Notebooks developed for the multiclass model

Tables 4.5.2.1 and 4.5.2.2 show all the notebooks developed in relation to the multiclass model.

Table 4.5.2.1: Notebooks created for the multiclass model (Part I)

Version	Task/goal	Splitting app.	Techniques
0 (initial 3 hidden layer DNN, 128 neurons)	Preliminary exploration	80/20	3 sets cross_val_score "score" method Manual KFold
A (initial 3 hidden layer DNN, 128 neurons)	Focus on the 3 sets strategy	80/20	3 sets
A_2 (initial 3 hidden layer DNN, 128 neurons)	Study effect of min-max scaling + 3 sets	80/20	3 sets MinMaxScaler
B (initial 3 hidden layer DNN, 128 neurons)	Focus on the manual CV strategy	80/20	Manual KFold
B_2 (initial 3 hidden layer DNN, 128 neurons)	Study effect of min-max scaling + manual CV	80/20	MinMaxScaler Manual KFold

Table 4.5.2.2: Notebooks created for the multiclass model (Part II)

Version	Task/goal	Splitting app.	Techniques
C (initial 3 hidden layer DNN, 128 neurons)	Version A + hot-encoded target labels	80/20	3 sets hot-encoded labels
D (initial 3 hidden layer DNN, 128 neurons)	Version B + hot-encoded target labels	80/20	hot-encoded labels Manual KFold Stratified KFold
E (initial 3 hidden layer DNN, 128 neurons)	Version A + preliminary error analysis	80/20	3 sets cross_val _predict confusion matrix + normalized version
F (start: initial 3 hidden layer DNN, 128 neurons)	Version A + first tuning step (layers + neurons)	80/20	3 sets RandomizedSearchCV GridSearchCV
G (start: initial 3 hidden layer DNN, 128 neurons)	Version B + first tuning step (layers + neurons)	80/20 85/15 (1st version)	Manual KFold GridSearchCV
H	Explore 1 layer with: 136, 134, 128, 150, 160 neur.	80/20	StratifiedKFold
I	Explore 1 layer with: 150, 160 142, 136 + 160 (test set) + optimizers (both sets)	85/15 (1st version)	StratifiedKFold
J (start: 1 hidden layer, 160 neur., models: J, J2, J3, J4, J5)	Estimate gen. error (J, J2) + second tuning step (J3, J4, J5)	85/15 (J) (first version) 80/20 (J2-J5)	3 sets (J, J2) GridSearchCV (J3-J5)
K (1 hidden layer, 160 neur.)	Explore optimizers with different learning rates (both sets)	80/20	StratifiedKFold
L (final model: 1 hidden layer, 160 neur., Adagrad opt, learning rate: 0.3)	Evaluate model performance (Final version: L3)	80/20	3 sets "predict" method confusion matrix + normalized version
M (final model: 1 hidden layer, 160 neur., Adagrad opt, learning rate: 0.3)	Evaluate model performance (Final version: L3, LOADED)	85/15 (2nd version)	3 sets "predict" method confusion matrix + normalized version

Notes:

- In Table 4.5.2.2, "neur." refers to "neurons", "gen." to "generalization" and "opt" to "optimizer".
- Just like the notebooks developed for the preliminary phase, the preprocessing stage includes standardization only except for versions A_2 and B_2.
- All files were named as: "Full multiclass contaminant model VER", followed by the version (the letter indicated in Tables 4.5.2.1 and 4.5.2.2).
- All notebooks except for Version L and M were designed to obtain exploratory/preliminary results. Therefore, all model versions were saved but only version L3 was loaded into notebook M to obtain more results about the final model.
- In Tables 4.5.2.1 and 4.5.2.2, the term "3 sets" is used exactly in the same way as before and also includes the usage of the "evaluate" method [19].
- When using KFold, StratifiedKFold and GridSearchCV the number of folds was set to five (parameter "cv" = 5; only the first runs of GridSearchCV used "cv" = 3).
  - However, when using (only once) RandomizedSearchCV, this parameter was set to 3. This is because this method was exploratory only and was quickly discarded.
  - When calling GridSearchCV, an object is created. Later, to obtain the results, one should use the best_params_and best_score_attributes of the object [3].
- Version 0:
  - Manual cross-validation was performed, using 5 folds.
  - Only the average validation accuracy per fold was calculated at this stage.
- Version C:
  - To transform the labels into hot-encoded labels, the keras.utils.to_categorical function was used as described in [2].
  - Also, it was necessary to change the loss function from "Sparse Categorical Crossentropy" to "Categorical Crossentropy".
- Version D:
  - It was also required to use the loss function: "Categorical Crossentropy".
  - Stratified KFold was explored in this notebook for the first time. Both average validation accuracy and peak validation accuracy per fold were obtained.
- Version G:
  - The first 85/15 approach was used here for the first time to try and expand the training set a bit more. All sets were stratified.
- Version H:
  - Two results were obtained when performing Stratified CV: the peak validation accuracy and the average validation accuracy per fold, just like when this method was first explored.

- Version I:
  - Sets were perfectly stratified as well.
  - The Stratified CV process would return the peak validation accuracy and the average validation accuracy per fold as well.
  - When performing cross-validation for the optimizers, the following combinations were considered:
    - SDG with learning rate = 0.1.
    - Adagrad with learning rate = 0.3.
    - Adam with learning rate = 0.03.
    - Adam with learning rate 0.003.
  - For the optimizers, Stratified CV was performed on both the training and the test sets as suggested by the mentor at Columbia University (Table 4.5.2.2).
  - In addition, after the results from the Stratified CV process were ready, the five averages and five peak accuracy values were averaged.
    - In consequence, it was possible to obtain one value for the average validation accuracy and the peak validation accuracy.
    - More details will be given in Chapter 5.
- Version L:
  - Version L contains three versions: L, L2, L3.
  - L was the first one and used the cross_val_predict function to obtain the predicted labels for the confusion matrix [20].
    - However, it was discarded for not being considered accurate enough since predictions are generated using cross-validation. Therefore, the model is retrained multiple times and the original one is lost.
  - L2 and L3 use the predict method from the Keras API [19], which is accurate.
  - L3 is the final model of this project.

As it's possible to appreciate, more techniques were added as development progressed. This, evidently, made the entire process more complex. It should be noted, however, that in general the methods for evaluating candidate models consisted in either using the three sets approach or cross-validation (CV).

It was for the final model, L3, that the confusion matrix was used to perform error analysis. An estimation of the generalization error was also calculated using the three sets approach.

Finally, the next section will give some further details of interest. These include the file formats used to save the model versions, the Keras wrapper for classifiers and the epochs used during training.

4.6 Additional details

4.6.1 File formats

Some more information that is relevant to consider relates to the file formats that were chosen, in general, to save the model versions. The types of files chosen include:

- .json: to save the architecture of the model.
- .h5: to save the entire model (architecture and weights).
- .h5: there was another .h5 file saved per model to save the weights only.

It should be noted that most notebooks, except for versions L and M, were exploratory. However, versions L and M focused on the final model and aimed to be precise and meticulous. For this reason, four files for this model were saved:

- A .json file for the architecture; this file was saved after completing training using the 80/20 data splitting approach.
- A .h5 file for the architecture and the weights; this .h5 was saved after completing training using the 80/20 approach.
- A .h5 file for the weights only, saved during training using a callback that was monitoring the parameter "val accuracy" and using the 80/20 technique.
  - This callback saved the best weights, in particular those that helped the model reach the highest validation accuracy during training.
- A .h5 file for the weights only, saved after training was finished using the 80/20 strategy.
  - Since rigorousness was needed for the final model, this fourth file was saved only for model L3 which was selected to continue to the hardware implementation phase.

It should be noted that in early notebooks related to Version B in Table 4.5.2.1 the models were not saved in .json files. Instead, the models were saved using .h5 after training and the weights separately with a callback (in .5). The same applies for the preliminary binary classifiers.

On the other hand, in early notebooks related to Version A the multiclass models were saved using the .json and .h5 formats after training. Also, the .h5 file was saved for the weights of these models using the checkpoint callback.

Furthermore, beginning notebook E the checkpoint callback, which is the callback used to save the best weights during training, was modified. Instead of monitoring the "accuracy" parameter, it was changed so that the "val accuracy" parameter would be monitored instead to save weights according to this metric.

4.6.2 Callbacks used

During the training phases of the models, including the preliminary binary classifiers, two main callbacks were used and a third one was only initialized. These callbacks were [26]:

- ModelCheckpoint: used to monitor a specific parameter and save the best weights according to that metric during training.
- Tensorboard: used to accumulate data related to the training phase of each model, including learning curves.
- EarlyStopping: initialized with parameters "patience" = 20 and "restore_best_weights" = True.
  - Since the number of epochs was relatively small, the number of samples was not extensive and the models were relatively simple, this callback was not used.
  - It was left however in the case of being required for future development.

4.6.3 Wrapper

To combine a Keras based neural network with Scikit-Learn, it's necessary to use a wrapper. The goal is to be able to interface the Keras model with Scikit methods [2]. Specifically, there are two different types of wrappers, one for regressors and one for classifiers.

In this project, the second was used: keras.wrappers.scikit_learn.KerasClassifier [27]. The KerasClassifier takes as input a function (that returns a model) and additional parameters, such as the number of epochs. Once called, the function returns an object to which Scikit methods can be applied to with no problems.

4.6.4 Epochs, dropout rate and batch size

Just to elaborate a bit more on this topic, for the case of the preliminary binary classifiers, 100 epochs were used for training. For the multiclass case, due to its higher complexity, 100 epochs were shown to be insufficient while developing the first version. Therefore, the number was increased to 200.

Tuning the epochs was briefly considered, as well as the dropout rate and the batch size. However, for the first case it was not really necessary since 200 was working well and the amount of samples was not excessive. Similarly, for the batch size, it was left as 32 (default) [19] for the same reasons. Moreover, in the case of the dropout rate, due to the relatively simple architecture of the neural networks it was suggested by the mentor at Columbia University to keep the layers fully connected.

4.7 Hardware implementation phase

Once the final Machine Learning model was selected, the hardware implementation phase of the project was initiated. This phase followed the hardware workflow described in Chapter 2. The first aspect to describe, then, relates to the models explored during the first part of the flow, which used hls4ml and Vivado HLS.

During this exploratory stage, certain values were kept constant for all models while others were changed. In particular, exploration consisted in changing the target clock, the precision, and board used (both a ZedBoard and a ZCU106 were considered).

Further details such as the chosen architecture, the strategy used, or the resource factors considered will be detailed in Chapter 6. Now, the models explored will be indicated, just like for the case of the Jupyter Notebooks.

4.7.1 Models developed during HLS

Table 4.7.1.1 shows the characteristics of the hardware models developed using HLS. In total, these were 29. Different data precision, target clock and board options were considered.

Table 4.7.1.1: Hardware models developed during HLS

Version	Board	Clock (ns)	Precision
1	ZedBoard	25	ap_fixed<16,8>
2	ZedBoard	20	ap_fixed<16,8>
3	ZedBoard	15	ap_fixed<16,8>
4	ZedBoard	10	ap_fixed<16,8>
5	ZedBoard	5	ap_fixed<16,8>
6	ZedBoard	3	ap_fixed<16,8>
7	ZedBoard	4	ap_fixed<16,8>
8	ZedBoard	20	ap_fixed<30,2>
9	ZCU106	20	ap_fixed<30,2>
10	ZCU106	15	ap_fixed<30,2>
11	ZCU106	10	ap_fixed<30,2>
12	ZCU106	5	ap_fixed<30,2>
13	ZCU106	4	ap_fixed<30,2>
14	ZCU106	3.50	ap_fixed<30,2>
15	ZCU106	3	ap_fixed<30,2>
16	ZCU106	2.50	ap_fixed<30,2>
17	ZCU106	2	ap_fixed<30,2>
18	ZCU106	20	ap_fixed<64,32>
19	ZedBoard	20	ap_fixed<64,32>
20	ZCU106	25	ap_fixed<64,2>
21	ZCU106	20	ap_fixed<32,2>
22	ZCU106	15	ap_fixed<32,2>
23	ZCU106	10	ap_fixed<32,2>
24	ZCU106	5	ap_fixed<32,2>
25	ZCU106	3	ap_fixed<32,2>
26	ZCU106	2	ap_fixed<32,2>
27	ZCU106	1	ap_fixed<32,2>
28	ZCU106	1.50	ap_fixed<32,2>
29	ZCU106	1.50	ap_fixed<30,2>

Chapter 5

Machine Learning results

In this chapter, the results of the project related to Machine Learning will be presented. These results will include the ones from the preliminary phase (where the six binary classifiers were trained, validated and tested) and from the multiclass development phase.

However, before presenting the results, there is an aspect that should be highlighted since it is relevant for hardware development (Chapter 6). This aspect is that the original data, from which results were obtained, was type "float64" for the samples and type "int64" for the labels.

5.1 Preliminary phase

The results obtained for the six binary classifiers developed are presented next in Table 5.1.1 and Table 5.1.2. The contaminants obviously refer to the ones indicated in Chapter 4.

Table 5.1.1: Results obtained for the six binary classifiers (Part I)

Result type	Pink plastic sphere	Cap shaped plastic	Glass fragment
Best test loss	0.001226	0.000557	0.215362
Best test accuracy	1.000000	1.000000	0.950000
Cross_val_score (all scores, one for every fold)	1.000000 0.968750 1.000000 1.000000 0.984375	1.000000 1.000000 1.000000 1.000000 1.000000	1.000000 0.937500 0.984375 1.000000 1.000000
.score (method)	0.987500	1.000000	0.962500
Manual KFold (average validation accuracy, one value for every fold)	0.993594 0.996719 0.995156 0.983906 0.981563	0.979688 0.995000 0.996719 0.997188 0.997188	0.957813 0.979219 0.961094 0.977656 0.986250

Table 5.1.2: Results obtained for the six binary classifiers (Part II)

Result type	Metal sphere	Small plastic sphere	Triangular plastic
Best test loss	0.281187	0.276277	0.080144
Best test accuracy	0.912500	0.962500	0.962500
Cross_val_score (all scores, one for every fold)	0.984375 0.953125 1.000000 1.000000 0.968750	0.984375 0.968750 0.984375 1.000000 1.000000	0.968750 0.968750 0.968750 0.937500 1.000000
.score (method)	0.925000	0.950000	1.000000
Manual KFold (average validation accuracy, one value for every fold)	0.990000 0.961719 0.972344 0.979531 0.979063	0.970469 0.995938 0.987031 0.995313 0.960469	0.959531 0.973750 0.956250 0.925469 0.940156

Notes:

- Results in Table 5.1.1 and Table 5.1.2 were rounded up to six decimal positions.
- The "best test loss" and "best test accuracy" results were produced by applying the "evaluate" method from the Keras API to each model, which took as input the test set.
  - These results are returned by the .evaluate method and correspond to the loss value and the metrics for a model in testing mode [19].
- The results coming from cross_val_score were obtained by applying the function While taking the training sets as inputs. The scoring metric chosen was "accuracy" (training accuracy).
- The .score method was applied to the model, which took as input the test sets. The metric evaluated was "accuracy".

Next, Table 5.1.3 shows the average values for the cross_val_score and Manual KFold (manual, non stratified, cross-validation) results. The notes related to it are presented on the next page.

Table 5.1.3: Average values obtained for cross_val_score and cross-validation with KFold in the preliminary phase

Type of contaminant	Average value for Cross_val_score	Average value for Manual KFold
Pink plastic sphere	0.990625	0.990188
Cap shaped plastic	1.000000	0.993156
Glass fragment	0.984375	0.972406
Metal sphere	0.981250	0.976531
Small plastic sphere	0.987500	0.981844
Triangular plastic	0.968750	0.951031

Notes:

- Like before, results in Table 5.1.3 were rounded up to six decimal positions.

In Table 5.1.3, it's possible to see that all values are above 0.95. The lowest value for the manual cross-validation is 0.951031 (the triangular plastic) while the highest value is 0.993156, which corresponds to the cap shaped plastic. In the case of the results for cross_val_score, the lowest is 0.981250 (metal sphere) while the highest value corresponds to 1.000000 (the cap shaped plastic).

Due to this last value, it might be possible to speculate about a possible situation of overfitting in the case of the cap shaped plastic. However, overall, since all average values are above 0.90 in the 0.95-0.99 range, it can be said that these preliminary results are satisfactory. Specifically, they are because they seem to show that there is the potential for identifying each contaminant type by separate to a good extent.

Furthermore, results in Table 5.1.1 and Table 5.1.2 are quite good as well. In the case of Table 5.1.1, the cross_val_score results are all above 0.90. Specifically, the lowest value is 0.937500 (glass fragment), and the highest one is 1.000000 again. The pink plastic sphere and the glass fragment have three folds that returned a value of 1.000000. Moreover, all five scores for the cap shaped plastic have a value of 1.000000. As mentioned before, this behavior seems to indicate a case of overfitting for this contaminant. For the others, since there are folds that scored a bit lower, it would be necessary to investigate further.

Manual KFold values appear to be very good, with all values above 0.95. The minimum is 0.957813 for the glass fragment, and the highest is 0.997188 for the cap shaped plastic. Similarly, best_test_accuracy and .score results are high as well. In the first case, the lowest is 0.950000 (glass fragment) with a value of 1.000000 for the other two contaminants. For the second case, the lowest is 0.962500 (glass fragment once again) and the highest is 1.000000 for the cap shaped plastic.

The best_test_loss results are quite low, which is a good indication. The lowest value is 0.000557 (cap shaped plastic) and the highest is 0.215362, for the glass fragment. This makes sense, due to the fact that the glass fragment seems a bit weaker in comparison to the other two contaminants regarding the other types of results. On the contrary, the cap shaped plastic has the best results, including the lowest (best) loss, which, as it has been said, all indicate a very likely case of overfitting.

Actually, if one goes back to Table 5.1.3, it's possible to see that among these three contaminants the glass fragment has the lowest averages while the cap shaped plastic has the highest ones. In the case of the glass, the average cross_val_score result is 0.984375 and the average CV is 0.972406. For the cap shaped plastic, these values are 1.000000 and 0.993156 respectively. Now, it would be interesting to analyze what happened in the case of the three remaining contaminants.

In the case of Table 5.1.2 now, the lowest value reported for cross_val_score is 0.937500 (triangular plastic). The highest one is 1.000000, which appears twice in the columns belonging to the metal and plastic spheres but only one in the case of the triangular plastic. In the case of the cross-validation results, the lowest value is 0.940156 (triangular plastic) while the highest is 0.995938, for the small plastic sphere.

The best_test_accuracy results in this case have a minimum value of 0.912500 (metal sphere) and a maximum value of 0.962500 (the other two contaminants). It would appear that the other three contaminants from Table 5.1.1 scored higher, with two values of 1.000000 out of three in total. Moreover, the .score results have a minimum value of 0.925000 (metal sphere) and a maximum of 1.000000 (triangular plastic) which would signify a perfect accuracy on the test set. These types of results in the case of Table 5.1.1 had the same maximum but a higher minimum.

The last type of result in Table5.1.2 is the best_test_loss. The lowest value is 0.080144 (triangular plastic) while the highest is 0.281187 (metal sphere). Since these results are also obtained using the test set, it's reasonable for the triangular plastic to have the minimum loss and the best accuracy. Also, it appears like the loss values are higher than the ones from Table 5.1.1 whose minimum was 0.000557 (cap shaped plastic) and the highest was for the glass fragment (0.215362), which also had the lowest test accuracy (0.950000).

Returning now to Table 5.1.3, it can be seen that out of these last three contaminants, the triangular plastic scores the lowest for both the average cross_val_score and manual cross-validation results. These values are 0.968750 and 0.951031 respectively. The contaminant that scored highest is the small plastic sphere, with an average cross_val_score result of 0.987500 and average CV of 0.981844.

Overall, considering again Table 5.1.3, the highest averages belong to the cap shaped plastic while the lowest to the triangular plastic. It's particularly important to analyze the cross_val_score and manual cross-validation results since they work with the technique of folds. The second one is very relevant to analyze since it allows for more control and is able to return the validation accuracy scores instead of just accuracy like the first one.

Finally, it can be said that, in general, the results obtained for the preliminary phase were good. In particular, cross_val_score and manual cross-validation results using KFold are all above 0.90 (or 90%). Consequently, it can be said that these preliminary results obtained indicate that each contaminant type could be identified fairly well by separate. This is, and was at the time, a satisfactory conclusion to start with the multiclass model.

5.2 Multiclass model development

Results in this section will be presented according to the notebook versions described in Table 4.5.2.1 and Table 4.5.2.2 (Chapter 4). As always, results in the next subsections were rounded up to six decimal places.

5.2.1 Version 0

Results for the first exploratory version of the multiclass model are presented in Table 5.2.1.1. Since the goal was to explore the multiclass approach for the first time, the same result types from before were considered. Also, as a reminder, it should be highlighted that this exploratory model used the 3 hidden layer architecture, where each layer had 128 neurons.

Table 5.2.1.1: Results obtained for Version 0

Result type	Value
Best_test_loss	0.536332
Best_test_accuracy	0.857143
Cross_val_score (all scores, one for every fold)	0.875000 0.906250 0.901786 0.861607 0.866071
.score (method)	0.882143
Manual KFold (average validation accuracy, one value for every fold)	0.833661 0.884487 0.897500 0.872388 0.883348
Cross_val_score (average)	0.882143
Manual KFold (average)	0.874277

It's immediately evident that increasing the model complexity decreased performance. As illustrated in Table 5.2.1.1, the average cross_val_score and average cross-validation results are 0.882143 and 0.874277 respectively. These values are both below 0.90, in contrast to the the six binary classifiers developed at the beginning.

Also, it's possible to see that no fold achieved an average validation accuracy higher than 90%: the highest value is 0.897500 while the lowest is 0.833661. The values reported per fold for the case of the cross_val_score results are slightly better: the highest value is 0.906250 while the lowest is 0.861607.

Similarly, the .score result in Table 5.2.1.1 is also below 90%, with a value of 0.882143. The best_test_accuracy value presents the same problem, with a value of 0.857143. Furthermore, the best_test_loss result is 0.536332, which is higher than the results obtained for the six binary classifiers.

It's at this point where a higher model complexity with a reduced number of samples became an important limitation. However, since results in Table 5.2.1.1 were close to the 90% range, it was decided to explore techniques that might improve performance to then focus on hyperparameter tuning.

5.2.2 Version A

It was first decided to explore the way in which the models were evaluated. The first step was to divide the evaluation methods to have two techniques (and notebook templates) running in parallel. Version A focuses on the 3 set approach and the results obtained can be found in Table 5.2.2.1.

Table 5.2.2.1: Results obtained for Version A

Result type	Value
Best_test_loss	0.687645
Best_test_accuracy	0.878571

As it's possible to see in Table 5.2.2.1, with respect to Version 0 the best_test_loss result increased to 0.687645 from 0.536332. Similarly, the best_test_accuracy increased to 0.878571 from 0.857143. Contrary to the first result type, this change can be considered positive.

It should be noted that Version A was trained again, so both models (Version 0 and Version A) are not exactly the same. However, the goal at this point was not to select a final model but rather to explore to obtain initial results. In particular, the objective for Version A was to focus on the 3 set approach as mentioned earlier, to later keep developing and improving this idea. Nevertheless, both models were saved.

5.2.3 Version A_2

Results for Version A_2, which used the min-max scaling technique and the 3 sets approach, are shown in Table 5.2.3.1.

Table 5.2.3.1: Results obtained for Version A_2

Result type	Value
Best_test_loss	6.988298
Best_test_accuracy	0.678571

As it can be seen in Table 5.2.3.1, the best_test_accuracy value decreased to 0.678571. This value is notably different from the result from Version A (0.878571) in Table 5.2.2.1 and Version 0 (0.857143) in Table 5.2.1.1. On the other hand, these two were different but very similar.

The best_test_loss result is 6.988298 which is very strange. The entire process was verified and run again but the obtained result was similar. Therefore, it was presented here to report it regardless of its value.

It can be said that, from these exploratory results, min-max scaling seemed to decrease performance for this project. This will be more evident once the results from cross-validation techniques are presented in the following subsections.

5.2.4 Version B

This version focused on using cross-validation to evaluate model performance and was run in parallel to Version A. The results obtained are shown in Table 5.2.4.1.

Table 5.2.4.1: Results obtained for Version B

Result type	Value
Manual KFold (average validation accuracy, one value for every fold)	0.850670 0.899085 0.895804 0.886049 0.871473
Manual KFold (average)	0.880616

Again, like in the case of Version 0, no fold achieved an average validation accuracy higher than 90%. For Version B, the highest value achieved was 0.899085, a bit higher than the one achieved by Version 0 (0.897500).

Similarly, for Version 0 the average validation accuracy was 0.874277 while for Version B this result was 0.880616. As it can be seen, even though the two models were different because they were trained separately (and, in any case, CV retrains a model), the results are very similar. Both models were saved.

5.2.5 Version B_2

This version combined the cross-validation technique from Version B in addition to min-max scaling. It was run in parallel to Version A_2. The results are shown in Table 5.2.5.1.

Table 5.2.5.1: Results obtained for Version B_2

Result type	Value
Manual KFold (average validation accuracy, one value for every fold)	0.802835 0.817946 0.823237 0.831719 0.823527
Manual KFold (average)	0.819853

In comparison to Version B, which achieved an average validation accuracy of 0.880616, Version B_2 achieved an average validation accuracy of 0.819853. This means performance decreased by 6.07%. The scores per fold, as shown in Table 5.2.5.1, are also lower, with the highest being 0.831719 and the lowest being 0.802835 for Version B_2.

From these results it's evident how min-max scaling does not improve performance for the multiclass case. Quite similarly, when comparing versions A and A_2, even though the two were trained separately, there was also quite a notable drop when it came to the accuracy results. In contrast, the accuracy results for Version A and Version 0 were similar.

Cross-validation, which retrains a model several times to apply it to the different fold sets, is useful in this case for two reasons. First, due to how it's performed, it can be used to observe that min-max scaling leads to worse performing behavior or a worse result trend. Secondly, it helps as evidence to make a decision, which is to discard min-max scaling for the project.

The reason why min-max scaling in this case might not be useful could be investigated further in the future. However, the reason why is likely due to the fact that, after observing the data, the new features among samples were all very similar. Therefore, for the given architecture, most likely more samples would be needed to better train the model.

5.2.6 Version C

The next technique that was explored was hot-encoding the target labels. Version C used the three sets approach in addition to the previously mentioned technique. Table 5.2.6.1 shows the obtained results.

Table 5.2.6.1: Results obtained for Version C

Result type	Value
Best_test_loss	0.851629
Best_test_accuracy	0.867857

It's important to compare these results to the ones obtained using Version A, which is the reference. For Version C, the best_test_accuracy result was 0.867857, as shown in Table 5.2.6.1. In comparison, the value for this result type was slightly higher for Version A: 0.878571 (Table 5.2.2.1).

Regarding the best_test_loss result, Version C had a result of 0.851629. For this same metric, with Version A a value of 0.687645 was obtained. Since this is a loss metric, evidently Version A performed better.

Even though results seemed to indicate that hot-encoded labels were not a good strategy to follow, the technique was not discarded at this point. The behavior trend was not very clear since the models were trained separately and the extent to which hot-encoding could degrade performance was not evident. For this reason, Version D was developed.

5.2.7 Version D

Version D combined manual cross-validation with the technique of hot-encoded labels. Stratified cross-validation was also explored. The results obtained with this model version are illustrated in Table 5.2.7.1.

Table 5.2.7.1: Results obtained for Version D

Result type	Value
Manual KFold (average validation accuracy, one value for every fold)	0.858058 0.888371 0.887969 0.895915 0.877054
Manual StratifiedKFold (average validation accuracy, one value for every fold)	0.869598 0.875067 0.864152 0.895491 0.888371
Manual StratifiedKFold (maximum validation accuracy, one value for every fold)	0.897321 0.901786 0.879464 0.915179 0.910714
Manual KFold (average)	0.881473
StratifiedKFold (average val. acc. and max. val.)	0.878536 0.900893

In Table 5.2.7.1, it should be explained that "val. acc" refers to "validation accuracy" and "max. val." to "maximum validation" (accuracy). As it can be observed, like always, for KFold the average validation accuracy values per fold and total average were calculated.

StratifiedKFold was run afterwards and a new result type was calculated, which was the maximum validation accuracy per fold. This was done at this point to introduce a new technique and therefore for exploratory purposes only.

However, StratifiedKFold was presented here since these results seemed quite good as it will be highlighted next. Moreover, because of this initial behavior and the use of stratified folds, this technique was used for the rest of the project. Peak validation accuracy will be reported later on in the following model versions as well.

Continuing now, it can be observed that the average validation accuracy using KFold was 0.881473. For StratifiedKFold, this result was 0.878536 (just slightly lower), and the average maximum validation accuracy using the same technique was 0.900893.

In Table 5.2.7.1, it can also be observed that no fold achieved an average validation accuracy higher than 0.90. Values that fell in this range belonged to the maximum validation accuracy result type, with 0.915179 being the best result.

Furthermore, Version B (Table 5.2.4.1) performed slightly worse when using the cross-validation technique with no hot-encoded labels. The result using manual KFold was 0.880616 on average instead of 0.881473 (Version D). However, both are practically the same.

Moreover, when comparing the results obtained using Version D to the ones obtained with Version B in Table 5.2.4.1 there's another important aspect to notice. The range for the average validation accuracy metric is more or less the same for both versions. However, for Version B the values per fold are closer to 90% than for Version D.

In particular, for Version B, there are two 0.89 values, one 0.88, one 0.87 and one 0.85. For Version D, in Table 5.2.7.1, there was one 0.89 value, two 0.88, one 0.87 and one 0.85. Now, if this analysis is considered instead, it would appear like Version B has a bit more potential.

Still, just as concluded for the comparison between Version A and C, performance seemed very similar between Version B and D. The differences in the results obtained were really marginal. Using cross-validation to compare Version B and D, which used hot-encoded labels, seemed to confirm the tentative conclusions from earlier to some extent.

Overall, the trends obtained seemed to not be very conclusive. Nevertheless, after analyzing the results for cross-validation per fold in detail, it was decided to keep the target indexes strategy instead of using hot-encoded labels. Ultimately, future versions of this project could use hot-encoded targets instead.

5.2.8 Version E

Version E consisted in a quick study of the initial architecture with 3 hidden layers and 128 neurons per layer. It was a preliminary analysis to observe how the architecture was (tentatively) behaving during error analysis.

It should be noted that the model was trained (and saved separately) again. This was not considered a limitation since the objective was to observe tentative results with respect to error metrics.

Next, Table 5.2.8.1 shows the results obtained after training when using the 3 sets approach.

Table 5.2.8.1: Results obtained for Version E

Result type	Value
Best_test_loss	0.622993
Best_test_accuracy	0.882143

As it can be seen in Table 5.2.8.1, the best_test_accuracy result was 0.882143. On the other hand, the best_test_loss result was 0.622993. Version A, which used the same approach, obtained similar results as shown in Table 5.2.2.1.

The new technique introduced at this point consisted in the confusion matrix. The values obtained for it were calculated using cross_val_predict, which is suitable for exploratory purposes.

It's important to remember at this point that the confusion matrix used by Scikit-Learn is transposed. This means that, contrary to confusion matrices found in theory texts, the matrix used by Scikit uses the rows as the actual classes and the columns as the predicted ones [10], [9].

The confusion matrix obtained for this version was:

$$\begin{bmatrix} 112 & 0 & 0 & 3 & 0 & 0 & 5 \\ 0 & 112 & 2 & 0 & 0 & 1 & 5 \\ 0 & 0 & 75 & 30 & 1 & 13 & 1 \\ 1 & 2 & 29 & 83 & 3 & 1 & 1 \\ 0 & 2 & 1 & 0 & 112 & 5 & 0 \\ 0 & 2 & 7 & 1 & 0 & 110 & 0 \\ 8 & 3 & 0 & 0 & 0 & 0 & 109 \end{bmatrix}$$

Here, the rows correspond, from top to bottom, to the following classes:

- Cap shaped plastic
- Non contaminated ("free") jar
- Glass fragment
- Metal sphere
- Pink, plastic sphere
- Small, plastic sphere
- Triangular plastic piece

Similarly, the columns correspond, from left to right, to the following classes:

- Cap shaped plastic
- Non contaminated ("free") jar
- Glass fragment
- Metal sphere
- Pink, plastic sphere
- Small, plastic sphere
- Triangular plastic piece

It can be observed that there seemed to be some confusion between the glass fragment and the metal sphere. On the contrary, other classes were correctly classified (mostly). For instance, 109/120 samples belonging to the triangular plastic class were predicted well. This was the problematic contaminant for the Binary Classifier previously developed [1].

In order to visualize the confusion matrix better, a color map was used as suggested in [2]. This version of the confusion matrix can be found in Figure 5.2.8.1.

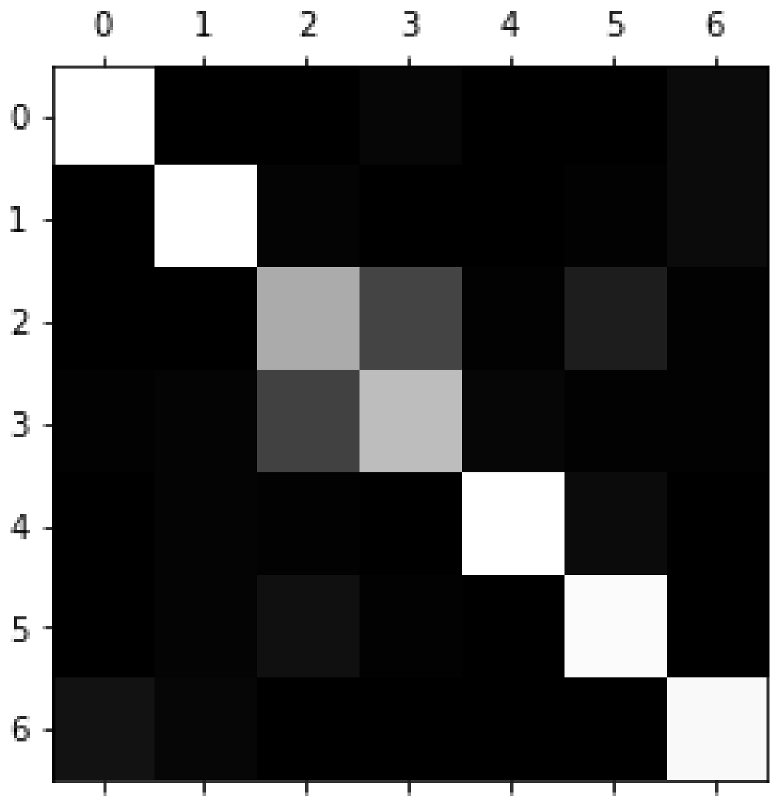


Figure 5.2.8.1: Version E: confusion matrix using a gray color map

As evident in Figure 5.2.8.1, the lighter the color of a cell, the higher the amount of samples in a given cell of the matrix. This way, it's possible to notice how most classes were classified correctly to a large extent.

Furthermore, the confusion between the glass fragment and the metal sphere classes is now also clear, as well as some other misprediction types. Some of these include the triangular plastic samples predicted as cap shaped plastic pieces or non contaminated jars. Another misprediction that is clear is, for example, the glass fragment samples that were confused for small plastic spheres and vice versa.

It's relevant to consider also the misprediction rate with respect to each class. Therefore, the normalized confusion matrix was also calculated and is presented below, with three decimal places for each entry:

$$\begin{bmatrix} 0.933 & 0 & 0 & 0.025 & 0 & 0 & 0.042 \\ 0 & 0.933 & 0.017 & 0 & 0 & 0.008 & 0.042 \\ 0 & 0 & 0.625 & 0.250 & 0.008 & 0.108 & 0.008 \\ 0.008 & 0.017 & 0.242 & 0.692 & 0.025 & 0.008 & 0.008 \\ 0 & 0.017 & 0.008 & 0 & 0.933 & 0.042 & 0 \\ 0 & 0.017 & 0.058 & 0.008 & 0 & 0.917 & 0 \\ 0.067 & 0.025 & 0 & 0 & 0 & 0 & 0.908 \end{bmatrix}$$

The previous matrix shows a proportion relative to the class (row) of samples that were classified as each class (column). In consequence, misprediction rates or percentages can be obtained. Additionally, it's also very useful to be able to easily visualize this information so once again the color map was used. Figure 5.2.8.2 corresponds to this representation of the normalized confusion matrix.

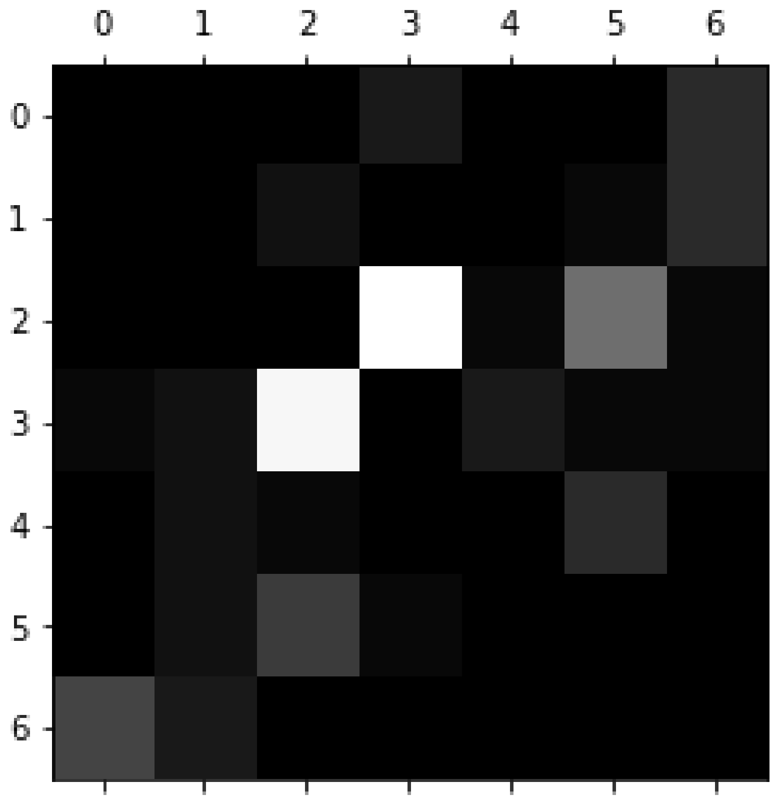


Figure 5.2.8.2: Version E: normalized confusion matrix using a gray color map

About Figure 5.2.8.2 above, it should be mentioned that before creating it the diagonals were filled with zeroes. This was done so that the error rates would be highlighted in exactly the same way as before in Figure 5.2.8.1. Therefore, the lighter the color of a cell in the matrix above, the higher the error rate.

From Figure 5.2.8.2 it's possible to reinforce the preliminary conclusions drawn before with the original confusion matrix. Firstly, this figure reinforces the idea of important confusion between the glass fragment and the metal sphere contaminants. Second, it would appear like other relevant types of mispredictions that are highlighted above are:

- The glass fragment and plastic sphere pair (both ways)
- The triangular plastic and the cap shaped plastic (both directions)
- The triangular plastic and the non contaminated jar (both directions)
- The pink sphere for the plastic sphere

These are, of course, preliminary conclusions based on a non optimized model. The final analysis will be given at the end of this section. However, these initial observations will be relevant once the error analysis for the final model is presented.

5.2.9 Version F

Version F initialized the first tuning step, for the number of layers and neurons. It used an 80/20 data splitting approach as described earlier in Chapter 4. The important results from this version, which were used later, were the ones coming from the optimization step using GridSearchCV.

It should be noted that while there were results obtained using the 3 sets approach, these were not used and were not considered to be indicative of any type of behavior in particular. Furthermore, regarding the RandomizedSearchCV method, this was discarded quickly since results were very random.

Therefore, the results obtained using GridSearchCV are presented next in Table 5.2.9.1 and Table 5.2.9.2. Results or scores, like always, are presented with six decimal places when applicable.

Table 5.2.9.1: Results obtained for Version F (Part I)

Result type: Best_params_	N# Layers	N# Neurons	N# Folds
Run 1	1	256	3
Run 2	2	256	3
Run 3	2	128	3
Run 4	2	256	3
Run 5	3	256	3
Run 6	3	256	5
Run 7	1	128	5

Table 5.2.9.2: Results obtained for Version F (Part II)

Result type: Best_score_	Total score	N# Folds
Run 1	0.877381	3
Run 2	0.870238	3
Run 3	0.869048	3
Run 4	0.873810	3
Run 5	0.866667	3
Run 6	0.882143	5
Run 7	0.897619	5

Initially, it was decided to use 3 folds since this is a very typical value used for GridSearchCV in most applications. However, since the amount of samples was limited, it was decided to change to 5 folds instead to increase the training set per fold (1120 samples instead of 933). Model versions that will be presented in the following subsections also used five folds.

As it's possible to see, the best scores were actually obtained using five folds, in Table 5.2.9.2. The highest score was 0.897619 (seventh run, 5 folds), and 0.882143 (sixth run, 5 folds). The third highest score was 0.877381 (first run, 3 folds).

These scores correspond, respectively, to the following architectures from Table 5.2.9.1:

- 1 hidden layer, 128 neurons
- 3 hidden layers, 256 neurons
- 1 hidden layer, 256 neurons

These results will be compared to the ones obtained with Version G, in the next subsection. Finally, it's important to emphasize at this point that the grid used for Version F was:

N# hidden layers: (1, 2, 3)
N# neurons: (16, 32, 64, 128, 256)

5.2.10 Version G

As explained in Chapter 4, in Table 4.5.2.2, Version G combined Manual KFold with GridSearchCV using both the 80/20 and 85/15 (first version) data splitting approaches. Results related to Manual KFold will not be presented here, since these results were not used. Furthermore, CV was performed before using GridSearchCV, which also performs cross-validation.

Next, Table 5.2.10.1 and Table 5.2.10.2 present the important results obtained with Version G in a similar fashion to the tables for Version F.

Table 5.2.10.1: Results obtained for Version G (Part I)

Result type: Best_params_	N# Layers	N# Neurons	N# Folds	Chosen strategy
Run 1	1	128	5	80/20
Run 2	1	132	5	80/20
Run 3	1	134	5	80/20
Run 4	1	138	5	80/20
Run 5	1	136	5	80/20
Run 6	1	132	5	80/20
Run 7	1	142	5	80/15 (1st)
Run 8	1	144	5	80/15 (1st)
Run 9	1	150	5	80/15 (1st)
Run 10	1	160	5	80/15 (1st)
Run 11	1	150	5	80/15 (1st)

Table 5.2.10.2: Results obtained for Version G (Part II)

Result type: Best_score_	Total score	N# Folds	Chosen strategy
Run 1	0.905357	5	80/20
Run 2	0.901786	5	80/20
Run 3	0.911607	5	80/20
Run 4	0.904464	5	80/20
Run 5	0.911607	5	80/20
Run 6	0.903571	5	80/20
Run 7	0.905882	5	80/15 (1st)
Run 8	0.901681	5	80/15 (1st)
Run 9	0.901681	5	80/15 (1st)
Run 10	0.909244	5	80/15 (1st)
Run 11	0.911765	5	80/15 (1st)

In Table 5.2.10.2, it's possible to find the highest scores for each startegy. In particular, for the 80/20 strategy, the three highest scores are: 0.911607, 0.911607 again, and 0.905357. For its part, in the case of the 85/15 strategy, the three highest scores are: 0.911765, 0.909244, and 0.905882.

All these scores correspond to a specific architecture from Table 5.2.10.1. In the case of the 80/20 strategy, the three highest scores mentioned before correspond to the following model architectures respectively:

- 1 hidden layer, 136 neurons
- 1 hidden layer, 134 neurons
- 1 hidden layer, 128 neurons

In the case of the scores mentioned earlier obtained for the 85/15 strategy, the corresponding model architectures are:

- 1 hidden layer, 150 neurons
- 1 hidden layer, 160 neurons
- 1 hidden layer, 142 neurons

Furthermore, before making any relevant comments, it should be highlighted that at the beginning the same grid from Version G was used. However, after the first run, the neuron array was expanded around 128 (the first candidate) using multiples of two.

For example, instead of using (16, 32, 64, 128, 256), a new array such as (122, 124, 126, 128, 130) could be used. Therefore, to search for new potential models in more detail, once a new candidate was obtained, the grid would be modified as described around the newest result. Evidently, this was an iterative process.

At some point, after noticing that the new candidates all presented architectures with 1 hidden layer, the entries for 2 and 3 hidden layers were erased from the hyperparameter grid. This way, at the end, the last grid produced, which was used to obtain the 11th candidate, was:

N# hidden layers: (1)
N# neurons: (150, 152, 154, 156, 158, 160, 162, 164, 166, 168, 170)

Finally, it's relevant to inspect the results obtained with this version and compare them to the results from Version F. In the case of Version G, it's evident how all candidates in Table 5.2.10.1 present only one hidden layer. Moreover, the maximum amount of neurons is 160 while the minimum is 128.

It's also interesting to note that the number of neurons when using the 85/15 strategy is higher. Specifically, the minimum amount is 142 and the maximum 160. On the other hand, for the 80/20 approach, the minimum number of neurons was 128 and the maximum 138 as shown in Table 5.2.10.1.

Version F, on the contrary, had candidates with 1, 2 or 3 hidden layers as illustrated in Table 5.2.9.1. However, the value for the number of neurons was either 256 or 128 so candidates were not as diverse as in the case of Version G. Ultimately, the best candidates were selected and evaluated using cross-validation. This will be presented now in the following subsections.

5.2.11 Version H

Version H combined a manual, stratified cross-validation process with the 80/20 data splitting approach using 5 folds. The architectures that were evaluated using Stratified CV were the three highest scores obtained using the 80/20 approach and the best candidate obtained using the 85/15 strategy, all from Version G.

These results were the ones considered since the search process was more detailed and consistent for Version G, in comparison to Version F. It should be noted, however, that the architecture with 1 hidden layer and 128 neurons, which was the third candidate from Version G, was the best candidate from Version F. This architecture version was therefore evaluated using Version H.

Also, as suggested by the mentor at Columbia University, the model with 1 layer and 160 neurons was also evaluated in addition to the already mentioned four cases. However, it was evaluated using the test set instead of the training set like the other four cases. This will be further addressed in the next subsection, for Version I.

Table 5.2.11.1 shows the results obtained after running StratifiedCV for all the models. It's also worth nothing that, contrary to manual KFold, in this case both the average validation accuracy and maximum validation accuracy per fold were calculated.

Table 5.2.11.1: Results obtained for Version H

Type of result (obtained for 1 layer models)	Value (136 n.)	Value (134 n.)	Value (128 n.)	Value (150 n.)	Value (160 n.)
StratifiedKFold (average validation accuracy, one value for every fold; manual process)	0.854821	0.867277	0.866004	0.877299	0.780179
	0.867589	0.877009	0.872924	0.853080	0.758304
	0.854152	0.839040	0.843661	0.843504	0.668929
	0.867723	0.876116	0.878616	0.870737	0.635982
	0.861652	0.854219	0.857946	0.873728	0.725268
StratifiedKFold (maximum validation accuracy, one value for every fold; manual process)	0.906250	0.919643	0.910714	0.924107	0.839286
	0.901786	0.919643	0.915179	0.892857	0.839286
	0.901786	0.879464	0.888393	0.883929	0.732143
	0.906250	0.919643	0.915179	0.915179	0.732143
	0.897321	0.888393	0.897321	0.906250	0.767857
StratifiedKFold (average val. acc. and max. val.)	0.861188	0.862732	0.863830	0.863670	0.713732
	0.902679	0.905357	0.905357	0.904464	0.782143

In Table 5.2.11.1, like for Table 5.2.7.1, it should be mentioned that "val. acc" refers to "validation accuracy" and "max. val." to "maximum validation" (accuracy). Additionally, "n." refers to "neurons".

Continuing now, it's evident that for all models except for the one with 160 neurons, performance was very similar. In fact, the average validation accuracy and average maximum validation accuracy values are almost identical. The differences are marginal.

Even then, it should be noted that the model which scored highest for average validation accuracy was the architecture with 128 neurons for the hidden layer. The average validation accuracy it reached was 0.863830. It also achieved the highest (average) maximum validation accuracy, with a value of 0.905357.

In particular for this model, average validation accuracy values ranged from 0.857946 to 0.878616. For the case of the maximum validation accuracy values, these fell in the range from 0.888393 to 0.915179.

On the other hand, the worst performing model was the architecture with 136 neurons. The average validation accuracy reported for it in Table 5.2.11.1 corresponds to 0.861188, with an average maximum validation accuracy of 0.902679. This last value was also the lowest for the previous metric.

Furthermore, in particular for this last model, the average validation accuracy values fall in the range between 0.854152 and 0.867723. Similarly, the range for the maximum validation accuracy values is from 0.897321 to 0.906250.

Overall, since all results were close to 90%, it can be said that they were quite good. Furthermore, at this point there were still hyperparameters left to tune. Therefore, there remained possibilities for improvement. However, it was not immediately clear which model to choose which is why Version I was developed. This notebook is discussed next.

5.2.12 Version I

Version I used StratifiedKFold to perform the first and second tuning steps using the 85/15 data splitting approaches. The results obtained are shown in Table 5.2.12.1, Table 5.2.12.2 and Table 5.2.12.3. In these tables, the same abbreviations used for Table 5.2.11.1 have been used as well.

The first table shows the results for different numbers of neurons in the hidden layer. The models chosen correspond to the best three candidates from Version G for the 85/15 strategy (150, 160 and 142 neurons), as well as the best candidate from the same version using the 80/20 approach (136 neurons). Just like Version H, the architecture with 160 neurons was evaluated using the test set instead.

For its part, the second table shows the results obtained for different optimizers (and learning rates) using the training set. Similarly, the third table also shows results for the optimizers explored but using the test set as suggested by the mentor at Columbia University. The candidates chosen will be specified when discussing the results now. Also, the reason why the specific optimizers were chosen will be explained in the following subsections.

Table 5.2.12.1: Results obtained for Version I (Part I)

Type of result (obtained for 1 layer models)	Value (150 n.)	Value (160 n.)	Value (142 n.)	Value (136 n.)	Value (160 n.)
StratifiedKFold (average validation accuracy, one value for every fold; manual process)	0.860147 0.892626 0.854937 0.868655 0.866134	0.874391 0.886408 0.861218 0.863655 0.868382	0.860168 0.881303 0.861954 0.863445 0.866954	0.862332 0.891912 0.857248 0.857437 0.870777	0.642738 0.748690 0.659881 0.615714 0.734881
StratifiedKFold (maximum validation accuracy, one value for every fold; manual process)	0.903361 0.932773 0.890756 0.907563 0.903361	0.903361 0.932773 0.890756 0.907563 0.903361	0.903361 0.932773 0.890756 0.907563 0.903361	0.911765 0.932773 0.894958 0.890756 0.911765	0.714286 0.857143 0.738095 0.690476 0.833333
StratifiedKFold (average val. acc. and max. val.)	0.868500 0.907563	0.870811 0.907563	0.866765 0.907563	0.867941 0.908403	0.680381 0.766667

In Table 5.2.12.1, it's possible to see that the model with 160 neurons achieved the highest average validation accuracy, with a value of 0.870811. Moreover, it also achieved the highest peak validation accuracy, with a value of 0.907563.

Furthermore, it can be observed that values for the average validation accuracy per fold fall in the range from 0.861218 to 0.886408 for this architecture. On the other hand, the range for its peak validation accuracy values goes from 0.890756 to 0.932773.

Contrary to this case, the model that performed the worst was the architecture with 142 neurons. Specifically, its average validation accuracy was 0.866765, and its average peak validation accuracy was 0.907563.

For this model, the average validation accuracy scores per fold go from 0.860168 to 0.881303. In the case of its maximum validation accuracy values achieved, the range goes from 0.890756 to 0.932773.

Now, it's also possible to compare the results from Table 5.2.12.1 for the model with 160 neurons with the results illustrated before in Table 5.2.11.1, both for the test sets. In the case of the first table, the average validation accuracy was 0.680381, and the average peak validation accuracy was 0.766667. In the case of the second table, these results were 0.713732 and 0.782143 respectively.

It can be said that performance for the test sets was worse when using the 85/15 approach. However, the average results were quite close. Moreover, these results also worked as an incentive to further improve the model to obtain better scores. Therefore, it was decided to choose the best performing model from this version to continue to the second tuning step.

The architecture selected based on the previous results was the one with 160 neurons which is why it was the only architecture to be later cross-validated using the test set (before continuing to the second tuning phase). The reason for this is that, considering Version H and I, the eight candidates evaluated with the training sets using StratifiedKFold performed similarly even though the splitting approaches were different. It was the model with 160 neurons that performed slightly better than all the other candidates.

After choosing this model, the second tuning step was started, where different optimizers and learning rates were evaluated. The search process to find the combinations of optimizer - learning rate to test will be detailed in later subsections. For now, it's relevant to highlight that the following pairs were considered for cross-validation:

- Option A: SGD with learning rate = 0.1.
- Option B: Adagrad with learning rate = 0.3.
- Option C: Adam with learning rate = 0.03.
- Option D: Adam with learning rate = 0.003.

These "options" (the term used to refer to the combinations) can be found with their results in Table 5.2.12.2 below. As mentioned earlier, this table shows the cross-validation results for the training set.

Table 5.2.12.2: Results obtained for Version I (Part II)

Type of result (obtained for 1 layer models)	Option A (SGD)	Option B (Adagrad)	Option C (Adam)	Option D (Adam)
StratifiedKFold (average validation accuracy, one value for every fold; manual process)	0.888130 0.894286 0.870357 0.858172 0.868445	0.903025 0.901492 0.878529 0.888592 0.897542	0.305609 0.339286 0.369391 0.427689 0.381008	0.361723 0.265861 0.255546 0.255294 0.125945
StratifiedKFold (maximum validation accuracy, one value for every fold; manual process)	0.932773 0.924370 0.907563 0.890756 0.903361	0.924370 0.920168 0.899160 0.903361 0.915966	0.323529 0.420168 0.436975 0.521008 0.487395	0.445378 0.315126 0.310924 0.336134 0.163866
StratifiedKFold (average val. acc. and max. val.)	0.875878 0.911765	0.893836 0.912605	0.364597 0.437815	0.252874 0.314286

It's immediately noticeable that Option C and D performed quite poorly in comparison to Option A and B. This means that the Adam optimizer, which had been used in the project until this point, did not perform very well for the architecture selected and with the learning rate values found during the grid search phase (that will be explained in the following subsections).

In particular, Option C achieved an average validation accuracy value of 0.364597, and an average peak validation accuracy value of 0.437815. For its part, Option D obtained an average validation accuracy with a value of 0.252874, and a peak validation accuracy (average) of only 0.314286.

It can also be observed that the ranges for the values obtained per fold for both of these options are also poor in comparison to Option A and B. These other two combinations, on the other hand, achieved higher results.

In particular, Option A (SGD) achieved an average validation accuracy of 0.875878, and a peak (average) validation accuracy of 0.911765. Option B (Adagrad), the best performing one, achieved a result of 0.893836 and 0.912605 respectively. As it can be seen, these four scores were evidently higher than the other two.

Naturally, the ranges for options A and B were much higher and close to their respective average values. For example, for the Adagrad optimizer, the best one, the range for its average validation accuracy values went from 0.878529 to 0.903025. For its peak validation accuracy values, the range went from 0.899160 to 0.924370. Next, Table 5.2.12.3 shows the results for this optimizers using the test set.

Table 5.2.12.3: Results obtained for Version I (Part III)

Type of result (obtained for 1 layer models)	Option A (SGD)	Option B (Adagrad)	Option C (Adam)	Option D (Adam)
StratifiedKFold (average validation accuracy, one value for every fold; manual process)	0.684643 0.759167 0.691190 0.650000 0.738929	0.699762 0.801667 0.701667 0.724286 0.809643	0.278926 0.578333 0.340000 0.336667 0.351548	0.201071 0.145833 0.146071 0.118690 0.174405
StratifiedKFold (maximum validation accuracy, one value for every fold; manual process)	0.761905 0.833333 0.785714 0.738095 0.833333	0.738095 0.857143 0.809524 0.761905 0.857143	0.380952 0.714286 0.404762 0.476190 0.452381	0.285714 0.190476 0.261905 0.190476 0.214286
StratifiedKFold (average val. acc. and max. val.)	0.704786 0.790476	0.747405 0.804762	0.377095 0.485714	0.157214 0.228571

Again, it's evident how, even for the test set, the results for the Adam optimizer are much lower than for the other two. For instance, Option C achieved an average validation accuracy of 0.377095 and an average peak validation accuracy of 0.485714. In the case of Option D, the average validation accuracy was the lowest for the entire project, with a value of just 0.157214. Similarly, the maximum (average) validation accuracy was 0.228571 for this combination.

The average results for Option A and B were fairly similar. For the average validation accuracy, the results were 0.704786 and 0.747405 respectively. In the case of the average maximum validation accuracy, the results obtained were 0.790476 and 0.804762. Once again, just like for the training set, the Adagrad optimizer performed the best.

In particular, for Option B (Adagrad), the range for the average validation accuracy per fold went from 0.699762 to 0.809643. For its part, maximum validation accuracy results ranged from 0.738095 to 0.857143.

Consequently, so far, since Option B was the best performing one both in the case of Table 5.2.12.2 and Table 5.2.12.3, it was the natural candidate to choose. However, this will be further addressed in the next subsections. There are some more results left to consider before making a final decision.

As a last point to discuss, it's interesting to compare the results for the architecture with 160 neurons using Adagrad instead of Adam (with the default learning rate of 0.001). It should be highlighted that this was the optimizer - learning rate combination used before starting the second tuning step.

Continuing now, if Table 5.2.12.1 is compared to Table 5.2.12.2, it's possible to notice that average results were higher for the Adagrad optimizer. Just to have a simple comparison, the average validation accuracy using the training set was 0.870811 with the default Adam but 0.893836 (almost 90%) with Option B.

The same applies to the results obtained using the test sets. For instance, for default Adam, in Table 5.2.12.1 the average validation accuracy reported was 0.680381. Moreover, in Table 5.2.11.1, when the 80/20 approach was used, the result was 0.713732. On the contrary, for Adagrad, in Table 5.2.12.3 the value reported was 0.747405 for the same metric, the highest of all.

These results were also evidence supporting the idea that Adagrad (with a learning rate of 0.3) was indeed a better option than using the Adam optimizer, for which three different learning rate values were studied (0.3, 0.001 and 0.003). It was particularly important to analyze the default Adam, seeing as changing the learning rate for this optimizer affected the cross-validation results significantly, as already shown.

5.2.13 Version J

As shown in Table 4.5.2.2, Version J was used to develop five different versions. The ones considered to be relevant to discuss are the last three which focused on the second tuning step. Each targeted one optimizer with different learning rates.

As indicated in the previously mentioned table, these versions used the 80/20 data splitting approach. Moreover, like before, the second hyperparameter optimization step was developed using GridSearch. The combinations and the results obtained can be found in Table 5.2.13.1 and Table 5.2.13.2.

Table 5.2.13.1: Results obtained for Version J (Part I)

Result type: Best_params_	Optimizer chosen	Learning rate	N# Folds
Run 1	Adam	0.030	5
Run 2	Adam	0.003	5
Run 3	Adam	0.003	5
Run 4	SGD	0.030	5
Run 5	SGD	0.100	5
Run 6	SGD	1.000	5
Run 7	SGD	0.100	5
Run 8	Adagrad	0.030	5
Run 9	Adagrad	0.300	5
Run 10	Adagrad	0.300	5

Regarding the first table, Table 5.2.13.1, the obtained learning rates for each run, for each optimizer chosen, are shown. There were three results obtained for the Adam optimizer, four for SGD, and three for Adagrad.

It should be explained how the grids were elaborated during this optimization step. For all three optimizers the initial grid was the same and included their default learning rate values (1e-3 for Adam, 1e-2 for SGD and Adagrad [7]):

(3e-4, 1e-4, 3e-3, 1e-3, 3e-2, 1e-2)

The grid for the next run, for each optimizer, was modified depending on the value obtained in the previous first run. Therefore, the grids for Adam were the following:

- Second run (Run 2): (3e-4, 1e-4, 3e-3, 1e-3, 3e-2, 1e-2, 3e-1, 1e-1)
- Third run (Run 3): (3e-5, 3e-4, 3e-3, 3e-2)

In the case of SGD, the grids were:

- Second run (Run 5): (3e-4, 1e-4, 3e-3, 1e-3, 3e-2, 1e-2, 3e-1, 1e-1)
- Third run (Run 6): (1e-3, 1e-2, 1e-1, 1)
- Fourth run (Run 7): (1e-2, 1e-1, 1, 10, 100)

For Adagrad, the grids used after its first run (Run 8) were:

- Second run (Run 9): (3e-4, 1e-4, 3e-3, 1e-3, 3e-2, 1e-2, 3e-1, 1e-1)
- Third run (Run 10): (3e-1, 1e-1, 1, 3)

As it's possible to see in Table 5.2.13.1, the learning rates obtained for Adam were 0.03 and 0.003. For SGD, the results were 0.03, 0.1 and 1. For Adagrad, the resulting learning rates were 0.03 and 0.3. It should be noted that these are the values reported by GridSearch that were different since some runs returned the same learning rate (for each optimizer).

Table 5.2.13.2: Results obtained for Version J (Part II)

Result type: Best_score_	Total score	N# Folds
Run 1	0.897619	5
Run 2	0.896429	5
Run 3	0.894048	5
Run 4	0.861905	5
Run 5	0.897619	5
Run 6	0.895238	5
Run 7	0.903571	5
Run 8	0.880952	5
Run 9	0.901190	5
Run 10	0.904762	5

The scores obtained by each optimizer - learning rate combination found are shown in Table 5.2.13.2. As it's possible to see, the best results for each optimizer were:

- Adam, learning rate = 0.03 (Run 1), with a score of 0.897619.
- SGD, learning rate = 0.1 (Run 7), with a score of 0.903571.
- Adagrad, learning rate = 0.3 (Run 9 and 10), with a score of 0.901190 and 0.904762 for each respective run.

The combinations chosen, after considering the results from Version J, were the ones shown previously for Version I in Table 5.2.12.2 and Table 5.2.12.3 (the "options"). In other words: Adam with a learning rate of 0.03 and 0.003 (Options C and D), SGD with a learning rate of 0.1 (Option A) and Adagrad with a learning rate of 0.3 (Option B).

It should be mentioned that in the case of the Adam optimizer, two learning rate values were chosen for CV instead of one. This decision was taken because until this point this optimizer had been used for development. Also, quite good results had been achieved (close to 90%). Therefore, there was special interest in observing how it could perform in comparison to the other choices.

5.2.14 Version K

As mentioned earlier in Table 4.5.2.2, Version K consisted in performing Stratified CV using the 80/20 data splitting approach for the optimizers. This notebook performed cross-validation using both the training and test sets. Table 5.2.14.1 shows the obtained results for the training set, and Table 5.2.14.2 for the test set.

Table 5.2.14.1: Results obtained for Version K (Part I)

Type of result (obtained for 1 layer models)	Option A (SGD)	Option B (Adagrad)	Option C (Adam)	Option D (Adam)
StratifiedKFold (average validation accuracy, one value for every fold; manual process)	0.872760 0.899010 0.894375 0.856771 0.877891	0.894453 0.921328 0.910078 0.874089 0.892369	0.885495 0.899583 0.926745 0.872500 0.888932	0.893438 0.909401 0.903385 0.874297 0.881562
StratifiedKFold (maximum validation accuracy, one value for every fold; manual process)	0.906250 0.937500 0.927083 0.880208 0.916668	0.911458 0.932292 0.937500 0.890625 0.906250	0.911458 0.927083 0.963542 0.895833 0.911458	0.916667 0.932292 0.927083 0.901042 0.916667
StratifiedKFold (average val. acc. and max. val.)	0.880161 0.913542	0.898464 0.915625	0.894651 0.921875	0.892417 0.918750

In Table 5.2.14.1, the same abbreviations from before are used again. Now, it's possible to see that the option that achieved the highest average validation accuracy for all 5 folds is Option B (Adagrad with a learning rate of 0.3). Specifically, this value corresponds to 0.898464, with an average peak validation accuracy for all folds of 0.915625.

Moreover, Option B has a range for the average validation accuracy values that goes from 0.874089 to 0.921328. For its part, the peak validation accuracy values fall in the range from 0.890625 to 0.937500.

On the contrary, the worst performing option in Table 5.2.14.1 is Option A (the SGD optimizer with a learning rate of 0.1). Its average validation accuracy across all folds is 0.880161, with an average peak validation accuracy of 0.913542. This is also the lowest result for all the options.

In particular, the values obtained using the SGD optimizer fall in the range from 0.856771 to 0.899010 for the average validation accuracy values. In the case of the peak validation accuracy results, the range goes from 0.880208 to 0.937500.

In general, it should be noted that values are close to 90%. However, the selected candidate should be analyzed further in more depth during the error analysis phase. It's also possible to state that, in comparison to Table 5.2.12.2 for Version I, results in this case seem more homogeneous, stable and similar, especially regarding Options C and D.

Furthermore, considering Table 5.2.12.2 again, also the best candidate was Option B. Moreover, it achieved similar values for the average validation accuracy for all folds (0.893836) and average peak validation accuracy (0.912605). These results, however, were slightly lower than the ones obtained using the 80/20 splitting approach in Version K.

Next, Table 5.2.14.2 shows the results for the same options (Version K) but evaluated on the test set instead.

In the case of Table 5.2.14.2, the best option resulted to be Option C (Adam optimizer with a learning rate of 0.03). The average validation accuracy for all folds is 0.793229, while the average peak validation accuracy is 0.837500.

Table 5.2.14.2: Results obtained for Version K (Part II)

Type of result (obtained for 1 layer models)	Option A (SGD)	Option B (Adagrad)	Option C (Adam)	Option D (Adam)
StratifiedKFold (average validation accuracy, one value for every fold; manual process)	0.752083 0.660625 0.784167 0.728750 0.792187	0.802083 0.660521 0.815417 0.780521 0.802500	0.783750 0.720312 0.818125 0.803646 0.840313	0.793854 0.712396 0.796771 0.755729 0.829271
StratifiedKFold (maximum validation accuracy, one value for every fold; manual process)	0.791667 0.708333 0.854167 0.770833 0.833333	0.833333 0.770833 0.875000 0.812500 0.875000	0.812500 0.791667 0.854167 0.854167 0.875000	0.833333 0.750000 0.833333 0.791667 0.875000
StratifiedKFold (average val. acc. and max. val.)	0.743562 0.791667	0.772208 0.833333	0.793229 0.837500	0.777604 0.816667

In particular, the range for the average validation accuracy values goes from 0.720312 to 0.840313. In the case of the peak validation accuracy results, these range from 0.791667 to 0.875000.

For its part, the worst performing option resulted to be Option A (SGD with a learning rate of 0.1), just like in the previous table (Table 5.2.14.1). It scored a value of 0.743562 for the average validation accuracy for all folds, and an average value of 0.791667 for the average peak validation accuracy.

Specifically, values for the average validation accuracy metric ranged from 0.660625 to 0.792187. In the case of the peak validation accuracy results, these ranged from 0.708333 to 0.854167.

Just like with the previous table for Version K, it can be said that results for Stratified CV were again more stable, similar, and homogeneous. Once more, this was especially true about Option C and Option D. It would appear like the 80/20 splitting approach balanced the total amount of data better.

Now, after considering all the results obtained for Version I and K, the best candidate resulted to be Option B (Adagrad with a learning rate of 0.3) in all cases except one. The different situation occurred in Table 5.2.14.2, when Option C was the best (Version K).

However, in that table, while Option C scored a value of 0.793229 for average validation accuracy for all folds, Option B scored a value of 0.772208. The respective average peak validation accuracy values were 0.837500 and 0.833333 respectively.

Therefore, the most significant difference occurred between the average validation accuracy results, which, in the end, were not very different and were still both close to 80%. Moreover, Option B performed significantly better than Option C in the case of Version I (85/15 data splitting approach) and slightly better when using the training set in Version K.

All in all, then, after considering all the obtained results coming from both Version I and K, the selected option was **Option B, Adagrad with a learning rate of 0.3**. Now, this selection, combined with the architecture of 1 hidden layer with 160 neurons, will be analyzed in depth in the following two subsections.

5.2.15 Version L

As indicated before in Table 4.5.2.2, Version L studied the optimized model. In particular, version L3 is the MLP with 1 hidden layer (160 neurons) that uses the Adagrad optimizer with a learning rate of 0.3.

This version used the 80/20 data splitting approach. It split the dataset into a training, validation and test set. The important results in this case correspond to the confusion matrices obtained, so these are illustrated next for each set. It should be noted that each row and column is associated to the same class as in the case of Version E.

5.2.15.1 Training set

The confusion matrix for the training set is shown next. Each class had 120 samples (840 samples in total).

$$\begin{bmatrix} 120 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 120 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 120 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 120 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 120 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 120 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 120 \end{bmatrix}$$

The confusion matrix is perfect, but this is not a very good sign. Most likely, the model overfitted the training set. Therefore, it's very important to see how the model performs when using the other sets. The case of the test set is shown next.

5.2.15.2 Test set

Now, the confusion matrix for the test set is illustrated. Each class had 40 samples (280 samples in total).

$$\begin{bmatrix} 39 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 37 & 2 & 0 & 0 & 1 & 0 \\ 1 & 0 & 32 & 7 & 0 & 0 & 0 \\ 2 & 0 & 7 & 28 & 1 & 0 & 2 \\ 1 & 0 & 0 & 0 & 39 & 0 & 0 \\ 0 & 0 & 5 & 0 & 0 & 35 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 38 \end{bmatrix}$$

In this case, there are 40 samples per class in the test set. Again, like in Version E, it can be seen that class 2 (row 3, glass fragment) is confused with class 3 (column 4, metal sphere) and viceversa. As a consequence, 32/40 and 28/40 samples are correctly classified for these two last classes respectively. Also, these are the lowest scores for the matrix.

On the other hand, class 0 (cap shaped plastic), class 1 (non contaminated jar), class 4 (pink sphere) and class 6 (triangle) are correctly classified for the most part. In particular, the ratios are 39/40, 37/40, 39/40 and 38/40 samples respectively.

Regarding class 5, the correct predictions are quite high as well, with 35/40 samples classified correctly. Now, just like for Version E, Figure 5.2.15.2.1 shows the gray scale map for the previous confusion matrix.

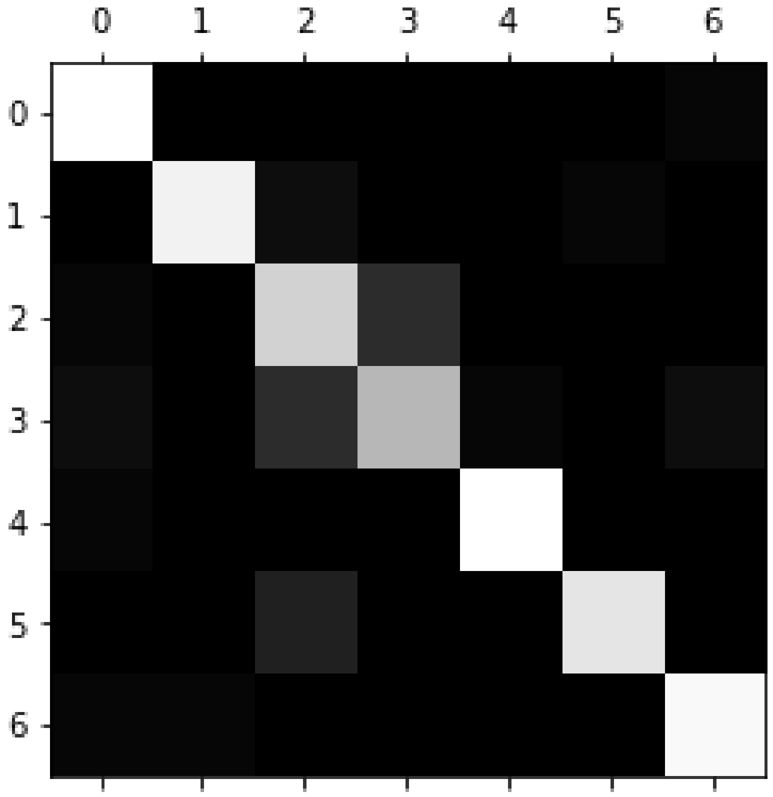


Figure 5.2.15.2.1: Final model: confusion matrix using a gray color map (test set, 80/20).

Again, using Figure 5.2.15.2.1, it can be observed that samples are classified correctly (mostly). It's possible to appreciate the confusion between classes 2 and 3. Also, the samples from class 5 (plastic sphere) wich are being misclassified as class 2 (glass) samples are brought to attention with this map.

Continuing now, the normalized confusion matrix for the test set is shown now. For its part, Figure 5.2.15.2.2 shows this matrix using a gray scale map.

$$\begin{bmatrix} 0.975 & 0 & 0 & 0 & 0 & 0 & 0.025 \\ 0 & 0.925 & 0.050 & 0 & 0 & 0.025 & 0 \\ 0.025 & 0 & 0.800 & 0.175 & 0 & 0 & 0 \\ 0.050 & 0 & 0.175 & 0.700 & 0.025 & 0 & 0.050 \\ 0.025 & 0 & 0 & 0 & 0.975 & 0 & 0 \\ 0 & 0 & 0.125 & 0 & 0 & 0.875 & 0 \\ 0.025 & 0.025 & 0 & 0 & 0 & 0 & 0.950 \end{bmatrix}$$

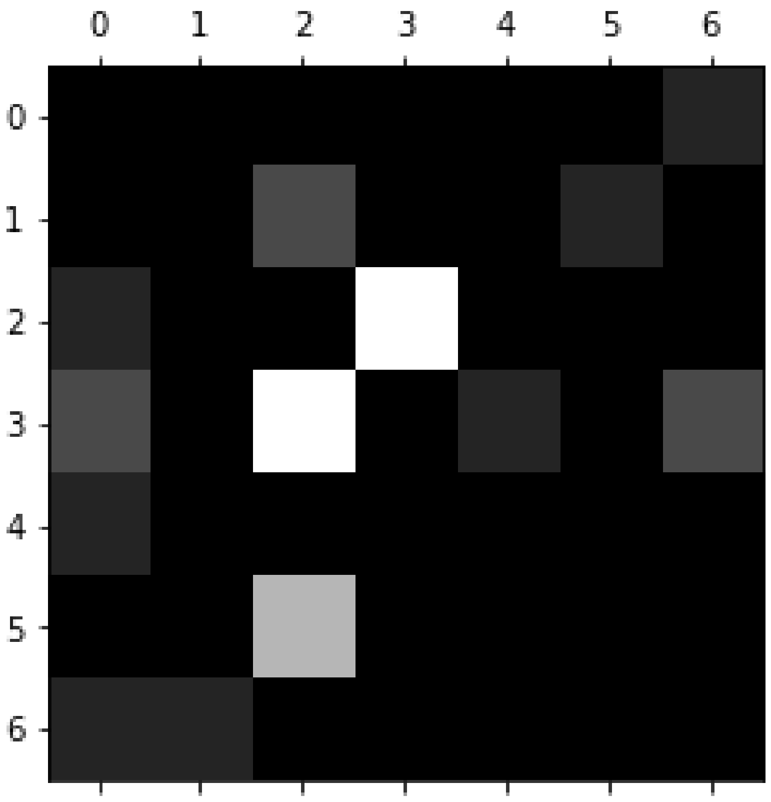


Figure 5.2.15.2.2: Final model: normalized confusion matrix using a gray color map (test set, 80/20).

Considering the previous matrix and Figure 5.2.15.2.2, once more it's possible to appreciate that the highest amount of error is focused on class 2 and 3, which are misclassified as each other. The second brightest cell in the map belongs to the samples from class 5 that were mispredicted as class 2 samples.

Next, it would be interesting to compare these results to the Binary Classifier developed by [1]. This is possible if the confusion matrix presented before is collapsed to a 2x2 dimension. To achieve this, simply all contaminant types should be considered as one class.

Therefore, for instance, a contaminant that is misclassified as another contaminant type will simply be considered as a correctly classified contaminant. Then, after collapsing the 7x7 matrix, the following confusion matrix for the test set was obtained:

$$\begin{bmatrix} 239 & 1 \\ 3 & 37 \end{bmatrix}$$

About this confusion matrix, it should be highlighted that the first row corresponds to the "contaminant" class, and the second to the "free" (non contaminated) jar class. About the columns, the first belongs to the "contaminant" class, and the second to the "free" class. In other words, the "contaminant" class is the Positive class while the "free" class is the Negative class.

If this new 2x2 matrix is taken into consideration, it can be appreciated that the misprediction ratios are quite low. In fact, the normalized confusion matrix in this case is:

$$\begin{bmatrix} 0.996 & 0.004 \\ 0.075 & 0.925 \end{bmatrix}$$

Since overall the total amount of contaminant samples is 240 (40 for 6 classes), and for the non contaminated samples the total is 40. As shown above, most samples are classified correctly for the collapsed, binary case.

Additionally, since in [1] the problematic contaminant was the triangular plastic, it's of special interest to analyze how many of these samples were misclassified as "free" jars. This can be done now by studying the 7x7 case in detail.

Therefore, considering now the first 7x7 confusion matrix presented at the beginning, only 1/40 out of all the triangular plastic samples were classified as "free jars". This is translated into an error of 2.5% with respect to the triangle plastic samples, in contrast to the 51.220% reported for the Binary Classifier [1] using a test set (21/41 samples).

5.2.15.3 Validation set

The confusion matrix for the validation set is shown below. In this case, each class had 40 samples (280 samples in total).

$$\begin{bmatrix} 37 & 0 & 0 & 0 & 0 & 0 & 3 \\ 0 & 40 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 33 & 5 & 1 & 1 & 0 \\ 0 & 0 & 16 & 23 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 39 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 38 & 0 \\ 2 & 3 & 0 & 0 & 0 & 0 & 35 \end{bmatrix}$$

Like before, it can be seen that class 2 and class 3 are being misclassified. On the other hand, most other classes are being classified correctly. Regarding these (all classes except 2 and 3), it's the last one (class 6, the triangular plastic) that has the lowest amount of correctly classified samples (35/40). Next, the gray scale map for this confusion matrix is shown in Figure 5.2.15.3.1.

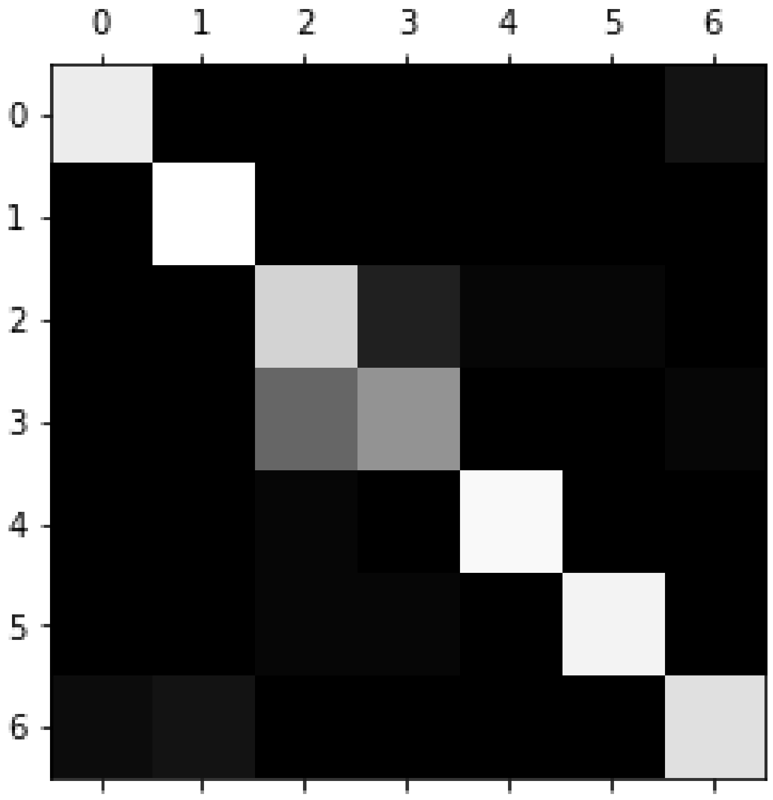


Figure 5.2.15.3.1: Final model: confusion matrix using a gray color map (validation set, 80/20).

In Figure 5.2.15.3.1, it can be appreciated again that, for the most part, samples were classified correctly. The most significant mispredictions concern classes 2 and 3; other errors seem less significant in comparison, with dark cells. To complement this information, the normalized confusion matrix is shown below.

$$\begin{bmatrix} 0.925 & 0 & 0 & 0 & 0 & 0 & 0.075 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.825 & 0.125 & 0.025 & 0.025 & 0 \\ 0 & 0 & 0.400 & 0.575 & 0 & 0 & 0.025 \\ 0 & 0 & 0.025 & 0 & 0.975 & 0 & 0 \\ 0 & 0 & 0.025 & 0.025 & 0 & 0.950 & 0 \\ 0.050 & 0.075 & 0 & 0 & 0 & 0 & 0.875 \end{bmatrix}$$

It's evident, in the matrix, how the biggest error proportions concern the confusion between classes 2 and 3. For instance, the proportion of class 2 samples classified as class 3 samples is 0.125. The result for the reversed case is 0.400. Now, concerning the other error ratios, these are all smaller, with a zero after the decimal point.

Next, like before, Figure 5.2.15.3.2 illustrates the gray scale map of the normalized matrix on the next page.

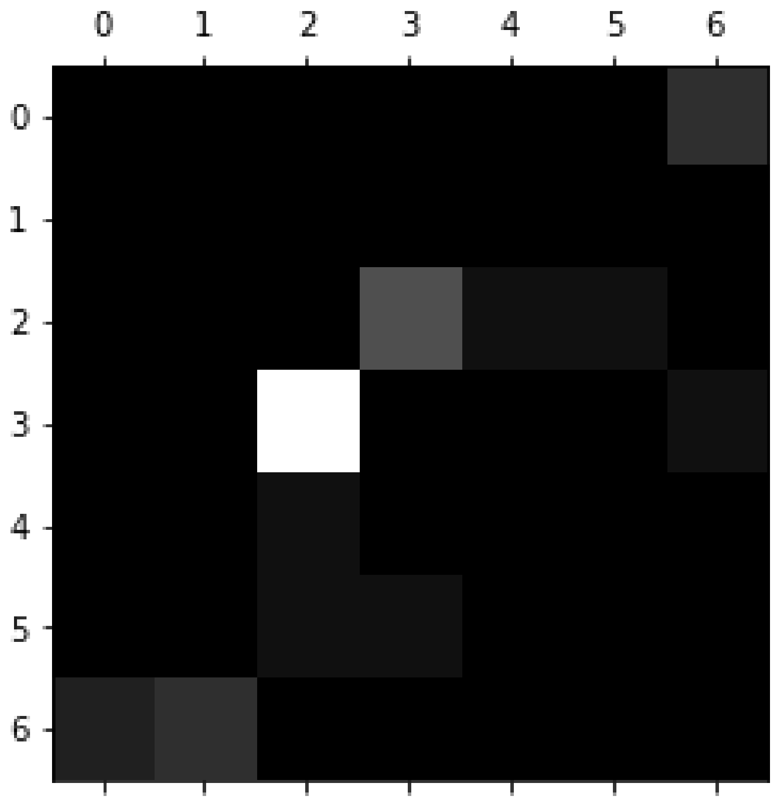


Figure 5.2.15.3.2: Final model: normalized confusion matrix using a gray color map (validation set, 80/20).

Figure 5.2.15.3.2 highlights where the main error ratios are in the matrix. The 0.400 value (samples from class 3 which were misclassified as class 2 samples) is the highest, and so is the brightest in the picture.

The map also highlights other cells where the error ratio is higher with respect to all the error ratios. In particular, cells with these coordinates: (2,3), (0,6) and (6,1) seem quite significant as well. Now, the collapsed 2x2 matrix for the validation set is illustrated next.

$$\begin{bmatrix} 237 & 3 \\ 0 & 40 \end{bmatrix}$$

Above, it's immediately noticeable that no "free" samples were misclassfied. This can be verified with the original matrix at the beginning of this section. Additionally, the number of mispredictions concerning the contaminanted class is very low: just 3 out of 240 samples.

In fact, this information is further emphasized in the normalized version of this confusion matrix:

$$\begin{bmatrix} 0.988 & 0.013 \\ 0 & 1 \end{bmatrix}$$

Therefore, the error for the validation set only involved the contaminated samples that were misclassified as "free" samples. The portion of contaminants (out of all the contaminant samples) that were misclassified was just 1.3%.

Finally, again, considering the triangular plastic now and the original 7x7 case, it's possible to observe that only 3 out of a total of 40 triangular plastic samples were classified as "free" jars. This is equivalent to an error of 7.5%.

5.2.16 Version M

As indicated previously in Table 4.5.2.2, Version M also focused on the final model. In fact, the trained model was loaded into this notebook (Version M). The difference was that Version M used the second 85/15 data splitting approach.

Like for Version L, the confusion matrices are presented next. The order will be the same: training, test, and validation set. Furthermore, all matrices associated each row and column to the same classes as before like in Version E and Version L.

5.2.16.1 Training set

The confusion matrix for the training set is shown below. In this case, each class had 136 samples, for a total of 952 samples in the training set.

$$\begin{bmatrix} 133 & 0 & 0 & 0 & 0 & 0 & 3 \\ 0 & 136 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 125 & 9 & 1 & 1 & 0 \\ 0 & 0 & 14 & 121 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 135 & 0 & 0 \\ 0 & 0 & 3 & 0 & 0 & 133 & 0 \\ 2 & 3 & 0 & 0 & 0 & 0 & 131 \end{bmatrix}$$

Once again, it's possible to observe that most mispredictions are the ones that confuse class 2 with 3 and viceversa. All other errors are small in comparison. In fact, the map shown in Figure 5.2.16.1.1 highlights this observation.

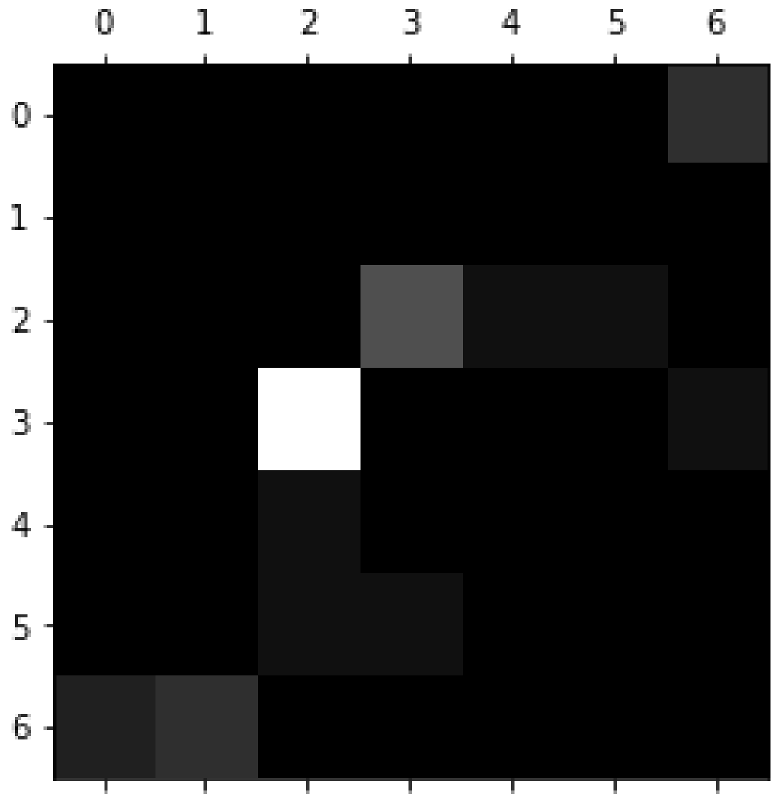


Figure 5.2.16.1.1: Final model: confusion matrix using a gray color map (training set, 85/15).

In fact, in Figure 5.2.16.1.1, the brightest cell (white) is that with coordinates (3,2). Here, there are 14 samples from class 3 that got misclassified as class 2 samples. The second brightest cell is the one with coordinates (2,3), where there are 9 samples.

There are other cells which are also quite bright in comparison to most, which are those with coordinates (6,0), (6,1) and (0,6). Next, the normalized confusion matrix for the training set is shown.

$$\begin{bmatrix} 0.978 & 0 & 0 & 0 & 0 & 0 & 0.022 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.919 & 0.066 & 0.007 & 0.007 & 0 \\ 0 & 0 & 0.103 & 0.890 & 0.007 & 0 & 0 \\ 0 & 0 & 0.007 & 0 & 0.993 & 0 & 0 \\ 0 & 0 & 0.022 & 0 & 0 & 0.978 & 0 \\ 0.015 & 0.022 & 0 & 0 & 0 & 0 & 0.963 \end{bmatrix}$$

The ratios presented in the normalized confusion matrix are consistent with the observations made earlier. In fact, the errors ratios with coordinates (3,2) and (2,3) are the highest ones (0.103 and 0.066 respectively). Figure 5.2.16.1.2 visualizes this quite well.

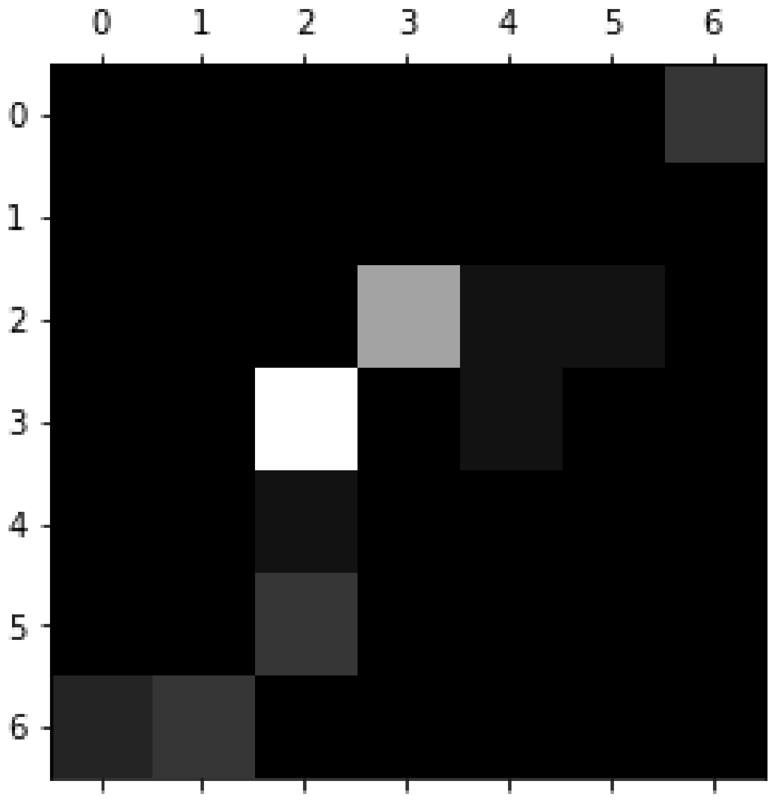


Figure 5.2.16.1.2: Final model: normalized confusion matrix using a gray color map (training set, 85/15).

In Figure 5.2.16.1.2, the brightest cells (highest error ratios) have coordinates (3,2) and (2,3), as expected due to the past behavior. Cells with coordinates (6,0), (6,1) and (0,6) are the second brightest.

Another cell which is notable and among the lightest in color is the one with coordinates (5,2). This type of misprediction was significant before once, in the case of the test set in Version L.

Now, the last step was obtaining the collapsed 2x2 confusion matrix. It's presented below.

$$\begin{bmatrix} 813 & 3 \\ 0 & 136 \end{bmatrix}$$

Once more, it's remarkable how no "free" samples were mispredicted as contaminated jars. Likewise, it's possible to observe that only 3 out of 816 contaminant samples were misclassified as free jars. In fact, the normalized confusion matrix is:

$$\begin{bmatrix} 0.996 & 0.004 \\ 0 & 1 \end{bmatrix}$$

As seen in the normalized confusion matrix, only 0.4% of all the contaminant samples were misclassified as "free" jars. Finally, like in the previous cases, special attention should be paid to the triangular plastic. As the 7x7 confusion matrix shows, only 3 out of 136 triangular plastic samples were misclassified as "free", which is equivalent to an error ratio of 2,206%.

5.2.16.2 Test set

The confusion matrix for the test set is shown next. In this case, each class had 30 samples, so there were a total of 210 samples in the test set.

$$\begin{bmatrix} 29 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 28 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 25 & 4 & 0 & 0 & 0 \\ 1 & 0 & 6 & 21 & 0 & 0 & 2 \\ 1 & 0 & 0 & 0 & 29 & 0 & 0 \\ 0 & 0 & 3 & 0 & 0 & 27 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 29 \end{bmatrix}$$

As with all the previous reported cases, once again the most significant mispredictions were between class 2 and 3. The cells (3,2) and (2,3) have, once more, the highest amount of misclassified values in them. Specifically, the values in these cells are 6 and 4 respectively. Now, like before, Figure 5.2.16.2.1 shows the color map for the confusion matrix.

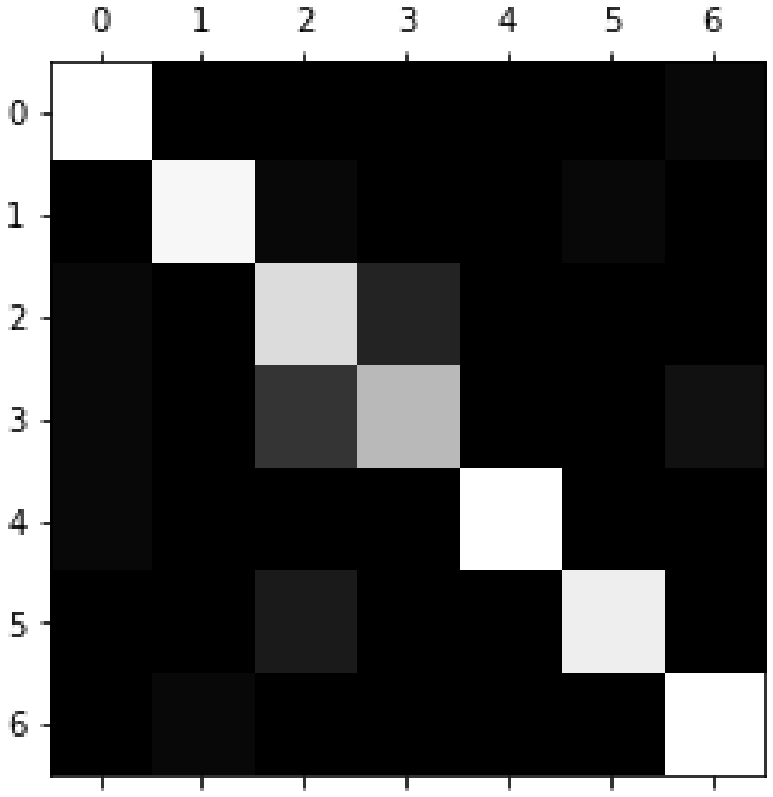


Figure 5.2.16.2.1: Final model: confusion matrix using a gray color map (test set, 85/15).

It's possible to appreciate in Figure 5.2.16.2.1 that most samples were classified correctly. The mispredicted samples mostly concern classes 2 and 3 with each other, as already mentioned. From the figure, it's already possible to appreciate that other important locations with misclassified data are cells (5,2), with 3 samples, and (3,6), with 2 samples.

Next, the normalized confusion matrix for this case is presented below.

$$\begin{bmatrix}
0.967 & 0 & 0 & 0 & 0 & 0 & 0.033 \\
0 & 0.933 & 0.033 & 0 & 0 & 0.033 & 0 \\
0.033 & 0 & 0.833 & 0.133 & 0 & 0 & 0 \\
0.033 & 0 & 0.200 & 0.700 & 0 & 0 & 0.067 \\
0.033 & 0 & 0 & 0 & 0.967 & 0 & 0 \\
0 & 0 & 0.100 & 0 & 0 & 0.900 & 0 \\
0 & 0.033 & 0 & 0 & 0 & 0 & 0.967
\end{bmatrix}$$

As expected, the highest error proportions are found in cells (3,2) with a value of 0.2 and (2,3) with a value of 0.133. For its part, cell (5,2) shows an error ratio of 0.1 and cell (3,6) an error ratio of 0.067. Now, in order to visualize these details better, Figure 5.2.16.2.2 shows the gray scale color map for the normalized confusion matrix.

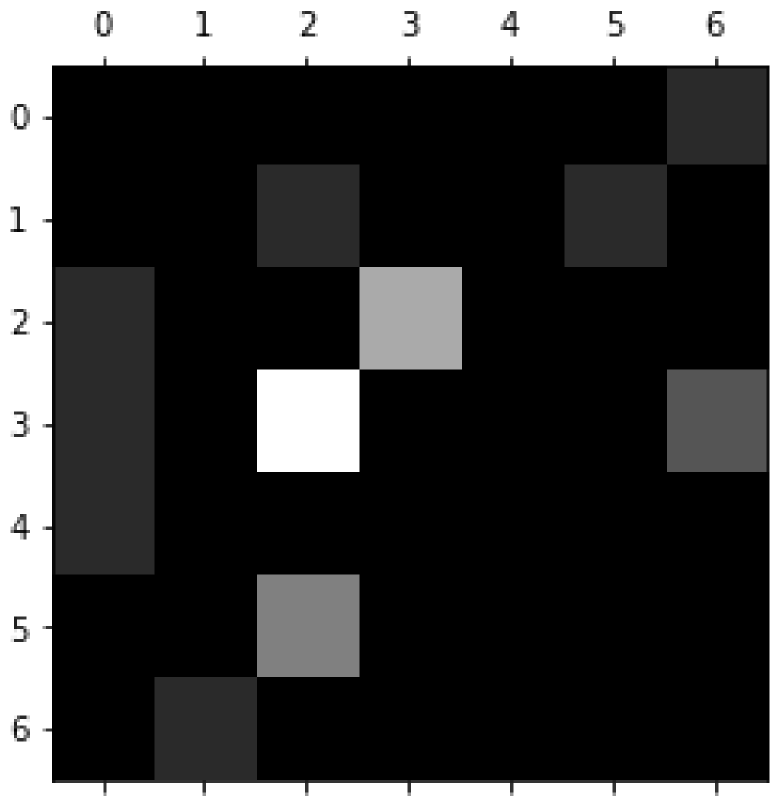


Figure 5.2.16.2.2: Final model: normalized confusion matrix using a gray color map (test set, 85/15).

In fact, in Figure 5.2.16.2.2, the brightest cells are again located at (3,2) and (2,3). The second most significant error contributions seem to come from cells (5,2) and (3,6) as mentioned before. There are, also, a few other mispredictions, which are not very significant in comparison to the previously mentioned ones. Moreover, it can be verified from the confusion matrix that the value of all these last cells is just 1 mispredicted sample.

Now, it's possible to continue and collapse the 7x7 confusion matrix to the 2x2 case. The binary confusion matrix is now illustrated:

$$\begin{bmatrix} 179 & 1 \\ 2 & 28 \end{bmatrix}$$

This time, there were contaminated samples that were mispredicted as "free" samples and viceversa. However, the error ratio was very low. In fact, the normalized 2x2 confusion matrix is:

$$\begin{bmatrix} 0.994 & 0.006 \\ 0.067 & 0.933 \end{bmatrix}$$

Therefore, it can be said that the error ratio describing contaminants that were misclassified as "free" jars is 0.6%, with respect to the total number of contaminants. Similarly, the error ratio for "free" jars that were mispredicted as contaminated jars is 6.7%, a bit higher, but still quite low.

Finally, regarding the triangular plastic piece, after checking the 7x7 confusion matrix it can be said that only 1/30 samples were mispredicted as "free" jars. This ratio is equivalent to saying that 3.33% of the triangular pieces were misclassified as "free".

5.2.16.3 Validation set

The 7x7 confusion matrix of the validation set is presented below. In this case, each class had 34 samples, which translate into a total of 238 samples for the entire set.

$$\begin{bmatrix} 34 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 33 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 34 & 0 & 0 & 0 & 0 \\ 1 & 0 & 3 & 30 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 34 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 34 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 34 \end{bmatrix}$$

Quite interestingly, as seen above, the mispredicted samples are not that many. For the first time, the cell at location (2,3) is 0 while the cell at (3,2) has a value of 3. In order to visualize this situation better, Figure 5.2.16.3.1 shows the map representing this confusion matrix.

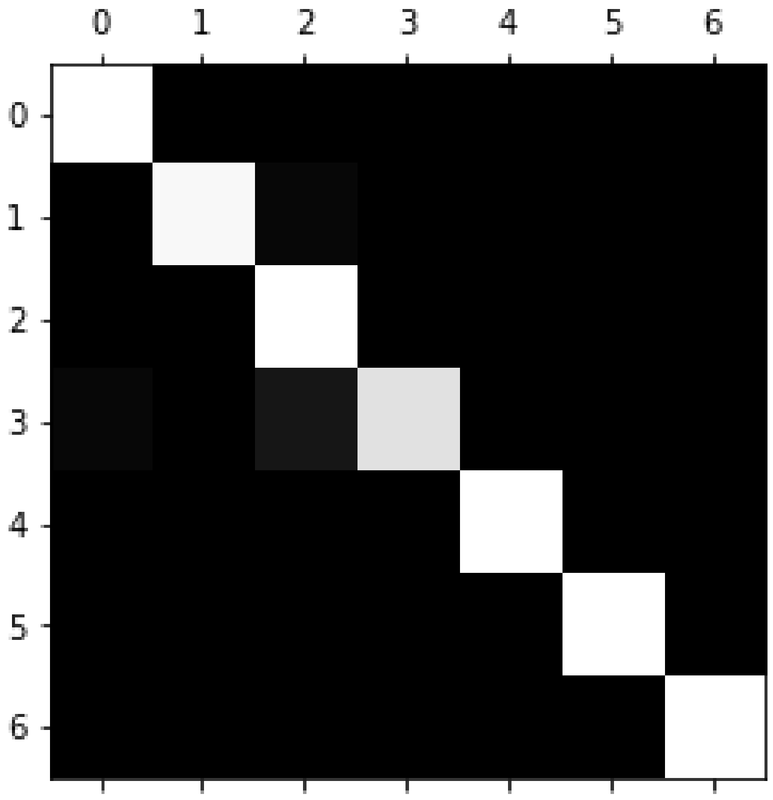


Figure 5.2.16.3.1: Final model: confusion matrix using a gray color map (validation set, 85/15).

The behavior described earlier can be appreciated better in Figure 5.2.16.3.1. Almost all the samples were classified correctly; this fact is evidenced by the presence of a bright diagonal. Cells with mispredicted samples are quite dark, except for the cell at (3,2) which is slightly lighter. This information can be complemented now with the normalized confusion matrix below.

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0.971 & 0.029 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0.029 & 0 & 0.088 & 0.882 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Above, the error ratios are quite low. The highest one is the one in the cell at (3,2), with a value of 0.088. The other two cells with mispredictions, cells (3,0) and (1,2), have both a value of 0.029. This situation is made even more evident with the color map of the normalized confusion matrix in Figure 5.2.16.3.2.

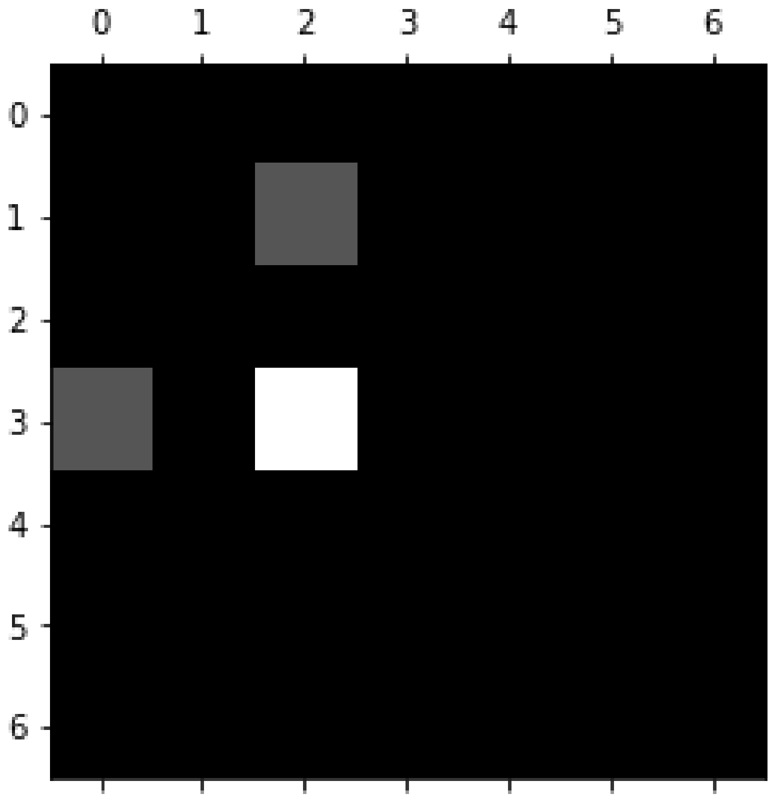


Figure 5.2.16.3.2: Final model: normalized confusion matrix using a gray color map (validation set, 85/15).

In fact, in Figure 5.2.16.3.2, there are only three cells that are not black. These cells are the ones that were mentioned earlier, out of which the one with the highest amount of mispredictions is the one at (3,2). Now, given these results it would be interesting to analyze the 2x2 confusion matrix, which is presented below:

$$\begin{bmatrix} 204 & 0 \\ 1 & 33 \end{bmatrix}$$

In this case, considering the binary classes, no contaminants were mispredicted. On the other hand, only one "free" sample got misclassified as a contaminant sample. Furthermore, the normalized 2x2 confusion matrix is:

$$\begin{bmatrix} 1 & 0 \\ 0.029 & 0.971 \end{bmatrix}$$

As it can be seen above, the error ratio for the "free" samples that were misclassified is only 2.9%, with respect to the entire set of "free" samples. On the contrary, the error ratio for the contaminated samples is 0%.

Finally, it's worth considering the case of the triangular plastic samples. Considering now the original 7x7 matrix again, it's possible to see that no triangular plastic pieces were misclassified as "free" jars.

Chapter 6

Hardware implementation

This chapter focuses on the hardware phase of the project. The workflow for this part starts by using **hls4ml**, which takes as input the .json and .h5 (for the weights) generated at the end of the ML worflow. Therefore, the files saved for the final, optimized model (L3) were used.

After obtaining the output files from hls4ml, **Vivado HLs** was used, and then finally **Vivado**. In other words, the steps taken were related to High Level Synthesis and Logic Synthesis, with bitstream generation.

6.1 hls4ml and Vivado HLS

6.1.1 Chosen options for the hardware models explored

As mentioned earlier in Chapter 2, with hls4ml it's possible to try different options for hardware implementation. Just as a reminder, it's possible to modify:

- The target clock.
- The data precision.
- The architecture type (parallelized or pipelined).
- The strategy to use (resource or latency) for the model and layers.
- The reuse factors for the model and for each layer.

There were 29 versions developed during this phase, as mentioned in Chapter 4. Some of these ended up using more than the available resources of the ZedBoard. For this reason, two boards were used: the ZedBoard (with part number: *xc7z020clg484-1*) and the ZCU106 (with part number: *xczu7ev-ffvc1156-2-e*).

Additionally, it should be mentioned that all versions shared the following:

- Parallel architecture (option chosen: *io_parallel*).
- Strategy chosen for the model and layer interfaces: Resource.
- Reuse factor for the first layer interface: 30.

- Reuse factor for the first layer interface: 160.

Therefore, during this phase, only options related to data precision and target clocks were explored. Furthermore, it should be explained that the previous list uses the word *interface* because hls4ml uses the mathematical representation of a neural network. In consequence, confusion should be avoided with representative diagrams.

Now, before presenting the results obtained for this part, as well as the different hardware models, it's relevant to discuss about the number of bits chosen for data representation. The next section focuses on this topic.

6.1.2 Bits used for data representation in hardware

As mentioned at the beginning of Chapter 5, the data used during model development with Python was type "float64" for the samples and "int64" for the labels.

Moreover, with hls4ml it's possible to define the number of integer and decimal positions, as well as the entire bit width of the data. For this reason, the following observations become relevant:

- The original samples had 6 decimal positions, could be positive or negative, and were smaller than 1 in absolute value (so between -1 and 1, both excluded).
- After preprocessing the data using StandardScaler, the preprocessed data samples had 8 decimal positions, could be positive or negative, and were smaller than 2 in absolute value (so between -2 and 2, both excluded)

Ultimately, it will be necessary to work with up to 8 decimal positions maximum. To be able to do this, at least 27 fractional bits are required. To make this number a multiple of two, then 28 fractional bits can be used.

Regarding the integer part of the data, 2 bits are enough. This is because 1 bit can be used for the absolute value of the number, and the remaining bit for the sign. Consequently, in total, 30 bits (in reality, 29) would be the minimum amount required to work with the NN model developed in hardware, at least regarding the samples.

In the case of the labels, it should be highlighted that the outputs of the multiclass model are values between 0 and 1 that are interpreted as probabilities. Therefore, considering this, the previous analysis should apply as well.

6.1.3 HLS hardware results

Table 6.1.4.1 and Table 6.1.4.2 show the results related to timing, obtained with Vivado HLS for the developed versions. For its part, Table 6.1.4.3 shows the results about resource usage, also obtained with Vivado HLS.

Table 6.1.4.1: Timing results of the hardware models after HLS (Part I)

Version	Target clock (ns)	Estimated (ns)	Uncertainty (ns)
1	25	19.936	3.12
2	20	16.681	2.50
3	15	12.947	1.88
4	10	8.457	1.25
5	5	4.375	0.62
6	3	3.853	0.38
7	4	3.962	0.50
8	20	17.172	2.50
9	20	10.722	2.50
10	15	10.722	1.88
11	10	8.705	1.25
12	5	4.324	0.62
13	4	3.500	0.50
14	3.50	3.062	0.44
15	3	2.625	0.38
16	2.50	2.188	0.31
17	2	1.711	0.25
18	20	14.422	2.50
19	20	17.172	2.50
20	25	13.120	3.12
21	20	11.744	2.50
22	15	11.744	1.88
23	10	8.097	1.25
24	5	3.783	0.62
25	3	2.625	0.38
26	2	1.711	0.25
27	1	1.170	0.12
28	1.50	1.312	0.19
29	1.50	1.312	0.19

Overall, results in Table 6.1.4.1 seem alright. However, there are cases, specifically versions: 6, 7, 13, 14, 15, 25, 27, 28 and 29, where either the estimated clock is bigger than the target clock, or the addition between the estimated clock and the uncertainty is bigger than the target clock.

Even though this might seem negative at the beginning, it really isn't since for these cases the target clock just acts as a point of reference. Ultimately, the timing restriction for this project is about latency, so trying new clock values is simply an exploratory activity. Therefore, even if the estimated clock is slightly bigger than the target one, the latency requirement could still be met.

Moreover, the lowest target corresponded to a clock period of 1 ns (Version 27), with an estimated clock period of 1.170 ns and an uncertainty value of 0.12 ns. The second fastest options were versions 28 and 29. These had target clocks with a period of 1.50 ns and estimated clock periods of 1.312 ns, with 0.19 ns of uncertainty. Also, there are other fast clocks in the table, with target periods of 2, 2.5, 3, 3.5, 4 and 5 ns.

On the other hand, the slowest target clock had a period of 25 ns (Version 1 and 20). All in all, then, even if there are several good options, which one to choose will depend on the latency values obtained and resource usage. Next, the latency results are shown in Table 6.1.4.2.

Table 6.1.4.2: Timing results of the hardware models after HLS (Part II)

Version	Min latency (us)	Max latency (us)
1	4.900	4.950
2	3.940	3.980
3	2.970	3.000
4	2.020	2.040
5	1.110	1.120
6	0.890	0.898
7	0.896	0.904
8	3.940	3.980
9	3.900	3.940
10	2.952	2.955
11	1.950	1.970
12	0.995	1.005
13	0.812	0.820
14	0.718	0.725
15	0.618	0.624
16	0.525	0.530
17	0.438	0.442
18	3.900	3.940
19	3.960	4.000
20	4.875	4.925
21	3.900	3.940
22	2.925	2.955
23	1.960	1.980
24	1.005	1.015
25	0.621	0.627
26	0.442	0.446
27	0.501	0.503
28	0.349	0.352
29	0.346	0.349

At this point, it's important to remember the latency constraint of 100 ms. Now, considering Table 6.1.4.2, it's better to consider the maximum latency ("Max latency") values and be as conservative as possible.

It can be seen in the table that the highest maximum latency value corresponds to 4.950 us (Version 1), which is equivalent to 0.00495 ms. On the contrary, the lowest maximum latency is 0.349 us (Version 29), which is equal to 0.000349 ms.

Naturally, all the minimum latency values are lower than the maximum latency values. Considering this, and what was described before, it can be stated that all the latency values are significantly smaller than the 100 ms restriction.

Therefore, which hardware model to choose at this point is more related to trying to find the best solution. This "best" solution should be the one that has, ideally, the fastest clock and smallest latency, and also that which uses the minimum required precision and minimum amount of resources.

However, resources, precisely, have not been analyzed yet. There were versions requiring to use more than the available resources of the board. Therefore, the resource usage estimation results from Vivado HLS are presented next in Table 6.1.4.3.

Table 6.1.4.3: Resource usage (%) of the hardware models after HLS

Version	BRAMs (%)	DSPs (%)	FFs (%)	LUTs (%)	URAMs (%)
1	54	75	16	86	0
2	54	75	15	86	0
3	54	75	16	93	0
4	54	75	23	92	0
5	54	75	42	93	0
6	54	75	47	92	0
7	54	75	43	95	0
8	94	303	28	105	0
9	42	38	6	24	0
10	42	38	6	24	0
11	42	38	6	24	0
12	42	38	9	25	0
13	42	38	10	25	0
14	42	38	16	27	0
15	42	38	16	27	0
16	42	38	18	27	0
17	42	38	19	24	0
18	85	144	12	44	0
19	191	1138	77	197	0
20	86	154	12	40	0
21	45	38	6	26	0
22	45	38	6	26	0
23	45	38	7	26	0
24	45	38	10	27	0
25	45	38	18	30	0
26	45	38	22	26	0
27	45	19	28	27	0
28	45	38	26	27	0
29	42	38	22	25	0

In Table 6.1.4.3, versions which utilized more than the available resources are marked in red. These versions are: 8, 18, 19, 20. The first one (8) was the first attempt at using the ap_fixed<30,2> precision, with the ZedBoard. Since the ZedBoard did not have enough resources, the ZCU106 was used instead.

Regarding options 18, 19 and 20, these were the only three models which tried to use a total bitwidth of 64 bits. Due to the results obtained while trying to use 64 bits, precision was lowered and kept at a total of 32 or less when continuing with the exploratory versions.

Now, it would be ideal to use as few resources as possible. From Table 6.1.4.3, it's possible to note that:

- BRAMs: the minimum usage is 42%, and the maximum is 86%.
  - The 86% corresponds to a version which uses more than the available resources.
  - Therefore, the highest usage %, which doesn't go past the available resource limit, is 54%.
- DSPs: the minimum usage is 19%, and the maximum is 75%.
- FFs: the minimum usage is 6%, and the maximum is 77%.
  - This 77% corresponds to a version which uses more than the available resources.
  - Therefore, the highest usage %, which doesn't go past the resource limit, is 47%.
- LUTs: the minimum usage is 24%, and the maximum is 93%.
- URAMs: all versions use 0%.

Chapter 7

Conclusions

In conclusion, it can be said that, considering all the work presented, results for the multiclass model are satisfactory despite the challenge of having a reduced amount of samples per class. Also, it meets the purpose of this thesis: specifically, a Machine Learning algorithm using samples from a Microwave Imaging system prototype was successfully developed to detect six different contaminants. In addition, it serves to reinforce the effectiveness and potential of combining MWI with ML for quality assurance.

Moreover, as shown in this text, the multiclass approach managed to solve the situation with the triangular plastic piece and also simplify the model architecture in the process. In particular, it decreased the error from 51.220% to 7.5% in the worst case, which corresponded to the confusion matrix of the validation set in Version L (Version L3).

Additionally, as reported in Chapter 6, estimated results from HLS indicate that the multiclass model can reach a minimum maximum latency of 0.000349 ms, with an estimated clock period of 1.312 ns. Instead, as mentioned in Chapter 1, the latency for the Binary Classifier was around 3 ms, with a 100 MHz clock. The HLS results are already faster than this final result.

However, the multiclass model developed in this thesis presented, as observed in Chapter 5, more significant confusion between the glass fragment and the metal sphere. It's thought that this could be due to insufficient data samples or similarities presented after data preprocessing.

Ultimately, the model developed in this thesis is not the real final version. This is because it could continue being developed in the future. Specifically, some ideas to develop in the future include:

- A hot-encoded version of the model, whose aim would be to compare performance in hardware.

- Integrate new hyperparameters for optimization, such as regularization parameters.

- Obtain more samples to enlarge the dataset and possibly retrain the model to observe the effect on performance (validation accuracy and classification metrics).

- Obtain samples in different positions in the jar while ensuring that all contaminant types have the sample amount of samples taken in the same positions.

- Go one step further by using another fluid instead of oil, such as chocolate spread or similar.

- Add new contaminant types.
- Once the model is more developed, try dynamic measurements as suggested by [1].

Before closing this thesis, there are three final aspects to discuss. The first is that the multiclass neural network developed offers companies the option to detect the specific contaminant type in a jar. As mentioned in Chapter 1, this allows a company to identify possible weak points in its quality control system if needed. In addition, it can also serve to obtain some statistics about the company's products.

The second aspect to emphasize can, in fact, be expressed with a question. This question is: what is a chocolate jar company really worried about? Is the true concern knowing the type of contaminant in a jar or whether the jar is contaminated at all? The answer really depends on what the company is *exactly* seeking to do with its quality control procedures, and to what extent.

Evidently, both are important questions but they will really depend on the needs and interests of the company as just said. If it's the first case, all models can be improved for more accuracy. In this sense, the multiclass neural network is no exception. If it's the second, collapsing the multiclass results to binary ones offers extremely good results, as shown in Chapter 5.

With this last observation comes the final aspect: what is worse for the company? A non contaminated jar that is predicted as contaminated, or vice versa? Which has a higher cost? A jar which is not contaminated, but predicted to be, can be thrown away (here, the economic loss is being ignored). However, the opposite case is more complicated.

This last point is truly important. It's the reason why the problem was originally a binary classification task. However, by changing the approach of the model and using multiclass instead, the binary performance was improved. Therefore, the more this initial multiclass NN is developed, from an algorithmic and/or data point of view, the better the performance will be both for its multiclass and binary results. This way, in the future, a model like this might even be launched in industry at a specific company.

Bibliography

[1] L. Urbinati. «Detection of food contaminants with Microwave Sensing and Machine Learning». MSc thesis in Electronics Engineering. MA thesis. Politecnico di Torino, Italy, Dec. 2019 (cit. on pp. 1, 2, 4, 42–45, 49–52, 54, 58, 62, 63, 81, 99, 100, 119).

[2] A. Géron. *Hands-On Machine Learning with Scikit-Learn, Keras &TensorFlow. Concepts, Tools, and Techniques to Build Intelligent Systems.* Sebastopol, CA: O'Reilly Media Inc, Sep. 2019, 2nd Ed. (Cit. on pp. 4, 5, 7, 8, 10–12, 14–18, 21, 26, 27, 29–36, 40, 52, 66, 69, 81).

[3] *sklearn.model$_s$election.GridSearchCV*. `https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html`. Accessed: March 3rd, 2020 (cit. on pp. 7, 59, 66).

[4] *sklearn.model$_s$election.RandomizedSearchCV*. `https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.RandomizedSearchCV.html`. Accessed: March 3rd, 2020 (cit. on pp. 7, 59).

[5] *What is underfitting and overfitting in machine learning and how to deal with it.* `https://medium.com/greyatom/what-is-underfitting-and-overfitting-in-machine-learning-and-how-to-deal-with-it-6803a989c76`. Accessed: March 22nd, 2020 (cit. on p. 13).

[6] A. Ng. *Machine Learning.* `https://www.coursera.org/learn/machine-learning`. Accessed: November 1st, 2019 (cit. on pp. 13, 15, 16, 18, 21, 29, 30, 32, 33, 39).

[7] *Usage of optimizers.* `https://keras.io/optimizers/`. Accessed: March 3rd, 2020 (cit. on pp. 18, 35, 58, 60, 93).

[8] *Metrics to Evaluate your Machine Learning Algorithm.* `https://towardsdatascience.com/metrics-to-evaluate-your-machine-learning-algorithm-f10ba6e38234`. Accessed: March 22nd, 2020 (cit. on p. 18).

[9] *Multi-Class Metrics Made Simple, Part I: Precision and Recall.* `https://towardsdatascience.com/multi-class-metrics-made-simple-part-i-precision-and-recall-9250280bddc2`. Accessed: March 12th, 2020 (cit. on pp. 19–23, 80).

[10] *Confusion matrix.* `https://scikit-learn.org/stable/auto_examples/model_selection/plot_confusion_matrix.html`. Accessed: March 12th, 2020 (cit. on pp. 23, 24, 80).

[11] *Multi-Class Metrics Made Simple, Part II: the F1-score.* `https://towardsdatascience.com/multi-class-metrics-made-simple-part-ii-the-f1-score-ebe8b2c2ca1`. Accessed: March 23rd, 2020 (cit. on pp. 24, 25).

[12] A. Géron. *Chapter 1. Introduction to Artificial Neural Networks.* `https://www.oreilly.com/library/view/neural-networks-and/9781492037354/ch01.html`. Accessed: March 23rd, 2020 (cit. on pp. 26–28).

[13] *CHAPTER 5: Why are deep neural networks hard to train?* `http://neuralnetworksanddeeplearning.com/chap5.html`. Accessed: March 24th, 2020 (cit. on p. 34).

[14] *sklearn.model$_s$election.cross_val_score*. `https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.cross_val_score.html`. Accessed: March 2nd, 2020 (cit. on pp. 37, 56).

[15] *Cross-validation: evaluating estimator performance*. `https://scikit-learn.org/stable/modules/cross_validation.html`. Accessed: March 4th, 2020 (cit. on pp. 37, 38, 64).

[16] *sklearn.model_selection.StratifiedKFold*. `https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.StratifiedKFold.html`. Accessed: March 2nd, 2020 (cit. on pp. 38, 57).

[17] *sklearn.preprocessing.StandardScaler*. `https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html`. Accessed: March 2nd, 2020 (cit. on pp. 39, 40, 54).

[18] *sklearn.preprocessing.MinMaxScaler*. `https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html`. Accessed: March 19th, 2020 (cit. on pp. 40, 52).

[19] *Model class API*. `https://keras.io/models/model/`. Accessed: March 5th, 2020 (cit. on pp. 41, 63, 66, 67, 69, 72).

[20] *sklearn.model$_s$election.cross$_v$al$_p$redict*. `https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.cross_val_predict.html`. Accessed: March 5th, 2020 (cit. on pp. 41, 67).

[21] *sklearn.datasets.load_files*. `https://scikit-learn.org/stable/modules/generated/sklearn.datasets.load_files.html`. Accessed: March 2nd, 2020 (cit. on p. 53).

[22] *numpy.savez*. `https://docs.scipy.org/doc/numpy/reference/generated/numpy.savez.html`. Accessed: March 2nd, 2020 (cit. on p. 53).

[23] *sklearn.model_selection.train_testsplit*. `https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html`. Accessed: March 2nd, 2020 (cit. on p. 53).

[24] *sklearn.model_selection.KFold*. `https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.KFold.html`. Accessed: March 2nd, 2020 (cit. on p. 56).

[25] *Metrics and scoring: quantifying the quality of predictions*. `https://scikit-learn.org/stable/modules/model_evaluation.html`. Accessed: March 4th, 2020 (cit. on p. 64).

[26] *Usage of callbacks*. `https://keras.io/callbacks/`. Accessed: March 5th, 2020 (cit. on p. 69).

[27] *Wrappers for the Scikit-Learn API*. `https://keras.io/scikit-learn-api/`. Accessed: March 5th, 2020 (cit. on p. 69).