INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

MESTRADO EM ENGENHARIA INFORMÁTICA

isep

# Framework para a configuração e treino de redes neuronais utilizando otimização Bayesiana

**TIAGO FILIPE ALVES DA SILVA**

Outubro de 2020

POLITÉCNICO
DO PORTO

Instituto Superior de Engenharia do Porto

# Framework for the configuration and training of neural networks using Bayesian Optimization

**Tiago Filipe Alves da Silva**

**Dissertation for the attainment of a Master's Degree in Computer Science, Specialization Area in Information and Knowledge Systems**

**Supervisor: Isabel Praça**

**Co-supervisor: Luis Gomes**

Porto, October 2020

*To everyone that has kept me in the path leading to this stage in my life, thank you.*

# Resumo

Redes neuronais existem há décadas, tendo sido primeiramente introduzidas nos anos 40 por dois cientistas que modelaram uma simples rede neuronal usando circuitos elétricos. Desde então, vários avanços têm sido feitos no campo de redes neuronais com o objetivo de as adaptar na resolução de tarefas cada vez mais complexas, por sua vez levando a que as suas arquiteturas se tornem gradualmente mais elaboradas. Esta progressão tem dificultado a melhoria da qualidade de redes neuronais por parte de utilizadores, visto haver cada vez mais hiperparâmetros (i.e. componentes arquiteturais) que requerem ajustes na tentativa de melhorarem a sua precisão.

A otimização de hiperparâmetros de uma rede neuronal é feita ajustando os mesmos de maneira a encontrar a arquitetura com os melhores resultados, podendo ser feita de forma tentativa erro, e guiada por algoritmos que o facilitem. Esta tese enquadra-se neste tema, apresentado uma solução que utiliza otimização Bayesiana como o algoritmo de otimização de hiperparâmetros para automaticamente configurar qualquer tipo de rede neuronal. O sistema desenvolvido não só otimiza os hiperparâmetros de redes neuronais, mas também localiza as caraterísticas mais relevantes de um conjunto de dados (também conhecido como seleção de caraterísticas) e aprende como cada hiperparâmetro e caraterística afeta o desempenho da rede, tornando-o útil na previsão do desempenho de uma configuração de uma rede neuronal sem sequer ter que a treinar e testar.

Os resultados observados na avaliação do sistema demonstram as suas fortes capacidades de aprendizagem e a sua habilidade de balancear a exploração de configurações com elevadas chances de ter um desempenho alto com a exploração de configurações menos familiares com um nível de desempenho mais imprevisível, de forma a evitar contentar-se com uma configuração suficientemente boa e tentar encontrar aquela com precisão máxima. Tanto o caso de estudo como a otimização de uma rede neuronal convolucional realizados demonstram a capacidade de adaptação do sistema a diferentes tipos de redes neuronais e de obtenção de resultados positivos em ambos os cenários. A avaliação do sistema demonstra o potencial do mesmo e com desenvolvimentos futuros poderá atingir um nível de qualidade e desempenho onde será capaz de encontrar configurações que superem aquelas provenientes tanto de abordagens manuais e automáticas existentes.

**Palavras-chave**: Redes Neuronais; Otimização de Hiperparâmetros; Seleção de Caraterísticas.

# Abstract

Neural networks have existed for decades, having first been introduced in the 1940s by two scientists modelling a simple neural network using electrical circuits. Since then, many advancements have been made in the field of neural networks with the intention of adapting them to solve increasingly more complex tasks, in turn leading to neural networks architectures gradually becoming more intricate. This progression has made it harder for users to improve the quality of neural networks, as there are ever more hyperparameters (i.e. architecture components) that require tweaking in an attempt to increase their accuracy.

In an attempt to overcome this issue, the concept of hyperparameters optimization emerged, where each hyperparameter of a neural network is adjusted manually or automatically by a system, so as to find the network architecture with the best results. This thesis delves into this subject, presenting a solution that employs Bayesian optimization as its hyperparameters optimization algorithm to automatically configure any type of neural network. The developed system not only optimizes the hyperparameters of neural networks, but it can also pinpoint the most relevant features in a dataset (also known as feature selection) and learn how each hyperparameter and feature affects the performance of the network, making it useful for predicting the performance of a neural network configuration without even having to train and test it in the first place.

The results observed in the evaluation of the system showcase its strong learning capabilities and its ability to balance the exploitation of configurations with an elevated chance of having a high performance and the exploration of unknown configurations with an unpredictable level of performance, in an attempt to avoid settling for a good enough configuration and find the best one. Both the undertaken case study and optimization of a convolutional neural network demonstrate the system's ability to adapt to different types of neural networks and obtain positive results in both scenarios. The system's evaluation demonstrates it has potential and with future work can reach a level of quality and performance where it can find configurations that surpass those of both existing automatic and manual approaches.

**Keywords**: Neural Networks; Hyperparameters Optimization; Feature Selection.

# Acknowledgments

I would first like thank my university, Instituto Superior de Engenharia do Porto, for providing me with the opportunity for pursuing a Master's in Computer Science. This degree has enabled me to extend my knowledge in fields such as software engineering, artificial intelligence, and information systems, and has also led me to work on and write the thesis here presented.

I would also like to thank both my supervisors, Isabel Praça and Luis Gomes, for guiding me throughout this stage of my life and ensuring the success of my thesis. Their constant supervision and feedback were vital in guaranteeing the utmost quality of the thesis. A special thanks goes out to Professor Luis, as the thesis would not be what it is now was it not for his continuous availability and assistance with any questions I might have had or any issue that I experienced. Furthermore, his support was critical in ensuring the writing of this document was as insightful as it was gripping.

I would also like to thank my girlfriend for always being there for me whenever I needed her the most, for keeping me motivated and encouraged, and for being a constant source of inspiration for what I should strive to be. Her emotional support was indispensable during the entire course of the thesis.

Last but not least, I would like to thank my family and friends for their determination and perseverance in keeping me focused on the thesis and assuring I worked as hard as possible to achieve something I could be proud of.

# Table of Contents


**1 Introduction ........ 1**

1.1 Background ........ 1

1.2 Problem ........ 2

1.3 Objectives ........ 2

1.4 Expected Outcomes ........ 3

1.5 Advocated Approach ........ 4

1.6 Report Structure ........ 5

**2 Context ........ 7**

2.1 Neural Networks ........ 7

2.2 Problem ........ 8

2.3 Value Analysis ........ 11

**3 State of the Art ........ 13**

3.1 Optimization Algorithms ........ 13
- 3.1.1 Grid Search ........ 13
- 3.1.2 Random Search ........ 14
- 3.1.3 Bayesian Optimization ........ 15
- 3.1.4 Genetic Algorithms ........ 16
- 3.1.5 Particle Swarm Optimization ........ 17
- 3.1.6 Bat Algorithm ........ 18

3.2 Researched Solutions ........ 20
- 3.2.1 Random Search for Hyper-Parameter Optimization of Neural Networks ........ 20
- 3.2.2 Algorithms for Hyper-Parameter Optimization of Deep Belief Networks ........ 21
- 3.2.3 Automatic Configuration of Deep Neural Networks ........ 23

**4 Design ........ 25**

4.1 Optimization Algorithm Analysis ........ 25
- 4.1.1 Surrogate Model ........ 27
- 4.1.2 Acquisition Function ........ 28
- 4.1.3 Outliers ........ 29

4.2 System Flow ........ 32

4.3 Configuration Search Space ........ 33

4.4 Parallelization of Configurations' Evaluation ........ 34

**5 Implementation ........ 37**

5.1 Technologies ........ 37

5.2 Human-computer Interaction ........................................................38
5.3 Search Space........................................................................39
5.4 Objective Function..................................................................40
5.5 Scalability..........................................................................40

**6 Evaluation............................................................................41**

6.1 Methodology .........................................................................41
6.2 Metrics..............................................................................43
6.3 Case Study - Detection of Sensor Vibrations ...........................................44
6.3.1 Datasets ..........................................................................44
6.3.2 Neural Network Structure and Search Space .........................................46
6.3.3 Human Fall Classification .........................................................47
6.3.4 Vibration Source Classification ...................................................51
6.4 Hyperparameters Optimization of Convolutional Neural Network ...........................55
6.4.1 Dataset ...........................................................................56
6.4.2 Convolutional Neural Network ......................................................56
6.4.3 Neural Network Structure and Search Space .........................................58
6.4.4 Analysis ..........................................................................59

**7 Conclusions ..........................................................................65**

7.1 Goals Accomplishment ................................................................65
7.2 Future Work..........................................................................67
7.3 Final Appreciation ..................................................................67

# List of Figures

Figure 1 – Thesis methodology. .......... 5
Figure 2 – Architecture of a neural network (Bre, et al., 2017) and structure of a neural network's neuron (Zhou, 2019). .......... 8
Figure 3 – Total number of configurations versus the number of features, exemplified using the static possible values of the three hyperparameters in Table 1. .......... 10
Figure 4 – Grid search of nine different configurations with two hyperparameters (yellow and green areas) (Bergstra & Bengio, 2012). .......... 14
Figure 5 - Random search of nine different configurations with two hyperparameters (yellow and green areas) (Bergstra & Bengio, 2012). .......... 15
Figure 6 – Example of the Bayesian optimization process at two different stages: on the left, after 2 configurations evaluations; on the right, after 8 configurations evaluations (Koehrsen, 2018). .......... 16
Figure 7 – Flow of a genetic algorithm (Saeed, 2017). .......... 17
Figure 8 – Flow of a bat algorithm. .......... 20
Figure 9 – An example regression with outliers present: on the left, using a Gaussian model; on the right, using a Student-t model (Vanhatalo, et al., 2009). .......... 30
Figure 10 – Flow of the system's optimization process. .......... 32
Figure 11 – Parallelization of evaluation of configurations: on the left, performed by the system; on the right, performed by the user. .......... 34
Figure 12 – Evaluation flow of a neural network configuration according to the thesis' evaluation methodology. .......... 42
Figure 13 – System performance results, throughout 3000 iterations, given different values for the trade-off parameter: 0, 4, 8, and 12, respectively, on the top left, top right, bottom left, and bottom right corners. Configurations deemed outliers by the system in the final iteration are not present. .......... 48
Figure 14 – Distribution of loss residuals and standard deviation of the system's predictions in all four tests for the last 500 iterations, outliers included. .......... 49
Figure 15 – Number of inliers and outliers in each of the four performed tests, according to the last iteration of each test. .......... 51
Figure 16 – System performance results, throughout 3000 iterations, given different values for the trade-off parameter: 4 and 8, respectively, on the left and right. Configurations deemed outliers by the system in the final iteration are not present. .......... 52
Figure 17 - Distribution of loss residuals and standard deviation of the system's predictions in both tests for the last 500 iterations, outliers included. .......... 53
Figure 18 - Number of inliers and outliers in the two performed tests, according to the last iteration of each test. .......... 55
Figure 19 – Handwritten digit images from the MNIST dataset (Lecun, et al., 1998). .......... 56
Figure 20 – Example structure of a CNN (Phung & Rhee, 2019). .......... 57
Figure 21 – System performance results, throughout 500 iterations, for a trade-off parameter of 1.3. Configurations deemed outliers by the system in the final iteration are not present. .......... 60

# List of Tables

Table 1 – Total number of possible configurations of an example neural network. .................. 9
Table 2 – Test set classification error of the best NN configuration found by each solution (Bergstra, et al., 2011)........................................................................................................ 22
Table 3 – Test set classification error of the best NN configuration in (Stein, et al., 2018) on the MNIST dataset, compared with other manually configured networks. .................................. 23
Table 4 – Test set accuracy of the best NN configuration in (Stein, et al., 2018) on the CIFAR-10 dataset, compared with other manually configured networks. ............................................... 23
Table 5 – Example configuration search space containing all possible data types. ................. 39
Table 6 – Example configuration for a given iteration. ............................................................ 40
Table 7 – Features of sensor vibrations' datasets.................................................................... 45
Table 8 – Search space of the vibrations' case study............................................................... 47
Table 9 – The 10 configurations found with the lowest loss, across all four trade-off parameter system tests, sorted by loss. ........................................................................................... 50
Table 10 - The 10 configurations found with the lowest loss, across both trade-off parameter system tests, sorted by loss. ........................................................................................... 54
Table 11 - Search space of the CNN optimization.................................................................... 59
Table 12 - The 10 configurations found with the lowest loss, sorted by loss. .......................... 61
Table 13 – Comparison of the thesis' system's best configuration with the best configuration of other HPO systems. ........................................................................................................ 62
Table 14 – Comparison of the thesis' system's best configuration with manually-configured neural networks. ........................................................................................................... 62
Table 15 – Level of accomplishment of the thesis' system's goals.......................................... 66

# List of Acronyms

| Acronym | Meaning |
|---|---|
| ANN | Artificial Neural Network |
| BO | Bayesian Optimization |
| CNN | Convolutional Neural Network |
| DBN | Deep Belief Network |
| GA | Genetic Algorithm |
| GECAD | Research Group on Intelligent Engineering and Computing for Advanced Innovation Development |
| GP | Gaussian Processes |
| HPO | Hyperparameters Optimization |
| ISEP | Instituto Superior de Engenharia do Porto |
| MNIST | Modified National Institute of Standards and Technology |
| NN | Neural Network |
| PI | Probability of Improvement |
| PSO | Particle Swarm Optimization |
| RNN | Recurrent Neural Network |
| TMDEI | Thesis / Master’s / Dissertation |
| TOP | Trade-off Parameter |
| TPE | Tree of Parzen Estimators |

# List of Symbols

| Acronym | Meaning |
|---|---|
| $\lambda$ | Set of hyperparameter values of a given NN configuration |
| $\Psi$ | HPO response function |
| $\Lambda$ | NN configuration space |

# 1 Introduction

This section introduces the thesis by first giving a brief overview of the history of neural networks and the problem this thesis contributes to, followed by a more detailed description of said problem. Following that, it lists the goals of the thesis, the expected outcomes, and the initial advocated approach in order to solve the identified problem. Lastly, the structure of this document is presented.

## 1.1 Background

Neural networks (more specifically, artificial neural networks) are systems inspired by the behavior of biological neural networks. Their history can be traced back to 1943, when a neurophysiologist named Warren McCulloch and a mathematician named Walter Pitts modelled a simple neural network using electrical circuits to describe how neurons in the brain may work (McCulloch & Pitts, 1943). Since then, neural networks (NNs) have involved into intricate structures of hundreds, thousands, or even millions of neurons, all working together to solve very complex tasks, such as detecting road lanes in self-driving car systems (McCall & Trivedi, 2006) and predicting Parkinson's disease in medical patients (Sadek, et al., 2019).

However, with the increase in complexity of neural networks over the years came the increase in difficulty to configure them. As the architecture of a NN becomes more intricate, users spend longer periods of time tweaking it in order to attempt to increase its performance. With NNs oftentimes having millions of possible different configurations, it becomes unfeasible and costly for users to experiment every single one.

In an attempt to solve this issue, the concept of hyperparameters optimization (HPO) emerged. The idea behind this technique is the automation of the configuration of neural networks using optimization algorithms, with a system having the capability of making informed decisions on what configurations to evaluate. Throughout the optimization process, the system tweaks the hyperparameters (i.e. architectural components) of the NN with the

intention of improving the network's results. The process by which the system decides what hyperparameters to tweak is dependent on the employed optimization algorithm.

## 1.2 Problem

Neural networks have been increasingly employed throughout recent years as tools for the identification and understanding of patterns and classification of data, having an autonomous reasoning capability in the resolution of the tasks for which they are trained. Algorithms of this kind do not possess a "hard-coded" logic; instead, they develop their own way of thinking throughout their training process.

Two of the most crucial tasks undertaken by developers when creating these types of systems are the configuration of their hyperparameters (e.g. activation functions, number of layers, layer types, etc.) and the selection of the most relevant attributes of the given dataset (also known as feature selection). Despite the vitality and importance of these processes, they have to be performed manually and are extremely time-consuming due to the semi-random trial-and-error approach taken by users in order to figure out which attributes and parameters provide the network with the best results (Stein, et al., 2018).

By automating this entire process, it can be performed in a more deliberate and knowledgeable way. An HPO system can keep track of every change it makes to every hyperparameter alongside the network performance that configuration led to, and then use that information to make new changes in the hyperparameters that will lead to improved network results. Moreover, if the system also has feature selection capabilities, it can choose to ignore certain features that it believes to either be irrelevant or detrimental towards the performance of the NN.

The thesis here presented, developed in the scope of the Thesis / Dissertation / Master's (TMDEI) class of the Master's in Computer Science at the Instituto Superior de Engenharia do Porto (ISEP), delves into the subjects of hyperparameters optimization and feature selection in neural networks. Proposed by two professors part of ISEP's GECAD (Research Group on Intelligent Engineering and Computing for Advanced Innovation Development) research center, the project consists in the implementation of a framework to automate the configuration of neural networks.

## 1.3 Objectives

The main goal of this thesis is to design and develop a software solution to tackle the time-consuming process of manually experimenting different configurations of a neural network. The system should automate the configuration and evaluation process of neural networks, thus alleviating end users from having to perform this process manually.

In order to accomplish this, it will be necessary to:

- Understand what a neural network is and how it functions;
- Explore the different types of existing neural networks and understand what differentiates them, the use cases for each, and the advantages and disadvantages of each one;
- Research problems of different natures solved through the usage of neural networks, in an attempt to try and find patterns in the configuration process of the networks and the feature selection process of the input data;
- Investigate already existing solutions for the automation of the configuration of neural networks, analyzing employed techniques, system performance and quality of generated networks;
- Apply the acquired knowledge in the design and architecture for the solution, adaptable to any use case and easily extensible to accommodate future additions;
- Implement the actual system based on the design and architecture previously specified;
- Incrementally evaluate and improve the final solution, reducing its time complexity and increasing its overall performance.

## 1.4 Expected Outcomes

Based on the objectives set out for the thesis and its system's development, the final version of the implemented framework is expected to have the ensuing capabilities:

- It should be able to take into account the user's data and the context of the problem at hand and find a neural network optimal for it. The obtained NN should perform comparably to a NN obtained through a manual process by a user;
- It should be able to create and configure multiple different types of neural networks, such as Convolutional Neural Networks (CNN) and Recurrent Neural Networks (RNN);
- It should be designed and implemented in a way where future extensibility (such as the ability to support other machine learning algorithms) is straightforward and simple;
- Due to the sheer complexity of iterating through and evaluating networks with different configurations until an optimal one is obtained, it is expected for the framework to operate for a long period of time[1]. With this in mind, it is also expected that the framework's time complexity is optimized in order to try and minimize this issue as much as possible.

[1] This will depend a lot on the problem to be solved, the type of, quality and quantity of the input data, the specification of the machine where the framework will operate on, among other factors.

## 1.5 Advocated Approach

In order to develop a system to automatically configure neural networks, a state-of-the-art study must be performed so as to investigate, evaluate and compare several different algorithms and techniques, both as a core part of the system's implementation, and as a way to optimize the system itself and reduce its complexity. This research will mainly focus on the core piece of the system, its optimization algorithm, delving into the various existing approaches, such as evolutionary and genetic algorithms, applied to both the context of hyperparameters optimization as well as other types of optimization. It will majorly consist of books and journal articles of the past ten years (i.e. between 2010 and 2020), as both neural networks and optimization algorithms are fast moving areas where it is vital to have the most recent research possible.

The system will then be designed in a way that conforms to the performed study, along with its expected behavior and characteristics (e.g. versatility and extensibility). It should adapt to any use case and be capable of optimizing the architecture of any type of neural network, with an optimization algorithm that can learn the most, the quickest, and have mechanisms that can lower the time complexity of computing the quality of a given NN configuration. Additionally, the system should have a mechanism to safeguard itself and its learning process against certain edge cases, such as data outliers and configurations with unexpected results.

The implementation will be closely guided by the design and how it is expected to be employed by a user, focusing on hiding its complexity from users and only exposing the features they need to tweak the system (such as its learning behavior, the configurations and data features to evaluate, and the criteria in charge of considering the optimization process finished). The user will have the freedom and control over how to specify each of the parameters the system will optimize and what the values of each of these parameters should look like.

Lastly, the entire system will undergo a series of experimental tests, where its performance will be evaluated in different case studies. These tests will deal with real world data and different types of neural networks, ensuring the system can learn the architectural patterns that make up the best performing NNs and obtain them. Not only that, but the feature selection capabilities of the system will also be assessed, to find out whether it can pinpoint the most relevant features of a dataset and the most redundant ones.

The entire methodology can be seen summarized by the process in the following figure:

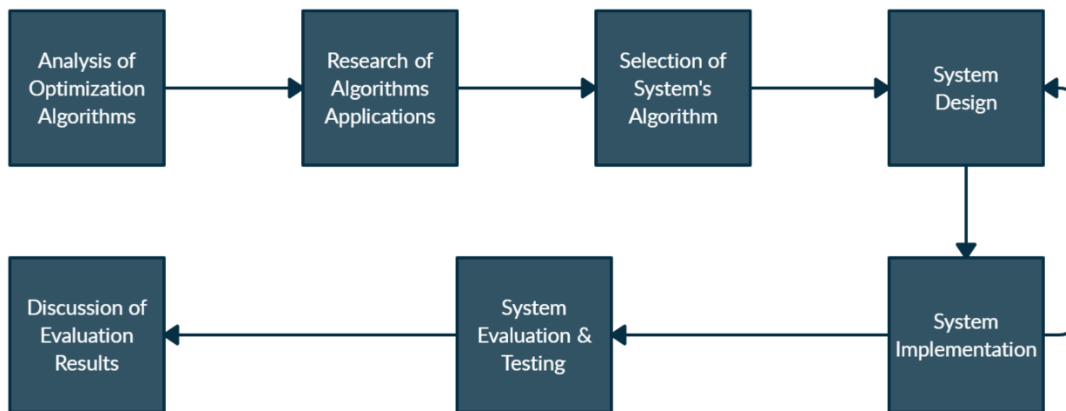


Figure 1 – Thesis methodology.

## 1.6 Report Structure

Section 1 starts off with the introduction of the thesis, where both the background behind neural networks and the optimization of hyperparameters are described, followed by a brief summarization of the thesis' problem. Afterwards, both the objectives of the thesis as well as its expected outcomes are listed, finalizing with an explanation of the advocated approach for the thesis.

Section 2 goes into a more thorough and deeper examination of the context behind the thesis. First, key basic concepts vital in understanding the entirety of the thesis, such as machine learning and neural networks, are clarified to the reader. Subsequently, the thesis' problem is explained in greater detail, emphasizing the existing struggles in the manual configuration of NNs, while also exposing the complications which arise from trying to automate this process. As a last point in this section, the business value of the thesis' system and the benefits of it to users is examined.

Section 3 presents the conducted state of the art, commencing with the undergone market study of existing optimization algorithms, where algorithms such as Bayesian optimization and particle swarm optimization are analyzed. Following that, various different researched solutions are discussed and compared where optimization algorithms are used in the optimization of hyperparameters in NNs.

Section 4 depicts the design of the system, starting off with the chosen optimization algorithm, Bayesian optimization, and going in depth over its main components: the surrogate model and the acquisition function. The flow of the entire system is also presented through a diagram, followed by the discussion on how the configurations to use in the optimization process should be specified. Lastly, the idea of parallelizing the evaluation of multiple configurations at the same time is deliberated.

Section 5 begins by listing the technologies employed in the development of the system, as well as how the human-computer interaction component of the system was handled. An overview of how the search space will work in the system follows, together with an explanation of the system's concept of objective function. Ending the section is an analysis of the scalability of the system.

Section 6 goes into the evaluation of the system, beginning with the methodology followed in the conduction of the system's experiments and the key metrics considered in the assessment of the system's performance. Afterwards, a case study on a dataset of vibrations detected by motion sensors is presented, with the system having to optimize a neural network in charge of understanding the cause of each vibration. Lastly, the system is put against other hyperparameters optimization systems and manually configured networks in the optimization of a convolutional neural network.

Section 7 presents the conclusions of the thesis, first listing its goals, and whether these were accomplished or not, and then delving into ideas for future work and development in the system. To finish off, a final appreciation over the entire thesis is made.

# 2 Context

This section introduces the reader to some fundamental concepts necessary to better understand the area in which this thesis will dive into, such as machine learning and neural networks. Following that, the problem of the thesis is explained in greater detail, together with the highlighting of a few key points that elucidate the difficulties of implementing a solution to the problem. Lastly, a value analysis on the project is presented, where both the benefits and drawbacks of the system are underlined.

## 2.1 Neural Networks

Machine learning, a process which neural networks are a part of, can be described as *"an application of artificial intelligence (AI) that provides systems the ability to automatically learn and improve from experience without being explicitly programmed"* (Expert System Team, 2017). Machine learning can contain multiple types of learning algorithms, such as supervised, unsupervised, and reinforcement learning, and each of these can include several different machine learning models, such as ANNs, decision trees, and support vector machines.

Neural networks are composed of multiple interconnected units called neurons that pass around information since they are given an input, until an output (i.e. prediction) is eventually obtained. Each neuron receives one or more inputs, performs various mathematical operations on it (depending on the type of neuron), and produces an output (Zhou, 2019) (see Figure 2). Groups of neurons are then put together in what are called layers, with the first layer being known as the input layer (where the input data is fed), the final layer as the output layer (where the prediction(s) are obtained), and any layer in between these two (assuming any exist) known as a hidden layer (see Figure 2).

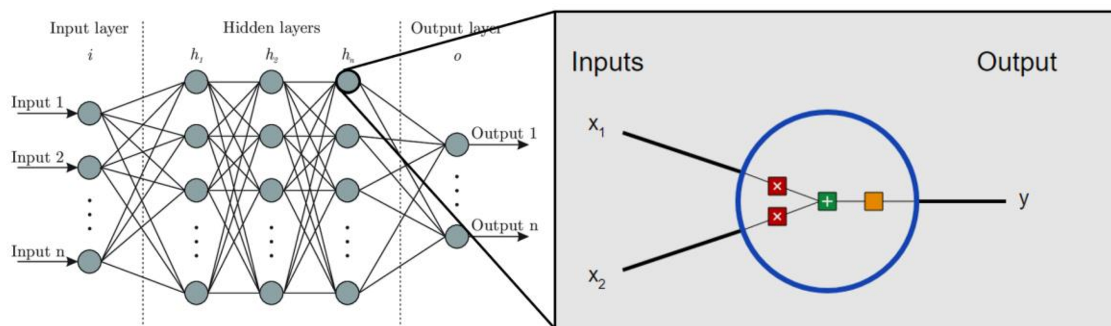


Figure 2 – Architecture of a neural network (Bre, et al., 2017) and structure of a neural network's neuron (Zhou, 2019).

Each neuron also attributes a weight to each one of its inputs, thus controlling how strongly neurons affects one another (aside from the input layer, every neuron's input must be another neuron's output). The weights of every neuron in a neural network are also known as the network's parameters, and they are the source behind its learning process: every weight starts out as a randomized value, and throughout the training process of the network, the error of the network's predictions is calculated (also known as a cost / loss function), with each weight getting slightly adjusted towards a value that will minimize the loss of the network and, therefore, improve its results (DeepAI, n.d.).

Just like a NN's parameters are a critical part of its thought process, its hyperparameters are also a critical part of its learning process. Examples of a network's hyperparameters are the number of hidden layers, the number of neurons per hidden layer, the network's learning rate, etc. Unlike the parameters of a NN, though, which are automatically tweaked throughout its training process to improve its accuracy, hyperparameters tend to be manually tweaked by users due to several complications brought about by attempting to automate it (see following section 2.2).

It is also relevant to talk about deep learning, as it will be the area where this system's benefits will be the most noticeable. Deep learning is a subset of machine learning comprised of neural networks with more complex architectures, constituted by multiple hidden layers meant to learn representations of data with multiple levels of abstraction. These are crucial in certain areas where datasets tend to be much more complex and difficult to understand, such as in speech recognition, natural language processing, and computer vision (LeCun, et al., 2015). The reason why the system will be the most useful in deep learning NNs is because these tend to have a much higher number of hyperparameters to configure, in comparison to NNs with simpler architectures.

## 2.2 Problem

The implementation of a neural network is a highly complex and laborious process, with users expending a vast amount of time since the initial network's design, until a final solution with

an expected optimal performance is obtained. This systematic procedure can be classified into the following steps:

1. Define the network's architecture (i.e. its hyperparameters), according to the problem it is meant to solve;
2. Implement the NN;
3. Train and test the network with the architecture initially specified;
4. Tweak the hyperparameters of the network;
5. Repeat steps three and four until the NN reaches optimal results.

The issue with this approach is the amount of time developers spend in step four, where they semi-randomly fine-tune each hyperparameter without having a good idea on what the results after those changes are going to be. By making the smallest change, the network may drastically improve its results, or it may worsen its performance significantly, or no change may even occur. It is a very time-consuming process with unpredictable outcomes. Not only that, but developers are not able to try out every single possible hyperparameter value, as there are too many combinations. Table 1 presents an example of a neural network with three hyperparameters, each having an arbitrary number of possible values, and a dataset containing four features, where each feature is either used or ignored by the NN. The hyperparameters and features equal a total of $(2 * 2 * 2 * 2 - 1) * 5 * 10 * 3 = 2250$ possible configurations (the combination of the features' possible values is decremented since the scenario with no features is not possible), a quantity rather impractical to be fully tested by developers. In real scenarios, however, it becomes even more complex, as datasets have more features and neural networks have dozens, or more, of hyperparameters with much larger ranges of values, resulting in millions of different possible configurations.

Table 1 – Total number of possible configurations of an example neural network.

| Features & Hyperparameters | Possible Values | Number of Possible Values |
|---|---|---|
| Feature 1 | Ignore or Do Not Ignore | 2 |
| Feature 2 | Ignore or Do Not Ignore | 2 |
| Feature 3 | Ignore or Do Not Ignore | 2 |
| Feature 4 | Ignore or Do Not Ignore | 2 |
| Number of Layers | From 1 to 5 | 5 |
| Number of Neurons per Layer | From 1 to 10 | 10 |
| Number of Epochs | From 1 to 3 | 3 |
| Total | | 2250 |

By keeping the same three hyperparameters in the above table with the exact same number of possible values, but varying the number of features in a dataset, the impact in the number of total configurations can be visually analyzed. The line chart in Figure 3 demonstrates the exponential increase in the total amount of configurations of a NN as the number of features increase. Whereas with one feature, there are 150 possible configurations, by ten features this value is upwards of 100 thousand (153450, to be exact).

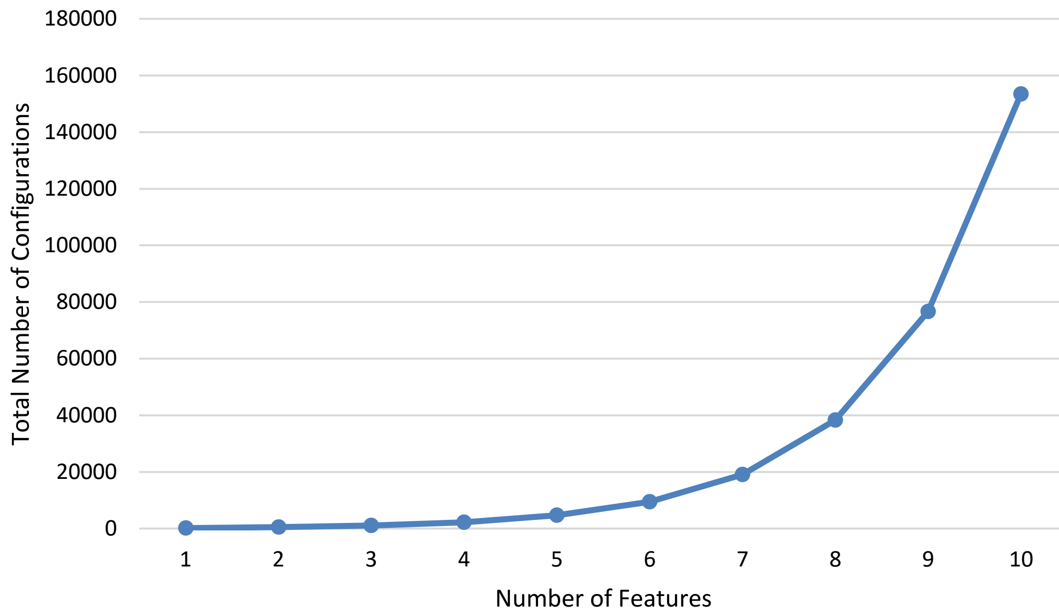


Figure 3 – Total number of configurations versus the number of features, exemplified using the static possible values of the three hyperparameters in Table 1.

The search automation of a NN's best hyperparameter values for a given dataset is a known subject in the area of machine learning known as hyperparameters optimization. This method can be expressed through the following equation:

$$\lambda^{(*)} = \underset{\lambda \in \Lambda}{argmin}\, \Psi(\lambda) \tag{1}$$

where we are trying to obtain a given set of hyperparameters $\lambda$, belonging to $\Lambda$, which minimize a given response function $\Psi$ (also referred as the objective function) that will be optimized by the system. The set $\Lambda$ (also known as the configuration space of the network) represents the array of hyperparameters $\{\lambda^{(1)} \ldots \lambda^{(S)}\}$ to be evaluated by the function $\Psi$, in order to obtain the hyperparameter values which provide the network with the lowest loss calculated by $\Psi$. Both the response function $\Psi$ and the set $\Lambda$ vary depending on the optimization algorithm and dataset used, as well as the tasks performed by the network over the dataset (Bergstra & Bengio, 2012).

Currently, the main obstacles with hyperparameter optimization are (Feurer & Hutter, 2019):

- Evaluations of the function $\Psi$ can be very time-and-resources-demanding, especially with more complex NNs and larger datasets;
- The search space[2] of a given hyperparameter can be extremely complex and high-dimensional;
- It is hard to know which hyperparameters require optimization and which ones do not;
- It is hard to know which hyperparameters are the most substantial in improving the network's results;
- It is not always possible to optimize the hyperparameters of a network through the usage of a cost function, like it is done in the training process of a NN.

## 2.3 Value Analysis

With the employment of a system like this, users will not have to spend a lot of their time manually tweaking hyperparameters of the network and re-training and re-testing it multiple times. The framework will automate the entire process, ensuring an optimal solution is eventually reached, thus freeing up users' time to work on other projects. This becomes even more obvious when it comes to deep learning networks, as these tend to be exponentially more complex to configure.

In view of the system's versatility, it can be employed in the automatic configuration of any type of neural networks, be it CNNs, RNNs, Feedforward NNs, etc. Furthermore, the system encourages experimentation, as it may try out network configurations that the user would never even consider testing.

Despite what is said in section 1.4 about the system being expected to run for long periods of time, it does not necessarily mean it will take longer than if the optimization process was performed manually by users. This is due to the fact that the system will employ metaheuristic techniques to predict the performance of not-yet-evaluated configurations, and subsequently use that knowledge to avoid evaluating configurations which it expects to have worse results than configurations that have already been evaluated. A user may not be capable of carrying out these assessments and end up spending a greater amount of time experimenting with worse-performing configurations.

[2] Domain of the hyperparameter being optimized (e.g. the number of hidden layers).

# 3 State of the Art

This section presents a state-of-the-art on existing algorithms and techniques used in the automatic configuration of neural networks. It starts off by summarizing and comparing multiple existing optimization algorithms that can be adapted to the optimization of hyperparameters, such as Bayesian optimization and genetic algorithms. Following that, various scientific publications are examined where systems with different optimization algorithms are employed in the automatic configuration of neural networks.

## 3.1 Optimization Algorithms

Throughout the years, many different optimization algorithms have been developed and employed in the hyperparameters optimization process of neural networks. This chapter introduces some of these existing algorithms and how each one of them operates.

### 3.1.1 Grid Search

Grid search is one of the most well-known hyperparameter optimization algorithms, consisting on the combination of every possible value of the search space of every hyperparameter (Bergstra & Bengio, 2012). As an example, if $A$ is the search space of a given hyperparameter, such that $A = \{1, 2\}$, and $B$ is the search space of another hyperparameter, such that $B = \{3, 4\}$, then, in accordance to eq. 1, $\Lambda = \{(1, 3); (1, 4); (2, 3); (2, 4)\}$. Figure 4 showcases how grid search would select nine configurations in an optimization process, only testing three distinct values on two hyperparameters (cf. Figure 5).

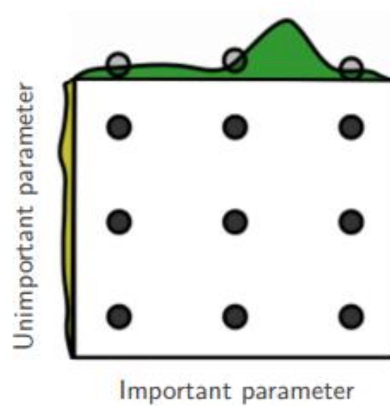


Figure 4 – Grid search of nine different configurations with two hyperparameters (yellow and green areas) (Bergstra & Bengio, 2012).

The simplicity of this algorithm comes with the cost of the *curse of dimensionality*, whereby the number of function evaluations exponentially increase with the dimensionality of the configuration space of the network (Feurer & Hutter, 2019). In other words, performing optimization using grid search becomes exponentially more expensive the more hyperparameters the network has and the larger the search space of each one is.

### 3.1.2 Random Search

Random search is an alternative to grid search which attempts to overcome its *curse of dimensionality* issue by randomly selecting configurations to evaluate, instead of evaluating every single possibility (Bergstra & Bengio, 2012). For example, if there was a search space $A = \{1, 2\}$ for hyperparameter $H_A$, and a search space $B = \{1, 2, 3, 4, 5\}$ for hyperparameter $H_B$, grid search would evaluate configurations sequentially, starting off by evaluating all the configurations where $H_A = 1,$ such as $(1, 1)$ and $(1, 2),$ and then all configurations where $H_A = 2,$ such as $(2, 1)$ and $(2, 2);$ random search, on the other hand, would evaluate configurations randomly, never selecting them in any specific order. Figure 5 showcases how random search would select nine configurations in an optimization process, testing an heterogenous range of values on two hyperparameters (cf. Figure 4).

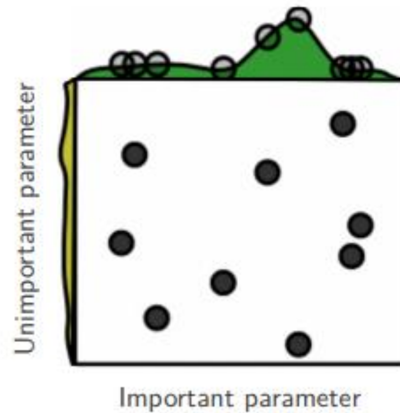


Figure 5 - Random search of nine different configurations with two hyperparameters (yellow and green areas) (Bergstra & Bengio, 2012).

Outside of that, it features most of grid search's characteristics, such as easy parallelization of the evaluation of different configurations (since each one is completely independent on the rest) and the need to specify the search space of every hyperparameter ahead of time.

### 3.1.3 Bayesian Optimization

Bayesian optimization (BO) algorithms avoid the complexity of calculating $\Psi$ by instead creating a surrogate function that approximates $\Psi$ and that is optimized and improved throughout the HPO process. This optimization process is accomplished on account of a history of past configurations evaluations maintained by the algorithm, allowing it to make informed choices on what hyperparameters to evaluate next based on past results (Hutter, et al., 2011).

Figure 6 demonstrates an example Bayesian optimization process at two different stages, with the dashed red line representing the real objective function of a given hyperparameter, the bold black line representing the surrogate model of the objective function, the black dots representing the results of evaluations made, and the grey area representing the uncertainty of the surrogate model. As can be seen, a BO algorithm optimizes its surrogate function, in every iteration, by "adding" the evaluation result of that iteration to it, slowly approximating it to the real objective function $\Psi$, whilst too lowering its uncertainty.

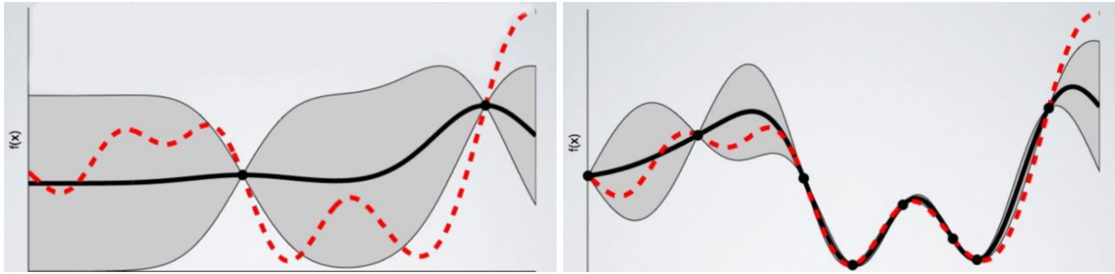


Figure 6 – Example of the Bayesian optimization process at two different stages: on the left, after 2 configurations evaluations; on the right, after 8 configurations evaluations (Koehrsen, 2018).

Bayesian optimization algorithms can have different implementations depending on two distinct aspects of the algorithm: how the surrogate function is built (e.g. Gaussian Processes (GP), Tree of Parzen Estimators (TPE), etc.); which criteria to use to select the next hyperparameters in each iteration of the process (i.e. acquisition function) (e.g. Probability of Improvement (PI), Expected Improvement (EI), etc.).

### 3.1.4 Genetic Algorithms

Genetic algorithms (GA) are a class of evolutionary algorithms pioneered in the 1960s and 1970s which, similarly to neural networks, take inspiration from biological processes. More specifically, GAs take inspiration from Darwin's theory of evolution, involving concepts such as natural selection, mutations and crossover (Yang, 2013).

Each solution to be evaluated by a GA is called an individual, and a group of individuals is called a population. Each individual possesses a chromosome representing the features of that individual. In the context of HPO, an individual would be considered a network configuration to be evaluated and its chromosome would be the hyperparameter values of that configuration. Each individual would also be part of a given population P, such that $P \subseteq \Lambda$.

In order to select the best individuals of a population, a fitness function is used to evaluate the performance of each one (this function is linked to the response function $\Psi$). The best individuals of each population are then added to a mating pool, where the higher the quality of an individual, the higher are the chances it is selected. In the selection process, multiple pairs of individuals are chosen to generate offsprings (i.e. children). Each offspring's chromosome will be a combination of its parents' chromosomes.

Since every offspring will always share a combination of its parents' characteristics, in order to introduce some randomness into the process, each offspring will suffer mutations too, where their chromosomes will suffer slight changes. The individuals obtained after the mutation process will then replace the previous generation as the new one.

Figure 7 shows a flowchart of a typical genetic algorithm. After a population is randomly initialized, its fitness is evaluated, with the best performing individuals being selected. These

individuals then undergo the crossover process to produce children, which, subsequently, have their chromosome mutated. These new individuals will compose the new population which will undergo the exact same process, until some given termination criteria are reached.

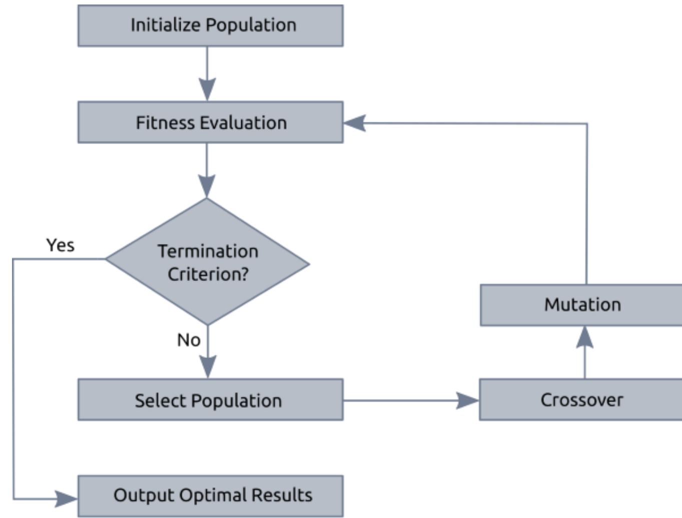


Figure 7 – Flow of a genetic algorithm (Saeed, 2017).

### 3.1.5 Particle Swarm Optimization

Particle swarm optimization (PSO) is an optimization algorithm introduced in the 1990s and inspired by the behavior observed in groups of social organisms, such as the coordinated flight of flocks of birds and the schooling of fish. PSO shares the concepts of individual and population also present in GA (also known as a particle and a swarm), where, iteratively, particles in a swarm move around in an attempt to find an optimal solution to a given problem (Martínez & Cao, 2019).

Each particle is defined by its current position and velocity—which stochastically change in every iteration—, in turn affecting its trajectory. The trajectory of each particle is also affected by the best position achieved by that particle and the swarm's best position. The position and velocity of each particle changes every iteration according to:

$$v_i^{t+1} = \omega v_i^t + c_1 r_1 \left(xBest_i - x_i^t\right) + c_2 r_2 \left(gBest - x_i^t\right) \tag{2}$$

$$x_i^{t+1} = x_i^t + v_i^{t+1} \tag{3}$$

where $x_i^t$ is the position of particle $i$ at iteration $t$, $v_i^t$ is the velocity of particle $i$ at iteration $t$, $xBest_i$ is the particle $i$'s best position, $gBest$ is the swarm's overall best position, and $\omega$, $c_1$, $c_2$, $r_1$, and $r_2$ are the inertia weight, two positive constants and two random parameters within [0, 1], respectively. The $\omega$, $c_1$ and $c_2$ parameters control the influence of different factors in the particle's velocity, with $\omega$ regulating the weight of the particle's previous

velocity, $c_1$ regulating the weight of the particle's best position, and $c_2$ regulating the weight of the swarm's best position. $r_1$ and $r_2$ are values selected randomly in every iteration and are meant to introduce randomness in the process and avoid particles converging to a local optimum.

### 3.1.6 Bat Algorithm

Bat algorithm (BA) is the most recent optimization algorithm here presented, having been introduced in 2010 by Xin-She Yang (Yang, 2010). Despite their blindness, through the mechanism of echolocation, bats are able to detect preys, avoid obstacles, and completely map out three-dimensional environments around them. They vary their echolocation pulses' frequency, loudness, and rate of emission in order to adapt to their surrounding environment and better perform tasks such as hunting. BA takes inspiration from this behavior of bats, in conjunction with other existing metaheuristic optimization algorithms, such as PSO and harmony search, establishing a novel population-based optimization algorithm.

In BA, the frequency, position, and velocity of each bat is updated in accordance with the following equation:

$$f_i = f_{min} + (f_{max} - f_{min})r_1 \tag{4}$$

$$v_i^{t+1} = v_i^t + (x_i^t - gBest^t)f_i \tag{5}$$

$$x_i^{t+1} = x_i^t + v_i^{t+1} \tag{6}$$

where $x_i^t$, $v_i^t$, and $r_1$ have the same meaning as in PSO's equations (eq. 2 and 3), $gBest^t$ is the bat population's best position at iteration $t$, $f_i$ is the pulse frequency of bat $i$, and $f_{min}$ and $f_{max}$ are the minimum and maximum frequencies allowed, respectively.

Up to this point, BA seems to follow a very similar logic as PSO. Where it starts to differentiate from it is through the concept of local search. It states that in every iteration, after the velocity and position of every bat is updated using the above equations, each bat should fly randomly. This random flight will involve two new parameters: the pulse emission rate $R_i^t$ and the loudness $A_i^t$. The local search will then be conducted either based on the current best solution or a randomly chosen one, depending on the bat's $R_i^t$, according to the following formula (Adarsh, et al., 2016):

$$x_i^{t+1,new} = \begin{cases} gBest^t + r_2 A_i^t, & r^3 > R_i^t \\ x_h^t + r_2 A_i^t, & otherwise \end{cases} \tag{7}$$

where $R_i^t$ is the pulse emission rate of bat $i$ at iteration $t$, $A_i^t$ is the loudness of bat $i$ at iteration $t$, $r_2$ is a random parameter within [-1, 1], $r_3$ is a random parameter within [0, 1], and $h$ is a random parameter within [1, 2, …, $N_b$], $h \neq i$ (where $N_b$ is the number of bats in the population), such that $x_h^t$ is the position of a bat in the population that is not bat $i$ at iteration $t$ picked randomly.

For each bat, it will be decided whether this new $x_i^{t+1,new}$ position or the previous $x_i^{t+1}$ position will be maintained according to:

$$x_i^{t+1} = x_i^{t+1,new} \Rightarrow r_4 < A_i^t \wedge \Psi\left(x_i^{t+1,new}\right) < \Psi\left(x_i^{t+1}\right) \quad (8)$$

where $r_4$ is a random parameter within [0, 1] and $\Psi$ is the objective function that is trying to be minimized in the HPO process. In case the "random walk" position of the bat becomes the bat's actual new position, the pulse emission rate and loudness of the bat will also be updated:

$$A_i^{t+1} = \alpha A_i^t \quad (9)$$

$$R_i^{t+1} = R_i^0[1 - \exp(-\gamma t)] \quad (10)$$

where $R_i^0$ is the initial pulse emission rate of bat $i$ at iteration $t$, and $\alpha$ and $\gamma$ are two positive constants. The initial loudness $A_i^0$ and pulse emission rate $R_i^0$ of a bat are randomly selected within [1, 2] and [0, 1], respectively.

Figure 8 summarizes the flow of a typical bat algorithm, starting off by defining the frequency and initial position, velocity, loudness, and emission rate of each bat. Following that, the position and velocity of each bat is updated, and each bat takes a walk starting off from the best bat's position or a random one, depending on a given condition. The new position of the bat is maintained if another given condition is met and if it is better than the previous bat position, in which case the bat's loudness and pulse emission rate are also updated. This process is repeated until some given termination criteria is reached.

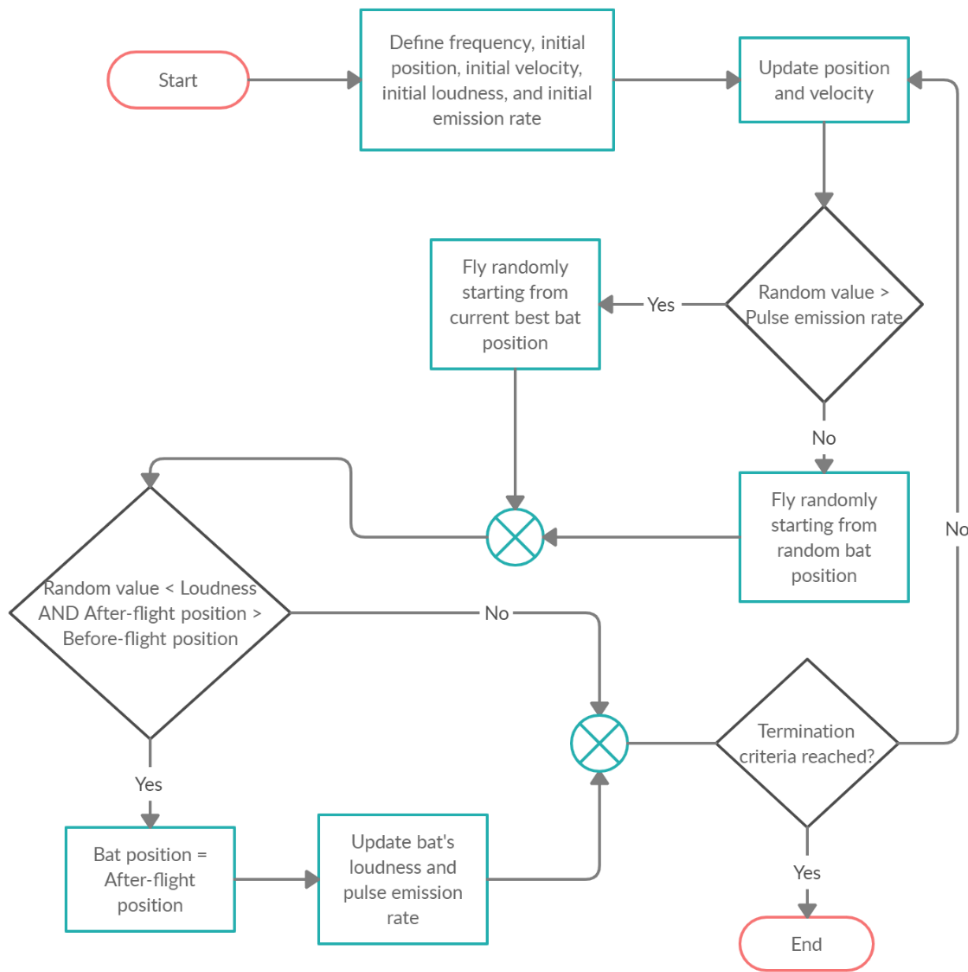


Figure 8 – Flow of a bat algorithm.

## 3.2 Researched Solutions

Applying the concepts behind the optimization algorithms presented in the previous section 3.1, numerous authors have designed and implemented their own systems to automatically configure neural networks. This section delves into various of these solutions, describing the technical details behind each one, their testing results, and how they fare against other previously developed systems.

### 3.2.1 Random Search for Hyper-Parameter Optimization of Neural Networks

In a study by Bergstra & Bengio, the authors implemented an automatic NN configurator using the random search algorithm (Bergstra & Bengio, 2012). The system was set up to configure seven hyperparameters on a one-layer NN, such as the type of input data preprocessing, number of hidden units, and learning rate of the network's stochastic gradient descent

optimization algorithm. The system was then compared to a grid search solution from (Larochelle, et al., 2007), in which, despite the hyperparameters to-be-optimized being different, the covered configuration search space was roughly the same size.

The tests were performed on eight different image datasets, five of which being the Modified National Institute of Standards and Technology (MNIST) dataset, a famous dataset of 70,000 28x28 greyscale images of handwritten digits,  and four other variants of it, such as with the images rotated or with random background images. For each dataset, the random search system was evaluated on varying numbers of trials (network configurations evaluations): 1, 2, 4, 8, 16, 32, and 64; and compared with the results of the grid search solution which, on average, ran for 100 trials.

Overall, random search managed to find a better network configuration than grid search at the end of eight configurations evaluations, less than one tenth the number of configurations tested by grid search. Even on test scenarios where random search had to perform more trials, it always managed to outperform grid search after reaching 64 evaluations.

The paper also presents experiments made using Gaussian processes to determine the relevance of each of the seven hyperparameters in the results and performance of the evaluated network configurations. Two important conclusions were reached through these experiments: only a small fraction of hyperparameters matter for any given dataset, and different hyperparameters matter on different datasets. These conclusions manage to better explain how the random search system managed to greatly outperform the grid search one with a much smaller number of trials: even though grid search evaluated more configurations, it did not evaluate certain important hyperparameters that random search did. This was due to the inherent grid search's limitations on the size of the configuration space of a network, as the algorithm puts the exact same weight on every hyperparameter and attempts to evaluate every possible configuration, a process that can take multiple days to finish.

### 3.2.2 Algorithms for Hyper-Parameter Optimization of Deep Belief Networks

In another study by the same two authors *et al* (Bergstra, et al., 2011)[3], grid search and random search were compared again in the automatic configuration of a Deep Belief Network (DBN) over six images datasets (all of them also used in (Bergstra & Bengio, 2012)). This random search system had to configure more hyperparameters than in (Bergstra & Bengio, 2012), including the number of hidden layers (between one and three), leading to a larger configuration search space.

The testing methodology was also very similar to (Bergstra & Bengio, 2012), with the implemented random search system being again compared with two other grid search solutions from (Larochelle, et al., 2007): a one-layer DBN and a three-layer DBN. The base

[3] Despite (Bergstra, et al., 2011) having been published the year before (Bergstra & Bengio, 2012), it is safe to assume it was written before (Bergstra & Bengio, 2012), as the authors constantly make reference to it. Chapter 3.2.2 was written with that assumption in mind.

configuration space of all three solutions was the same, with (Bergstra, et al., 2011) making some slight implementation changes which expanded its own configuration search space.

Tests results reveal that random search, unlike in (Bergstra & Bengio, 2012), did not manage to outperform grid search in every dataset—albeit, in most cases, it still managed to converge to a maximum after around 32 trials. In one of the datasets, even after the maximum allowed of 128 trials, random search never managed to obtain results as good as the three-layer grid search's best NN. This may suggest that the expanded search space of the random search system may not include configurations with improved performance.

Still in the same study (Bergstra, et al., 2011), two more systems are presented to configure a Multi-Layer Perceptron (MLP) on 10 hyperparameters: one with a BO algorithm using GP, and another using a BO algorithm too, but with Tree of Parzen Estimators instead. Both systems always started out with the first 30 configurations being randomly selected, after which the BO acquisition function took over the process of selecting the configurations to evaluate.

These two systems' results were compared with the random search solution previously introduced in this paper and the grid search solution from (Larochelle, et al., 2007) on two of the six datasets also used in the random search's earlier experiments. Each system was allowed to run for up to 200 trials. Each trial was executed on one of four different kinds of GPUs: NVIDIA GTX 285, GTX 470, GTX 480, and GTX 580; with a one-hour time limit per trial, independently of the GPU.

The results, seen in Table 2, showcase the two BO systems as being the top-performing solutions, with the system using TPE to build the surrogate function finding the configuration with the lowest classification error. These results reveal how the modelling approach of BO and the capability of selecting new configurations to evaluate based on past results can perform better than the brute-force approach of grid search or the random selection approach of random search.

Table 2 – Test set classification error of the best NN configuration found by each solution (Bergstra, et al., 2011).

| **Algorithm** | **Convex dataset** | **MRBI dataset** |
|---|---|---|
| BO w/ TPE | 14.13 ± 0.30% | 44.55 ± 0.44% |
| BO w/ GP | 16.70 ± 0.32% | 47.08 ± 0.44% |
| Grid Search | 18.63 ± 0.34% | 47.39 ± 0.44% |
| Random Search | 18.97 ± 0.34% | 50.52 ± 0.44% |

Time-complexity wise, both BO systems took about 24 hours to run, with up to five configurations being evaluated in parallel. By using the surrogate model of BO on the two systems to predict the performance of a given configuration, after 200 trials, the system with GP and the system with TPE were able to predict $\Psi$ in 150 and 10 seconds, respectively. These

represent very positive results as a configuration can usually take hours or even days to be evaluated.

### 3.2.3 Automatic Configuration of Deep Neural Networks

In (Stein, et al., 2018), a solution based on Bayesian optimization is presented, implemented using random forests to build the surrogate model and Moment-Generating Function (MGF) as the acquisition function. Additionally, the system uses parallelization in order to evaluate multiple different configurations at the same time, where, in every iteration, five configurations are evaluated in parallel using NVIDIA K80 GPUs.

The system is applied in the automatic configurations of CNNs and is tested on two very famous image datasets: MNIST and CIFAR-10, a dataset of 60,000 32x32 coloured images containing one of ten different objects, such as airplane, deer, or horse. For each of the two datasets, the system's results were compared with three other manually configured networks, as seen on Table 3 and Table 4.

Table 3 – Test set classification error of the best NN configuration in (Stein, et al., 2018) on the MNIST dataset, compared with other manually configured networks.

| Algorithm | Error | Epochs |
|---|---|---|
| (Ciresan, et al., 2012) | 0.23% | 800 |
| (Graham, 2014) | 0.32% | 250 |
| (Stein, et al., 2018) | 0.61% | 10 |
| (Yang, et al., 2015) | 0.71% | Unknown |

Table 4 – Test set accuracy of the best NN configuration in (Stein, et al., 2018) on the CIFAR-10 dataset, compared with other manually configured networks.

| Algorithm | Accuracy | Epochs |
|---|---|---|
| (Graham, 2014) | 95.59% | 250 |
| (Springenberg, et al., 2014) | 95.59% | 350 |
| (Stein, et al., 2018) | 86.46% | 50 |
| (Zeiler & Fergus, 2013) | 84.87% | 500 |

For both datasets, the system managed to find the optimal network configuration after approximately 50 evaluations. Despite the seemingly worse performance of the best network configuration found by the paper's solution, it is important to note that the evaluated configurations were only allowed to run up until a number of epochs drastically smaller than those of the manually configured NNs (10 and 300 epochs in the MNIST and CIFAR-10 datasets, respectively). This was done in order to speed up the optimization process of the system. The results suggest that, if allowed to run for a longer number of epochs, the system would have been able to find better performing configurations.

# 4 Design

After the research made in the previous section 3, and with the knowledge gathered through it, this chapter will now delve into the design of the system to be developed. It starts out by analysing the studied optimization algorithms, weighing their pros and cons, and selecting which one will be implemented in the system. It also delves into the two essential parts of the chosen algorithm, Bayesian optimization: the surrogate model, that will keep a history of evaluated configurations and build a model around it that best represents it; and the acquisition function, in charge of, at every iteration, picking the configuration to be evaluated next that it believes will provide the best results. The final point relating to the optimization algorithm will be outliers, how these can strongly influence the surrogate model and how the system is going to handle them as to ensure the model does not behave erratically due to them.

The succeeding section presents the designated flow of the system, from the moment its optimization process begins, until it finishes. Afterwards, two different approaches are discussed on the definition of the configurations search space, together with each one's strongest and weakest points. As the final design component of the system, an examination on the parallelization of the evaluation of network configurations is made.

## 4.1 Optimization Algorithm Analysis

The core component of the system presented in this thesis is the HPO algorithm, as it will be in charge of selecting the configurations to be evaluated based on certain criteria that differ in each algorithm. This can be a very intricate task as it is important for the algorithm to balance exploration and exploitation, a trade-off between the system evaluating configurations similar to past ones that have delivered positive results (i.e. exploitation), and the system attempting to try out novel configurations, in the hope of finding even better performing ones (i.e. exploration) (Berger-Tal, et al., 2014).

This dilemma leads straight into the grid search and random search optimization algorithms, which were some of the first optimization algorithms employed in the HPO process and that, of those studied, are the simplest to implement too. Unfortunately, none of these two algorithms have the aforementioned capability of balancing exploration and exploitation, as neither one keeps track of past configurations evaluations to make informed decisions on selecting new configurations. Grid search employs a brute-force approach, in which, once the configuration space is defined, every single configuration in it is evaluated. This approach easily becomes impractical in the real world as the dimensionality of the configuration space increases (i.e. more hyperparameters with more possible values are added to a NN). This grid search drawback (described in chapter 3.1.1) becomes an even bigger predicament in the system to-be-implemented in the thesis, which is expected to be employed by users for any kind of dataset and type of neural network and, as such, no assumptions can be made about the dimensionality of the configuration space. Adding to this, the systems analysed in chapters 3.2.1 and 3.2.2 demonstrate how easily other optimization algorithms can outperform grid search in a shorter amount of time.

Looking at random search, despite it solving the *curse of dimensionality* issue of grid search by randomly selecting configurations to evaluate, instead of evaluating every single one, it still suffers from not having the ability to learn from past configurations evaluations and use that knowledge to select new candidates that have a higher chance of performing well. Since configurations are always randomly selected, the results of the system can often be unpredictable, where running the system on the exact same dataset with the exact same configuration space can lead to varying results every time.

Unlike grid search and random search, population-based optimization algorithms (GA, PSO, and BA), along with the BO algorithm, take into consideration past trials in order to select new configurations to evaluate. Moreover, they inherently possess exploration versus exploitation mechanisms that can be tweaked in order to better adapt them to each network configuration scenario. Through this, they are able to make smarter and more knowledgeable decisions upon what configurations could have the most potential at any given point.

As the last point of consideration, the elevated time-complexity of HPO can be pinpointed to the evaluation of the response function $\Psi$. Calculating this function involves the training and testing of a neural network from scratch with a given set of hyperparameters $\lambda$, a process that can sometimes take hours or even days to finish. This issue is still one of the current biggest complications of the automatic configuration of NNs hindering it from being employed in the real world more often (albeit manual configuration of NNs also suffers from this). The BO algorithm possesses a mechanism that helps deter this problem by building a surrogate model of $\Psi$ which is much easier and faster to calculate than $\Psi$ (more details about this on section 3.1.3). Using the surrogate model, BO can avoid having to resort to $\Psi$ to evaluate a given configuration as much as possible, thus drastically reducing the time taken by the system to find an optimal configuration and allowing it to run more trials without as much of a time penalty.

Having all of these advantages and disadvantages in mind, in addition to the versatility and intelligent decision-making skills of the BO algorithm and the promising results observed in 3.2.2 and 3.2.3, it was adopted as the HPO algorithm of the thesis system.

### 4.1.1 Surrogate Model

For a given set of points $\{ Y(x) \mid x \in X \}$, indexed by a set $X$, there is a possibly limitless amount of functions that could describe the distribution of these points. The surrogate model of the system, Gaussian Processes, attempts to solve this problem by assigning a given probability to each of these functions in order to try and find the one that best describes the dataset. It achieves this by extending a multivariate Gaussian distribution, which is specified by a mean vector and a covariance matrix, to an infinitely dimensional Gaussian distribution, specified instead by a mean function and a covariance (also known as kernel) function (Ebden, 2015) (see eq. 11). In the given implementation of the system, the mean function was defined as 0 for any value of $x$—as GPs are able to model the mean arbitrarily well (Krasser, 2018)—(see eq. (12)) and the covariance function was defined as the square exponential kernel (see eq. (13)).

$$f \sim GP(\mu, k) \tag{11}$$

$$\mu(x) \equiv 0 \; \forall \; x \tag{12}$$

$$k(x, x') = \exp\left(-\frac{d(x,x')^2}{2l^2}\right) \tag{13}$$

where $l$ is the length scale of the kernel and $d(x, x')$ is the Eucledian distance between points $x$ and $x'$.

Considering $T$ as the training data (configurations evaluated so far) and $T_*$ as the testing data (configurations yet to be evaluated), the three presented equations can be used: to define the *prior* distribution $P_{T_*}$, used to make predictions before any training data has yet to be seen; the posterior distribution $P_{T_*|T}$, used to make predictions based on already evaluated configurations (Görtler, et al., 2019). To calculate the posterior, one must first look at the joint distribution $P_{T_*,T}$:

$$P_{T_*,T} = \begin{bmatrix} T \\ T_* \end{bmatrix} \sim N\left(\begin{bmatrix} \mu \\ \mu_* \end{bmatrix}, \begin{bmatrix} \mathrm{K} & \mathrm{K}_* \\ \mathrm{K}_*^T & \mathrm{K}_{**} \end{bmatrix}\right) \tag{14}$$

where $\mu$ and $\mu_*$ stand for the means of the training and testing data, respectively, $K$ and $\mathrm{K}_{**}$ are the covariance matrices for the training and testing data, respectively, and $\mathrm{K}_*$ and $\mathrm{K}_*^T$ are the covariance matrices between the training and testing data, normal and transposed, respectively. Knowing the value for $T$, one can calculate $P_{T_*|T}$ (i.e. the posterior distribution) using:

$$P_{T_*|T} \sim N(\mu_* + K_*^T K^{-1}(T - \mu), K_{**} - K_*^T K^{-1} K_*) \tag{15}$$

According to eq. 12, since the mean is considered to be 0 for every configuration, eqs. 14 and 15 can thus be simplified, respectively, to:

$$P_{T_*,T} = \begin{bmatrix} T \\ T_* \end{bmatrix} \sim N\left(0, \begin{bmatrix} \mathrm{K} & \mathrm{K}_* \\ \mathrm{K}_*^T & \mathrm{K}_{**} \end{bmatrix}\right) \tag{16}$$

$$P_{T_*|T} \sim N(K_*^T K^{-1} T, K_{**} - K_*^T K^{-1} K_*) \tag{17}$$

### 4.1.2 Acquisition Function

The goal of an acquisition function is to select the next configuration to evaluate on every iteration according to select criteria. The chosen acquisition function for the system, Probability of Improvement, estimates the probability of improvement for a given configuration by calculating the probability (between zero and one) that it will perform better than the best configuration obtained thus far. It uses the surrogate function to predict, according to the knowledge of past configurations evaluated, what the configuration's result value will be and how certain it is of it (represented by the respective standard deviation). Its formula is as follows (MathWorks, n.d.):

$$PI(x, Q) = \Phi\left(\frac{\mu_Q(x_{best}) - \mu_Q(x)}{\sigma_Q(x)}\right) \tag{20}$$

where $Q$ is the posterior distribution function of the surrogate model (in our case, according to eq. 17, $P_{T_*|T}$), $x$ is the configuration, $x_{best}$ is the best configuration found so far, $\mu_Q$ and $\sigma_Q$ are the posterior mean and standard deviation, respectively, of the configuration, and $\Phi$ is the unit normal cumulative distribution function.

For the main thesis' solution, the metric to be used by both the surrogate model and the acquisition function as the one to be optimized will be the loss (also known as the error or cost) of a neural network with a given configuration. As such, the optimization problem at hand is one of minimization, and the best configuration will be considered to be the one with the lowest loss and the one with highest probability of improvement as the one with the highest chances of having a smaller loss. This is important to mention, as eq. 20 showcases the minimization version of the PI formula, not the maximization one.

As mentioned in section 4.1, one of the biggest advantages of Bayesian Optimization over other optimization algorithms is its ability to balance exploration and exploitation of configurations over time. The acquisition function of the BO algorithm is the one in charge of balancing this mechanism in a manner that best optimizes results and increases the chances of finding the global optimum. The formula for the acquisition function of the system presented in the previous section (eq. 20) currently has no such mechanism, being purely exploitational.

The idea of using a variable called the trade-off parameter (TOP) to balance out exploration versus exploitation in the PI acquisition function was first introduced in (Kushner, 1964). Since then, many other authors have explored the importance of this parameter in various different

domains (Törn & Zilinskas, 1989) (Jones, 2001) (Lizotte, 2008). With the introduction of the parameter, eq. 20 of the PI function becomes:

$$PI(x, Q, t) = \Phi\left(\frac{\mu_Q(x_{best}) - \mu_Q(x) + \xi(t)}{\sigma_Q(x)}\right) \tag{21}$$

where $t$ is the current iteration of the optimization process, and $\xi(t)$ is the trade-off parameter, whose value will depend on the iteration. As $\xi(t) \rightarrow +\infty$, the acquisition function prioritizes configurations with higher posterior standard deviation, thus encouraging exploration. Conversely, as $\xi(t) \rightarrow 0$, the acquisition function prioritizes configurations with higher posterior mean, thus encouraging exploitation. Thus, the parameter should be adapted depending on the user's preferences.

In (Kushner, 1964), Kushner suggests tweaking this value over time, starting off with it quite high, to encourage the exploration of regions of higher interest, and to decrease it throughout the optimization process, in order to slowly search more the regions of interest previously explored and converge to the best value. It does not, however, provide an algorithm for how one could achieve this. For the thesis, the approach taken was to linearly decrement the value of the TOP at every iteration, so it reaches 0 at the last one. Thus, the formula is:

$$\xi(t) = \xi(t-1) - \frac{\xi(0)}{n} \tag{22}$$

where $n$ is the number of iterations the system will run for, and $\xi(0)$ is the initial value of the trade-off parameter. The importance of $\xi(0)$ is explored in section 6.3, where experimental system tests are performed with different TOP values to evaluate the influence it has over the entire system's behaviour.

### 4.1.3 Outliers

Gaussian Processes works with the expectation that every variable involved in the optimization process follows a normal distribution which, when joined together, form a multivariate normal distribution (Vanhatalo, et al., 2009). The problem with this expectation is its non-robustness, as a single outlier can drastically reduce the accuracy of the model when making predictions. This issue can be observed in Figure 9 (a), where the black line represents the real function, the blue line represents the surrogate model, the red dashed line represents the standard deviation of the surrogate model, and the blue dots represent the sample data points.

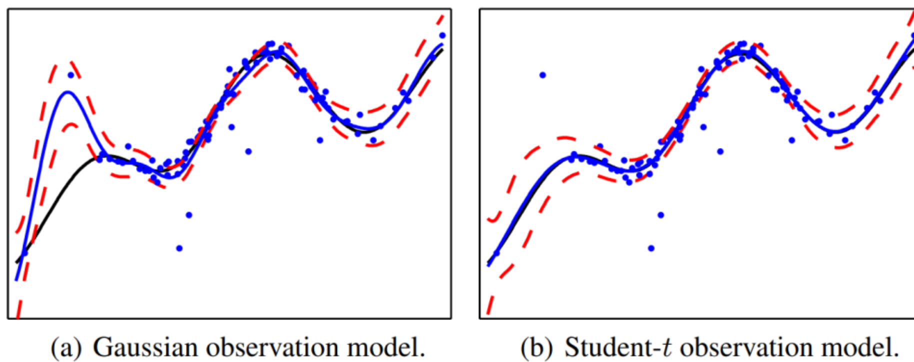


Figure 9 – An example regression with outliers present: on the left, using a Gaussian model; on the right, using a Student-t model (Vanhatalo, et al., 2009).

In (Martinez-Cantin, et al., 2017), the authors mention two ways to handle outliers: robustness of inference to outliers, which consists on developing models which are capable of including outliers without allowing them to dominate non-outlier data; outlier diagnostics, consisting on analyzing the data for any anomalies and excluding them, ensuring the surrogate model is built only on standard data. Robustness of inference to outliers tends to be more computationally expensive as the surrogate model must have extra logic in order to handle outliers, whereas with outlier diagnostics the model can be kept as is and the discovery and removal of outliers can be done separately, which tends to be faster to perform.

Both (Xia, 2017) and (Vanhatalo, et al., 2009) demonstrate approaches taken on the issue of handling outliers based on model robustness. In (Xia, 2017), a Student-t Process is used in place of a Gaussian Process, which is similar but uses student-t distributions instead of normal distributions, capable of fitting outliers without skewing the model completely towards them (see Figure 9, (b)). In (Vanhatalo, et al., 2009), a modified version of a Gaussian Process is presented that employs student-t likelihood. On the other hand, (Martinez-Cantin, et al., 2017) describes an approach based on detecting and removing outliers before fitting the data on the surrogate model. It fits the data on a GP with student-t likelihood, similar to (Vanhatalo, et al., 2009), but instead of using that as the real surrogate model, it uses it to find outliers, removes them, and then the outlier-free data is fitted on the real ordinary GP model. This approach is quicker as the real surrogate model does not have to be robust to outliers, leading to faster predictions.

Martinez-Cantin *et al* paired up their system against two other robust systems similar to (Xia, 2017) and (Vanhatalo, et al., 2009) in four separate experiments, with their system coming out on top in every experiment, performing almost as well as when there were no outliers in the data. This was most likely due to the fact that their surrogate model did not have adapt to the existence of outliers (due to these having been removed beforehand) and, as such, managed to perform more accurate predictions and, thus, get closer to a global optimum.

Unfortunately, (Martinez-Cantin, et al., 2017) does not provide the technical details of the algorithm used on their system. As such, the thesis will employ an outlier diagnostics approach, similar to (Martinez-Cantin, et al., 2017), but using a Grubb's test instead.

The Grubb's test is a statistical test introduced by Frank Grubbs in 1950 (Grubbs, 1950) that detects outliers in a dataset originating from a normal distribution. It tests a null hypothesis that a dataset has no outliers versus an alternative hypothesis that one outlier is present in the dataset. It detects outliers one at a time, retesting the entire dataset every time it finds an outlier, until no more outliers are detected. The two-sided version of Grubb's test was employed, which checks whether the point furthest away from the mean (eq. 23) is an outlier or not (eq. 24). If eq. 24 is proven to be true, the null hypothesis is rejected, and the value is considered an outlier.

$$G = \max_{i=1,\ldots,N} \frac{|X_i - \bar{X}|}{s} \tag{23}$$

$$G > \frac{N-1}{N} \sqrt{\frac{v^2}{N-2+v^2}}, \quad v = t_{\frac{\alpha}{2N}, N-2} \tag{24}$$

where $N$ is the number of observations in the dataset, $\bar{X}$ and $s$ are the mean and standard deviation of the dataset, respectively, $\alpha$ is the significance level, and $v$ is the upper critical value of the student-t distribution with significance level $\frac{\alpha}{2N}$ and $N-2$ degrees of freedom.

Due to a lack of sufficient data, Grubb's test, like other outlier tests, can be extremely sensitive with few data points, frequently classifying most of them as outliers, and, as such, is not advised to be used in the first few iterations. Taking that into consideration, like in (Martinez-Cantin, et al., 2017), the diagnostic of outliers is not executed for the first ten iterations. Unlike (Martinez-Cantin, et al., 2017), however, the diagnostics mechanism is ran on every iteration from then onwards, instead of only every two iterations, as it is cheap enough to do so, leading to a more performant surrogate model.

Given that the configurations of the neural networks in the search space are generated inside a predictable and controlled environment—based on the interval of hyperparameter values specified by the user—, there will never be noisy samples and, as such, will not need to be checked for outliers. The outliers detection-and-removal process is only applied to the objective value of the surrogate model (i.e. the loss of a neural network with a given configuration).

Similarly to (Martinez-Cantin, et al., 2017), no outlier is ever permanently removed from the data history. Every data point, at every iteration, has the chance to be reclassified as either an outlier or an inlier and, consequently, be removed or added to the surrogate model, respectively. As the system evaluates more configurations, its judgement changes on which configurations it considers to be outliers and which ones it does not.

## 4.2 System Flow

In view of the previous section 4.1, mainly the selected optimization algorithm and outlier handling mechanism, the flowchart seen on Figure 10 was devised, showcasing the logic and flow of the system's optimization process.

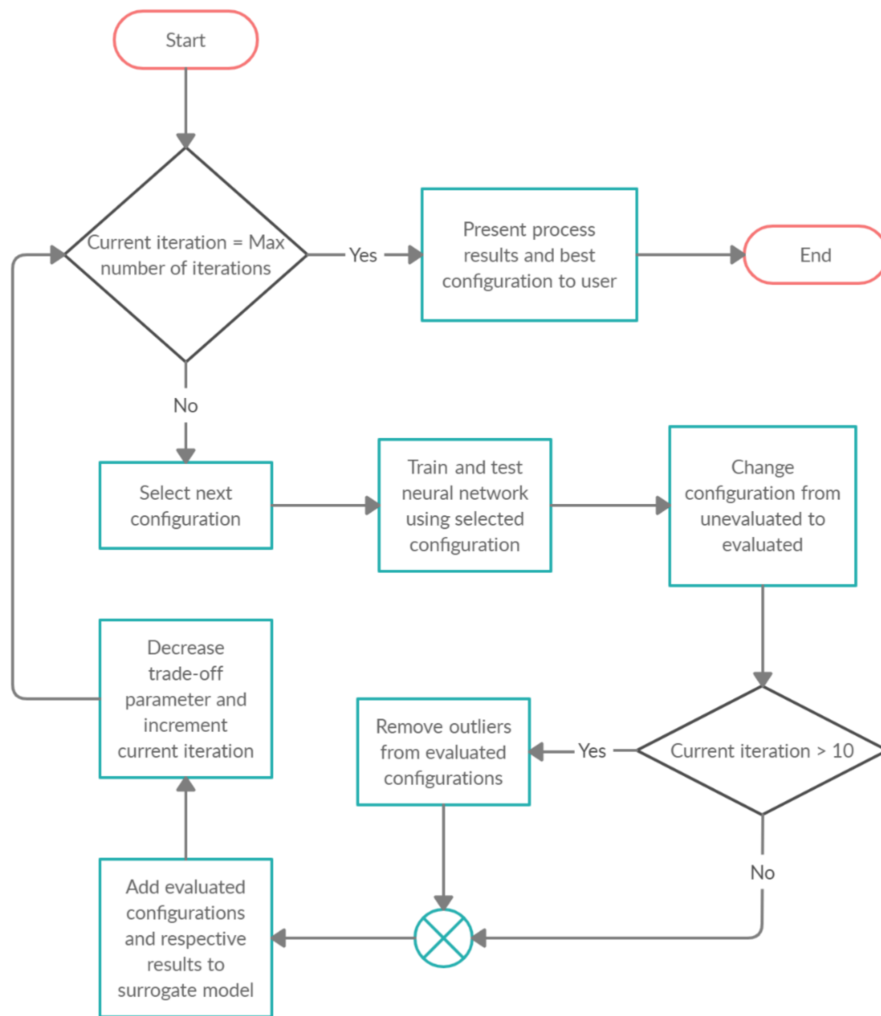


Figure 10 – Flow of the system's optimization process.

The system starts by checking if it has reached the maximum number of iterations (the designed stopping criterion) it is mean to run for and, if so, immediately presents the optimization process' results to the user, together with the best neural network configuration it found. However, if the current iteration is not the last one, it will use the acquisition function to select the next configuration, train and test a neural network using the given configuration, and mark said configuration as evaluated. If the system has undergone at least ten iterations, it will detect outliers in the previously evaluated configurations and remove them before adding the evaluated configurations to the surrogate model. Finally, it decreases the value of the trade-off parameter and redoes the entire process again for the next iteration.

## 4.3 Configuration Search Space

Every studied solution ( (Larochelle, et al., 2007), (Bergstra, et al., 2011), (Bergstra & Bengio, 2012), (Stein, et al., 2018)) took the same approach when defining the configuration search space of a neural network: the authors defined it directly in their system themselves. The definition of the search space came from research made by the authors or from the authors' own past experiences solving problems with similar datasets and/or similar neural networks. This approach comes with a few complications for the thesis system:

1. **The configuration search space has to be manually defined by the system's developer for every NN type –** Given that the thesis system is expected to be easily adaptable for any type of neural network, requiring the developer to first define the configuration search space directly in the system can become an obstruction to this. In order to manually define the search space, the developer must first do a lot of research on the type of neural network in order to know what are the key hyperparameters of the network and their respective search spaces. Not only is this a very time-consuming process, especially if the developer does not already possess some knowledge on the network type, but it is also not very versatile. This is due to the fact that even if the developer ends up defining a broad and suitable search space, chances are, it will not work for every single dataset. As proven in (Bergstra & Bengio, 2012) (see section 3.2.1), different hyperparameters have varying degrees of importance depending on the dataset in question. As such, manually implementing a "one size fits all" configuration search space for every type of NN is not a feasible choice;
2. **The system's user has no control over the configuration search space –** Since the developer is the one in charge of specifying the configuration search space in the system, the user will not be able to modify it. This is not necessarily an issue for the researched articles, as the solutions presented in them were implemented for scientific experimentation purposes, and not meant to be used directly by anyone rather than the authors, but it is for the thesis system, as it is meant to be open to any user. Even if the user has interest in expanding or shrinking the search space of a hyperparameter, add and/or remove hyperparameters, or change how the search space is explored by the system, they will be constrained to how all of this was defined by the developer.

With these points in mind, a different approach will be taken to describe the configuration search space of the system: users will specify it themselves. The biggest drawback of this approach is that users will still be involved in the process of configuring a neural network, whereas instead of directly tweaking hyperparameters values and manually re-training and re-testing the network, they will instead have to specify the search spaces of each hyperparameter. This disadvantage is also the method's most significant benefit: users have complete control over what the configuration search space of the system will be. Not only that, but this approach also ensures the system can be more easily expanded to other neural

network types, as the developer will not have to worry about deeply researching the subject and having to come up with a search space that will most likely not suit every use case.

## 4.4 Parallelization of Configurations' Evaluation

By default, BO (and other optimization algorithms) evaluate network configuration sequentially. This means that even if the system has multiple GPUs and/or CPUs at their disposal, it will only use one at a time to evaluate a given configuration. In the case of BO, evaluating configurations through a sequential manner ensures that there will be the most feedback about previous evaluations' results when selecting new configurations. On the other hand, parallelizing this process reduces the time taken by the system and allows for the possibility of running more trials in the same time span (see Figure 11, left side), but comes with the complication of having less information in the configuration selection process. For example, if five configurations are being evaluated in parallel in five different GPUs, once the first configuration finishes being evaluated, a sixth one will have to be selected, which will only be based on the results of the configuration that has concluded, and not on the other four that are still being evaluated.

Another possible approach to the parallelization of the evaluation of a configuration is done on the user's side, wherein they setup the training and testing of a configuration beforehand in such a way that the work of it is split among multiple different devices (GPUs and/or CPUs) (see Figure 11, right side). So instead of the system evaluating, for example, three configurations at the same time in three separate devices, only one evaluation will be evaluated at any given time, but its evaluation effort will be split among the three devices. In general, this will cut the configuration evaluation time by a third, although this depends a lot on the parallelization strategy applied and how the devices coordinate among themselves.

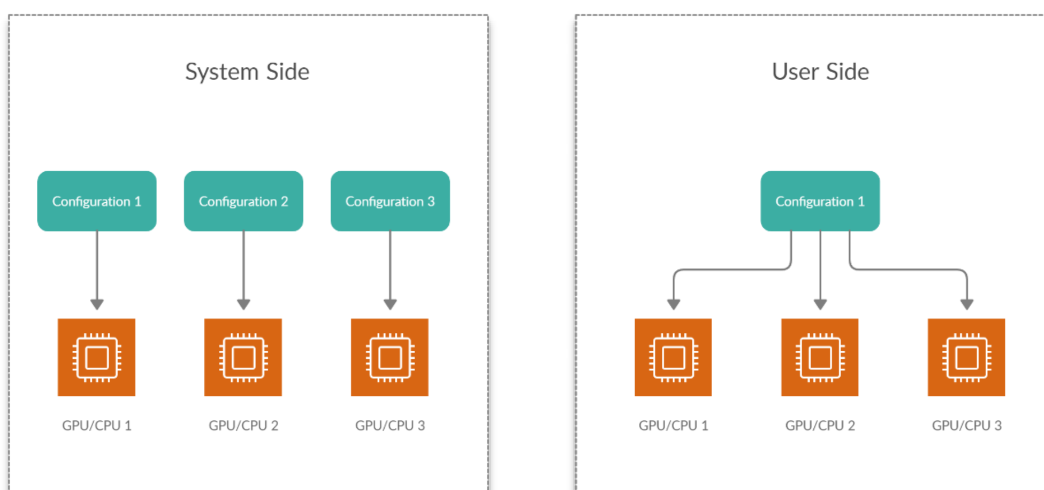


Figure 11 – Parallelization of evaluation of configurations: on the left, performed by the system; on the right, performed by the user.

With the user side approach, the user has full control over whether to follow a parallelization approach or not, as they may want to sacrifice the time reduction in the optimization process' execution with the aim of having the most information for the system's acquisition function. Not only that, but the user also has the freedom to select the parallelization strategy they intend to use and that best adapts to the problem at hand. With these points in mind, in addition to the fact that the decrease in the time complexity deriving from either approach is relatively the same, it was determined to not implement a mechanism to evaluate configurations in parallel in the system, and instead leave this decision up to the system's user.

# 5 Implementation

This section presents the technical implementation of the system, based on the design earlier described and the selected hyperparameters optimization algorithm. It starts off by presenting the technologies employed in the coding of the system, such as the programming language, followed by its human-computer interaction component, listing what data the user and system will provide to each other to ensure an optimal workflow. Two sections detailing how the search space was implemented and how it relates to the user-implemented objective function follow, concluding with a scalability issue of the system and how it was determined to be tackled.

## 5.1 Technologies

The system was implemented from the ground up using Python, a recognized programming language commonly employed in the fields of data science and machine learning. Wherever possible, already well established, tested, and documented frameworks and libraries were used, as long as these were not an impediment towards the quality and end goals of the system. The employed libraries are as follows:

- **scikit-learn –** Provides a Gaussian Process surrogate model that can be trained on existing data and used to perform predictions on unseen data;
- **SciPy –** Provides a function to calculate the cumulative distribution function of a normal distribution, used in the acquisition function;
- **NumPy –** Manipulates data (configurations, losses, etc.) as multi-dimensional arrays and performs mathematical operations on them;
- **outlier_utils –** Provides a function for the two-sided Grubb's test;
- **Pandas –** Creates a summary of the evaluated configurations and respective results as a dataframe for the user to consult.

As can be seen, no library or framework was employed in the system in relation to the implementation of neural networks. This is because the system was made in such a way that it

can be used to optimize the architecture of any machine learning algorithm, not just neural networks. Nevertheless, this was not a system requirement for the thesis and, as such, tests were only performed on neural networks and conducted with the following tools:

- **Pandas –** Reads structured data from CSV files and manipulates it as dataframes;
- **Tensorflow –** Sets up datasets as batches for training and evaluation of neural networks;
- **Keras –** Runs Tensorflow under the hood, simplifying the implementation, training, and evaluation of NNs;
- **Matplotlib –** Draw charts containing the results of the optimization system.

## 5.2 Human-computer Interaction

With the intention of keeping the system as accessible and user-friendly as possible, the system was implemented in such a way that the user only interacts with it through a single function. By calling this function with the required parameters, the system will immediately start the optimization process and output its results as it goes along, returning a summary of the entire procedure once it finishes.

The required parameters are:

- **Search space –** A dictionary specifying the configurations search space. More details in section 5.3;
- **Objective function –** A user-defined function, which receives as parameter a dictionary consisting of the selected neural network configuration for the current iteration. More details in section 5.4;
- **Number of iterations –** The stopping criterion of the optimization process. The system will execute for the specified number of iterations;
- **Trade-off parameter –** The initial value of the acquisition function's trade-off parameter ($\xi(0)$, according to eq. 22). The higher the value, the more the system will explore the search space, and vice-versa;
- **Outlier threshold –** Alias for the alpha value of the Grubb's outliers test ($\alpha$, according to eq. 24). The higher the value, the more sensitive the system will be to outliers and the more easily it will classify them as such.

The values returned by the system are:

- **Evaluated configurations –** A dataframe containing all the evaluated configurations at every iteration and their respective loss, accuracy, and the surrogate model's predicted loss and respective prediction standard deviation. The user can use this dataframe for diagnostic purposes, collect statistics, draw graphs, etc.;

- **Surrogate model –** The surrogate model with the knowledge gained by all the evaluated configurations. The user can save this model and later use it to predict other configurations;
- **Best configuration –** A fully trained neural network with the best performing configuration. The user can save the neural network or deploy it to start using it right away.

Despite the simple human-computer interaction of the system, as it grows in complexity and configurability in the future, it can easily be extended to possess an internal state and give room for more convoluted interactions with the user.

## 5.3 Search Space

As specified earlier in section 4.3, the user of the system will have complete control over the configurations search space the system will use. This search space will be specified through a dictionary, where each item's key and value will be, respectively, the name of a hyperparameter and the hyperparameter's search space, consisting of a vector of either numeric (integer or floating-point) or textual values (but not both at the same time). The numeric values will be kept as is when feeding them into the optimization algorithm, but the textual values will instead be considered as categorical data and converted to natural numbers ranging from zero to the number of values in the respective search space minus one, as the surrogate model only understands numeric values (see Table 5). The search space of all the hyperparameters combined will represent the overall configurations search space.

Table 5 – Example configuration search space containing all possible data types.

| Hyperparameter Name | Original Search Space | Converted Search Space |
|---|---|---|
| Number of Layers | [1, 2, 3] | [1, 2, 3] |
| Learning Rate | [0.1, 0.01, 0.001] | [0.1, 0.01, 0.001] |
| Training Optimizer | [Adam, AdaGrad] | [0, 1] |

The search space of all the hyperparameters combined will represent the overall configurations search space. An example of a possible configuration at a given iteration can be seen in Table 6.

Table 6 – Example configuration for a given iteration.

| Hyperparameter Name | Hyperparameter Value |
|---|---|
| Number of Layers | 2 |
| Learning Rate | 0.01 |
| Training Optimizer | 1 |

## 5.4 Objective Function

Since the user is in charge of specifying the configurations search space, they are the one that knows where each hyperparameter is meant to be used. Even if a given hyperparameter has the name "Number of Layers", the system will not know what it represents or how to use it. This ensures the user is not constrained on specifying only hyperparameters the system knows and supports, but instead has full freedom and flexibility on using whichever hyperparameters they wish.

To accomplish this, one of the parameters the user must pass to the system is a function defined by them, referred to as the objective function. At every iteration of the optimization process, the system will call this function and inject the value of each of the configuration's hyperparameters for that iteration as function arguments. The code the user wrote for the function will then be responsible for using each hyperparameter wherever the user intended it to be used. For example, if the function has an argument for the number of layers of a neural network, somewhere in the function's code could be a loop that creates the number of layers of the network based on that variable.

## 5.5 Scalability

The biggest weak point of the system's surrogate model, Gaussian Processes, is its scalability. Since it has a cubic time complexity ($O(n^3)$) (Feurer & Hutter, 2019), it can become extremely costly to calculate its posterior distribution the more training data there is, which, in turn, affects the time taken by the model to make predictions for new configurations. Since the acquisition function, at every iteration, uses the surrogate model's predictions in order to pick the next configuration to evaluate, it becomes unattainable to do this for every configuration available in the search space at that point (since search spaces can easily get to millions of configurations). Instead, based on some manual tests performed, it was decided to cap the number of configurations for the acquisition function to evaluate to five thousand. As such, at every iteration, a maximum of five thousand configurations (less if the available search space is smaller than that) are randomly selected as candidates for the acquisition function.

# 6 Evaluation

This chapter dives into the performance of the system and discussion of its results, starting off by describing the methodology used in the execution of the experiments and enumerating the key metrics employed in the assessment of the system's performance. Following that, a case study on sensor vibrations is presented, split into two parts: detecting whether a vibration was caused by a human fall or not; detecting the source of the vibration. Lastly, the system is tested in the optimization of a convolutional neural network using the MNIST dataset, with its best configuration then being compared with the best configuration found by other HPO systems and manually configured networks.

## 6.1 Methodology

For the performed case study (section 6.3), where the system is employed in the detection of sensor vibrations, the original dataset was split into three separate groups at a ratio of 60%, 20%, and 20%, respectively: training, validation, and test datasets. For a given iteration of the optimization process, the neural network is first exposed to the given training dataset and learns from it, followed by its performance evaluation via the validation dataset, where its predictions are compared with the real values. This procedure occurs for every epoch the neural network is designed to train and evaluate for. Once the last epoch is finished, the NN's performance is evaluated one last time, but against the test dataset instead, and the loss obtained from this last evaluation is what is considered as the final loss of the network and fed into the surrogate model of the system as the results of the configuration. The figure below demonstrates this entire flow:

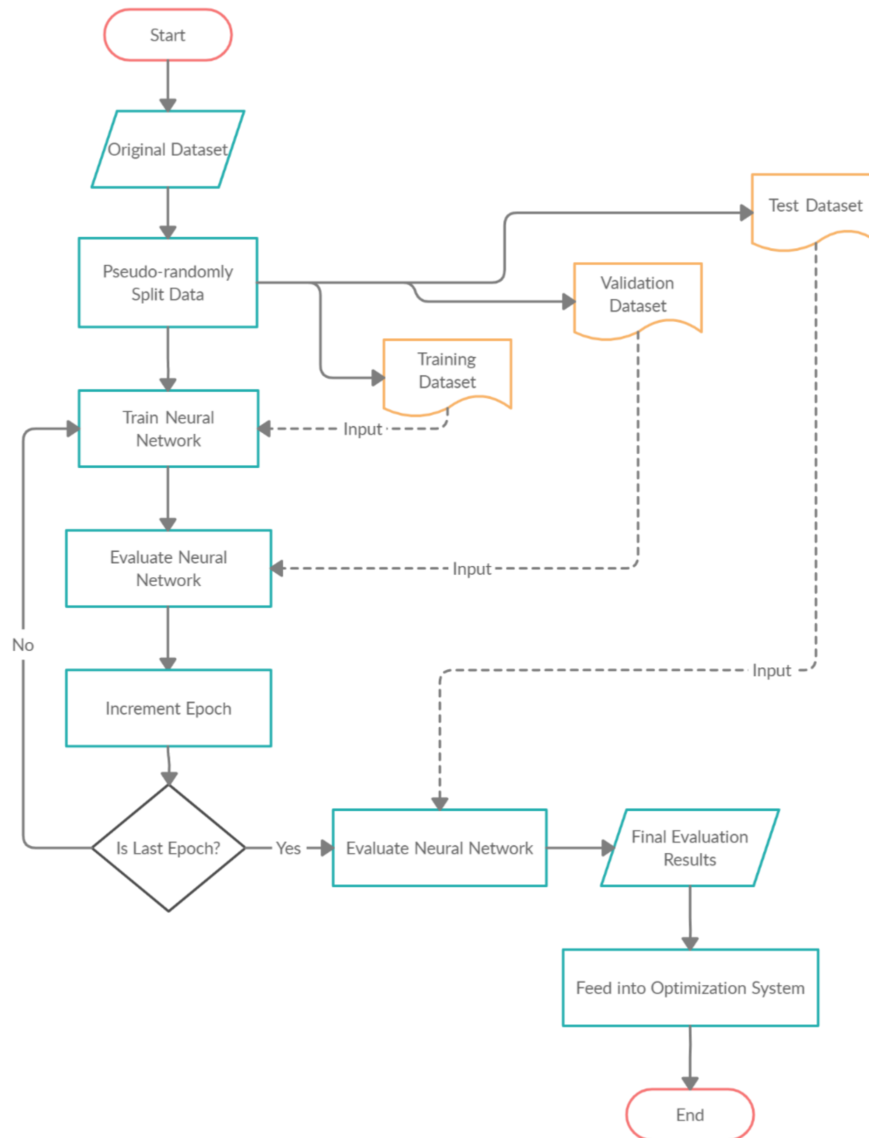


Figure 12 – Evaluation flow of a neural network configuration according to the thesis' evaluation methodology.

The training, validation, and test datasets are obtained by pseudo-randomly slicing parts of the original dataset on every evaluated configuration, always using the same seed. This ensures that all datasets are randomized, but in a predictable manner, guaranteeing every single configuration always uses the same samples and avoiding an increase or decrease in performance between configurations not because of the different architecture or set of features, but because of the different data samples used. After splitting the three datasets, the training dataset is randomized again (not pseudo-randomized), so even though every single evaluated configuration uses the same samples for training, they may be exposed to them in a different order throughout different epochs, leading to different learning processes.

With regard to the optimization system, as aforementioned, it will take into consideration the loss of a given neural network configuration when evaluated against the test dataset as the

objective value to be minimized. As such, its goal will be to find the combination of hyperparameters and input features that will lead to the lowest test loss possible. Regarding the outliers' diagnosis mechanism of the system, every test will be performed with an outlier threshold of 0.05.

Lastly, regarding the more technical side of the opted evaluation methodology, every test was performed on the same machine provided by GECAD, ISEP's research center which proposed the thesis here discussed. The machine has the following specifications:

- Intel Xeon E5-2697 v2 processor, with 2.70GHz;
- Four NVIDIA Tesla K20c graphics cards, each with 5GB GDDR5 video memory;
- 64 GB of RAM.

## 6.2 Metrics

With the aim of judging the performance of the implemented system in the conducted experiments, multiple different criteria and measures were considered. The first set of criteria concern the results of the configurations evaluated by the system, such as their accuracy and loss. These criteria will be obtained at the end of every configuration evaluation, after its training and testing, and used at the end of the optimization process to determine the quality of configurations the system finds and, therefore, the quality of the system itself. These criteria are as follows:

- **Best NN configuration found –** A system to automatically configure neural networks will only be as good as the best network configuration it finds. The accuracy and error rate of the optimal configuration will be one of the most vital measurement of the system's results;
- **Configurations performance over time –** In order to understand if the system is improving its results over time by slowly converging to better performing configurations, a history of the results of every configuration will be kept and, subsequently, analysed and assessed.

The second set of measures concern the system itself and its own mechanisms, centring around its learning behaviour and predictions' accuracy, crucial for ensuring the system remains useful for predicting future configurations, in addition to its ability to handle outliers without being neither too sensitive nor impervious to them. Both measures will be kept track of throughout the optimization process and obtained after every configuration's evaluation (similar to the first set of measures), which is when the system's makes its own prediction about the evaluated configuration's loss and when it tests the history of evaluated configurations for outliers and removes them. These measures are as follows:

- **Optimization system's predictions accuracy and confidence –** Even if the system does not find the best performing configurations during the carried out tests, if it manages to become smarter (i.e. predict the results of configurations with a low standard deviation), it will still have developed the intellect to find those configurations, as it was able to accurately learn how each hyperparameter and feature affected a configuration's results;
- **Outliers' detection-and-removal mechanism –** It is fundamental that the system is able to accurately pinpoint outlier results from the evaluated configurations and ignore them, as these can drastically affect the performance of the system's predictions. Whether a value should be considered an outlier or not can be a subjective decision, but, nonetheless, the system's judgement on this matter will be a metric to consider.

For every conducted experiment, each one of these metrics will be looked at and discussed from various points of views and through different methods depending on the test itself and what it is meant to accomplish.

## 6.3 Case Study - Detection of Sensor Vibrations

The case study performed uses a dataset of vibrations detected by a sensor, along with a multitude of other tools that obtain data about the vibration itself, such as its acceleration and orientation. The dataset was provided by GECAD to be used in the case study here presented, as the research center was interested in putting the system to the test with its own data and in obtaining the best performing NN configuration the system could find for their own applications.

This case study is split into two separate parts: firstly, the system will be used to find the best configuration for a binary classification scenario in which the neural network will have to detect whether a given vibration sample was caused by a human fall or not (human fall classification); secondly, in a more challenging setting, the system will be used to find the best configuration for a multiclass classification problem of recognizing what type of object caused the vibration (vibration source classification). Each of these tasks comes with its own separate dataset, albeit the two are extremely similar (more details on this in the following section), and the results and respective discussion of both can be seen in sections 6.3.3 and 6.3.4.

### 6.3.1 Datasets

Both datasets come in a structured format, split into dozens of comma-separated values (CSV) files with a varying number of rows each. The human fall and vibration source datasets have a total of 5,926 and 1,535 samples and 12 and 9 features each, respectively. Table 7 below

describes the features of both datasets, their data types, and in which datasets each one is present.

Table 7 – Features of sensor vibrations' datasets.

| Features | Data Type | Human Fall Dataset | Vibration Source Dataset |
|---|---|---|---|
| Time | Numeric | ✓ | ✓ |
| Accelerometer 1 X-axis | Numeric | ✓ | ✓ |
| Accelerometer 1 Y-axis | Numeric | ✓ | ✓ |
| Accelerometer 1 Z-axis | Numeric | ✓ | ✓ |
| Accelerometer 2 X-axis | Numeric | ✓ | ✓ |
| Accelerometer 2 Y-axis | Numeric | ✓ | ✓ |
| Accelerometer 2 Z-axis | Numeric | ✓ | ✓ |
| Gyroscope X-axis | Numeric | ✓ | ✕ |
| Gyroscope Y-axis | Numeric | ✓ | ✕ |
| Gyroscope Z-axis | Numeric | ✓ | ✕ |
| Sound | Numeric | ✓ | ✓ |
| Doppler | Numeric (Categorical) | ✓ | ✓ |

Each of the 12 total features can be summarized as:

- **Time –** Corresponds to the time the vibration was detected at (in Unix time);
- **Accelerometers –** Correspond to the two accelerometers used to detect the acceleration of the vibration, in each of the three-dimensional axes;
- **Gyroscope –** Represents the three-dimensional rotation of the vibration;
- **Sound –** Vibration sound detected by a microphone, varying between 0 and 255, and symbolizing the sonic intensity of the vibration;
- **Doppler –** Obtained by running the signal of a doppler sensor through a function that outputs a categorical value between 0 and 16, representing the strength and abruptness of the vibration.

The target value of each of the two datasets also differs: for the human fall, it is a binary value of whether it was caused by a human fall or not; for the vibration source, it can have one of three values, depending on the object that triggered the vibration: water bottle, chair, or smartphone. For the human fall dataset, the vibrations of the fall were simulated by the dropping of a doll consisting of a thick cardboard tube with a diameter of 20 centimetres, holding ten 1.5 litters water bottles inside of it (simulating the approximate density of human flesh), and a three-kilogram iron block on top (simulating a human head), all wrapped in

clothing. All other non-human fall vibrations originate from random activities and sources, such as walking and clapping.

Given the relatively small amount of samples for each dataset and the complexity of the problems at hand, it is not expected that even the best neural network the system finds has an exceptionally high accuracy (above 90%). The main goal of this case study is to perform an initial assessment of the system's performance and learning capabilities and to experiment and discuss different settings of the system, such as its trade-off parameter.

### 6.3.2 Neural Network Structure and Search Space

Both parts of this case study share a similar neural network structure. Both will consist of a feed forward neural network (FFNN), a type of neural network where the connections between neurons do not form a cycle (similar to the NN in Figure 2), with the following characteristics:

- **Input layer –** Receives the dataset as input, with a neuron per data feature;
- **Hidden layers –** One or more hidden layers, depending on the respective hyperparameter. The number of neurons in each of these layers and their respective activation function is always the same for a given configuration and are too dependent on their respective hyperparameters;
- **Output layer –** For the human fall, this layer is comprised of one neuron with the *sigmoid* activation function; for the vibration source, it is instead comprised of three neurons (one for each possible classification class) with the *softmax* activation function;
- **Cost function –** Cross-entropy loss.

For the output layer, both *sigmoid* and *softmax* output probabilities between zero and one, signifying the certainty the network has that a given class is present. *Sigmoid* outputs values independent among multiple neurons, making it more suitable for both binary and multilabel classification problems. *Softmax*, however, outputs values dependent among themselves that always sum to one, making it more fitting for multiclass classification problems. As for the cost function, cross-entropy loss calculates the performance of a network in which the output(s) is(are) between zero and one—thus being applicable to both aforementioned activation functions—by measuring the distance between the network's prediction(s) and the real value(s).

Table 8 demonstrates the established hyperparameters and respective search spaces of each of the two datasets, which, when combined with every feature of the corresponding dataset, will equate to the configurations search space of that dataset. With that in mind, the human fall optimization will have a search space of 5,503,680 configurations and the vibration source optimization will have a search space of 2,759,400. Despite the larger search space of some of the hyperparameters for the vibration source problem, it manages to have roughly half of the

total number of configurations as the human fall problem, which has to take into consideration three extra features.

Table 8 – Search space of the vibrations' case study.

| Hyperparameters | Data Type | Human Fall Search Space | Vibration Source Search Space |
|---|---|---|---|
| Epochs | Numeric | [1, 2, 3, 4] | [1, 2, 3, 4, 5] |
| Training Optimizer | Textual | [Adam, SGD, RMSProp] | [Adam, SGD, RMSProp] |
| Batch Size | Numeric | [16, 32] | [16, 32, 64] |
| Nr. of Hidden Layers | Numeric | [1, 2, 3, 4] | [1, 2, 3, 4, 5] |
| Nr. of Neurons per Hidden Layer | Numeric | [1, 6, 11, 16, 21, 26, 31] | [1, 6, 11, 16, 21, 26, 31, 36] |
| Hidden Layers Activation Function | Textual | [ReLU, Sigmoid] | [ReLU, Sigmoid, TanH] |

### 6.3.3 Human Fall Classification

For the human fall classification, four runs of the system were executed, each with a different value for the trade-off parameter. Given the influence this parameter has in the entire system, being solely in charge of managing the exploration versus exploitation mechanism, it was vital to understand how it influences the learning behaviour of the system.

*Influence of the Trade-off Parameter*

Figure 13 showcases the results obtained from the four runs of the system made with four different values for the TOP: 0, 4, 8, and 12. Each test was carried out over the course of 3,000 iterations, exploring ≈0.00005% of the total search space, and taking, on average, around 8 hours to complete. Each chart in the figure can be interpreted as: blue line corresponding to the real loss of the evaluated configurations; red line corresponding to the prediction made by the system of the evaluated configurations' loss; light red area corresponding to the standard deviation of the system for each prediction made.

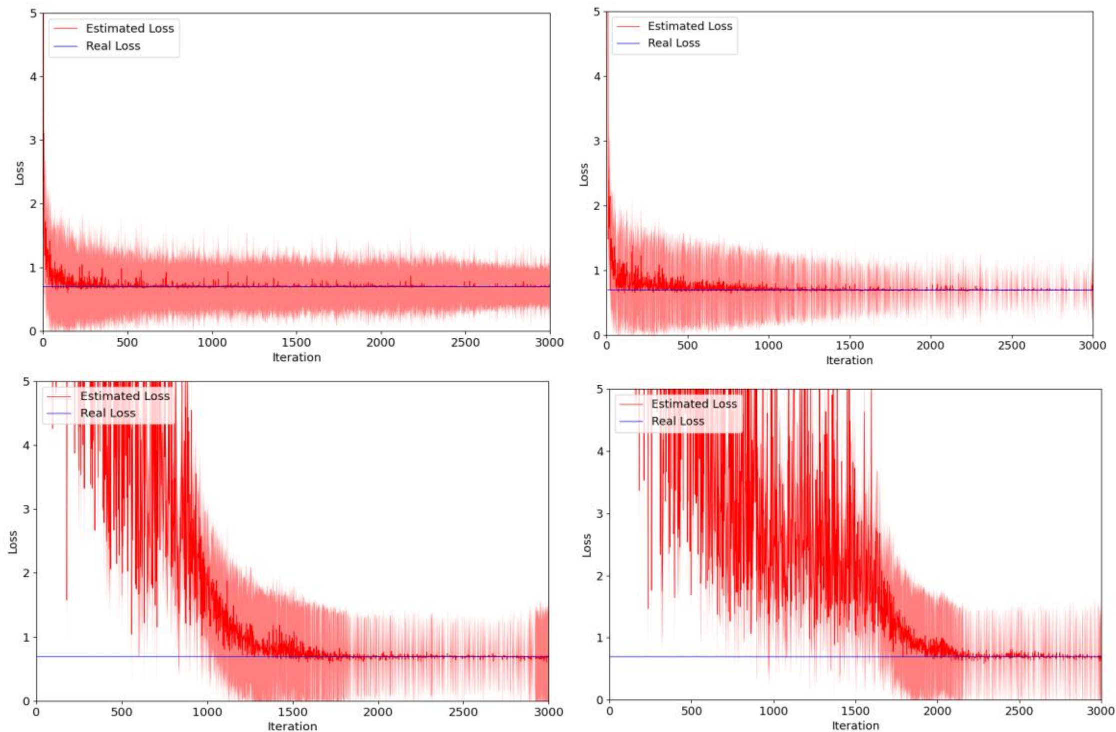


Figure 13 – System performance results, throughout 3000 iterations, given different values for the trade-off parameter: 0, 4, 8, and 12, respectively, on the top left, top right, bottom left, and bottom right corners. Configurations deemed outliers by the system in the final iteration are not present.

The first noticeable observation is that the system managed to learn with every one of the four TOP values, having more and more accurate predictions over time, whilst too lowering its predictions' uncertainty (standard deviation). It is, however, hard to tell whether the system managed to find better configurations over time, as any configurations deemed an outlier by the system in its last iteration is not presents in the graphs, bringing about the seemingly constant line of the real evaluated configurations losses in all of the graphs. It can also be seen that the higher the trade-off parameter, the higher (and less accurate) the estimated loss of the system at earlier iterations is. This behaviour is expected, as the system explores the search space earlier on and, as such, is constantly evaluating configurations very distinct from each other, in an attempt to find a global minimum instead of a local one.

Except when the TOP = 0, the system manages to become proficient at making predictions, in the sense that not only does it have predictions spot on with the real loses, but it too is aware of its own accuracy, as its extremely low standard deviation of those predictions is proof of. On the other hand, when the TOP = 0, the system never quite manages to get very certain of its predictions, with the average standard deviation of its predictions hovering around 0.49. This can be due to the fact that since the parameter is zero, the system follows a 100% exploitational methodology and, as such, is always avoiding risks and selecting very similar configurations at every iteration. In turn, this leads to the system never being certain of its predictions, as it was never exposed to other more distinct configurations in the search space.

Looking at the system's performance towards the end of the optimization process—when it has the most gathered knowledge—, Figure 14 demonstrates the loss residuals (absolute difference between the system's predicted losses and respective real losses) and standard deviation for the last 500 evaluated configurations. TOP = 4 has both the lowest residuals and standard deviation, with medians of virtually zero and minute interquartile ranges (IQRs), confirming what was established with Figure 13. It is interesting to see how despite having the highest median standard deviation, as previously determined, TOP = 0 manages to have loss residuals comparable with the tests using values of 8 and 12, but with a much less spread out and more concentrated distribution.

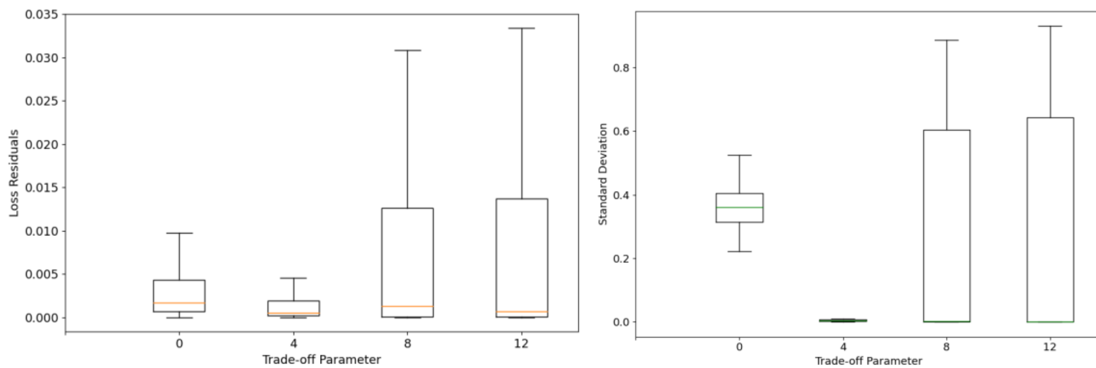


Figure 14 – Distribution of loss residuals and standard deviation of the system's predictions in all four tests for the last 500 iterations, outliers included.

Only taking into account the findings from Figure 13 and Figure 14, the trade-off parameter with the value of four appears to be the best one, where despite it not being the one where its predictions converge towards the real values the fastest—TOP = 0 is—it is the one that has the most precise predictions and the highest degree of certainty of said predictions for the longest period of time. Putting it simply, it is with this TOP value that the system learns the best.

*Evaluated Configurations*

Taking a deeper look at the actual evaluated configurations in all four system tests, Table 9 lists the ten configurations found with the lowest loss, together with the iteration and test they were found in (according to the trade-off parameter used), and their respective loss and accuracy. Instead of the accuracy, the loss of the neural network was the metric chosen to find the best ten configurations due to its more comparative and less fluctuating nature.

Table 9 – The 10 configurations found with the lowest loss, across all four trade-off parameter system tests, sorted by loss.

| Trade-off Parameter | Iteration | Loss | Accuracy |
|---|---|---|---|
| 8 | 2677 | 0.60 | 64.5% |
| 8 | 497 | 0.64 | 64.7% |
| 8 | 62 | 0.66 | 54.6% |
| 0 | 1380 | 0.66 | 60.5% |
| 0 | 855 | 0.67 | 67.9% |
| 8 | 33 | 0.67 | 63.2% |
| 12 | 869 | 0.67 | 50.3% |
| 12 | 557 | 0.68 | 67.7% |
| 12 | 1579 | 0.68 | 58.2% |
| 8 | 1495 | 0.68 | 52.2% |

Despite the earlier assessments that the trade-off parameter of four was the one where the system learned the best, it is not the one where the best configurations were found, as not a single configuration in the top 10 comes from this test. The system trial with a TOP value of eight managed to find 5 of the 10 best configurations, with 3 of them being the three best ones found, meaning that even though the system did not learn as well with this trade-off parameter compared with when it was four, the knowledge the system did gain may have been more valuable. As a result, a TOP value of eight may have the best balance between the learning of the system and the search for high performing configurations, as a higher TOP value indicates a deeper exploration of the search space, in turn leading to a slower progression of the system's predictions' accuracy, but also to a higher chance of finding the highest quality configurations as well.

However, all of these conclusions do not imply that TOP = 4 is bad, as many of these top configurations were found in early iterations of the system, while it was still unintelligent, so the likelihood of them having been found by chance is high; but, then again, this is exactly the kind of behaviour that can be expected when the system follows a more exploratory approach.

Considering the features and hyperparameter used by each of the best configurations, it is important to analyse whether patterns emerge or not. These details of the configurations are not present in Table 9, as there are too many to list, but can instead be observed in Appendix 1, which has the configurations listed in the same order as Table 9. Looking at the table in the appendix, one feature can be seen as being ignored by all of the best configurations: the time. This feature was expected ahead of time to be irrelevant towards classifying the vibration as a human fall or not, but it was still added to the feature selection mechanism of the system to observe whether the system picked up on this too or not. Outside of this feature, though, there is no other apparent pattern on the features and hyperparameters of the best configurations. A peculiar case is the second-best configuration, which only has one hidden

layer with one neuron, but manages to have a loss and accuracy akin to the other top-performing configurations.

*Diagnosis of Outliers*

According to the results present in Figure 15, the outliers' detection-and-removal mechanism seems to be working as anticipated. The number of outliers when the trade-off parameter is 8 or 12 (roughly one third of the total evaluated configurations) may indicate the mechanism to be too sensitive, but given that these two tests explore the search space more and the other two tests with TOP values of 0 and 4 have a more sensible amount of outliers, the mechanism looks to be acting exactly how it ought to.

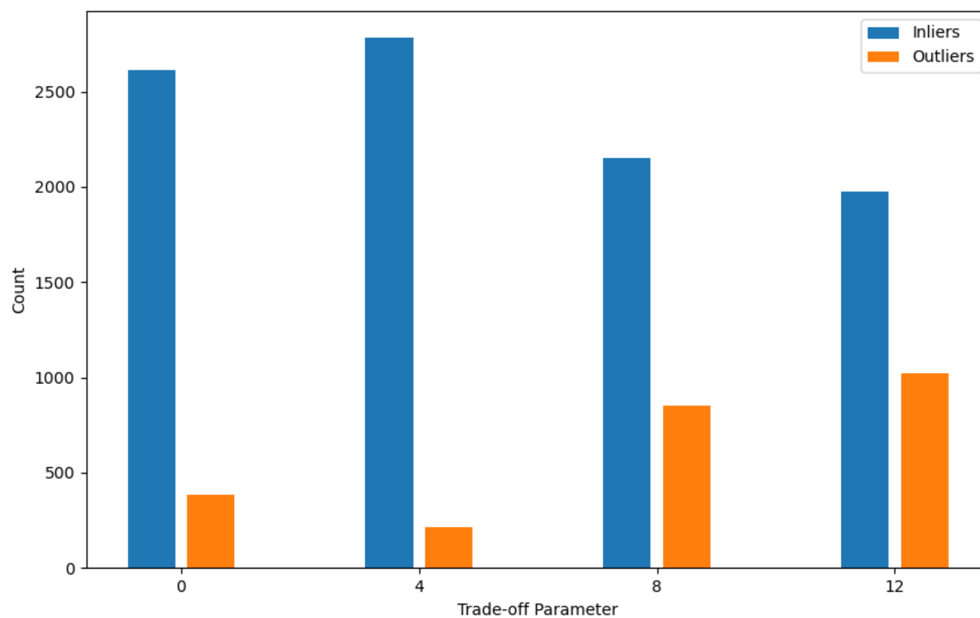


Figure 15 – Number of inliers and outliers in each of the four performed tests, according to the last iteration of each test.

Another remark about the outliers' mechanism is how, by the last iteration in the respective test, it classified the 30 configurations with the lowest loss as outliers. This was expected given the two-sided nature of the employed Grubb's test, which considers both minimum and maximum values as possible outliers. The second part of this case study will experiment with using the one-sided version of the Grubb's test that only looks at maximums to locate outliers, in order to assess whether this will have an impact in the system's learning behaviour.

### 6.3.4 Vibration Source Classification

The second part of this study, the vibration source classification, has similar results' breakdown and discussion as the first part, deepening the analysis of the system in a different classification problem variant. Only two experiments were performed in this part, with the trade-off parameters values of four and eight, as they were the ones that showed the best

results in the previous part, according to both the system's performance over time and the quality of configurations found. Each test ran for 3,000 iterations, exploring ≈0.001% of the total search space, over an average of 6 hours.

*Influence of the Trade-off Parameter*

Commencing by observing Figure 16, the learning behaviour of the system is extremely similar to what was seen in the human fall classification in both trade-off parameters. Despite this, there are some noticeable differences between both parts' rounds of tests. In the vibration source classification, both charts show how the system is more certain of its predictions after the halfway mark of the total number of iterations, evident by the less prominent light red areas. This is not necessarily meaningful, as it may have been an effect caused by the different search space, dataset, type of classification task, or some other variable, but there is also the likelihood it was caused by the change in the Grubb's test of the system's outliers mechanism (see Diagnosis of Outliers section of the earlier tests). Since the system no longer diagnosis any value below the average as an outlier, the surrogate model is able to keep a longer history of evaluated configurations, leading to the system's higher confidence in its predictions, as it retains more knowledge than in the first study part.

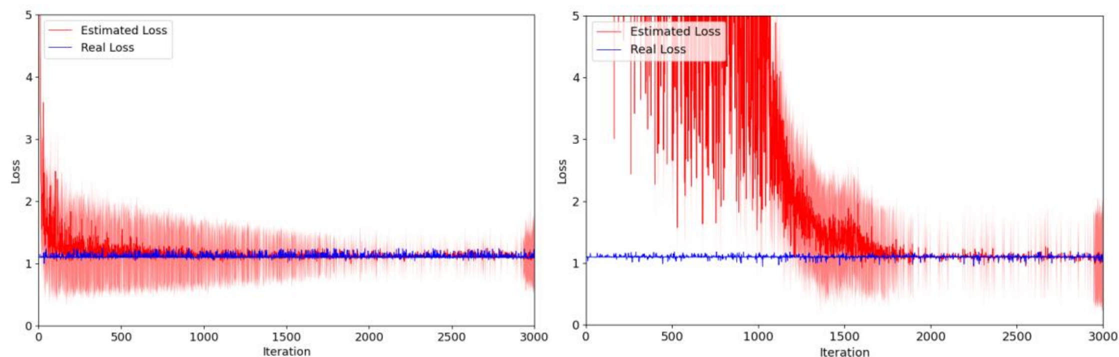


Figure 16 – System performance results, throughout 3000 iterations, given different values for the trade-off parameter: 4 and 8, respectively, on the left and right. Configurations deemed outliers by the system in the final iteration are not present.

The blue line representing the real loss of the evaluated configurations can also be perceived to be less constant and more turbulent compared to the previous tests. The explanation can be summed up to the same as the previous paragraph, where it may just be a consequence of the different problem at hand, or it may be a result of the alteration in the outliers' mechanism, as it caused the system to become less sensitive to outliers, leading to more variation in the evaluated configurations shown in the charts (as outliers are not present).

Despite the higher confidence in its predictions displayed by the system, Figure 17 demonstrates how its predictions for the last 500 evaluated configurations and with both TOP values are worse compared to the human fall classification. Going off of the deduction conceived in the previous paragraph, the higher degree of variation in inlier evaluated configurations made it harder for the system to understand the feature and architectural patterns that lead to the respective configurations' results.

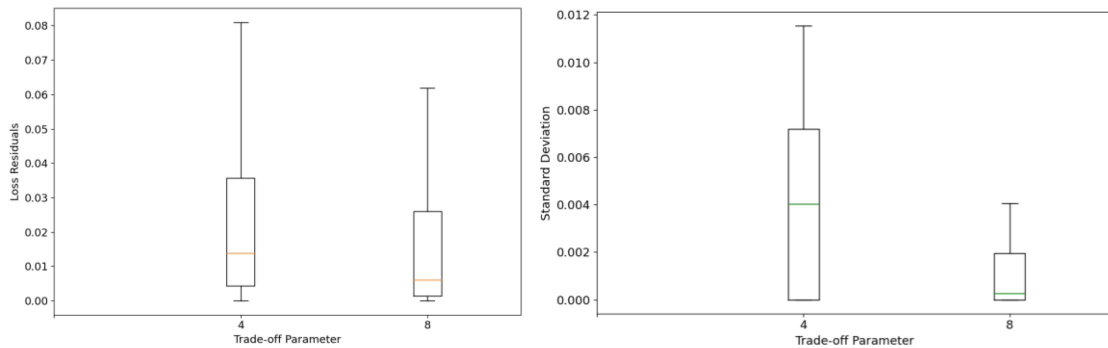


Figure 17 - Distribution of loss residuals and standard deviation of the system's predictions in both tests for the last 500 iterations, outliers included.

Summarizing the analysis made in this section, the system became more positive of its predictions due to less of them being classified as outliers and, thus, ignored by the surrogate model; in turn, however, the system's accuracy deteriorated due to the fact that the higher number of configurations it was able to learn from was also more diverse in their losses compared to the first part of the study, so understanding how each feature and hyperparameter lead to a certain configuration result was a more complex task.

*Evaluated Configurations*

Examining the results of the best 10 evaluated configurations across the two performed trade-off parameter tests, the patterns become more apparent than those analysed in the first part of the study. The trend of none of the best configurations having been found with TOP = 4 remains, but, this time, every one of the configurations was found with TOP = 8, as seen in Table 10. Furthermore, 9 out of the 10 configurations were found past iteration 1000, presumably on account of the balance between the system's exploration of the search space together with all the knowledge gathered by the system up to that point.

Table 10 - The 10 configurations found with the lowest loss, across both trade-off parameter system tests, sorted by loss.

| Trade-off Parameter | Iteration | Loss | Accuracy |
|---|---|---|---|
| 8 | 1025 | 0.95 | 56.4% |
| 8 | 2250 | 0.96 | 55.7% |
| 8 | 1472 | 0.96 | 50.5% |
| 8 | 1119 | 0.97 | 54.7% |
| 8 | 1702 | 0.97 | 49.8% |
| 8 | 2281 | 0.98 | 54.1% |
| 8 | 2552 | 0.99 | 55.7% |
| 8 | 2194 | 0.99 | 56.0% |
| 8 | 445 | 0.99 | 55.4% |
| 8 | 1895 | 1.00 | 50.5% |

Looking at the features and hyperparameters of the top configurations (see Appendix 2), one can spot more obvious patterns on the values preferred by the system compared to what was discussed in the human fall classification: none of the configurations used the time column, just like in the first part; the X axis of the second accelerometer and the microphone sound are always used; the Z axis of both the first and second accelerometers are never used; the Doppler value is not used 80% of the time; there were 36 neurons per layer in 9 configurations; the activation function of the neurons was always the hyperbolic tangent (TanH). Given that 36 was the maximum allowed value for number of neurons on the hidden layers, there is the possibility that increasing this value could lead to configurations with better results.

Just like mentioned in the previous section of this part of the study, the more obvious patterns of the best configurations can be owed to the Grubb's test change. As the system does not classify minimums as outliers anymore, it can absorb their information and attempt to search for other similar configurations with even lower losses.

*Diagnosis of Outliers*

With the change made in the outliers' mechanism, the number of diagnosed outliers by the last iteration on both experiments lowered compared to the human fall classification, as shown in Figure 18. Compared to Figure 15, the total amount of outliers was reduce by about half, which was expected given that around half of the evaluated configurations are no longer outlier candidates (those with a loss below the mean).

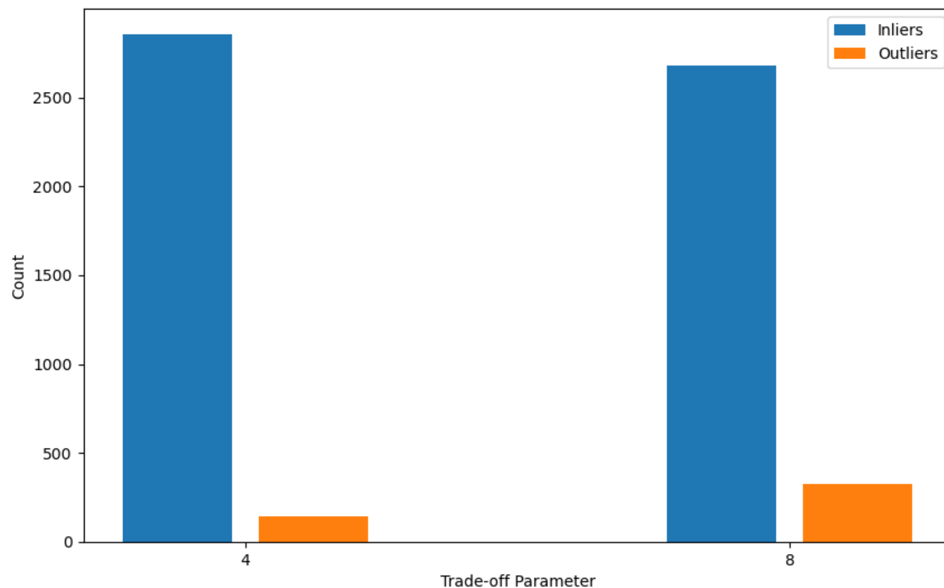


Figure 18 - Number of inliers and outliers in the two performed tests, according to the last iteration of each test.

Unlike in the human fall classification, none of the configurations listed in Table 10 were classified as outliers; in fact, across the 6000 total iterations of both tests and when sorting by loss, the 3338th best configuration was the first to be classified as an outlier.

In spite of this part of the study having less values classified as outliers, it does not necessarily mean the system's diagnosis-and-removal of outliers' mechanism improved, as it could be categorizing certain configurations as inliers which could, in turn, negatively affect the system's performance and predictions' accuracy. However, in this case, the system's learning behaviour does seem improved in comparison to the first part of the study, probably due to the system having the possibility of learning the features and hyperparameters which constitute the best configurations found. As such, the one-sided Grubb's test was kept for the following section's tests.

## 6.4 Hyperparameters Optimization of Convolutional Neural Network

Following the undertaken case study, the system was put to the test against other HPO systems and manually configured neural networks. The system will be in charge of optimizing a convolutional neural network—a type of NN different from the previous experiments—using the MNIST dataset earlier introduced in section 3.2.1 of the state of the art. The structure of this section will be similar to the previous section 6.3, where the dataset is first described in greater detail, followed by an explanation of how a CNN works, the base structure of the neural network and the designated search space, concluding with the analysis and discussion of the experiments' results.

### 6.4.1 Dataset

The MNIST dataset is a dataset widely used in the scientific community to examine the performance of machine learning algorithms applied in the field of computer vision. It consists of 70,000 black-and-white 28x28 images of handwritten digits from 0 to 9 (see Figure 19), split between a training dataset of 60,000 images and a testing dataset of 10,000 images.

Figure 19 – Handwritten digit images from the MNIST dataset (Lecun, et al., 1998).

It originates from a 1998 journal article where two separated NIST datasets, named Special Database 1 and Special Database 3, were combined, giving origin to the MNIST dataset (Lecun, et al., 1998). The authors decided to mix samples from both databases with the intent of having more variation after realizing that Special Database 1 had been obtained among high school students, whereas Special Database 3 had been collected from American Census Bureau employees.

As the MNIST dataset already comes presplit into a training and a test dataset, in order to adhere to the evaluation flow of a neural network as shown in Figure 12, the test dataset will also be used as the validation dataset.

### 6.4.2 Convolutional Neural Network

One of the most compelling features of a convolutional neural network is its capability to not only individually analyse the pixels of a given image, but also to look at them as groups of neighbouring pixels and understand the features that they may identify together. Not only that, but CNNs reduce the dimensionality and complexity of images as one goes deeper into the network, resulting in reduced computational cost for processing the data and training the network.

In order to achieve this, a CNN is usually built through the combination of three different types of layers (see Figure 20 for an example CNN structure showing every layer type):

- **Convolution layer –** As the name implies, a convolution layer convolutes the input it receives using a kernel that scans the input for certain features, reducing its dimensionality in the process. If a given image has a size of 5x5, for example, a convolution layer may go through it with a kernel of size 3x3, outputting a 3x3 image for the following layer. A convolution layer can have multiple kernels, each in the charge of identifying either different features or the same set of features but in different locations in the input. The size of the kernel(s) determines how many neighbouring pixels to analyse at once: the larger the kernel, the bigger the group of pixels evaluated together, and vice-versa;
- **Pooling layer –** A pooling layer uses a kernel mechanism that scans its input, similar to a convolution layer, but has a different internal implementation compared to a convolution layer's kernel. Depending on the type of pooling, as the kernel goes through the image, it selects the maximum or average value in its area on a maximum pooling or average pooling layer, respectively. This does not only reduce the size of the input, like with the convolution layer, but it also ensures the network becomes impervious to changes in the rotation and position of the image in addition to suppressing any existing noise in the input;
- **Fully connected layer –** A fully connected layer works in the same way as a hidden layer in a FFNN, where every neuron in the layer is connected to every other neuron in the subsequent layer (see Figure 2). The purpose of this type of layer in a CNN is to use all the features knowledge obtain by the network thus far through the other two types of layers and reason about what all the identified features could represent.

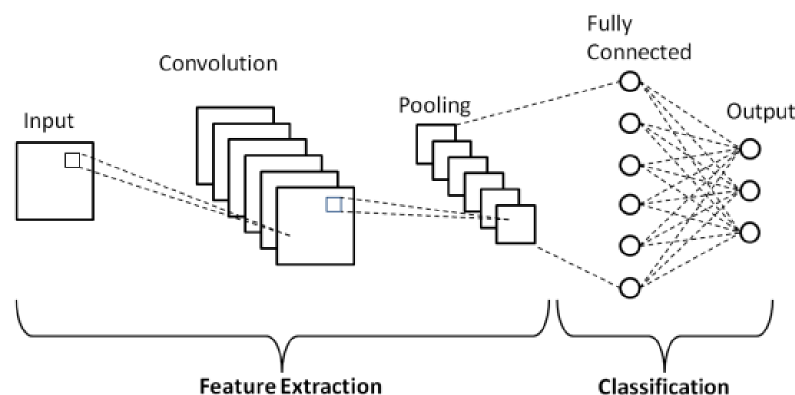


Figure 20 – Example structure of a CNN (Phung & Rhee, 2019).

To better understand how the brain of a CNN works, as an example, for the current scenario of identifying the digit present in an image, the network could start off with a convolution layer in charge of identifying edges in the image. Following that, another convolution layer uses the knowledge of the previous layer about the presence or lack thereof of edges in the

image to identify corners. A third convolution layer could use the obtained information about corners in the image to figure out shapes, such as circles. Finally, a group of one or more fully connected layers could then use all of this data to figure out what the number in the image is. Concerning the pooling layers, any of the aforementioned convolution layers could be followed by a pooling layer to ensure changes in the rotation or position of the image does not affect the convolution layer's ability to identify features.

### 6.4.3 Neural Network Structure and Search Space

Following the overview and explanation of how convolutional neural networks work, the ensuing lists presents the base structure employed for the CNN used in the performed experiments:

- **Convolution and maximum pooling layers pairs –** The network starts off with one or more pairs of layers—depending on the respective hyperparameter—, each consisting of a convolution layer followed by a maximum pooling layer. The number of kernels, kernel size, and activation function of each convolution layer will be the same across all layers for a given configuration and are dependent on the respective hyperparameters; likewise, the kernel size of every maximum pooling layer of a configuration will be the same for all layers and will too depend on its hyperparameter. The input of each convolution layer will be the output of the previous maximum pooling layer, except for the network's first convolution layer, which will act as the input layer of the network and directly receive the MNIST dataset's images;
- **Fully connected layers –** Following the convolution and maximum pooling layers pairs are the fully connected layers. The number of these layers, as well as the number of neurons in each layer—which is the same for all layers in a configuration—, are dependent on the respective hyperparameters to be optimized;
- **Output layer –** Finally, connected to the last fully connected layer is the output layer, comprising of 10 neurons with the *softmax* activation function, where each neuron is in charge of outputting the likelihood of a given image having a certain digit (similarly to the output layer in the vibration source classification of the case study);
- **Cost function –** Cross-entropy loss.

Table 11 showcases the search space used for the optimization of the CNN in the performed experiments. The combination of the search space of all hyperparameters leads to a total of 699,840 possible configurations. Despite the kernel of both convolution and maximum pooling layers being two-dimensional, Table 11 presents a one-dimensional search space for both layer types' kernels. This is due to the fact that for the base structure of the network, all kernels were considered to always be squared and, as such, the same value is used for both the kernels' width and height.

Table 11 - Search space of the CNN optimization.

| Hyperparameters | Data Type | Search Space |
|---|---|---|
| Epochs | Numeric | [3, 6, 9, 12, 15] |
| Training Optimizer | Textual | [SGD, Adagrad, Nadam] |
| Learning Rate | Numeric | [0.1, 0.01, 0.005, 0.001] |
| Batch Size | Numeric | [16, 32, 64] |
| Nr. of Convolution and Max Pooling Layers Pairs | Numeric | [1, 2, 3] |
| Nr. of Kernels per Convolution Layer | Numeric | [1, 2, 3, 4] |
| Convolution Layers Kernel Size | Numeric | [2, 3] |
| Convolution Layers Activation Function | Textual | [ReLU, Sigmoid, ELU] |
| Max. Pooling Layers Kernel Size | Numeric | [2, 3] |
| Nr. of Fully Connected Layers | Numeric | [1, 2, 3] |
| Nr. of Neurons per Fully Connected Layer | Numeric | [100, 150, 200] |
| Fully Connected Layers Activation Function | Textual | [Sigmoid, TanH, ELU] |

### 6.4.4 Analysis

In order to assess the system's performance in the optimization of a CNN, the learning behaviour of the system will first be observed and discussed, followed by an analysis of the best 10 configurations it found and their respective architectures, similarly to the case study. The system will then be matched against other HPO systems and manually configured networks by comparing the accuracy of the best configuration it finds to the accuracy of the best configuration found by the other HPO systems and the accuracy of the networks manually configured by users.

*Learning Behaviour*

There are a few factors that make the evaluation of a configuration in this section's experiments lengthier than those in the case study in section 6.3: the more intricate semi-structured nature of the images dataset; the more complex architecture of a CNN; the higher number of epochs the network trains for. Due to these factors, in addition to the need in having a procedure more similar to other HPO systems with which results will later be compared, the stopping criterion of the system was set to 500 iterations.

Due to the reduction in the number of system iterations to one sixth of the value in the case study, the trade-off parameter had to be adjusted too. As the TOP value of 8 was considered the best performing one, given that it managed to find the 3 best configurations in section 6.3.3 and the 10 best configurations in section 6.3.4, it was too reduced to one sixth of its original value, leading to $\frac{8}{6} \approx 1.3$. This change was meant to adapt the exploratory and

exploitative balance of the TOP = 8 to the decreased number of iterations in this section's experiments.

With those alterations in mind, the optimization procedure took around 31 hours to complete, with an exploration of ≈0.004% of the total configurations' search space. The performance of the system throughout the entire process can be seen in Figure 21.

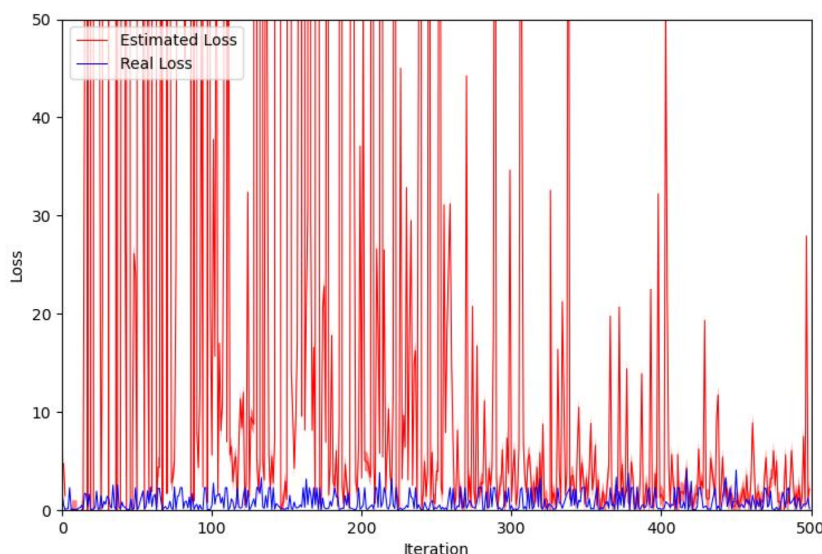


Figure 21 – System performance results, throughout 500 iterations, for a trade-off parameter of 1.3. Configurations deemed outliers by the system in the final iteration are not present.

Looking at the graph, it can be seen that the system did not manage to get as accurate as it did in the case study tests. However, not only did the system have less data to learn from, as it only ran for 500 iterations, compared to the 3000 iterations in the case study, but the variation in each configuration's loss was more spread out too—as perceived by the blue line—, making it more difficult to predict the loss of a given configuration. Thus, despite the system not achieving the precision it did in the case study by the end of the optimization process, its predictions managed to converge towards the real values faster than in the case study. Two possible causes for this behaviour may have been the smaller trade-off parameter and the smaller configurations' search space.

*Evaluated Configurations*

Table 1 showcases the 10 best configurations in the optimization process. As can be seen, all but one configuration have an accuracy between 98% and 99%, as well as very similar losses among themselves, with 8 out of 10 configurations having been found before the system reached the halfway mark of the NN's optimization.

Table 12 - The 10 configurations found with the lowest loss, sorted by loss.

| Iteration | Loss | Accuracy |
|---|---|---|
| 45 | 0.04 | 98.6% |
| 47 | 0.05 | 98.8% |
| 240 | 0.05 | 98.5% |
| 125 | 0.05 | 98.3% |
| 10 | 0.06 | 98.6% |
| 86 | 0.06 | 98.7% |
| 111 | 0.06 | 98.0% |
| 257 | 0.06 | 98.1% |
| 91 | 0.06 | 98.0% |
| 304 | 0.07 | 97.9% |

Looking at the hyperparameters of each of these configurations through the table in Appendix 3, a few architectural patterns can be noted: 6 out of 10 configurations used the maximum number of epochs available, 15, including the 4 best configurations; 8 configurations used 0.1 as the learning rate; 9 configurations used 1 convolution and maximum pooling layers pair, the minimum available; 7 configurations used 4 kernels in the convolution layers, the maximum available; 8 configurations used a kernel size of 3x3 and 2x2 in the convolution and maximum pooling layers, respectively; none of the top 10 configurations used the smallest available number of neurons in the fully connected layers, 100.

The obtained results suggest that even better configurations could have possibly been found if the number of epochs available in the search space were higher. The fact that most configurations used only one pair of convolution and maximum pooling layers could be due to the relatively small resolution of the images (28x28), whereby having more convolution and/or maximum pooling layers would reduce the resolution of the images to a point where they are not usable anymore. The high number of kernels in the convolution layers can also imply that, at every convolution layer, multiple relevant features were detected in the input by the network. Finally, the higher number of neurons in the fully connected layers is most likely proof that more neurons were necessary to process the data coming from the convolution and maximum pooling layers in order to understand the digit present in each image.

*Best Configuration Comparison*

The most accurate configuration found by the system during the optimization process is the second configuration seen in Table 12, having an accuracy of 98.76%. This configuration was compared with the best configuration found by other HPO systems (see Table 13) and network architectures crafted by users manually (see Table 14).

Looking at Table 13, one can see the systems of (Stein, et al., 2018) and (Larochelle, et al., 2007)—which have already been discussed in the state of the art section of this report—with an EGO and a grid search algorithm, respectively, as their optimization algorithms. (Han, et al., 2020) and (Yoo, 2019), on the other hand, have a genetic algorithm and a univariate dynamic encoding algorithm for searches (uDEAS) as their optimization algorithms.

Table 13 – Comparison of the thesis' system's best configuration with the best configuration of other HPO systems.

| **Configuration** | **Optimization Algorithm** | **Nr. of System Iterations** | **Epochs** | **Accuracy** |
|---|---|---|---|---|
| Thesis | BO | 500 | 15 | 98.76% |
| (Stein, et al., 2018) | EGO | 200 | 10 | 99.39% |
| (Han, et al., 2020) | Genetic Algorithm | Unknown | Unknown | 99.28% |
| (Yoo, 2019) | uDEAS | 402 | 20 | 99.11% |
| (Larochelle, et al., 2007) | Grid Search | Unknown | Unknown | 96.06% |

The displayed results showcase how the best configuration found by the thesis' system did not manage to reach the level of precision of the other configurations, only outperforming that of (Larochelle, et al., 2007) with an accuracy difference of more than 2%. Specifically, (Stein, et al., 2018) managed to find a more accurate configuration despite the smaller number of iterations in the optimization process (200 versus 500) and the smaller number of epochs the configuration trained for (10 versus 15).

Compared with manually configured NNs, the thesis' best configuration did not manage to have better results, having an accuracy worse than every other configuration (see Table 14). Despite the more accurate configuration from (Tabik, et al., 2017) with 5 less epochs of training, the thesis' best configuration was capable of getting within an accuracy difference of 0.21% with (Ciresan, et al., 2011) despite the extra 485 epochs the network was able to train for.

Table 14 – Comparison of the thesis' system's best configuration with manually-configured neural networks.

| **Configuration** | **Epochs** | **Accuracy** |
|---|---|---|
| Thesis | 15 | 98.76% |
| (Tabik, et al., 2017) | 10 | 99.07% |
| (Ciresan, et al., 2011) | 500 | 98.98% |
| (Graham, 2014) | 250 | 99.68% |

More epochs of training does not necessarily translate to more accurate configurations (as it is proof the case study in section 6.3), but given the complexity of the problem at hand and the research made, it is safe to assume expanding the epochs' search space of the system would have led to the finding of better configurations. Increasing the search space of other hyperparameters and the value of the trade-off parameter could have possibly led to the discovery of more precise configurations too, although it would also have negatively impacted the learning behaviour of the system.

# 7 Conclusions

Beginning this chapter, a delineation over the thesis' goals is done, where initially planned objectives are presented alongside other initially unforeseen objectives that were achieved either as a side effect of the approaches taken in the development of the system or as a necessity to support these same approaches. Following that, a balance is made over future improvements that can be made to the system in order to enhance its performance, versatility, and intelligence. Lastly, a summary over the work accomplished throughout the entire thesis is made, concluding with a final judgment over this work and its final obtained results.

## 7.1 Goals Accomplishment

At the beginning of the thesis, a list of goals was laid down to help guide the system's development and to help achieve final positive results. Table 15 lists the goals for the system's thesis and their respective level of accomplishment, including those not initially planned.

Table 15 – Level of accomplishment of the thesis' system's goals.

| System Goal | Level of Accomplishment | Initially Planned |
|---|---|---|
| Optimization of Hyperparameters | Accomplished | Yes |
| Feature Selection | Accomplished | Yes |
| Learning Capabilities | Accomplished | No |
| Time Complexity Reduction | Accomplished | Yes |
| Handling of Outliers | Accomplished | No |
| Adaptable to Any Type of Neural Network | Accomplished | Yes |
| Adaptable to Other Machine Learning Algorithms | Accomplished | No |
| Best Configuration Superior to Other Optimization Systems | Not Accomplished | Yes |
| Best Configuration Superior to Manually Configured Networks | Not Accomplished | Yes |

Starting off with the key objectives of the thesis necessary for the automatic configuration of NNs, the system has both the capabilities of optimizing any neural network hyperparameter and of selecting the most relevant features in structured datasets (feature selection). The selected optimization algorithm, Bayesian optimization, allowed the system to not only make informed decisions on the selection of configurations, but also to learn from them and understand how each hyperparameter and feature affected the final results of a given configuration. Despite not being a goal initially planned for the thesis, not only did it increase the value of the system, but it also helped reduce its time complexity, as once the system's predictions start lining up with the real values, it can be used to predict the results of a configuration with even having to train and test the network with that configuration.

With the system's learning capability also came the need to ensure the quality of the data it learned from. Given the system's surrogate model, Gaussian Processes, sensitivity to outliers, a mechanism not initially considered had to be implemented in order to pinpoint any possible outlier configurations and exclude them from the list of configurations the system learned from. Concerning the system's adaptability to any type of neural network, not only was the goal accomplished, but it was implemented in such a way that it can be used to optimize any other machine learning algorithm, such as support vector machines.

Finally, the main method through which the quality of the system was planned to be evaluated was by comparing the best configuration it could find (i.e. the most accurate) against the best configuration found by other HPO systems and against networks manually configured by users. Neither of these goals were accomplished, as can be seen in section 6.4.4, although the results obtained demonstrate the potential of the implemented system.

## 7.2 Future Work

One of the biggest possible points of improvement for the system is how its scalability is handled (see section 5.5). Currently, the acquisition function only has to consider five thousand configurations, chosen randomly at every iteration, as possible candidates for evaluation at that iteration. This ensures the acquisition function does not spend a long time assessing the probability of improvement of the entire pool of available not-yet-evaluated configurations, which could have millions of configurations. This mechanism could be enhanced by synchronizing it with the surrogate model, so that the five thousand configurations are not picked completely randomly and are instead chosen according to what the system believes are the best possible candidates. Not only that, but instead of always selecting five thousand configurations, the system could automatically adapt this value depending on the size of the total configurations' search space and the computational capabilities of the machine it is running on.

In order to help further mitigate the time complexity of automating the configuration of neural networks, more techniques could be researched and implemented. One such technique could be the system keeping track of the time it takes to evaluate each configuration and subsequently use that data to learn and predict how long future configurations will take to evaluate. Based on that information, it can prioritize faster-to-evaluate configurations that it predicts will have the same results as other configurations that may take longer to evaluate. This would help the system avoid unnecessarily complex network architectures that have results equally as good as simpler ones.

As more general points of improvement for the system, other existing surrogate models and acquisitions functions should be investigated and experimented with. There are multiple available options for each of these two vital components of Bayesian optimization, with only one of each having been tried out in the thesis. Similarly, there are other methods to detect data outliers which could be researched and used in place of the employed Grubb's test. Every change made in the system should then be followed by several tests on different types of neural networks and datasets to ensure their versatility and adaptability to any use case.

## 7.3 Final Appreciation

The topic of the thesis delved into multiple different subjects, such as machine learning, neural networks, hyperparameters optimization, and feature selection, all modern and valuable disciplines that keep maturing every day. With this, it was possible to have an enriching experience on how neural networks came to be, how they function, why their manual configuration can be a problematic and time-consuming task, and how one can go about creating a solution to fix this problem.

The framework initially envisioned for the thesis started off with its core, the optimization of hyperparameters and the selection of features in datasets, and then was expanded and

perfected through other features that improved its performance and reduced its inherent complexity. The undertaken state of the art study gave an overview of existing optimization algorithms and existing works using these algorithms applied in the optimization of hyperparameters in neural networks. With the design following that, critical decisions about the inner workings of the thesis' system were taken, such as the chosen optimization algorithm and how configurations' search spaces should be specified. Finally, the implementation then built upon the design guidelines to create an easy-to-use solution that was capable of performing its duty with minimal user intervention.

In the evaluation stage of the system, the performed case study managed to establish a deeper understanding of the system and how one of its most crucial mechanisms, the exploration versus exploitation of configurations, can be tweaked through a single value with a big impact in system's entire optimization process. Subsequently, the optimization of a convolutional neural network not only showed how the system can be successfully applied in the optimization of a different type of neural network, but also how the best configurations it finds can have results comparable to those of other optimization systems and of manually obtained configurations.

The value of the system for users is clearly present, and with extra future research and developments, it can reach a level of quality and performance permitting its general usage by the public. To conclude, both the thesis and the system through it implemented are considered a success.

# References

Adarsh, B. R., Raghunathan, T., Jayabarathi, T. & Yang, X.-S., 2016. Economic dispatch using chaotic bat algorithm. *Energy,* Volume 96, pp. 666-675.

Berger-Tal, O., Nathan, J., Meron, E. & Saltz, D., 2014. The Exploration-Exploitation Dilemma: A Multidisciplinary Framework. *PLOS ONE,* Volume 9, pp. 1-8.

Bergstra, J., Bardenet, R., Kégl, B. & Bengio, Y., 2011. *Algorithms for Hyper-Parameter Optimization.* Red Hook, NY, USA, Curran Associates Inc., pp. 2546-2554.

Bergstra, J. & Bengio, Y., 2012. Random Search for Hyper-Parameter Optimization. *Journal of Machine Learning Research,* 13(Feb), p. 281–305.

Bre, F., Gimenez, J. M. & Fachinotti, V. D., 2017. Prediction of wind pressure coefficients on building surfaces using Artificial Neural Networks. *Energy and Buildings,* Volume 158.

Ciresan, D. et al., 2011. Flexible, High Performance Convolutional Neural Networks for Image Classification. *International Joint Conference on Artificial Intelligence IJCAI-2011,* pp. 1237-1242.

Ciresan, D., Meier, U. & Schmidhuber, J., 2012. Multi-column Deep Neural Networks for Image Classification. *2012 IEEE Conference on Computer Vision and Pattern Recognition,* pp. 3642-3649.

DeepAI, n.d. *Weight (Artificial Neural Network).* [Online]
Available at: https://deepai.org/machine-learning-glossary-and-terms/weight-artificial-neural-network
[Accessed 28 12 2019].

Ebden, M., 2015. Gaussian Processes: A Quick Introduction. *arXiv,* Volume arXiv preprint arXiv:1505.02965.

Expert System Team, 2017. *What is Machine Learning? A definition.* [Online]
Available at: https://expertsystem.com/machine-learning-definition/
[Accessed 22 12 2019].

Feurer, M. & Hutter, F., 2019. Hyperparameter Optimization. In: F. Hutter, L. Kotthoff & J. Vanschoren, eds. *Automated Machine Learning: Methods, Systems, Challenges.* s.l.:Springer International Publishing, pp. 3-33.

Görtler, J., Kehlbeck, R. & Deussen, O., 2019. *A Visual Exploration of Gaussian Processes.* [Online]
Available at: https://distill.pub/2019/visual-exploration-gaussian-processes/
[Accessed 26 6 2020].

Graham, B., 2014. Fractional Max-Pooling. *arXiv preprint arXiv:1412.6071.*

Grubbs, F. E., 1950. Sample Criteria for Testing Outlying Observations. *The Annals of Mathematical Statistics,* 21(1), pp. 27-58.

Han, J.-H., Choi, D.-J., Park, S.-U. & Hong, S.-K., 2020. Hyperparameter Optimization Using a Genetic Algorithm Considering Verification Time in a Convolutional Neural Network. *Journal of Electrical Engineering & Technology,* 15(2), pp. 721-726.

Hutter, F., Hoos, H. H. & Leyton-Brown, K., 2011. *Sequential Model-Based Optimization for General Algorithm Configuration.* Berlin, Heidelberg, Springer-Verlag, pp. 507-523.

Jones, D. R., 2001. A Taxonomy of Global Optimization Methods Based on Response Surfaces. *Journal of Global Optimization,* Issue 21, pp. 345-383.

Koehrsen, W., 2018. *A Conceptual Explanation of Bayesian Hyperparameter Optimization for Machine Learning.* [Online]
Available at: https://towardsdatascience.com/a-conceptual-explanation-of-bayesian-model-based-hyperparameter-optimization-for-machine-learning-b8172278050f
[Accessed 4 2 2020].

Krasser, M., 2018. *Gaussian processes.* [Online]
Available at: http://krasserm.github.io/2018/03/19/gaussian-processes/
[Accessed 14 6 2020].

Kushner, H. J., 1964. A New Method of Locating the Maximum Point of an Arbitrary Multipeak Curve in the Presence of Noise. *Journal of Basic Engineering,* Volume 86.

Larochelle, H. et al., 2007. An empirical evaluation of deep architectures on problems with many factors of variation. *Proceedings of ICML,* Volume 227, pp. 473-480.

LeCun, Y., Bengio, Y. & Hinton, G., 2015. Deep learning. *Nature,* Volume 521, p. 436–444.

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

Lizotte, D. J., 2008. *Practical Bayesian Optimization,* Edmonton, Alberta, Canada: University of Alberta.

Martinez-Cantin, R., Tee, K., McCourt, M. & Eggensperger, K., 2017. Filtering Outliers in Bayesian Optimization.

Martínez, C. M. & Cao, D., 2019. Integrated energy management for electrified vehicles. In: C. M. Martínez & D. Cao, eds. *Ihorizon-Enabled Energy Management for Electrified Vehicles.* s.l.:Butterworth-Heinemann, pp. 15-75.

MathWorks, n.d. *Bayesian Optimization Algorithm.* [Online]
Available at: https://www.mathworks.com/help/stats/bayesian-optimization-

algorithm.html#bva8rde
[Accessed 9 July 2020].

McCall, J. C. & Trivedi, M. M., 2006. Video-based lane estimation and tracking for driver assistance: survey, system, and evaluation. *IEEE Transactions on Intelligent Transportation Systems,* 7(1), pp. 20-37.

McCulloch, W. S. & Pitts, W., 1943. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics,* 5(4), pp. 115-133.

Phung, H. V. & Rhee, E. J., 2019. A High-Accuracy Model Average Ensemble of Convolutional Neural Networks for Classification of Cloud Image Patches on Small Datasets. *Applied Sciences,* Volume 9, p. 4500.

Sadek, R. M. et al., 2019. Parkinson's Disease Prediction Using Artificial Neural Network. *International Journal of Academic Health and Medical Research,* 3(1), pp. 1-8.

Saeed, A., 2017. *Using Genetic Algorithm for optimizing Recurrent Neural Network.* [Online] Available at: http://aqibsaeed.github.io/2017-08-11-genetic-algorithm-for-optimizing-rnn/
[Accessed 6 February 2020].

Springenberg, J. T., Dosovitskiy, A., Brox, T. & Riedmiller, M., 2014. Striving for simplicity: The all convolutional net. *arXiv preprint arXiv:1412.6806.*

Stein, B. v., Wang, H. & Bäck, T., 2018. *Automatic Configuration of Deep Neural Networks with EGO,* s.l.: ArXiv 2018.

Tabik, S., Peralta, D., Herrera-Poyatos, A. & Herrera, F., 2017. A snapshot of image Pre-Processing for convolutional neural networks: Case study of MNIST. *International Journal of Computational Intelligence Systems,* Volume 10, pp. 555-568.

Törn, A. & Zilinskas, A., 1989. *Global Optimization.* 1st ed. s.l.:Springer-Verlag Berlin Heidelberg.

Vanhatalo, J., Jylänki, P. & Vehtari, A., 2009. Gaussian process regression with Student-t likelihood. In: *Advances in Neural Information Processing Systems 22.* s.l.:Curran Associates, Inc., pp. 1910-1918.

Xia, Q. T. a. L. N. a. Y. W. a. T. D. a. W. A. a. J. C. a. S.-T., 2017. Student-t Process Regression with Student-t Likelihood. In: *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence.* s.l.:s.n., pp. 2822-2828.

Yang, X.-S., 2010. A New Metaheuristic Bat-Inspired Algorithm. In: J. R. González, et al. eds. *Nature Inspired Cooperative Strategies for Optimization (NICSO 2010).* Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 65-74.

Yang, X.-S., 2013. Optimization and Metaheuristic Algorithms in Engineering. In: X. Yang, A. H. Gandomi, S. Talatahari & A. H. Alavi, eds. *Metaheuristics in Water, Geotechnical and Transport Engineering.* Oxford: Elsevier, pp. 1-23.

Yang, Z. et al., 2015. Deep Fried Convnets. *Proceedings of the IEEE International Conference on Computer Vision,* pp. 1476-1483.

Yoo, Y., 2019. Hyperparameter optimization of deep neural network using univariate dynamic encoding algorithm for searches. *Knowledge-Based Systems,* Volume 178, pp. 74-83.

Zeiler, M. D. & Fergus, R., 2013. Stochastic Pooling for Regularization of Deep Convolutional Neural Networks. *arXiv preprint arXiv:1301.3557.*

Zhou, V., 2019. *Machine Learning for Beginners: An Introduction to Neural Networks.* [Online]
Available at: https://victorzhou.com/blog/intro-to-neural-networks/
[Accessed 23 12 2019].

# Appendix 1 – Human Fall Classification Best Configurations

| T | Ax 1 | Ay 1 | Az 1 | Ax 2 | Ay 2 | Az 2 | Gx | Gy | Gz | S | D | E | TO | BS | NpHL | HL | HLAF |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| × | × | ✓ | × | ✓ | × | × | × | ✓ | × | ✓ | × | 2 | RMSProp | 16 | 1 | 3 | ReLU |
| × | × | ✓ | × | × | ✓ | × | × | ✓ | ✓ | ✓ | ✓ | 2 | Adam | 32 | 1 | 1 | Sigmoid |
| × | × | ✓ | × | ✓ | × | ✓ | × | × | × | ✓ | ✓ | 3 | Adam | 32 | 1 | 2 | ReLU |
| × | × | ✓ | ✓ | × | × | ✓ | ✓ | ✓ | ✓ | × | × | 4 | RMSProp | 16 | 16 | 2 | Sigmoid |
| × | ✓ | ✓ | ✓ | × | × | ✓ | ✓ | ✓ | × | ✓ | × | 4 | RMSProp | 16 | 16 | 2 | Sigmoid |
| × | ✓ | × | ✓ | × | ✓ | × | × | × | × | × | × | 1 | SGD | 16 | 31 | 1 | Sigmoid |
| × | ✓ | ✓ | × | × | ✓ | × | × | × | × | ✓ | × | 2 | SGD | 16 | 1 | 2 | Sigmoid |
| × | × | × | × | ✓ | × | ✓ | ✓ | ✓ | × | × | ✓ | 1 | RMSProp | 32 | 1 | 1 | ReLU |
| × | ✓ | × | ✓ | ✓ | × | ✓ | ✓ | × | × | × | ✓ | 3 | Adam | 32 | 16 | 1 | Sigmoid |
| × | ✓ | ✓ | ✓ | × | ✓ | × | ✓ | × | ✓ | × | ✓ | 3 | Adam | 16 | 1 | 3 | ReLU |

**T** – Time
**Ax 1** – Accelerometer 1 X-axis
**Ay 1** – Accelerometer 1 Y-axis
**Az 1** – Accelerometer 1 Z-axis
**Ax 2** – Accelerometer 2 X-axis
**Ay 2** – Accelerometer 2 Y-axis
**Az 2** – Accelerometer 2 Z-axis
**Gx** – Gyroscope X-axis
**Gy** – Gyroscope Y-axis
**Gz** – Gyroscope Z-axis
**S** – Sound
**D** – Doppler
**E** – Epochs
**TO** – Training Optimizer
**BS** – Batch Size
**NpHL** – Nr. of Neurons per Hidden Layer
**HL** – Nr. of Hidden Layers
**HLAF** – Hidden Layers Activation Function

# Appendix 2 – Vibration Source Classification Best Configurations

| T | Ax 1 | Ay 1 | Az 1 | Ax 2 | Ay 2 | Az 2 | S | D | E | TO | BS | NpHL | HL | HLAF |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| × | ✓ | × | × | ✓ | ✓ | × | ✓ | × | 5 | Adam | 32 | 36 | 2 | TanH |
| × | ✓ | × | × | ✓ | ✓ | × | ✓ | × | 5 | Adam | 32 | 36 | 2 | TanH |
| × | ✓ | ✓ | × | ✓ | × | × | ✓ | × | 4 | Adam | 16 | 31 | 3 | TanH |
| × | × | × | × | ✓ | ✓ | × | ✓ | × | 1 | Adam | 16 | 36 | 4 | TanH |
| × | ✓ | × | × | ✓ | × | × | ✓ | × | 3 | RMSProp | 16 | 36 | 4 | TanH |
| × | ✓ | × | × | ✓ | ✓ | × | ✓ | × | 5 | Adam | 32 | 36 | 1 | TanH |
| × | ✓ | × | × | ✓ | ✓ | × | ✓ | × | 5 | Adam | 32 | 36 | 2 | TanH |
| × | × | ✓ | × | ✓ | ✓ | × | ✓ | ✓ | 2 | RMSProp | 32 | 36 | 5 | TanH |
| × | × | ✓ | × | ✓ | ✓ | × | ✓ | ✓ | 2 | RMSProp | 16 | 36 | 5 | TanH |
| × | ✓ | × | × | ✓ | × | × | ✓ | × | 3 | RMSProp | 16 | 36 | 4 | TanH |

**T** – Time
**Ax 1** – Accelerometer 1 X-axis
**Ay 1** – Accelerometer 1 Y-axis
**Az 1** – Accelerometer 1 Z-axis
**Ax 2** – Accelerometer 2 X-axis
**Ay 2** – Accelerometer 2 Y-axis
**Az 2** – Accelerometer 2 Z-axis
**S** – Sound
**D** – Doppler
**E** – Epochs
**TO** – Training Optimizer
**BS** – Batch Size
**NpHL** – Nr. of Neurons per Hidden Layer
**HL** – Nr. of Hidden Layers
**HLAF** – Hidden Layers Activation Function

# Appendix 3 – Convolutional Neural Network Best Configurations

| E | TO | BS | LR | CPL | KpCL | CKS | CAF | PKS | FCL | NpFCL | FCLAF |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 15 | Adagrad | 16 | 0.01 | 1 | 4 | 3 | Sigmoid | 2 | 3 | 150 | TanH |
| 15 | Adagrad | 16 | 0.1 | 1 | 4 | 3 | ELU | 3 | 1 | 150 | ELU |
| 15 | Adagrad | 16 | 0.01 | 1 | 4 | 3 | Sigmoid | 2 | 3 | 150 | TanH |
| 15 | SGD | 16 | 0.1 | 1 | 2 | 3 | Sigmoid | 2 | 1 | 150 | Sigmoid |
| 12 | SGD | 16 | 0.1 | 1 | 3 | 3 | ELU | 2 | 3 | 200 | TanH |
| 15 | Adagrad | 16 | 0.1 | 1 | 4 | 3 | Sigmoid | 2 | 3 | 150 | TanH |
| 3 | SGD | 32 | 0.1 | 2 | 4 | 3 | ELU | 2 | 3 | 150 | ELU |
| 15 | SGD | 64 | 0.1 | 1 | 2 | 3 | Sigmoid | 2 | 2 | 200 | TanH |
| 6 | Adagrad | 32 | 0.1 | 1 | 4 | 2 | ELU | 2 | 2 | 150 | TanH |
| 6 | Adagrad | 32 | 0.1 | 1 | 4 | 2 | ELU | 3 | 1 | 200 | TanH |

**E** – Epochs
**TO** – Training Optimizer
**BS** – Batch Size
**LR** – Learning Rate
**CPL** – Nr. of Convolution and Max Pooling Layers Pairs
**KpCL** – Nr. of Kernels per Convolution Layer
**CKS** – Convolution Layers Kernel Size
**CAF** – Convolution Layers Activation Function
**PKS** – Max. Pooling Layers Kernel Size
**FCL** – Nr. of Fully Connected Layers
**NpFCL** – Nr. of Neurons per Fully Connected Layer
**FCLAF** – Fully Connected Layers Activation Function