University of Sheffield

Applications of Hyper-parameter Optimisations for Static Malware Detection.

Fahad T. ALGorain

A thesis submitted to the University of Sheffield in partial fulfilment
for the degree of Doctor of Philosophy

The University of Sheffield
Faculty of Engineering
Department of Computer Science

March 31, 2023

Declaration

I Fahad T. ALGorain, confirm that the thesis is my own work. This work has not been previously been presented for an award at this, or any other, university. Some parts of this thesis have been published by the author.

Fahad T. ALGorain

Acknowledgments

Firstly, my eternal and deepest appreciation goes to my wife Hind Aldossary and my kids (Joud, Ahmed and Farah) who have supported me during my studies. For their unconditional love and patience when I was not in the mood to do anything and my studies have piled up in the office. Without your presence in my life, I do not know how to proceed and make the correct choices. When I think about some moments in the PhD, where I felt down, you came and pulled me out of it. With you guys in my life, many hard challenges were successfully navigated. I thank God for the moment our lives intertwined like this. Thanks for everything! Also, I would like to extend my thanks to my parents and siblings as well for their motivational talks, calls and prayers. In addition, I would like to express my sincere gratitude to my supervisor John A Clark for his guidance and support during my PhD studies. He helped me realize how to successfully collaborate on research and how a PhD student can create original and rigorous work of relevance to modern-day concerns. I can say with my head held high that I am really grateful for every intellectual conversation that we had during the course of this PhD. I wish him eternal blessings during his life. Furthermore, I would like to thank my research group for their insights during the course of my work. Special thanks go to my colleagues Abdullah Alsharif, Ibrahim Althmali, Nasser Albunian, Aryan Pasikhani, Abdulmonem Alshahrani, Abdullah ALQahtani and Mohammed Al-Hamidi for their continuous support, invaluable advice and encouragement that they have given during this endeavour.

Abstract

Malware detection is a major security concern and a great deal of academic and commercial research and development is directed at it. Machine Learning is a natural technology to harness for malware detection and many researchers have investigated its use. However, drawing comparisons between different techniques is a fraught affair. For example, the performance of ML algorithms often depends significantly on parametric choices, so the question arises as to what parameter choices are optimal. In this thesis, we investigate the use of a variety of ML algorithms for building malware classifiers and also how best to tune the parameters of those algorithms – a process generally known as hyper-parameter optimisation (HPO). Firstly, we examine the effects of some simple (model-free) ways of parameter tuning together with a state-of-the-art Bayesian model-building approach. We demonstrate that optimal parameter choices may differ significantly from default choices and argue that hyper-parameter optimisation should be adopted as a 'formal outer loop' in the research and development of malware detection systems. Secondly, we investigate the use of covering arrays (combinatorial testing) as a way to combat the curse of dimensionality in Gird Search. Four ML techniques were used: Random Forests, xgboost, Light GBM and Decision Trees. cAgen (a tool that is used for combinatorial testing) is shown to be capable of generating high-performing subsets of the full parameter grid of Grid Search and so provides a rigorous but highly efficient means of performing HPO. This may be regarded as a 'design of experiments' approach. Thirdly, Evolutionary algorithms (EAs) were used to enhance machine learning classifier accuracy. Six traditional machine learning techniques baseline accuracy is recorded. Two evolutionary algorithm frameworks Tree-Based Pipeline Optimization Tool (TPOT) and Distributed Evolutionary Algorithm in Python (Deap) are compared. Deap shows very promising results for our malware detection problem. Fourthly, we compare the use of Grid Search and covering arrays for tuning the hyper-parameters of Neural Networks. Several major hyper-parameters were studied with various values and results. We achieve significant improvements over the benchmark model. Our work is carried out using EMBER, a major published malware benchmark dataset of Windows Portable Execution (PE) metadata samples, and a smaller dataset from kaggle.com (also comprising of Windows Portable Execution metadata). Overall, we conclude that HPO is an essential part of credible evaluations of ML-based malware detection models. We also demonstrate that high performing hyper-parameter values can be found by HPO and that these can be found efficiently.

Contents

Acknowledgments **ii**

1 Introduction **1**

1.1 Malware and its Detection . . . 1
1.2 Machine Learning-Based Detection and Hyper-parameter Optimisation . . . 1
1.3 Scope of the Thesis . . . 3
1.4 Thesis Hypothesis . . . 3
1.5 Thesis Contributions . . . 4
1.6 Structure of the Thesis . . . 4
1.7 Publications . . . 5

2 Background and Literature Review **6**

2.1 Malware and Windows Portable Executable Files . . . 6
2.1.1 What is Malware? . . . 6
2.1.2 Portable Executable (PE) Files . . . 6
2.1.3 PE File Format and Structure . . . 7
2.2 Machine Learning (ML) algorithms . . . 9
2.3 Supervised ML algorithms . . . 9
2.3.1 Linear Models . . . 9
2.3.2 Logistic regression (LR) . . . 9
2.3.3 Stochastic Gradient Descent (SGD) . . . 9
2.3.4 K-nearest neighbour (KNN) . . . 10
2.3.5 Naive Bayes (NB) . . . 10
2.3.6 Tree-based Models . . . 10
2.3.7 Random Forests (RFs) . . . 11
2.3.8 Gradient Boosting Decision Trees (GBDTs) . . . 11
2.3.9 Extreme Gradient Boosting (xgboost) . . . 11
2.3.10 Light Gradient Boosting Machines (LightGBM) . . . 11
2.4 Deep Learning (DL) Models . . . 12
2.4.1 Neural Network Structural Model . . . 13
2.4.2 Training of A Neural Network . . . 16
2.4.3 ML-Based Static Malware Detection Related Literature . . . 16

2.5 Formal Definition of HPO and Motivation for its Use in Malware Classification . . . 17
2.6 Hyper-Parameter Optimisation (HPO) 18
2.6.1 Main Process of HPO 18
2.7 General Categories of Hyper-Parameter Optimisation 19
2.7.1 Traditional Model-Free Blackbox approaches 19
2.8 Model-Based approaches to HPO 20
2.8.1 Bayesian Optimisation (BO) 20
2.8.2 Sequential Model-Based Optimisation (SMBO) 20
2.8.3 Tree-Structured Parzen Estimators (TPE) 21
2.9 HPO using Deep Learning 22
2.9.1 Selection of Hyper-parameter Optimisation for Deep Learning 22
2.9.2 Approaches of Hyper-parameter Optimisation for Deep Learning 22
2.10 HPO using Metaheuristic Approaches 22
2.10.1 Genetic Algorithms (GAs) 23
2.10.2 Particle Swarm Optimisation (PSO) 24
2.10.3 Distributed Evolutionary Algorithm in Python (Deap) 24
2.10.4 Tree-Based Pipeline Optimisation Tool (TPOT) 25
2.11 HPO-Related Literature 25
2.11.1 A Comparison between Different HPO Approaches for ML models 26
2.12 Covering Arrays: An Experimental Design Approach to HPO 28
2.12.1 What is Combinatorial Testing (CT)? 28
2.12.2 Covering Arrays - Related Literature 29
2.13 Overview of the cAgen Tool 29
2.13.1 Workspaces 30
2.13.2 Input Parameter Model (IPM) 30
2.13.3 Generate 31
2.14 Datasets Background 32
2.15 Summary and Links to the Research Hypotheses 38

3 Bayesian Hyper-Parameter Optimisation 39
3.1 Introduction 39
3.2 Research Question 40
3.3 Experiments 40
3.3.1 Execution Environment 40
3.3.2 Experiments with Default Settings 40
3.3.3 Model Hyper-parameter Optimisation 41
3.4 Results 41
3.5 Discussion 52
3.6 Conclusions 52
3.7 Summary 52

4 Covering Arrays ML HPO for Static Malware Detection 54
4.1 Introduction . . . 54
4.1.1 Covering Arrays: Dealing with the Curse of Dimensionality . . . 54
4.1.2 Generating Covering Arrays . . . 55
4.2 Research Question . . . 56
4.3 The cAgen Array Generator . . . 56
4.3.1 Parameter Specification and Array Generation . . . 56
4.3.2 Array Indexing . . . 57
4.4 Methodology . . . 58
4.5 Experiments . . . 58
4.5.1 Execution Environment . . . 59
4.5.2 Experiments Settings . . . 59
4.5.3 Implementation Details . . . 59
4.6 Results . . . 60
4.6.1 Comparing the Results of CA against another Benchmark . . . 65
4.7 Conclusions . . . 65
4.8 Summary . . . 65

5 HPO for Evolutionary Algorithms 66
5.1 Introduction . . . 66
5.2 Aim of this chapter . . . 66
5.3 Research Question . . . 66
5.4 Experiment Setup . . . 67
5.4.1 Initial Experiments and Evaluation Metrics . . . 67
5.4.2 Deap Setup . . . 67
5.4.3 TPOT Setup . . . 68
5.4.4 TPOT Pre-built Configurations . . . 68
5.4.5 TPOT Custom Configuration . . . 68
5.4.6 Target ML Algorithms . . . 69
5.5 Results and Discussion . . . 69
5.5.1 Deap Results . . . 69
5.5.2 TPOT Results . . . 71
5.6 Conclusion . . . 71
5.7 Summary . . . 72

6 Using CA and Grid Search with Deep Learning 73
6.1 Introduction . . . 73
6.1.1 Aim of this Chapter . . . 73
6.1.2 Chapter Contribution . . . 73
6.2 Research Question . . . 74
6.3 Experiment Setup and Early Notes . . . 74
6.3.1 Experiment Setup . . . 74

6.3.2 Baseline Model 74
6.4 Grid Search versus cAgen for NN Hyper-parameter Optimization Tasks 75
6.4.1 Grid Search 75
6.4.2 cAgen 75
6.4.3 Hyper-parameter Grid Search Configurations 76
6.5 Results 76
6.6 Discussion 78
6.6.1 Tuning the Number of Neurons 78
6.6.2 Tuning the Number of Layers 78
6.6.3 Tuning of Activation Functions 78
6.6.4 Tuning Dropout relay 79
6.6.5 Choice of Optimisers 79
6.6.6 Tuning the Number of Epochs and Batch Size 79
6.7 Conclusion 80
6.8 Summary 80

7 Conclusions and Future Work **81**
7.1 Context and Motivation 81
7.2 Recap: the Research Hypotheses 83
7.3 Evaluating the Evidence for the Hypotheses 83
7.4 Limitations and Future Work 84
7.4.1 Dataset Issues 84
7.4.2 Other Limitations and Future Work 85
7.5 Conclusions 86
7.6 Acknowledgments of the Use of Freely Available Software 87
7.7 And Finally 88

List of Figures

2.1 PE File Structure [1]. 8
2.2 A simple Feed Forward Neural Network showing (a) structure, (b) how a weighted sum is calculated and fed into an activation function to produce a node value (output), and (c) common activation functions [2] 13
2.3 Softsign and Hard_sigmoid Activation Functions Example [3] 14
2.4 Softmax Activation function Example [4] 15
2.5 Linear Activation function Example [3] 15
2.6 Example of how TPOT would work with RF Classifier 26
2.7 Comparison of Common HPO Algorithms (courtesy of [5]) 28
2.8 Workspaces with one selected to work on 30
2.9 Shows the input parameter of xgboost ML model 31
2.10 Shows a t1 test set of xgboost ML model 32
2.11 courtesy of [6] 34
2.12 courtesy of [6] 35
2.13 courtesy of [7] 36

3.1 Highest Validation Score at each Iteration for AHBO-TPE (yellow) and Random Search (blue) (Ember Dataset). 43
3.2 ROC AUC Comparison for AHBO-TPE (Cyan), Random Search (Yellow), and Default Benchmark Model (Red) applied to the Ember Dataset. 48
3.3 FPR and TPR Comparison for AHBO-TPE (Cyan), Random Search (Yellow), and Default Benchmark Model (Red) applied to the Ember Dataset. 49

4.1 Full Parameter Specification for ABC Example 57
4.2 Array Generation for ABC example above with t=2 57
4.3 cAgen ML Models Results Comparison for Strength $t = 2$ 63
4.4 cAgen ML Models Results Comparison for Strength $t = 3$ 64
4.5 cAgen ML Models Results Comparison for Strength $t = 4$ 64

6.1 cAgen Implementation Detail for both Ember and the Kaggle Datasets 75

List of Tables

2.1 Common Techniques, Main Hyper-parameters, HPO Methods and Available Libraries 27
2.2 Previous Related Work to Datasets 37

3.1 Score Comparison of ML Models with Default Parameters (Ember Dataset). 41
3.2 Score Comparison of ML Models Before / After Optimisation (Ember Dataset). . . . 42
3.3 Score Comparison of the Remaining ML Models using AHBO-TPE (Ember Dataset). 43
3.4 Completion Time Results for Selected ML Models with AHBO-TPE (Ember Dataset). 44
3.5 Score comparisons for the Application of HPO (Kaggle Dataset). 44
3.6 LightGBM Grid Search Hyper-parameter Results (Ember Dataset). 45
3.7 LightGBM Random Search Hyper-parameter Results (Ember Dataset). 45
3.8 LightGBM AHBO-TPE Search Hyper-parameter Results (Ember Dataset). 46
3.9 SGD Model AHBO-TPE Search Hyper-parameter Results (Ember Dataset). 46
3.10 RF Model AHBO-TPE Search Hyper-parameter Results (Ember Dataset). 47
3.11 LR Model AHBO-TPE Search Hyper-parameter Results (Ember Dataset). 47
3.12 KNN Model AHBO-TPE Search Hyper-parameter Results (Ember Dataset). 47
3.13 Confusion matrix for three classification models 49
3.14 False positive rate (FPR) and false negative rate (FNR) for each classification model . 50
3.15 ML Models Hyper-parameter results using AHBO-TPE (Kaggle Dataset). 51

4.1 ML Models cAgen configurations 59
4.2 RF Model cAgen Results Comparison 61
4.3 LightGBM Model cAgen Results Comparison 61
4.4 DT Model cAgen Results Comparison 62
4.5 Xgboost Model cAgen Results Comparison 62

5.1 Pre-initialised Deap Hyper-parameter Configuration Ranges 67
5.2 Pre-initialised TPOT hyper-parameter configurations (Config-dict) 68
5.3 ML Model Accuracy: Comparison using Default and Deap Optimization 69
5.4 ML Model and Deap Search Space / Hyper-parameter results 70
5.5 TPOT built-in Configuration Search Space Results. 71

6.1 DL Model Grid Search Hyper-parameter Configurations (Ember Dataset) 76

6.2 DL Model Grid Search Configuration Space (Kaggle Dataset) 76
6.3 DL Model cAgen Results Comparison (Kaggle Dataset) 77
6.4 DL Model cAgen Results Comparison (Ember Dataset) 77

Chapter 1

Introduction

1.1 Malware and its Detection

Malware is any malicious software that causes harm to the users of a computer system. It is one of the most pressing problems in modern cybersecurity, and its detection has been a long-standing focus for academic and commercial research and development [8]. There are many types of malware, and each has its own characteristics [9, 10]. New malware families and new variants of existing families are constantly emerging. This means detection techniques must evolve and improve too. There are three approaches to malware detection: static, dynamic, and hybrid detection. **Static malware detection** analyses malicious binary files without executing them. This is the focus of this thesis. **Dynamic malware detection** uses features of run-time execution behaviour to identify malware. **Hybrid detection** combines the previous two approaches. Detection must be adequate, i.e. exhibit low false positives and negatives, but also efficient, particularly in areas such as forensics or threat hunting where vast file storage may need to be scanned for malware. Furthermore, the malware environment constantly changes, so detectors' (re-)training or reconfiguration speed is also important [11].

1.2 Machine Learning-Based Detection and Hyper-parameter Optimisation

Machine learning (ML) is one of the highest-profile technologies of our age. Its theory develops apace, and the number of successful applications to modern-day problems is huge. It also has the potential to play a critical role in detecting malware by any of the three approaches identified above. Machine Learning is an obvious avenue to pursue, with various advantages to harnessing it for malware detection and categorisation: an ML approach can significantly reduce manual effort in developing detectors, giving more rapid deployment; ML can play a critical role in the extraction of insight from malware samples; and ML-based detectors can detect some unseen malware, e.g. unseen malware that has features that are similar to those of known malware may be detected because of the loose pattern matching that underpins many ML classification approaches.

Malware *detection* is the focus of this thesis and aims to determine if a file or behaviour is malicious. Ground truth labels (indicating whether an instance of software is malware or not) are not always available or necessary for this task. Both supervised machine learning approaches (where labels are available) and unsupervised machine learning approaches (when labels are not available) have been applied. (Semi-supervised learning, which can be applied when some labels are available, has received much less attention. Malware *classification*, on the other hand, involves categorising malware into specific families or types. This generally requires ground truth labels.

A large number of ML techniques have indeed been brought to bear on the malware detection problem, often attaining good results. However, ML must not be seen as a toolkit that can be thrown at any problem. Many ML techniques are parameterised, and the choice of parameters may significantly affect performance. In modern, widely used ML tool-kits, algorithms often have many tens of parameters (and sometimes more). This leads to the thorny issue of how such parameters may be best set, a problem generally referred to as hyper-parameter optimisation (HPO). Suitable HPO has the potential to improve on the results obtained by a specific detection approach but also to enable fair comparison of techniques that are not the specific focus of the investigation. Manual tuning is often simply impossible. (As an aside, we observe that many commercial ML users spend a great deal of time tuning for their specific needs).

Existing ML toolkits address this problem to some extent by adopting *default values* for parameters; these values have been shown to work plausibly over many problems. However, for any specific problem, it is far from clear that the default values will be the best, or even good, choices. We have significant domain incentives to gain the best possible results for malware detection.

The negative impact of false positive (FP) and false negative (FN) classifications differs significantly. Academic research typically treats the relative importance as equal (e.g., using accuracy or F1-score as a target performance metric). It almost universally fails to indicate what levels of false negatives and false positives are acceptable. The anti-malware industry, however, associates a much higher cost with FPs than with FNs. FPs are largely considered unacceptable, and FNs are largely considered a limitation. So in practice, the anti-malware industry does not seek to minimize both errors but to minimize FNs while keeping FPs at zero. This is a perfectly reasonable stance by the anti-malware industry. There is a general informed acceptance that anti-malware will not protect against all malware (and so some level of FNs is to be expected). The negative impact of FPs, e.g. the waste of skilled analysis time and system inconveniences incurred by management action before full resolution of the raised issues, may bring the anti-malware itself into disrepute and destroy faith in its general operation. The risk managers in individual organisations are free to deviate from this position, but in practice, the same issues are often faced.

We observe that in a more general engineering context, there are many cases where false alarms (FPs) have led to operator actions that simply increase risk. Safety system alerts, for example, have been switched off by operators if they are found to waste their time. Where anti-malware is widely distributed the reputation damage that may be incurred by FPs is multiplied.

Whatever the relative costs assumed for each type of error, the detection models developed by ML algorithms depend on choices of its parameters: making high-performing hyper-parameter choices matter. In this thesis, we explore various ML techniques applied to malware classification. We aim

to demonstrate that hyper-parameter tuning is important for ML users in the research community focusing on security applications, particularly static malware detection. For comparison purposes, our experiments typically use the evaluation criteria adopted in reviewed research works.

1.3 Scope of the Thesis

The work presented in this thesis explores various approaches to hyper-parameter optimisation in the context of ML-based static detection of malware. It focuses on a prevalent form of malware: Windows Portable Execution (PE) files. Windows is the most common user operating system; malware writers often target it. The thesis uses two publicly available datasets for its work. One is available from the ML competition website kaggle.com [12]. The other is EMBER, a much larger dataset. We use the 2018 version of EMBER, which, its authors say, was curated to be specifically challenging for ML-based approaches. We do not formally explore types of malware other than Windows PE files. The application of the techniques deployed in this thesis to other types of malware is left as future work. The encouragement to pursue hyper-parameter optimisation came from several sources, e.g. [13, 5]. The scope for hyper-parameter optimisation (HPO) seems excellent. To the best of our knowledge, there is no significant exploration of the application of HPO in Static PE malware detection.

1.4 Thesis Hypothesis

The main thesis hypothesis is:

- ***Main Hypothesis**:* HPO can significantly improve the performance of static malware detectors based on machine learning.

We explore three general approaches to HPO:

- A Bayesian approach where the results of trials of sets of hyper-parameter values are used to inform the selection of the next candidate set;
- Covering Arrays, a technique that provides a concise but diverse set of candidate parameter sets to evaluate and which may be regarded as a *Design of Experiments* approach; and
- Optimisation-based approaches using evolutionary algorithms (DEAP and TPOT) to search the hyper-parameter space.

More specific research hypotheses are:

- ***Hypothesis1**:* AHBO-TPE is an efficient and effective technique for hyper-parameter optimisation of ML-based malware detectors. It can find high-performance hyper-parameter vectors more quickly than comparable techniques.

- ***Hypothesis2****:* A Covering Array is an efficient technique for hyper-parameter optimisation of ML-based malware detectors. It can find significantly better hyper-parameters in comparison to Grid Search and do so with reduced computation.
- ***Hypothesis3****:* An Evolutionary Algorithm can find new or improved hyper-parameter vectors that give better performance than the defaults of ML-based malware detectors.
- ***Hypothesis4****:* Grid Search and Covering Arrays can be used to efficiently achieve high-performing parameter choices for Deep Neural Network based malware detectors.

The first three hypotheses are evaluated using a variety of underpinning ML classification approaches. The fourth hypothesis extends our investigation into HPO for optimising the model parameters of neural networks. All assume that the context is static malware detection of Windows PE files.

1.5 Thesis Contributions

In the context of static ML-based Windows PE malware detection, the contributions of this thesis are as follows:

- The development of benchmarks for the use of various ML techniques with their default model parameters. We demonstrate that optimal parameter choices may differ significantly from default choices. We argue that hyper-parameter optimisation should be adopted as a 'formal outer loop' in the research and development of malware detection systems.
- A demonstration of the effectiveness and the time taken by established model-free approaches to the hyper-parameterisation of such models. Specifically, we apply Grid Search and Random Search for HPO purposes.
- Demonstration of the efficiency and effectiveness of a specific Bayesian approach to such HPO.
- The demonstration of the efficiency and effectiveness of Covering Arrays applied to the model parameter domains of ML approaches.
- A demonstration of the efficiency and effectiveness of two evolutionary algorithms for the HPO of ML-based malware detectors.

1.6 Structure of the Thesis

The rest of the thesis is structured as follows:

- Chapter 2 surveys the background literature relevant to the thesis work, covering malware (particularly Windows PE files), its detection, ML techniques and provides an introduction to Covering Arrays.

- Chapter 3 provides an exploration of classical ML classifier techniques applied with default parameters and a newer ML approach. These are subject to classic model-free HPO approaches (Random Search and Grid Search). A specific Bayesian HPO approach is evaluated.
- Chapter 4 explores the use of Covering Arrays as an HPO approach for the classical ML techniques of interest.
- Chapter 5 explores the use of two evolutionary algorithm approaches (DEAP and TPOT) to enhance and optimise selected ML models' default parameters.
- Chapter 6 explores HPO approaches for Deep Neural Networks (specifically, the use of Covering Arrays).
- Chapter 7 provides conclusions and identifies future work.

1.7 Publications

Works in this thesis have appeared in the following publications:

1. **Fahad T. ALGorain and John A. Clark. "Bayesian Hyper-Parameter Optimisation for Malware Detection." (2021)."`https://ceur-ws.org/Vol-3125/paper6.pdf`"**
2. **F.T ALGorain and J.A. Clark, "Bayesian Hyper-Parameter Optimisation for Malware Detection." Electronics 2022, 11, 1640. "`https://doi.org/10.3390/electronics11101640`"**
3. **F.T. ALGorain and J.A. Clark, "Covering Arrays ML HPO for Static Malware Detection." Eng 2023, 4, 543-554. "`https://doi.org/10.3390/eng4010032`"**

Chapter 2

Background and Literature Review

This chapter provides background and reviews literature concerned with Windows Portable Execution (PE) files, HPO, and its application in static malware detection.

2.1 Malware and Windows Portable Executable Files

2.1.1 What is Malware?

Malware is any malicious software that causes harm to the users of a computer system. Recently, malware has become more sophisticated and anti-malware software is frequently becoming circumvented or deceived. Malware authors use various ways to bypass current detectors, e.g. using various *obfuscation techniques* such as binary-packers. Special measures are needed to find such malware. In this thesis, we are concerned with static malware detection approaches. In particular, our work deals with the detection of Windows Portable Executable (PE) files, a very common form of malware.

2.1.2 Portable Executable (PE) Files

The Microsoft file format PE, which stands for portable executable, is used for executable files, object files, dynamic-link library files (DLLs), and some other types of files. Since the release of Windows NT 3.1, this format has been utilised by Windows operating systems. Any file with the following extensions—.cpl,.dll,.exe,.ocs,.scr, and. sys—has a PE file format. This format makes it possible for Windows to manage executable code. These PE files contain data that instructs the Windows operating system on how to load and run them. They are trustworthy, harmless files that are essential to all of Microsoft's operating systems. When the PE files are contaminated with malicious code then this trust is misplaced.

When arbitrary or harmful code is added to a portable executable, PE infection results. It is not difficult to insert malicious code into PE files because the PE format was not intended to be resistant to code change. PE files are frequently infected by various types of malware, e.g. Trojans, backdoors, ransomware, worms, and advanced persistent threat (APT) malware [14, 15, 16]. Once malware has access to a computer's PE files, it can often run undetected by the user.

2.1.3 PE File Format and Structure

The PE file format is the most common executable format for Windows OS and its executables, DLLSs and FONs (font files). It comprises a number of standard headers (PE-32 format), see the structure of PE in 2.1, followed by different *sections* [17]. In the Header section, we can see the PE file structure COFF (Common Object File Format). This file contains sensitive information such as machine type (windows, MAC etc.), the format of the file (e.g. DLL or EXE), the number of sections, symbols etc. The optional header has information about the linker version, code size, initialisation size and uninitialised data, and address entry pointers to the sections that follow. Pointers to components like export tables, import tables, resources, exceptions, debug information, certificate information, and relocation tables are kept in data directories. As a result, they summarise the contents of any executables [18]. The last part consists of Section Tables which show the name, offset and size of all sections in a PE file. PE sections have codes and initialised data that the Windows loader uses to navigate into the executables or readable/writable memory pages. The same also goes for imports, export and resources defined by the file. Different sections have a header that specifies the size and the address. The import table address instructs the loader which functions should be imported statically.

The resources section contains important information about user interfaces, e.g. cursors, fonts, icons, and menus. Normally a PE file contains a .text code section, and others (.dat, .rdatat or. bss). Section .reloc is the place in which relocation tables usually reside. This gives the windows loader the ability to reassign the base address from the preferred base of the executables. The section .tls has special thread local storage (TLS) to store thread-specific local variables. These have been exploited to redirect the entry point of an executable in order to check if a debugger or any other tool is being run [19]. For more information about PE files, the reader is referred to [20]

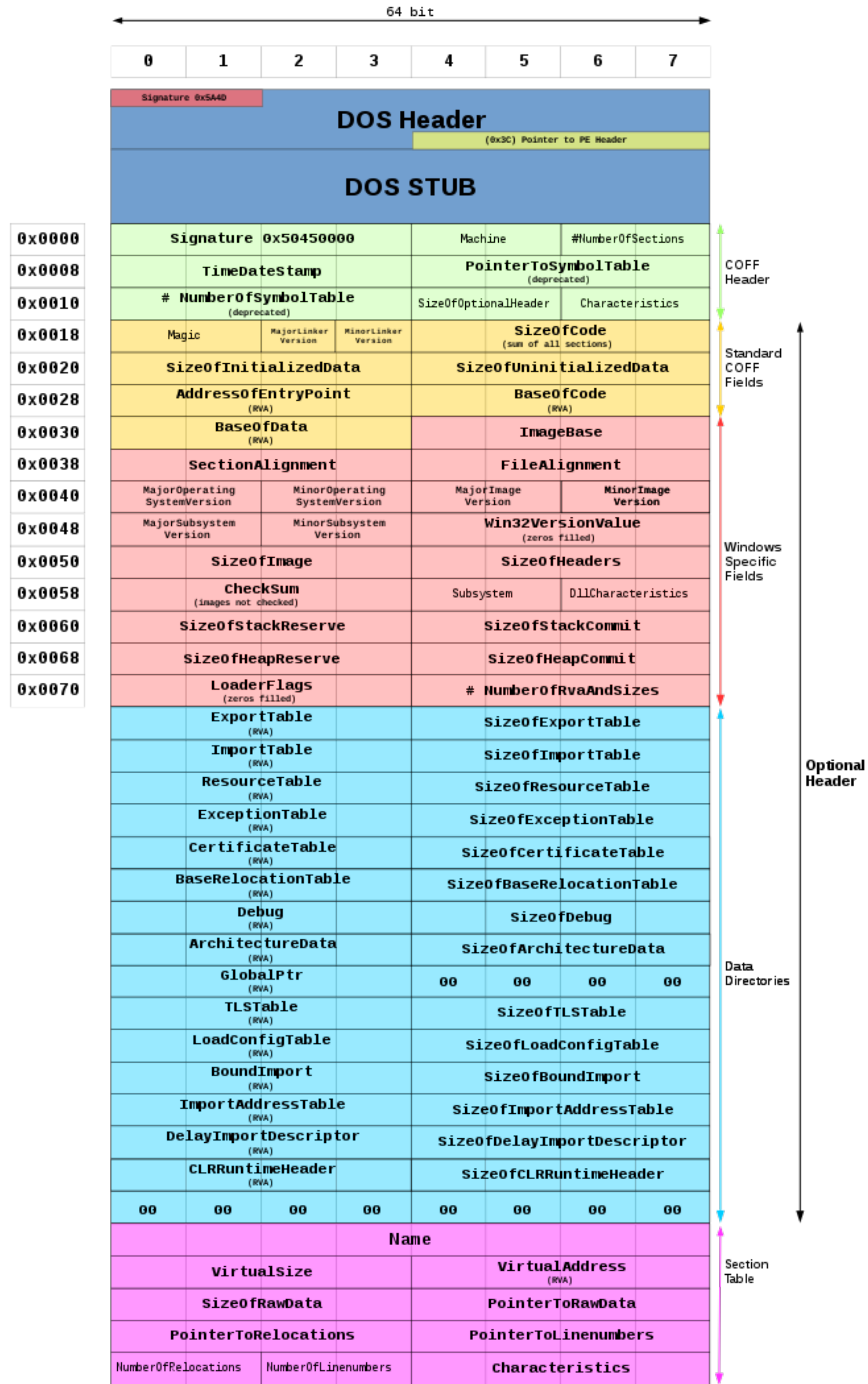

Figure 2.1: PE File Structure [1].

2.2 Machine Learning (ML) algorithms

ML algorithms are of three forms: supervised learning, which needs data to be labelled in order to train a model; unsupervised learning where unlabeled data are typically used to learn patterns of 'similar' data inputs (i.e. data classes); and semi-supervised learning, in which the previous two are combined to get the desired output.

2.3 Supervised ML algorithms

In any given supervised learning technique, both labels x (features) and y (target) are available. The aim here is to find an optimal predicted value of a model function to minimise a specific cost function that models an error between the estimated output and the true labels (a.k.a ground truth). There are several loss functions in supervised learning models (euclidean distance, cross-entropy, information gain etc.) [21]. Furthermore, different ML algorithms have a different predictive model architecture that is based on the hyper-parameters space. These will be discussed below.

2.3.1 Linear Models

Generally, supervised learning models can be either classification or regression techniques; these are used to predict continuous (regression) or discrete variables (classification). We will start with logistic regression from [22] because it is one of the chosen ML models for our scope.

2.3.2 Logistic regression (LR)

LR [23] is one of the models that can be used for classification problems. LR cost function can take many forms depending on the regularisation method chosen for the penalty. Three types of method of regularisation methods exist in LR (L1, L2 and elasticnet) [24]. In order to tune LR, the first hyper-parameter is the regularisation method used in the penalisation (l1, l2, elasticnet or non); it is called a penalty in sk-learn. C (the coefficient) is another important hyper-parameter to tune, this determines how strong the regularisation for the model needs to be. Furthermore, the solver type represents the type of optimisation algorithm to be used (i.g. newton-cg, lbfgs, liblinear, sag or saga) in LR. These solvers have correlations with a penalty and C, this relation is called conditional hyper-parameters.

2.3.3 Stochastic Gradient Descent (SGD)

SGD [22] is an optimisation algorithm that is used for classification or regressions. The gradient here is the slope or slant on a surface (gradient descent means descending a slope to reach the lowest point on that surface). It is an iterative technique. It typically starts from a random point of a function's response surface and travels down in steps until it reaches the lowest point of that function (or at least a local optimum). With SGD the gradient loss is calculated by taking each sample at a time and then the model gets updated with decreasing strength schedule (learning rate). To achieve the best results, SGD uses minibatch learning (online or out of core) by the partial_fit method; the data should have

zero mean and unit variance. SGD has a function that is added to the loss called a *regularizer* (it works as a penalty to the loss). The penalty shrinks the parameters towards the zero vector using three choices: L2 (squared euclidean norm), L1 (absolute norm) or a combination of both called (Elasticnet). When the parameter gets updated and crosses a value (0.0) of the regulariser, this leads to online feature selection of the sparse learning models [22].

2.3.4 K-nearest neighbour (KNN)

K-nearest neighbour (KNN) is one of the commonly used ML algorithms for classification tasks. It uses a set (the training set) of labeled data items to make a prediction for new (unlabeled) data items. The predicted class of each new item is usually set to the most highly occurring class in its k nearest neighbours in the training set. The technique adopts a distance metric to determine distances between data items.

The most important hyper-parameter is k, the number of considered nearest neighbours [25]. There are a couple of problems that might arise if k is too small or too large. If k is too small the model will under-fit, alternatively, if it is too large, it will overfit and require more time to be trained. Depending on different problems, the weighted function used in prediction can be either uniform or distance. With uniform, the points are equally weighted, and with distance, they are weighted using the inverse of their distance. One more distance and power metric can be tuned for minor improvements called Minkowski. Finally, the algorithms that compute the nearest neighbours are taken from three choices: ball tree, k-dimensional tree, or brute force search. There is also a common choice in which we can set it to auto mode in sk-learn [22].

2.3.5 Naive Bayes (NB)

Naive Bayes (NB) [26] form a set of supervised learning techniques based on Bayes' theorem. It shows a way in which we can calculate the probability of a hyper-parameter belonging to a given score. It uses previous results to form a probabilistic model that is based on the probability of the score given a vector of hyper-parameters. We can say that Bayes theorem is denoted by the following: P (score|hyper-parameter) = (P (hyper-parameter|score) * P (score)) / P(hyper-parameter). P (score|hyper-parameter) is the probability of the score given the provided hyper-parameter value. There are four main types of NB: Bernoulli NB [27], Gaussian NB [28], multinomial NB and complement NB [29]. Interested readers should refer to [30] for more information.

2.3.6 Tree-based Models

The Decision Tree (DT) [31] is a widely used classification technique which utilises a tree-based structure to model a decision and consequently summarise the set of classification rules from the data. There are three main components for a DT: the root node to represent the whole data, multiple nodes that show decision tests and sub-nodes split over features to make a decision, and lastly, result classes represented by various leaf nodes [32]. Commonly these algorithms recursively split the training set with better values from the features to achieve good decisions over each subset. In order to prevent over-fitting in a DT, pruning is used to remove a few subsets from the decision

nodes. To control the complexity of a DT, there is a hyper-parameter called max_depth [33]. There are several hyper-parameters that can be tuned to build a better DT model [34]. First, to control the split parameter quality we use a measuring function called *criterion* in scikit-learn. Two main types of measuring functions can be selected: *Gini impurity* and *Information gain*. Furthermore, there are also two choices for the split method, it is either *best* (for best split) or *random split* (to split randomly). There is a feature selection method called *max_features* for choosing the best features. Moreover, there are a few more hyper-parameter for the splitting process: *min_samples_split* (to acquire a minimum number of data points to split) and *min_samples_leaf* (to acquire a minimum number of leaves to obtain); *max_leaf_nodes* (maximum number of leaf nodes), and the *min_weight_fraction_leaf* (this is the minimum weighted fraction of total weights). These also are tuned to improve model performance [22], [34]. Similarly, it is possible to combine multiple singular ML models to achieve better performance. Models derived in this way are known as ensembles. Such models include Random Forest (RF), Extra Trees (ET), Extreme Gradient Boosting (xgboost) and Light Gradient Boosted Machines (LightGBM). We will focus on xgboost, LightGBM and RF.

2.3.7 Random Forests (RFs)

As an ensemble learning technique, RF [35] employs a technique called bagging to combine multiple decision trees together. The majority-voted classes are chosen as the final classification in RF, which use standard DTs constructed from a large number of randomly generated subsets [36]. Similar to RF, ET [37] employs a randomly selected feature set and the entire sample set to construct DTs. It also chooses the split at random while RF seeks optimal results.

2.3.8 Gradient Boosting Decision Trees (GBDTs)

A GBDT uses the boosting function and it facilitates classification 'difficult' samples. This is due to putting more weight on using these difficult samples during training. GBDT is a version of Gradient Boosting Machines GBM (where all weak classifiers are regression trees). There are two techniques that are based on GBDT: xgboost and LightGBM.

2.3.9 Extreme Gradient Boosting (xgboost)

Xgboost [38] is one of the popular tree-based ensemble models; it is designed for speed and performance improvement. It uses boosting and gradient descent methods to combine basic DTs. It grows trees depth-wise (one of the main important traits to differentiate it from LightGBM). Xgboost trains various models on several subsets of the training dataset and then gives out the vote for the best-performing model.

2.3.10 Light Gradient Boosting Machines (LightGBM)

LightGBM [39] is an implementation of GDBT that is used to preserve the accuracy of the model. One of the main differences between it and xgboost is that it grows trees leaf-wise. LightGBM is very

suitable for large datasets, that is due to how fast and efficiently it produces results. Gradient-based side sampling is a way to weigh samples during training. Data instances with larger gradients contribute more to the information gain used for building a tree. Thus, keeping instances with the highest gradients during training and using random sampling instances with smaller gradients. It has an exclusive feature bundling to reduce the number of features. Many features are exclusive and can be bundled together (meaning they do not take non-zero values simultaneously). This is used to lower the dimension of features. Thus, classifiers train quicker while having the same accuracy. In the ML detection field, most models are required to handle bigger data sets. LightGBM is very effective in this aspect, as it can handle big data. This is one of the reasons why this specific model has been used in this thesis.

2.4 Deep Learning (DL) Models

Deep Learning (DL) techniques are applied to many areas such as computer vision, machine translation and natural language processing; this is due to the success in solving many types of related problems. It is based on the artificial neural network (ANNs) theory. DL comes in various forms such as deep belief networks (DNNs), feedforward neural networks (FFNNs), convolutional neural networks (CNN), recurrent neural networks (RNNs) and more [40]. All of these models have the same hyperparameters because they have the same base architecture (ANN structure). DL approaches have many hyper-parameters that require tuning. Two main hyper-parameters are the number of layers and the number of neurons, which in turn raises the complexity of the DL model [41]. Mainly, DL models should have the capability to model the objective function and avoid over-fitting. Then setting up the type of function needed for the problem (i.e. binary cross entropy for classifications or RMSE for regression etc.). After that, we need to set up the required activation function. There are many types of activation functions to be used depending on the problem: Softmax, rectified linear unit (Relu), sigmoid, tanh, or soft sign, hard_sigmoid etc. (These are used to model non-linear problems). Finally, the optimiser type, it can be set as follows: stochastic gradient descent (SGD), adaptive moment estimation (Adam) and others [42]. There are other hyper-parameters that are closely related to the optimisation and training process of DL Models. The learning rate is one of the most important parameters to tune in DL models [43]. It makes use of the step size of each iteration that enables the objective function to converge. Even though having a large learning rate speeds up the training process, however, the gradient may achieve a local minimum value or even cannot converge. Furthermore, a lower learning rate converges somewhat smoothly, but in turn, it will increase the model training time and require more epochs. The optimal learning rate should be one that enables the objective function to reach a global minimum within budget. Another important hyper-parameter to tune is the drop-out rate (it is used to combat over-fitting). With drop-out, a random sample of neurons is removed and it should be tuned accordingly. Batch size (number of processed samples before training) and Epochs (number of complete cycles throughout the entire training set) are other hyper-parameters that we need to tune [44]. The number of iterations and the resource requirement for training affects the Batch size. Epoch tuning usually depends on the type of the training set and normally it is tuned while increasing its value slowly (this is until validation accuracy starts

decreasing) this means, the model started over-fitting. One of the methods to combat over-fitting is called Early stopping (it is a form of regularization in which if model validation accuracy starts decreasing it will stop training the model in advance after a number of epochs). Commonly, the DL model converges fast with lower epochs and to prevent over-fitting the model early stopping is used. Another hyper-parameter that is used to lower the training time of the model is called Patience. These are the main hyper-parameters to tune to achieve the desired optimal values for any DL model.

2.4.1 Neural Network Structural Model

The common neural network (NN) structural model, e.g. as described in [2], has an input layer, an output layer, and a number (possibly none) of intermediate or 'hidden' layers. Each layer has a number of nodes or 'neurons'. Different layers may have different numbers of neurons. A Feed Forward Neural Network is illustrated in figure 2.2.

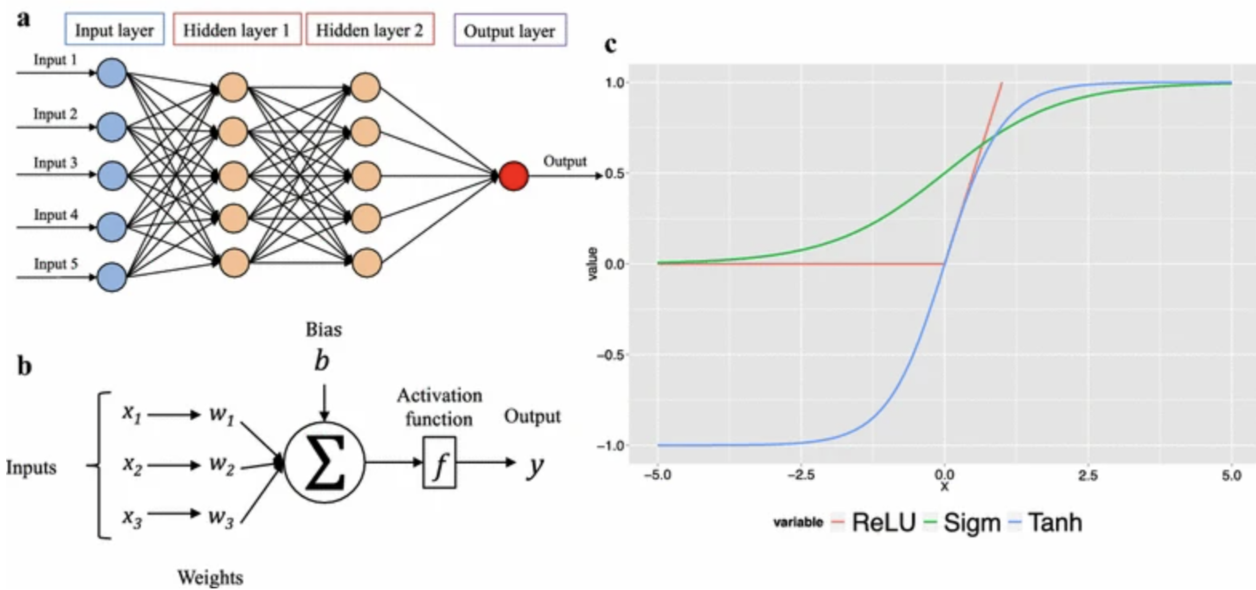

Figure 2.2: A simple Feed Forward Neural Network showing (a) structure, (b) how a weighted sum is calculated and fed into an activation function to produce a node value (output), and (c) common activation functions [2]

The number of input and output neurons is typically defined by the problem. For example, application metadata comprising 80 features with (normalised) values in the interval (0,1) would naturally give rise to 80 input neurons. Similarly, if the problem is binary classification (e.g. malware or non-malware), then a single output neuron would be very common. If the task is to identify the type of malware family, a *multi-class* problem, then an output layer with a neuron for each considered family would be usual.

Values are associated with neurons. These are either problem input values or else calculated as indicated below. Neurons in an intermediate layer are connected to neurons in the previous layer

and each connection is associated with a *weight* (see Figure 2.2 (b)). There is an additional constant weight which is a *bias* term. The neurons in the previous layer to which a neuron is connected can be thought of as its ‘input neurons’. The weighted sum of the values stored at the neuron’s inputs plus a bias term forms the input to an *activation function* that computes the stored value of the neuron in question. There are many activation functions, which typically implement a non-linear response. Three are shown in Figure 2.2 (c). Commonly used ones are linear, softmax, rectified linear unit (Relu), sigmoid, tanh, softsign and hard-sigmoid (e.g. see c in 2.2). Figures 2.3, 2.4 and 2.5 illustrate the remaining activation functions for reference (softsign, linear, softmax and hard_sigmoid).

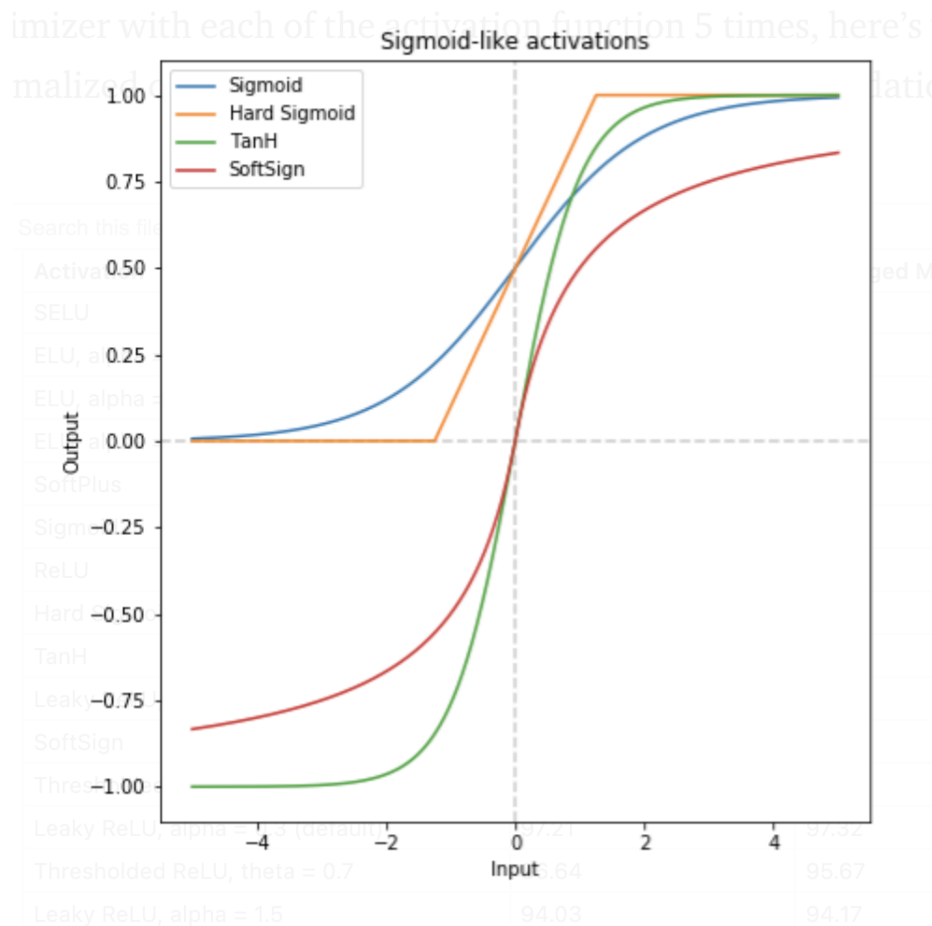

Figure 2.3: Softsign and Hard_sigmoid Activation Functions Example [3]

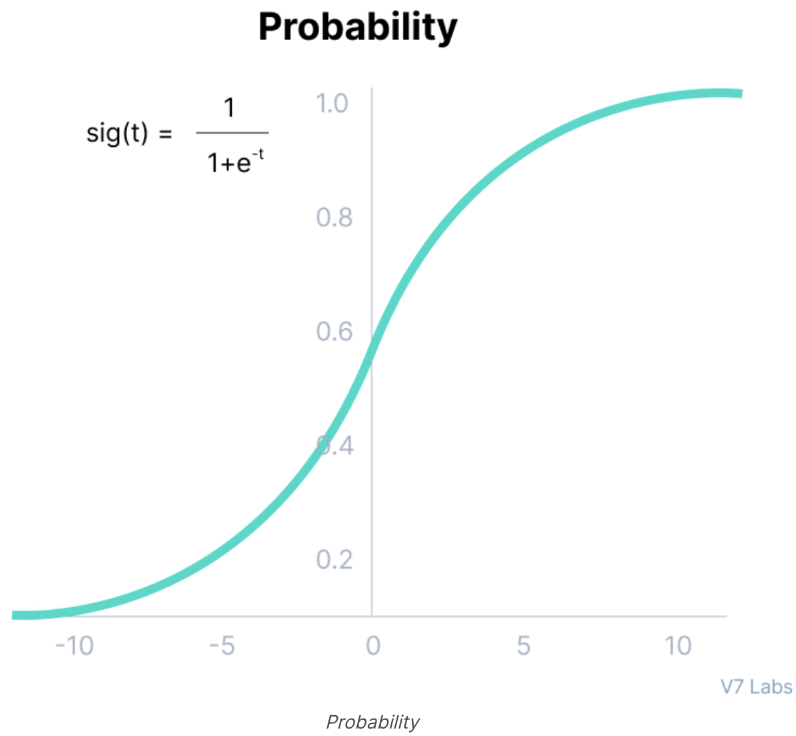

Figure 2.4: Softmax Activation function Example [4]

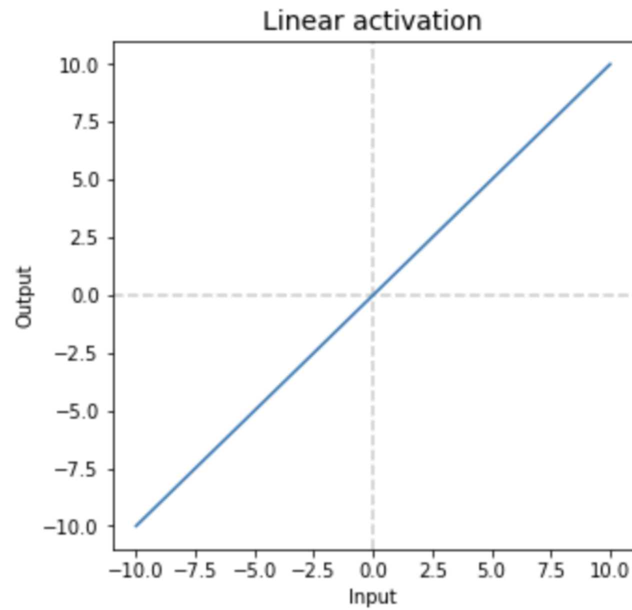

Figure 2.5: Linear Activation function Example [3]

The number of hidden layers and the numbers of neurons in them are the two main hyper-parameters that affect performance and require tuning to give excellent performance for the problem at hand.

2.4.2 Training of A Neural Network

Training a neural network requires a ‘loss function’ to measure how far the performance of the model deviates from the idea. Loss functions come in various forms (binary cross entropy for binary classifications, multi-class cross entropy for multi-classification and root mean squared error (RMSE) for regression problems). In our case, we will focus on binary cross entropy. Binary cross entropy (also called log loss) compares the predicted probability p to the actual class output y (represented as 0 or 1.0). It is the negative average of the log of the corrected predicted probabilities (meaning how far or close we are from the actual value). The formula for binary cross entropy is:

$$-(y\log(p)+(1-y)\log(1-p)) \tag{2.1}$$

Deep neural networks are typically trained, by updating and adjusting neurons weights and biases, utilizing the supervised learning back-propagation algorithm in conjunction with an optimisation technique such as stochastic gradient descent [45, 46]. Dropout addresses the problem of over-fitting by preventing neuron co-adaption. Basically, it works by taking away a (usually) small number of neurons in a random manner. The random deletion occurs with a specified probability that is a tunable parameter of the network. Another hyper-parameter to be set is the optimiser. This could be stochastic gradient descent (SGD), adaptive moment estimation (Adam) or root mean square propagation (RMS prop), etc [42]. There are other dependent hyper-parameters to the optimiser such as Learning Rate (LR). This is to be set during the training or optimisation of the DL model. The learning rate is one of the most important hyper-parameters for DL models that need to be tuned [43]. It directly affects the convergence of the objective function. It specifies the step size at each iteration of the search. Hence, if the LR were lower, there is a chance that it will lead to a slower convergence rate. However, if it is too high it could cause the model to never converge or become stuck at a local minimum [47]. Based on the data set used, the right LR should be found by looking through a number of values until a reasonable one is found. No single LR could work for every problem, it needs to be modified accordingly.

2.4.3 ML-Based Static Malware Detection Related Literature

Several works have explored the use of machine learning for Windows PE malware detection, e.g., [48, 49, 50], but work has often been hampered by the absence of a standard benchmark dataset. The publication of the Ember dataset [6] has resolved this problem. The dataset is accompanied by various Python routines to facilitate access. Ember’s authors have also provided baseline applications of various ML techniques to their datasets. **This paper [6] explicitly identifies the potential for HPO in future work**. In [51], the authors considered imbalanced dataset issues and model training duration. They also applied a static detection method using a Gradient-Boosting Decision Tree Algorithm. Their model achieved better performance than the baseline model with less training time

(They used feature reduction based on the recommendation of the authors of [6].) Another approach used a subset of the Ember dataset for their work and compared different ML models [52]. Their work is mainly concerned with scalability and efficiency. Their goal was to identify malware families. The proposed Random Forest model achieved a slightly better performance than the baseline model. [53] The authors here utilised a hybrid of two datasets, Ember (version 2017) and another dataset from the security partner of Meraz's 18 techno-cultural festivals (IIT Behali). A feature selection method was used to improve their model's performance Fast Correlation-based Feature Selection method (FCBF). Thirteen features (with high variance) were selected. Several ML models (Decision Trees, Random Forest, Gradient boost, AdaBoost, Gaussian Naive Bayes) were introduced to be trained. Random forest achieved the highest accuracy 99.9%.

2.5 Formal Definition of HPO and Motivation for its Use in Malware Classification

Hyper-parameters are parameters of a model that are not updated during the learning process [54]. The HPO problem is defined in a common way by many researchers as a search to find x^* defined in Equation (2.2).

$$x^* = \arg\min_{x \in X} f(x), \tag{2.2}$$

where $f(x)$ is an objective function. Commonly, $f(x)$ is an error rate of some form evaluated on the validation set, e.g., the Root Mean Square Error (RMSE). x^* is the hyper-parameter vector that gives rise to the lowest objective score, and x can be any vector of parameters in the specified domain. HPO seeks the hyper-parameter values that return the lowest score. For malware and similar classification tasks, suitable choices for the objective functions are holdout and cross-validation errors. Furthermore, if we consider a loss function for the same problem, then a possible choice is the misclassification rate [55]. For our proposed model in chapter 3, the loss function is defined by Equation (2.3).

$$f(x) = (ROC_AUC - 1) \tag{2.3}$$

where *ROC_AUC* is the Receiver Operating Characteristic (with cross-validation) Area Under the Curve. *ROC_*-related criteria are common in malware detection. For an in-depth background about validation protocols, see [56]. Our work also aims to investigate evaluation time. There are three clear ways to do this. The first is to use a subset of folds in testing an ML algorithm [57]. The second is to use a subset of the dataset, especially if the data set is large [58, 59] or the third is to use fewer iterations.

Although HPO has a great deal to offer, it comes at a computational price. For every hyper-parameter evaluation, we must train the model, make predictions on the validation set, and then calculate the validation metrics. Developing a robust ML-based classifier for Windows PE with a credibly sized and diverse dataset such as Ember is not yet introduced, therefore, it is a significant undertaking. The computational costs involved act as a disincentive to implementing Bergstra et al.'s formal outer loop. There is a pressing need for traversing the hyper-parameter space efficiently, and we demonstrate how a leading HPO approach allows us to do so.

Here, Windows PE files are a means to an end; the same issues apply to detecting other malware. Although malware is our major interest, our work also seeks to motivate consideration of HPO, and the use of state-of-the-art approaches, in particular, more widely in the application of ML in cybersecurity. For more information about HPO, interested readers should refer to [55].

2.6 Hyper-Parameter Optimisation (HPO)

2.6.1 Main Process of HPO

The normal process to utilise HPO is as follows:

1. Select an objective function and the metrics that will be used.
2. Choose the hyper-parameters for tuning and select the required optimisation method.
3. Train the model using the default parameters as a baseline.
4. Select a search space to check for applicability based on intuition or manually.
5. Narrow down the search space if it is too large, based on the plausible parameters found and/or explore new ones
6. Return the highest performing hyper-parameters.

However, there are a few issues that hinder the progress of the most commonly used optimisation techniques [21]. This is because HPO problems are unique [60] as follows:

First, the optimisation target is usually a non-convex and non-differentiable function, whereas many traditional methods are configured to solve convex and non-differentiable functions. This can lead easily to getting stuck in a local rather than the global optimum. Further, the lack of smoothness in the optimisation target leads to poor performance [5].

Second, hyper-parameters of different ML models have different forms (e.g. continuous, discrete, conditional etc.). This makes many traditional optimisation methods with a focus on continuous or numerical variables infeasible for HPO [61].

Third, it is computationally expensive to train ML models on large datasets. Sometimes HPO methods make use of data sampling to obtain approximate values for the objective function. Consequently, these approximate values should be usable with the appropriate optimisation techniques (specific to the HPO problem). A limitation is that the evaluation time for a given function will be ignored by black-box optimisation (BBO) models. Thus, they might need exact instead of approximate values for the objective function. This makes various BBO algorithms unsuitable for HPO problems with a limitation on time and budget.

Furthermore, there is a need to find effective optimisation methods to be applied to the HPO problem. This is in order to identify optimal hyper-parameter space for any ML models. We will discuss BBO further in section 2.7.1

2.7 General Categories of Hyper-Parameter Optimisation

The goal of HPO is to automate the process of hyper-parameter-tuning and enhance the practical applicability of ML techniques to problems [60, 5]. Further rationale for using HPO can be found in [62]. Hyper-parameter techniques come in various forms: babysitting [63], Grid search (GS)[64], Random-Search(RS)[65], Gradient Based Optimisation [66], Bayesian Optimisation(BO) [67], (BO-GP) Gaussian Process [67], Sequential Model Based Algorithm Configurations(SMAC) [63], and Tree Structured Parzen Estimators using BO (BO-TPE) [64]. This thesis will explore the use of GS, RS and BO-TPE.

2.7.1 Traditional Model-Free Blackbox approaches

Perhaps the two most common HPO methods are Random Search and Grid Search. These require only an evaluation function to work, i.e., they are what is commonly referred to as 'blackbox' techniques.

Random Search selects values randomly from the domain of each hyper-parameter. Usually, the values selected from different domains by Random Search are independent, i.e., the value selected for one parameter does not affect the value selected for a different parameter. Furthermore, for an individual parameter, all values have the same probability of being selected. (Selection is said to be *uniform*.) It is possible to relax such properties, producing what is referred to as a *biased* stochastic search. Such bias often encodes for domain insight, which is not in the spirit of a blackbox approach. In our work, we adopted a standard unbiased Random Search.

In Grid Search the individual parameters are discretised, i.e., a number of specific values are selected as 'covering' the particular parameter space. For example, the elements in the set $\{0.0, 0.25, 0.5, 0.75, 1.0\}$ could be taken to cover a continuous parameter in the range [0.0, 1.0]. Grid Search evaluates the function over the cross-product of the discretised hyper-parameter domains and so suffers from the 'curse of dimensionality' [68]. As the number of parameters increases or finer grain discretisation is adopted, the computational complexity mushrooms.

Random Search and Grid Search do not learn from past evaluations; we generally refer to such approaches as being *uninformed*. Consequently, they may spend a great deal of time evaluating candidates in regions where the previous evaluation of candidates has given rise to poor objective values. Random Search will search the specified space until a certain number of evaluations, time, or budget has been reached. It works better than Grid Search when we know the promising hyper-parameter regions, and so we can constrain the stochastic selection of candidates to lie in such regions [69, 70]. Combining Random Search with complex strategies allows a minimum convergence rate and adds exploration that can improve model-based searches [55, 71].

It is not surprising that uninformed methods can be outperformed by methods that use evaluation history to judge where to try next; indeed, such guided searches usually outperform uninformed methods [64, 72, 73]. Thus, motivates our interest in the use of Bayesian optimisation approaches, which we now explore.

2.8 Model-Based approaches to HPO

Below we discuss model-based approaches to HPO, specifically Bayesian optimisation and its variants.

2.8.1 Bayesian Optimisation (BO)

BO has emerged recently as one of the most promising optimisation methods for expensive blackbox functions. It has gained a lot of traction in the HPO community, with significant results in areas such as image classification, speech recognition, and neural language modelling. For an in-depth overview of BO, the reader is referred to [54, 74]. BO is an informed method that takes into consideration past results to find the best hyper-parameters. It uses those previous results to form a probabilistic model that is based on the probability of the score given a vector of hyper-parameters. This is denoted by the formula: $P(score|hyperparameter)$. [75] refers to the probabilistic model as a *surrogate* for the objective function denoted by $P(y|x)$, the probability of y given x. The model or surrogate is more straightforward to optimise than the objective function. BO works to find the next hyper-parameters to be evaluated using the actual objective function by selecting the best-performing hyper-parameters on the surrogate function. A five-step process to do this is given by [75]. The first step builds a surrogate probability model of the objective function. The second finds the hyper-parameters with the best results on the surrogate. The third applies those values to the real objective function. The fourth updates the surrogate using this new real objective function result. Steps 2–4 are repeated until the maximum iteration or budgeted time is reached [76]. BO has two primary components: a probabilistic model and an acquisition function to decide the next place to evaluate. Furthermore, BO trades off exploration and exploitation; instead of assessing the costly blackbox function, the acquisition function is cheaply computed and optimised. There are many choices for the acquisition function. In this thesis, we use the most common—expected improvement (EI) [77]. The goal of Bayesian reasoning is to become more accurate as more performance data is acquired. The previous five-step process is repeated to keep the surrogate model updated after each evaluation of the objective function [64]. BO spends a little more time generating sets of hyper-parameter choices that are likely to provide real improvements whilst keeping calls to the actual objective function as low as possible. Practically, the time spent on choosing the next hyper-parameters to evaluate is often trivial compared to the time spent on the (real) objective function evaluation. BO can find better hyper-parameters than Random Search in fewer iterations [72]. In this thesis, we investigate whether AHBO-TPE, a specific variant of BO, can, for ML-based Windows PE file malware detection purposes, find better hyper-parameters than Random Search and with fewer iterations.

2.8.2 Sequential Model-Based Optimisation (SMBO)

There are several options for the SMBO's evaluation of the surrogate model $P(y|x)$ [64]. One of the choices is to use Expected Improvement (EI), defined in Equation (2.4).

$$EI_{y^*}(x) = \int_{-\infty}^{y^*} (y^* - y)P(y|x)dy \tag{2.4}$$

Here y^* is the threshold value of the objective function, x is the vector of hyper-parameters, y is the actual value of the objective function using the hyper-parameters x, and $P(y|x)$ is the surrogate probability model expressing the probability (density) of y given x. The goal is to find the best hyper-parameters under function $P(y|x)$. The threshold value y^* is the best objective value obtained so far. We aim to improve (i.e., get a lower value than) the best value obtained so far. For such minimisation problems, if a value y is greater than the threshold value, then it is not an improvement. Only values less than the threshold are improvements. For a value y less than the threshold y^*, the improvement is $(y^* - y)$. Integrating over all such improvements, weighted by the density function, $P(y|x)$ gives the overall expected improvement given the vector of hyper-parameter values x. When better values of x are found (i.e., giving rise to actual improvements in the real objective function) the threshold value y^* is updated. The above description is an *idealised* view of Expected Improvement. In practice, the choice of the threshold value is more flexible, i.e., y^* need not be the best objective value witnessed so far; this is actually the case for the Tree-Parzen Estimator approach outlined immediately below.

2.8.3 Tree-Structured Parzen Estimators (TPE)

The Tree-Structured Parzen Estimators approach constructs its model using Bayesian rules. Its model $P(y|x)$ is built from two model components, as shown in Equation (2.5). One component, $l(x)$, models values less than a threshold and the other, $g(x)$, models value greater than that threshold.

$$P(x|y) = \begin{cases} l(x) \;\; if \;\; y < y^* \\ g(x) \;\; if \;\; y >= y^* \end{cases} \tag{2.5}$$

TPE uses y^* to be some quantile γ of the observed y values, i.e., such that $P(y < y^*) = \gamma$ [78]. This allows data to be available to construct the indicated densities. $l(x)$ is the density based on the set of evaluated values of x that have been found to give objective values less than the threshold. $g(x)$ is the density based on the remaining evaluated x values. Here, $P(x|y)$ is the density of hyper-parameter x given an objective function score of y. Following [64] it is expressed as shown in Equation (2.6).

$$P(y|x) = \frac{P(x|y) * P(y)}{P(x)} \tag{2.6}$$

Reference [64] also shows that to maximise improvement, we should seek parameters x with high probability under $l(x)$ and low probability under $g(x)$. Thus, they seek to maximise $g(x)/l(x)$. The best such x outcome is then evaluated in the actual objective function and will be expected to have a better value. The surrogate model estimates the objective function; if the hyper-parameter that is selected does not make an improvement, the model will not be updated. The updates are based on previous history/trials of the objective function evaluation. As mentioned before, the previous trials are stored in (score, hyper-parameters) pairs by the algorithm after building the lower threshold density $l(x)$ and higher threshold density $g(x)$. It uses the history of these previous trials to improve the objective function with each iteration. The motivation to use TPE with SMBO to reduce time and find better hyper-parameters came from leading HPO papers [64, 72, 79, 5]. SMBO uses Hyperopt

[78]—a Python library that implements BO or SMBO. Hyperopt makes SMBO an interchangeable component that could be applied to any search problem. Hyperopt supports more algorithms, but TPE is the focus of our work. Our contribution lies in the demonstration of the usefulness of SMBO using TPE for PE malware classification purposes.

2.9 HPO using Deep Learning

2.9.1 Selection of Hyper-parameter Optimisation for Deep Learning

Hyper-parameters are selected either manually or automatically [80]. The former is based on the experience of a researcher. The latter does not need much understanding from the user but is usually computationally heavy. In our case, we wanted to investigate how and which hyper-parameters would affect our model performance. This is in order to understand as much as possible which of these parameters would add value to our goal.

2.9.2 Approaches of Hyper-parameter Optimisation for Deep Learning

There are several ways in which we can incorporate optimisation techniques for ML models [69, 64, 66, 67, 63, 81]. However, we are interested in two methods Grid Search optimisation [82] and another unique approach utilising a tool for Covering Arrays (cAgen) [83] (we discuss this further in section 2.12). Grid Search is one of the most common optimisation search methods. It is relatively simple to perform but has some limitations [84]. It requires the user to pre-define a search space beforehand and it does an exhaustive search to find the best-performing values. One issue with the search space is that if the discretised search space does not contain the best values the performance would suffer. Covering Arrays are used for combinatorial testing (in the software testing field), in which the number of tests is defined by the value of strength t (t-way testing). It is used to reduce the number of tests needed to produce results. A recent paper by [85] revealed that a covering array can be used to optimise the parameters of a Convolutional Neural Network. The authors investigated the use of mixed-level covering arrays in experimental design to determine optimal parameter settings and concluded that this method is very promising.

2.10 HPO using Metaheuristic Approaches

Metaheuristic approaches are derived from biological theories and are predominantly used in optimisation problems [65]. They aid in solving non-convex, non-continuous and non-smooth optimisation problems. They are typically computationally intensive. One major class is population-based optimisation (PBO). PBO approaches include Genetic Algorithms (GAs), Particle Swarm Optimisations (PSOs), Evolutionary Algorithms (EAs) and Evolutionary Strategies (ESs). PBO usually involves creating and updating a specific population, where every generation and single instance in it would be evaluated as part of the process to find a global optimum [79]. The difference between various PBOs is the way in which they choose the populations [86]. PBO can parallelise with ease. Because if we take a specific population with n individuals the evaluation process can be n

machines in parallel [5, 87]. From within EA specifically, PSO and GAs are the most frequently used techniques [88, 89].

2.10.1 Genetic Algorithms (GAs)

GA is a commonly used approach that is based on evolutionary theory. It implements a survival of the fittest regimen, where the individuals with the best survivability traits (the best-performing individuals) are more likely to survive and pass their traits to the next generation. Generations are successively evolved. Once the generation's budget reaches the end, the best members of the population are identified as the global optimum. (In some cases memory is maintained of the best performer in any generation and this individual is returned.)

In order to apply a *GA* to *HPO* problems, each *chromosome* or instance represents a *hyper-parameter* vector and the value of each hyper-parameter vector is used to evaluate that individual. A chromosome is typically a string (sequence) of elements of some data type. Simple GAs will use binary strings as chromosomes. More sophisticated ones may use sequences of floats or have mixed data types. Sub-strings will be *interpreted* as values to be used as inputs to an evaluation. For example, the first 8 bits of a chromosome may represent an ML model parameter in the range (0,1). We might map the value of the first 8 bits to an integer in the range (0,127) with a natural unsigned binary interpretation and then divide by 127 to normalise to the range (0,1). It is common to refer to the string of values as the *genotype* of the chromosome, whilst its interpretation in terms of the actual model is referred to as the *phenotype*. The phenotype is what it 'means' in the real (problem) world. We refer to an interpretable sub-string as a *gene*, with individual lower-level data types as *alleles*.

The gene crossover and mutation methods are then performed on the genes associated with each chromosome. The fitness function is what makes the evaluation metrics, while the initialised parameter ranges include all possible values for evaluation [90]. In order to identify the optimum, selection, crossover, and mutation operation operations are performed on the population's chromosome. Both methods introduce new gene values into the population.

Crossover generates new chromosomes by swapping parts of different chromosomes. For example, suppose A and B are two binary chromosomes with 100 bits. *One-point crossover* may pick a random internal point along two chromosomes and swap the parts after that point. Thus, if 43 is selected as the point, then bits 44-99 are exchanged between the two. There are many crossover variations. In theory, crossover, allows highly effective genes from two chromosomes to combine (hopefully to good effect, though this is problem dependent and is not guaranteed).

Mutation randomly modifies genes within a chromosome (often with a small probability) [5]. Mutation is important for a GA because it is frequently the only means by which a population can gain a chromosome with a particular gene value. If a gene value is not present in any chromosome of a population, then selection will not introduce one, since it merely samples from what is already there. Furthermore, the crossover will not introduce new values if crossover points are aligned along gene boundaries, which is commonly the case.

Note that even if a gene value is actually present in a population, it may disappear from the evolution process if the individuals that possess it are not sufficiently high-performing, i.e. they may

not be selected for the next generation. Mutation maintains the potential for the introduction of absent values.

Crossover and mutation ensure that different generations have different characteristics which in turn relieves the pressure of missing high-performing traits [91].

A typical GA approach to HPO is given below:

1. [65]: Randomly initialise a set (population) of chromosomes (candidate strings of hyper-parameter values). This is generation 0.

2. Evaluate each instance in the current generation using the fitness function. The fitness function will be a measure of how well those hyper-parameter values work when the ML technique at hand uses them.

3. Carry out selection, crossover, and mutation operations on the population and its chromosomes to produce the next generation (next set of hyper-parameter settings for evaluation).

4. Repeat procedures 2 and 3 until the budget ends.

5. Stop the search and return the best performing hyper-parameters, i.e. the chromosome(s) with the highest fitness.

As illustrated above, the initial population is usually generated randomly [92]. But this is not always the case. A higher-performing initial population may be engineered in some way. This can speed up convergence and so reduce compute time to find optima or high-performing solutions [5]. Further details can be found in [93].

Therefore, GAs usually do not require strict and specific initialisation. Implementation is also fairly easy. However, they can be computationally expensive ($(O(n)^2)$ [94]) and convergence may be slow. Furthermore, GAs have their own model parameters, e.g. probability of crossover, probability of mutation, the number of chromosomes in the population, and the budgets available. These may have a critical effect on their performance.

2.10.2 Particle Swarm Optimisation (PSO)

Another approach is PSO, which is derived from the biological population of both social and individual behaviour [22, 95]. Particle (swarm) searches in the defined space in a semi-random manner [86]. PSO finds the optimum based on knowledge sharing and cooperation between different particles in a specific group. PSO is dependent upon specific and proper initialisation, this is true for discrete hyper-parameters [96]. With such a problem, a proper initialisation depends upon the previous experience of the user.

2.10.3 Distributed Evolutionary Algorithm in Python (Deap)

Deap [97] is a computational evolutionary framework that supports fast and easy-to-use testing of new ideas. The framework is built with the Python programming language with the goal of providing

tools to produce custom-made EA in a form of pseudo-code [98]. It is a type of black box framework, built with different compatible existing techniques and provides various optimisation approaches GAs, Genetic Programming (GP), Evolutionary Strategies (ESs), Particle Swarm Optimisation (PSO), Differential Evolution (DE), Estimation of Distribution Algorithm (EDA) [99].

There are two main structures for Deap, a creator and a toolbox. The *creator* is a meta-factory that produces classes through inheritance and composition during run-time. In practice, it aids the making of genotypes and populations from data structures like lists, sets, dictionaries etc. The main uses of the creator is to facilitate the different types of EAs, as previously mentioned. The second structure is the *toolbox*, a collection of tools or operators to be accessed by the user according to the needs of that user's problem. Moreover, the core functions include four common algorithms in Evolutionary computation (EC) pointed out in [100]. For more information about Deap please refer to [98]. Deap has been successful in different venues [101, 102, 103, 104, 105, 106] which led to the decision to utilise it for hyper-parameter optimisation of ML models targeting PE static malware detection. To the best of our knowledge, no one has used Deap for HPO to improve ML model accuracy for PE static malware problems.

2.10.4 Tree-Based Pipeline Optimisation Tool (TPOT)

TPOT [107] is an open-source Tree-Based Pipeline Optimization Tool, a type of automated machine learning (Auto-ML) tool that utilises genetic programming for optimisation. It also integrates a Pareto optimisation that produces compact ML pipelines without sacrificing classification accuracy. The goal behind TPOT is to construct and enhance several series of data transformation and ML models to maximize the classification accuracy for a given dataset. TPOT is built on top of scikit-learn [108] a common ML python library [109]. Below is an example picture that depicts how TPOT would work [107]. TPOT proved to be effective for optimizing supervised ML model pipelines constructions in [110, 107, 103]. For more information about TPOT readers should refer to [111, 107]. To the best of our knowledge, no one has yet used TPOT for HPO of ML models specific to PE static malware detection problems.

2.11 HPO-Related Literature

Multiple works in the general optimisation literature have demonstrated the potential of HPO. For example, [69] indicated the importance of parameter tuning for increasing accuracy, indicating that Random Search works better than Grid Search when tuning neural networks. Further, [112] used standard tuning techniques to the application of a decision tree on 102 datasets and calculated the accuracy differences between tuned and traditional models. For all datasets, the experiments showed that tuning could achieve better performance than with the defaults. References [113, 114] are concerned with greedy forward search, which seeks to identify the most important hyper-parameter to change next. Reference [115] stressed the importance of single hyper-parameters after using sequential model-based optimisation (SMBO) tuning. ANOVA was used to measure hyper-parameter importance. The authors of [64, 116] assessed the performance of hyper-parameters across different datasets. Both have highlighted the importance of knowing which parameters to include in the

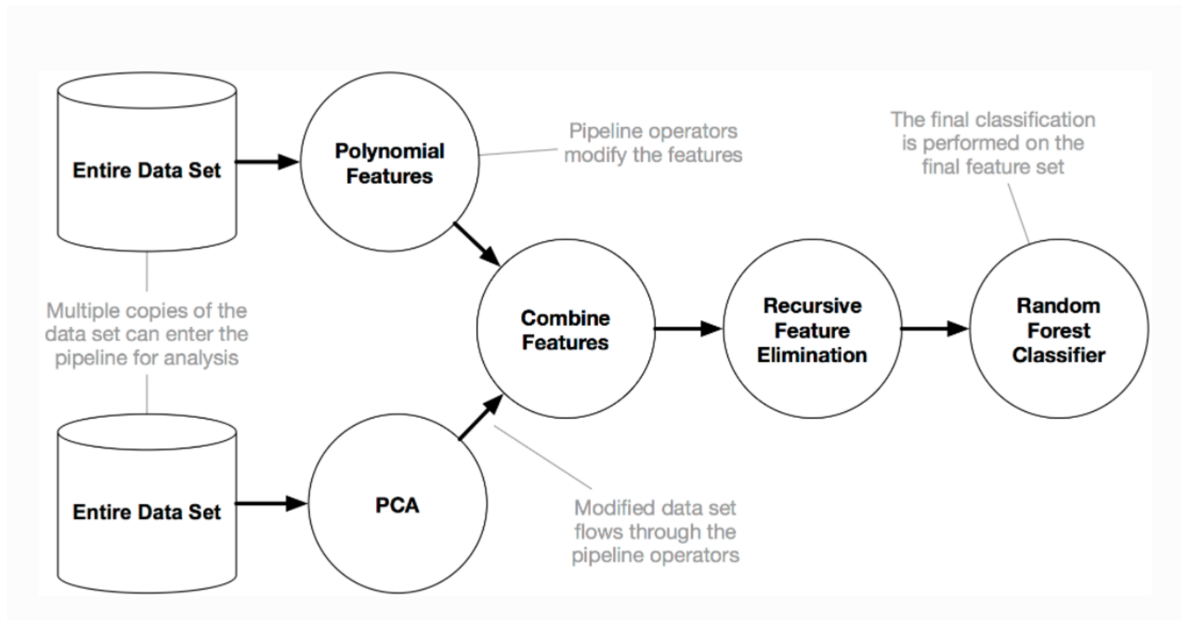

Figure 2.6: Example of how TPOT would work with RF Classifier

hyper-parameter search space in order to see an improvement. Reference [116] also used surrogate models that allow setting randomly chosen hyper-parameter configurations based on a limit on the number of evaluations carried out. A hyper-parameter search based on Bayesian Optimisation (BO) was used in [64, 54] to improve the speed of the search. The literature reveals that HPO, and in particular BO approaches, have much to offer. Readers are encouraged to refer to the survey paper [5] for a wider assessment of different HPO methods. HPO has clearly given excellent results across many parameter optimisation problems. We envisage that it can do the same for ML-based Windows PE static malware detection.

2.11.1 A Comparison between Different HPO Approaches for ML models

For more information about the common HPO algorithms and a comprehensive overview of common ML models, hyper-parameters, suitable optimisation techniques and available libraries, please refer to [5]. The table in figure 2.7 shows the common HPO algorithms, and another table 2.1 shows the common ML techniques, hyper-parameters, optimisation techniques and Python libraries.

Table 2.1: Common Techniques, Main Hyper-parameters, HPO Methods and Available Libraries

ML Algorithms	Main HP	HPO Methods	Libraries
RF	n_estimators, max_depth, criterion, min_samples_split, min_samples_leaf, max_features	GA, PSO, AHBO-TPE, SMAC, GS, RS, BOHB	TPOT, OPTUNITY, SMAC, BOHB, HYPEROPT
SGD	penalty, loss, max_iter, alpha	GS, RS, AHBO-TPE	HYPEROPT
LightGBM	num_leaves, Min_child, samples, n_estimators, boosting_type, learning_rate, Subsample_for_bin, Colsample_bytree, feature_fraction, Bagging_fraction, Reg_alpha, Reg_lambda, Is_unbalance, Objective	GS, RS, AHBO-TPE	HYPEROPT
KNN	n-neighbours	GS, RS, BO, AHBO-TPE, HYBERBAND	SKOPT, HYBERBAND, SMAC, HYBEROPT
NB	N/A	N/A	N/A
Logistic Regression	Penalty C Solver	GS, RS, AHBO-TPE, SMAC	SMAC, HYBEROPT
DL	Number of hidden layers, units per layer, loss, optimiser, Activation Function, learning_rate, Dropout_relay, epochs, batch_size, patience	GS, RS, PSO, AHBO-TPE, BOHB	OPTUNITY, BOHB, HYPEROPT

HPO Method	Strengths	Limitations	Time Complexity
GS	Simple.	Time-consuming, Only efficient with categorical HPs.	$O(n^k)$
RS	More efficient than GS. Enable parallelization.	Not consider previous results. Not efficient with conditional HPs.	$O(n)$
Gradient-based models	Fast convergence speed for continuous HPs.	Only support continuous HPs. May only detect local optimums.	$O(n^k)$
BO-GP	Fast convergence speed for continuous HPs.	Poor capacity for parallelization. Not efficient with conditional HPs.	$O(n^3)$
SMAC	Efficient with all types of HPs.	Poor capacity for parallelization.	$O(nlogn)$
BO-TPE	Efficient with all types of HPs. Keep conditional dependencies.	Poor capacity for parallelization.	$O(nlogn)$
Hyperband	Enable parallelization.	Not efficient with conditional HPs. Require subsets with small budgets to be representative.	$O(nlogn)$
BOHB	Efficient with all types of HPs. Enable parallelization.	Require subsets with small budgets to be representative.	$O(nlogn)$
GA	Efficient with all types of HPs. Not require good initialization.	Poor capacity for parallelization.	$O(n^2)$
PSO	Efficient with all types of HPs. Enable parallelization.	Require proper initialization.	$O(nlogn)$

Figure 2.7: Comparison of Common HPO Algorithms (courtesy of [5])

2.12 Covering Arrays: An Experimental Design Approach to HPO

Below we motivate the use of covering arrays.

2.12.1 What is Combinatorial Testing (CT)?

Testing of systems generally seeks to stress a system in a rigorous fashion with the intention of revealing flaws. It is typically infeasible to execute all possible inputs to test a system and so choices must be made. However, some sets of inputs are more likely to reveal flaws than others. Various 'coverage criteria' have been developed that specify characteristics of effective test sets. Combinatorial testing is one such approach. It provides requirements on which combinations of parameter values must be present in the test set. A car, for example, may be travelling at different speeds (low, medium, high), on various road surfaces (dry, wet, snow, ice), with different tyres (summer, wet, all-purpose) at different states of tyre wear (8mm, 6mm, 4mm, 2mm), with a variety of optional systems (traction control - yes/no, cruise control - yes/no, and so on). In modern cars, the range of options is significant. Testing the car under all possible combinations is simply infeasible. The software supporting modern cars is similarly varied, with a distinct desire to satisfy user preferences for configuration. Thus, in both system and software contexts choices need to be made about which configurations are actually tested. This has led to testing using formally defined subsets of the full combinatorial space, i.e. specifying which combinations of choices (parameters) must be present in the test set.

One such technique is the Covering Array. The columns of the array represent specific parameters. The rows of the array represent specific tests. Each parameter has a set of values. The Cartesian product of all parameter sets gives complete combinatorial coverage. A covering array can provide a subset of that with a particular strength t. For any subset of t parameters, each possible t-tuple occurs in at least one row (test). This is called t-way testing. Orthogonal Arrays (OAs) are the optimal version of CAs where each t-tuple occurs exactly once (rather than at least once). For some problems, an OA may not actually exist. Pairwise testing ($t = 2$) is widely used. Furthermore, it has been found that small values of t can actually give high performance in fault-finding. As t increases the size of the covering array increases too. The test set reduction achieved by covering arrays compared with a full combinatorial Grid Search may be very significant. In this thesis, we investigate whether the clear efficiency benefits of a covering array approach can be brought to bear on the ML-based static malware detection problem.

2.12.2 Covering Arrays - Related Literature

There are many problems for CAs [117] where construction of optimal values is known to be the hardest [118]. Various methods for generating covering arrays have been proposed. These include AETG [119], deterministic density algorithm (DDA) [120],[121], IPO [122], ACT [123], each with its own advantages and disadvantages. Interested reader are referred to [119], [120], [121], [122] and [124] respectively for more information. The In-Parameter-Order (IPO) strategy grows the covering array column by column, adding rows where needed to ensure full t-way coverage. Various research on improving covering array generation with the In-Parameter-Order strategy have been made. The original aim of the strategy has been the generalisability of generating covering arrays of certain arbitrary strength [125] resulting in the IPOG algorithm. In [126] a modification to IPOG resulted in smaller covering arrays in some instances and faster generation times. In [127] proposed a combination of IPOG with a recursive construction method that reduces the number of combinations to be enumerated. In [128] use of graph-colouring schemes was proposed to reduce the size of covering arrays. In [129] IPOG was modified with additional optimisations aimed at reducing don't-care values in order to have a smaller number of rows. Most of these presented works aim primarily at reducing generated covering array sizes. Alternatively, improving the performance of test generation even though it is of importance, has not gained much attention when applied in real-world scenarios (e.g. in software testing). The FIPOG technique has been shown to outperform the IPOG implementation (called ACTs) in all benchmarks and improved test generation times by up to a factor of 146 [130]. In the next section, we describe the cAgen tool [83] which implements the FIPOG technique.

2.13 Overview of the cAgen Tool

CAgen [83] is a quick and very efficient (in-memory) tool for combinatorial t-way test set generation. It has two settings available: a Web-GUI and a command line tool (fipo-cli). We use the Web-GUI tool in our research and describe three of its critical features below.

2.13.1 Workspaces

Models are defined in 'workspaces'. Figure 2.8 shows a current tool system configuration with two workspaces (models) in the workspace tab. (Note: the figure is for illustration purposes only.) One of them is selected.

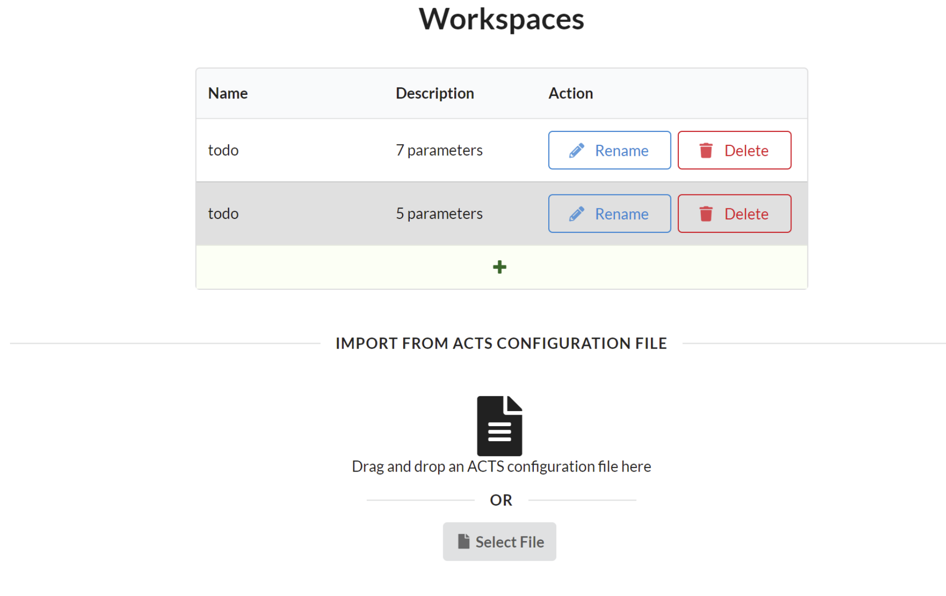

Figure 2.8: Workspaces with one selected to work on

In the workspaces tab, you can add a new workspace and select a workspace to work with. A workspace can be created by either clicking the plus sign or by importing an ACTS configuration file. Once a workspace is added to the list, you can click on it and it will redirect you to the Input Parameter Model.

2.13.2 Input Parameter Model (IPM)

Figure 2.9 shows the input parameter of the xgboost ML model.

Input Parameter Model

Export IPM...

Name	Values	Cardinality
min_child_weight	0,1,2,3,4,5,6,7	8
gamma	0,1	2
max_leaves	0,1,2,3,4	5
reg_alpha	0,1,2,3,4,5	6
max_depth	0,1,2,3,4,5,6,7,8	9

+ Add Type Name

Constraints

Figure 2.9: Shows the input parameter of xgboost ML model

The Input Parameter (IPM) tab is used to alter the parameters of your model. You can create new parameters by selecting a type, adding a name, or specifying the values of the model. Clicking the add button will then add the parameter to the IPM. The tool supports the following parameter types: Boolean, here in the boolean value, only the name needs to be set; Enumeration, a list of comma-separated values; Range, a range of values (e.g Range (2,6) which equals 2,3,4,5,6. The last parameter is called Exponential writing.

The first three types (Boolean, Enumeration and Range) are straightforward. Exponential parameters allow a faster way of adding several parameters at once: 3^2, 2^5 would specify 2 parameters with 3 values and 5 binary parameters, so a total of 7 new parameters are added to the model upon a click [83]. Moreover, we can click on the added parameters to delete or edit them. Constraints can be added in the space below the input parameters as shown in Figure 3.2 above. Constraints allow you to deem certain parameter combinations invalid and prevent their appearance in the test sets generated by CAgen. For example, if we do not want test value 0 in min_child_weight we can add in the constraints section an in input such as (min_child_weight => "1, 2, 3, 4, 5, 6, 7").

2.13.3 Generate

Figure 3.3 shows an example of generating a test set of arbitrary values of t with strength 1 of the xgboost ML model.

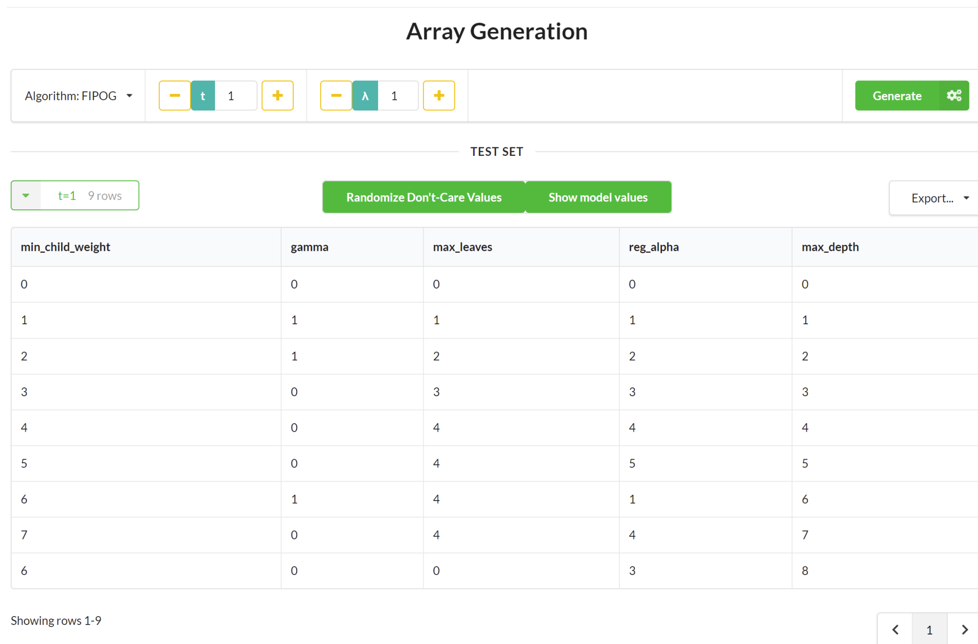

Figure 2.10: Shows a t1 test set of xgboost ML model

The generate tab makes it easy to choose the required algorithmic settings (FIPOG, FIPG-F and FIPOG-F2) and initiate the generation of the test set for the selected workspace and IPM. We can set other parameters accordingly: for example, strength t shows the strength of combinatorial testing (with a max of $t = 8$). Index lambda gives the coverage of t-way interactions, i.e. the minimum number of times each combination should appear in the generated test set. After clicking the generate button, the job is passed to a web worker (it constructs the combinatorial test set for the given parameters/models) using the fipo.wasm web-assembly. Once that is done, the array is displayed with two options: either randomize "Do not care values" or switch between values in the model and IPM. In the end, exporting the resulting arrays as comma-separated values (.csv) or directly into Python and Matlab formats is possible.

2.14 Datasets Background

Our work uses EMBER: An Open Dataset for Training Static PE Malware Machine Learning Models [6]; and another dataset from Kaggle [12]. In EMBER, the authors have outlined the features' details and how to access them. The 2018 version (version 2 of the release) [13] is used in our work. The authors stressed that the 2018 version of the dataset will present a significant challenge to ML-based algorithms. It comprises 1m samples. We used 300k benign (represented in blue) and 300k malicious (represented in red) samples for training, with 100k benign and 100k malicious samples for testing purposes. The 200k unlabelled (represented in green) examples of the dataset were not used in our experiments. (Our work concerns supervised learning only.) The training and testing sets were created using a stratified sampling approach. The samples were divided into equal-sized subsets based on

their labels (benign or malicious) and then randomly sampled from each subset to create the training and testing sets. This approach ensures that the distribution of benign and malicious samples is similar in the training and testing sets. However, there might still be potential biases in the dataset. *Spatial bias* refers to the bias introduced by the distribution of samples across different malware families. The malicious samples are drawn from diverse families in the Ember 2018 dataset. Still, some families might be over-represented while others are under-represented. This could potentially affect the performance of machine learning models trained on this dataset, as they may become biased towards more prevalent families and perform poorly on under-represented families. *Temporal bias* refers to the bias introduced during the dataset's creation period. Malware evolves over time, and a dataset created at a specific time may not represent current malware trends. (This might also apply to the second dataset used, i.e. the kaggle.com dataset). The Ember 2018 dataset was created using samples from 2017 to 2018, so it may not capture more recent developments in malware (This might also apply to the other dataset). The EMBER dataset is a collection of features from PE files that serve as a benchmark dataset for researchers. The dataset contains features from 1 million PE files witnessed before/during 2018. The repository makes it easy to reproduce and train the benchmark models, extend the provided feature set, or classify new PE files with the benchmark models. EMBER consists of the following:

- A collection of JSON line files, each containing a single JSON object.
- Each object includes the following types of data: the sha256 hash of the original file - a unique identifier.
- Coarse time information (monthly resolution) that establishes an estimate of when the file was first seen.
- A label, which may be 0 for benign, 1 for malicious, or -1 for unlabelled.
- Eight groups of raw features that include both parsed values and format-agnostic histograms.
- The dataset's samples have 2381 features in total.

The following two pictures illustrate the dataset training/testing sets and the years the samples were acquired, respectively.

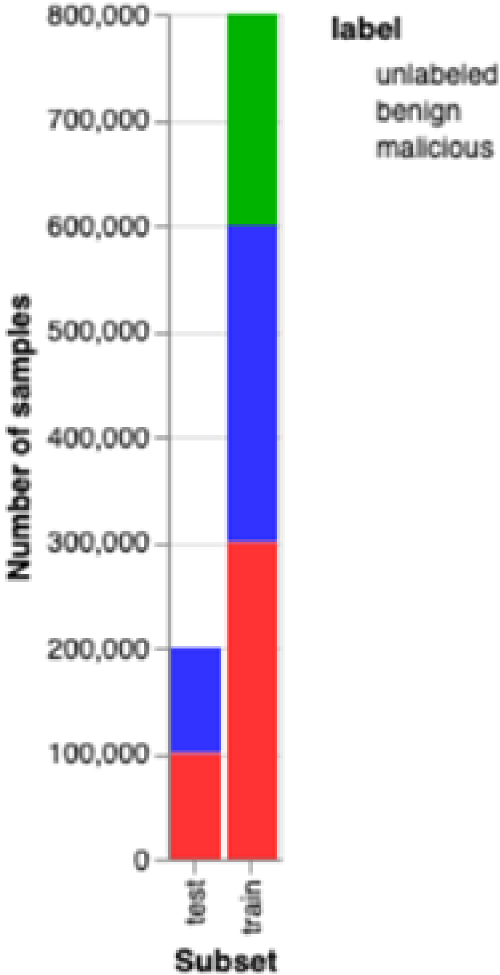

Figure 2.11: courtesy of [6]

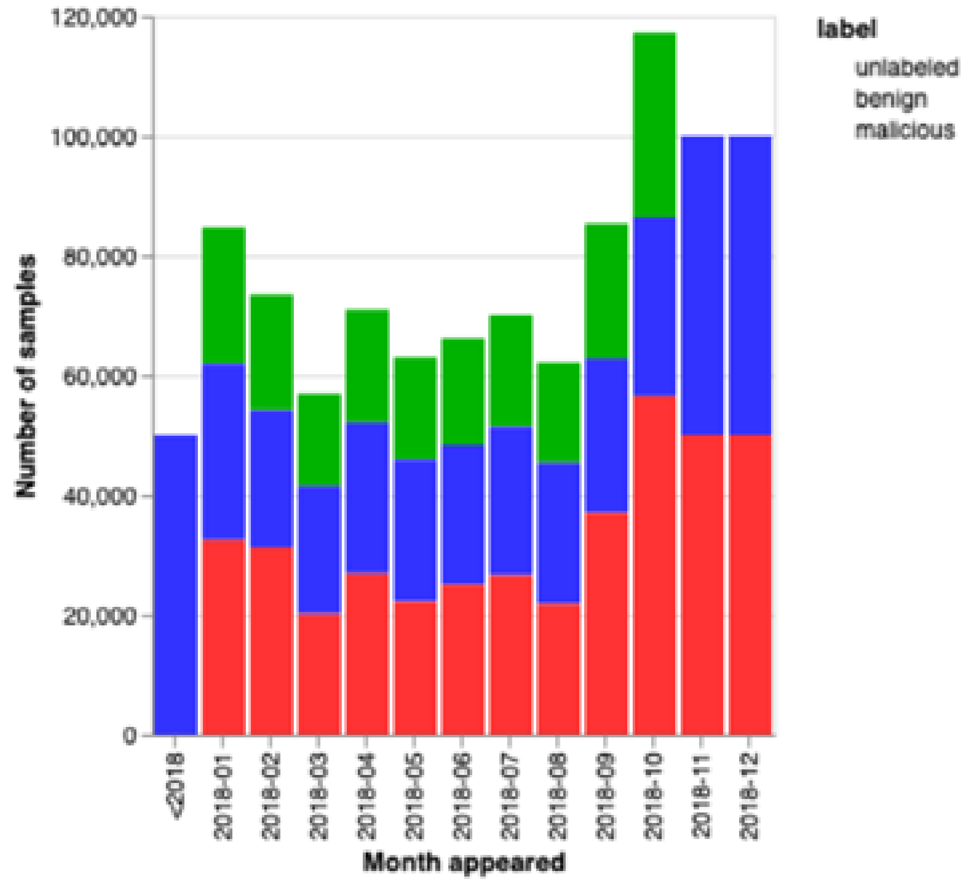

Figure 2.12: courtesy of [6]

The work also uses a second dataset built by [12] using PE files from [7]. The dataset has 19,611 samples labelled (malicious or benign) from different repositories (such as VirusShare). The samples have 75 features. It is split into 80% training and 20% testing and can be found in [12]. All results were obtained using Jupyter Notebook version 6.1.0 and Python version 3.6.0. Furthermore, implementation details of our experiments presented in chapter 3 are available via our GitHub repository [131].

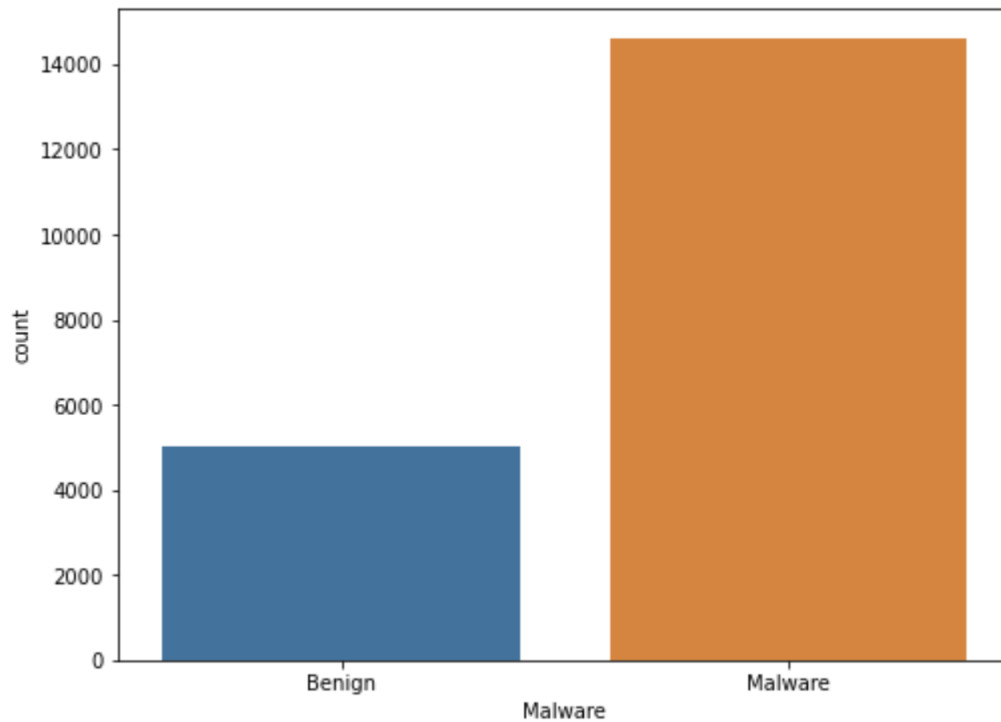

Figure 2.13: courtesy of [7]

Table 2.2: Previous Related Work to Datasets

Authors	**Paper Title**	**Brief**	**Score**
[51]	Static PE malware detection using gradient boosting decision trees algorithm (Ember 2017)	The authors made use of feature reduction techniques (reducing the number of features from 2381 to 1711). The ML technique used was the Gradient boosting decision trees algorithm with Area Under the Curve (AUC) as the score. The authors do not state the model parameters.	AUC 0.999678
[132]	Static PE Malware Type Classification Using Machine Learning Techniques (Ember 2017)	The authors observed how the literature did not consider malware classification by family type. They have used Ember version 2017 and some relabeling using VirusTotal to confirm the malware type. The best-performing ML technique investigated was Random Forest. The authors do not state the model parameters.	F1 0.96
[133]	EMBER-Analysis of Malware Dataset Using Convolutional Neural Networks (Ember 2017)	The author implemented two algorithms Convolutional Neural Networks and Feed Forward Neural Networks and assembled the results in terms of accuracy. Feed Forward Neural Network proved to be the best-performing classifier. The authors do not state the model parameters.	F1 CNN 0.95 FNN 0.97
[134]	DeepMalNet Evaluating shallow and deep networks for static PE malware detection (Ember 2017)	This work statically detects malicious Windows Portable Executable files, it primarily compares the performance of shallow (classical ML techniques) and deep networks (PE) files. Experiments with these deep model configurations are run for up to a thousand iterations at different learning rates in the range of 0.01% and 0.50%. The outcomes of deep networks have been shown to be superior to those of shallow networks. This work considers a random subset of the data set. This subset contains 25 035 benign and 24 965 malicious files for training and 25 094 benign and 24 906 malicious files for testing. The techniques that were used (techniques that are similar to those investigated in this thesis) are LR, RF, NB, KNN, DT, and DNN. The author uses variations of a single model parameter (LR).	Accuracy LR 0.544 RF 0.969 NB 0.531 KNN 0.942 DT 0.963 DNN 0.989
[135]	Evaluating Performance Maintenance and Deterioration Over Time of Machine Learning-based Malware Detection Models on the EMBER PE Dataset (Ember2018)	In this paper, the authors investigated how to keep ML-based malware detection models for PE executables running at a good level of performance. They have shown, in particular, that different decision tree-based models kept their performance pretty well over time, and that ensemble models based on decision trees kept their performance even better in some cases. This paper claims that it is the first published work to compare the rate at which different machine learning-based malware detection models lose performance over time on such a large real-world malware dataset. The best techniques with 10-fold validation (compared to this thesis) are LightGBM, RF, DT and KNN. The authors do not state the model parameters.	Accuracy LightGBM 0.948 RF 0.923 DT 0.971 KNN 0.872
[136]	Windows PE Malware Detection Using Ensemble Learning (Kaggle dataset)	This study suggests an ensemble learning-based method for malware detection. A stacked ensemble of fully-connected, one-dimensional convolutional neural networks (CNNs) performs the initial stage classification, while a machine learning algorithm handles the final stage classification. They evaluated 15 different machine learning classifiers in order to create a meta-learner. Several machine learning techniques were utilised for this comparison (compared to our thesis): NB, DT, RF, GB, KNN, SGD and Neural Nets The outcomes of tests conducted on the Windows Portable Executable (PE) malware dataset. An ensemble of seven neural networks with the ExtraTrees classifier as the last-stage classifier performed the best at (1). The authors do not state the model parameters.	Accuracy NB 0.972 DT 0.989 RF 0.984 KNN 0.986 SGD 0.979 NN 0.979

2.15 Summary and Links to the Research Hypotheses

This chapter summarised the background and research literature relevant to the work presented in this thesis. Portable executable files (PE files), which underpin the various Windows OSs, were described and identified as a particular problem. Machine learning was identified as a highly promising basis for the development of static PE detectors and various supervised learning algorithms were explained (LR, RF, SGS, KNN, NB, GBDT, xgboost, LightGBM, and DL). The application of ML to PE malware detection was summarised, specifically in the context of the use of two datasets (EMBER and a dataset from Kaggle), presented in Table 2.2. The research literature concerned with applying ML to Windows PE malware detection suggests that systematic hyper-parameter optimisation is lacking in the field. The HPO literature suggests that default parameter choices for ML models are unlikely to be optimal and the adoption of HPO approaches could bring major benefits to ML-based static PE malware detection. We hypothesise that such benefit can come from the use of HPO approaches that are common in other deployment domains (both applied ML areas and further afield). Specifically, we hypothesise that Bayesian methods, standard model-free methods (Grid Search and Random Search), covering arrays, and evolutionary algorithms, have the potential to bring significant benefits. The above has led to the main research hypothesis and its refinement into more detailed research hypotheses, as presented in chapter 1. We now move to investigate the first of our detailed hypotheses, which is concerned with the exploitation of a specific Bayesian method for HPO.

Chapter 3

Bayesian Hyper-Parameter Optimisation

This chapter thoroughly explores classical ML models applied to the default parameters and an alternative, albeit newer, approach. It demonstrates the use of some model-free approaches (e.g. Grid Search and Random Search) and Bayesian HPO approach evaluations.

3.1 Introduction

In this chapter, we explore the use of ML techniques applied to the classification of a specific form of malware: Windows Portable Execution (PE) files. We show that a specific technique is highly promising and that HPO still significantly affects its malware detection performance. We argue that HPO should be essential in ML-based malware detection research and development and security applications more widely. The contributions of this chapter are:

1. A demonstration of how well various ML-based Windows Portable Executable (PE) file classifiers perform when trained with default parameters.
2. An evaluation of various HPO approaches applied to this problem, including:
   (a) established major model-free techniques (Grid Search and Random Search); and
   (b) a state-of-the-art Bayesian optimisation model-based approach (Bayesian Optimisation with Tree-Structured Parzen Estimators).
3. A demonstration for our target problem that the optimal choices of ML hyper-parameters may *vary considerably from the toolkit defaults.*

Windows PE files are an important malware vector, and their detection has been the focus of significant research. The work described in this paper primarily uses the Ember dataset [6]—a recently published dataset comprising a header and derived information from a million PE files. The dataset contains samples of malware, benign software, and software of unknown status. (We use only benign and malicious samples.)These samples are labelled accordingly. Ember is now a major resource for the research community. We augment our Ember-focused work with work on a smaller PE dataset available from the high-profile competition website kaggle.com.

3.2 Research Question

In this chapter, we investigate a simple research question:

RQ: Can AHBO-TPE provide a highly efficient and effective means of hyper-parameter optimisation for machine learning-based Windows PE malware detectors?

Section 3.3 details the experiments performed. The results are given in section 3.4, and section 3.5 discusses the work. Section 3.6 provides conclusions.

3.3 Experiments

Here we outline the experiments carried out and provide sample data and execution environment details. Discussion of the results is given in Section 3.4.

3.3.1 Execution Environment

Our work uses two powerful toolkits: scikit-learn [22] and Hyperopt [78]. The experiments were carried out using the Windows 10 operating system, with 8GB RAM, AMD Ryzen 5 3550 H with Radeon Vega Mobile Gfc 2.10 GHz, 64-bit operating system, and an x64-based processor. Further, we used a MacBook Air (running Catalina version 10.15), 1.8 GHz Dual-core Intel i5, 8 GB 1600 Mhz DDR3, Intel HD graphics 6000 1536 MB.

3.3.2 Experiments with Default Settings

Table 3.1 shows the results when various ML techniques are applied with default parameter settings. The techniques include well-established approaches: Stochastic Gradient Descent classifier (SGD), Logistic Regression classifier (LR), Gaussian Naïve Bayes (GNB), K-nearest Neighbour (KNN), and Random Forest (RF) [22, 137]. A state-of-the-art approach—LightGBM [39]—is also used. LightGBM has over a hundred parameters and so introduces major challenges for hyper-parametrisation. Some of its categorical parameters (e.g., boosting type) give rise to conditional parameters. For initial experiments, we adopted the default parameter settings adopted by the Scikit-Learn toolkit for all techniques other than LightGBM (which has its own defaults). The evaluation metric is Area Under the Receiver Operating Characteristic Curve (ROC AUC) [138]. ROC AUC plays an important role in many security classification tasks, e.g., it occurs frequently as an evaluation metric in intrusion detection research. More specifically we have chosen the same metrics used by Ember's authors.

Table 3.1: Score Comparison of ML Models with Default Parameters (Ember Dataset).

ML Model	Time to Train	Score (AUC-ROC)
GNB	11 min 56 s	0.406
SGD	11 min 56 s	0.563
LightGBM Benchmark	26 min	**0.922**
RF	57 min and 52 s	0.90
LR	1 h and 44 min	0.598
KNN	3 h 14 min 59 s	0.745

3.3.3 Model Hyper-parameter Optimisation

The most promising of the evaluated ML algorithms, taking into account functional performance and speed of training, was LightGBM. We choose to further explore hyper-parameter optimisation on this technique. Since LightGBM has over 100 parameters, some of which are continuous, we simply cannot do an exhaustive search. Accordingly, we have had to select parameters as a focus in this work. We focused on what we believe are the most important parameters. For Grid Search, we had to be particularly selective in what we optimised. Moreover, for Random Search, we specified a budget of 100 iterations. We examine Grid Search, Random Search, and AHBO-TPE as HPO approaches. We, therefore, compare model-free (blackbox) approaches (Grid Search and Random Search) and AHBO-TPE, an approach that uses evaluation experience to continually update its model and suggest the next values of the hyper-parameters. We applied AHBO-TPE in two phases, the first one we initially set to 3 iterations, while the second was allowed 100 more iterations for fair comparison (with Random Search).

3.4 Results

The 2018 version of Ember was developed to include samples that present challenges to ML classification approaches [13]. It can, therefore, present an excellent means to stress-test available ML-based malware detection approaches. Table 3.1 shows the result of applying a variety of ML approaches, instantiated with their corresponding default parameters, to classify the samples of this dataset. All results were obtained under the MacBook Air environment described in Section 3.3.1. Table 3.1 also shows that the various ML techniques vary hugely in their suitability for the classification of PE files. We can see that LightGBM is clearly the best-performing approach, taking both time and score into account.

The subsequent tables summarise our attempts to apply HPO approaches to the most promising of the original ML techniques. Table 3.2 gives the results of applying a variety of HPO techniques. The LightGBM Benchmark results are those given in Table 3.1. Grid Search results were also obtained using the MacBook environment. The remaining results (AHBO-TPE and Random Search) were obtained using the Windows 10 laptop. The number of objective evaluations indicates the default number of evaluations of the approach for LightGBM, the total number of evaluations of the Grid

Search, and the index of the evaluation at which the best result was achieved for AHBO-TPE. Random Search and AHBO-TPE were allowed 100 evaluations. Grid Search required 965 evaluations. The ranges for parameters subject to variation are shown later in Table 3.6 (for Grid Search) and Table 3.7 (for Random Search). Random Search was allowed to explore a greater number of parameters and performed well. The meaningful application of Grid Search to this extended set of varied parameters would be computationally infeasible.

Table 3.2: Score Comparison of ML Models Before / After Optimisation (Ember Dataset).

Search Methods	**Best ROC AUC Score**	**Number of Objective Evaluations**	**Time to Complete Search**
Benchmark LightGBM Model	0.922	100	26 min (MacBook)
Grid Search	0.944	965	Almost 3 months (MacBook)
Random Search	0.955	60	15 days, 13 hrs and 12 min (Windows 10)
AHBO-TPE with 100 iterations (results after 3 iterations)	**0.957** (0.955)	26 (3)	27 days (4 h) (Windows 10)

We can see that HPO can offer significant improvements. Random Search performs very well, and so does AHBO-TPE. We can see that the initial optimisation for AHBO-TPE is far more efficient, with the technique achieving 0.955 after only three objective evaluations. Note that the time to completion is for information only. The LightGBM and Grid Search are evaluated on a Mac, and the remaining approaches were evaluated on a laptop running Windows (as described earlier).

AHBO-TPE achieves a very good result very quickly, i.e., after 3 iterations. Figure 3.1 illustrates the best score values achieved by Random Search and AHBO-TPE for each iteration (up to 100).

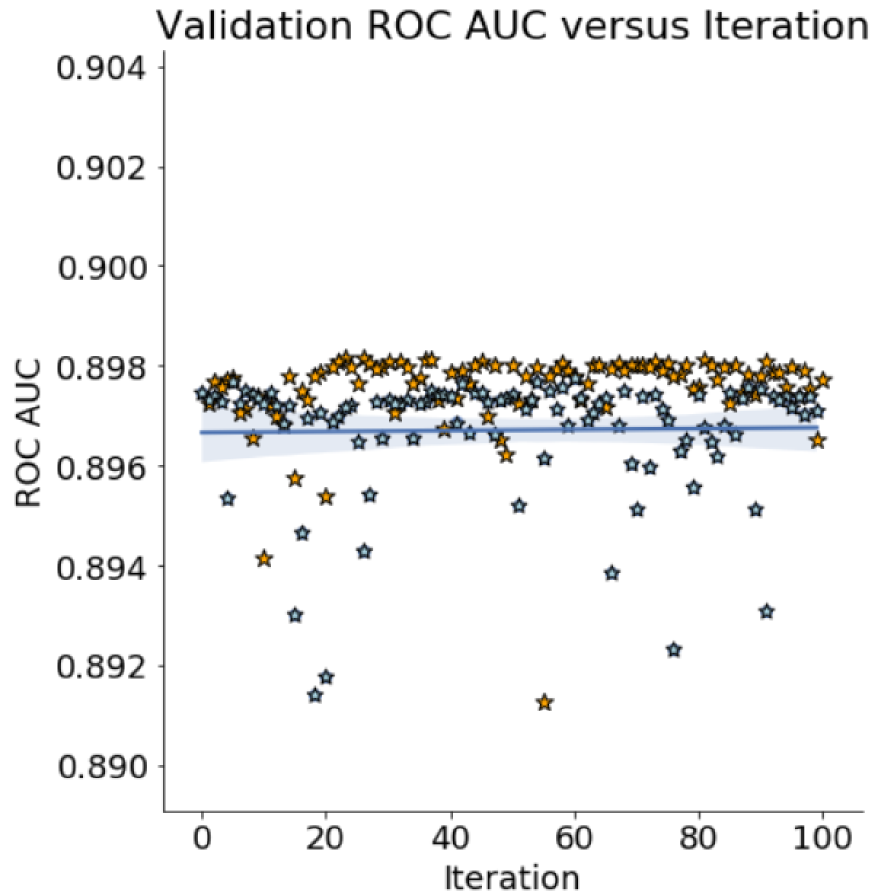

Figure 3.1: Highest Validation Score at each Iteration for AHBO-TPE (yellow) and Random Search (blue) (Ember Dataset).

Table 3.3 shows the performance of the remaining ML models using default parameter values and after parameter optimisation (using AHBO-TPE). All results were obtained under the Windows 10 environment indicated in Section 3.3.

Table 3.3: Score Comparison of the Remaining ML Models using AHBO-TPE (Ember Dataset).

ML Model	Score (AUC-ROC)	Score (AUC-ROC) After Optimisation
GNB	0.406	same
SGD	0.563	0.597
RF	0.901	**0.936**
LR	0.598	0.618
KNN	0.745	0.774

Table 3.4 gives the performance in training time based on the results attained by using

AHBO-TPE.

Table 3.4: Completion Time Results for Selected ML Models with AHBO-TPE (Ember Dataset).

ML Model	Time to Train	Training Time Reduction
GNB	11 min 56 s	same
SGD	**4 min 35 s**	14 min 35 s
LightGBM Benchmark	18 min 30 s	7 min 30 s
RF	31 min 14 s	26 min
LR	1 h 5 min 37 s	38 min
KNN	4 h 37 min and 30s	increased by 1 h 23 min 29s

Table 3.5 provides the default parameter results for various ML techniques applied to the kaggle.com dataset together with results after parameter optimisation using AHBO-TPE. (For LightGBM, it also gives results where Random Search and Grid Search were used to optimise parameters) . The LightGBM and RF Classifiers performed comparably, giving the highest AUC ROC scores (0.97914 and 0.97965). GNB and LR classifiers performed worst (0.54479 and 0.5072). The KNN classifier performs well (0.9595). SGD achieved a reasonable score (0.8432). The increase in the score under AHBO-TPE for LightGBM is considerable (0.97914 to 0.99755). The tool's default parameter choices cannot be relied upon to produce the best or even good results.

Table 3.5: Score comparisons for the Application of HPO (Kaggle Dataset).

ML Model	Default AUC ROC Score	Grid Search Optimised AUC ROC Score	Random Search AUC ROC Score	AHBO-TPE AUC ROC Score
LightGBM	0.97914	0.98247	0.99809	**0.99755**
RF	0.97965	N/A	N/A	0.97819
KNN	0.94888	N/A	N/A	0.95954
LR	0.5	N/A	N/A	0.50729
SGD	0.84065	N/A	N/A	0.84322
* GNB	0.54475	N/A	N/A	Same

There are no hyper-parameters for GNB; hence AHBO-TPE results are the same value as for defaults.

Tables 3.6 and 3.8 illustrate the difficulty of manually tuning parameters. In some cases, the defaults and the best-found values are at the opposite ends of the parameter ranges, e.g., the bagging fraction in Table 3.6. Many are significantly different from the default value, e.g., *num_leaves* in Tables 3.7 and 3.8 and *n_estimators* in Table 3.8. Some binary choices are reversed, e.g., *objective* and *is_unbalance* of Table 3.7.

Table 3.6: LightGBM Grid Search Hyper-parameter Results (Ember Dataset).

Hyper-Parameter	**Grid Search Best Hyper-Parameter Settings**	**Range**	**Default Value**
boosting_type	GBDT	GBDT, DART, GOSS	GBDT
num_iteration	1000	500:1000	100
learning_rate	0.005	0.005:0.05	0.1
num_leaves	512	31:2048	31
feature_fraction	1.0	0.5:1.0	1.0
bagging_fraction	0.5	0.5:1.0	1.0
objective	binary	binary	None

Table 3.7: LightGBM Random Search Hyper-parameter Results (Ember Dataset).

Hyper-Parameter	**Random Search Best Hyper-Parameter Settings**	**Range**	**Default Value**
boosting_type	GBDT	GBDT or GOSS	GBDT
num_iteration	60	1:100	100
learning_rate	0.0122281	0.005:0.05	0.1
num_leaves	150	1:512	31
feature_fraction	0.8	0.5:1.0	1.0
bagging_fraction	0.8	0.5:1.0	1.0
objective	binary	binary only	None
min_child_samples	165	20:500	20
reg_alpha	0.102041	0.0:1.0	0.0
reg_lambda	0.632653	0.0:1.0	0.0
colsample_bytree	1.0	0.0:1.0	1.0
subsample	0.69697	0.5:1.0	1.0
is_unbalance	True	True or False	False

The hyper-parameters giving the best performance for each ML model are given in Tables 3.8–3.12. Here, AHBO-TPE was used as the HPO approach. The results are shown with 10 iterations (a constraint imposed for reasons of computational practicality) and 3-fold cross-validation. All results were obtained using the Windows 10 environment indicated in Section 3.3.

Table 3.8: LightGBM AHBO-TPE Search Hyper-parameter Results (Ember Dataset).

Hyper-Parameter	**Random Search Best Hyper-Parameter Settings**	**Range**	**Default Value**
boosting_type	GBDT	GBDT or GOSS	GBDT
num_iteration	26	1:100	100
learning_rate	0.02469	0.005:0.05	0.1
num_leaves	229	1:512	31
feature_fraction	0.78007	0.5:1.0	1.0
bagging_fraction	0.93541	0.5:1.0	1.0
objective	binary	binary only	None
min_child_samples	145	20:500	20
reg_alpha	0.98803	0.0:1.0	0.0
reg_lambda	0.45169	0.0:1.0	0.0
colsample_bytree	0.89595	0.0:1.0	1.0
subsample	0.63005	0.0:1.0	1.0
is_unbalance	True	True or False	False
n_estimators	1227	1:2000	100
Subsample_for_bin	160,000	2000: 200,000	200,000

Table 3.9: SGD Model AHBO-TPE Search Hyper-parameter Results (Ember Dataset).

Hyper-Parameter	**AHBO-TPE Search Hyper-parameter Results**	**Range**	**Default Value**
Penalty	L2	L1, L2, elasticnet	L1
Loss	Hinge	hinge, log, modified-huber, squared-hinge	Hinge
Max-iterations	10	10:200	1000

Table 3.10: RF Model AHBO-TPE Search Hyper-parameter Results (Ember Dataset).

Hyper-Parameter	AHBO-TPE Search Hyper-Parameter Results	Range	Default Value
n_estimators	100	10:100	10
max_depth	30	2:60	None
max_features	auto	auto, log2, sqrt	auto
min_samples_split	10	2:10	2
min_samples_leaf	30	1:10	1
criterion	gini	gini, entropy	gini

Table 3.11: LR Model AHBO-TPE Search Hyper-parameter Results (Ember Dataset).

Hyper-Parameter	AHBO-TPE Search Hyper-Parameter Results	Range	Default Value
max_iter	200	10:200	100
C	8.0	0.0:20.0	auto
solver	sag	liblinear, lbfgs, sag, saga	lbfgs

Table 3.12: KNN Model AHBO-TPE Search Hyper-parameter Results (Ember Dataset).

Hyper-Parameter	AHBO-TPE Search Hyper-Parameter Results	Range	Default Value
n_neighbors	15	1:31	5

In Figures 3.2 and 3.3, a comparison is given between the benchmark model results and those obtained using AHBO-TPE and Random Search to optimise parameters.

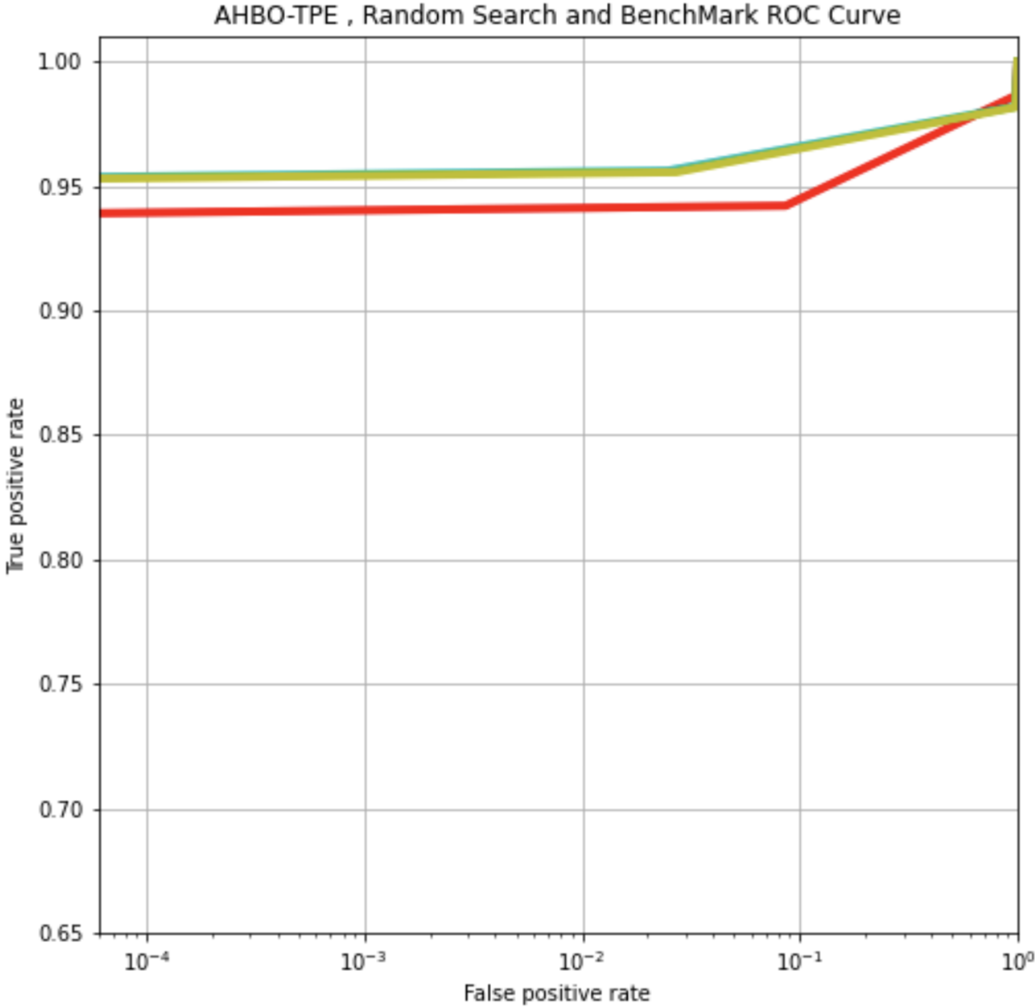

Figure 3.2: ROC AUC Comparison for AHBO-TPE (Cyan), Random Search (Yellow), and Default Benchmark Model (Red) applied to the Ember Dataset.

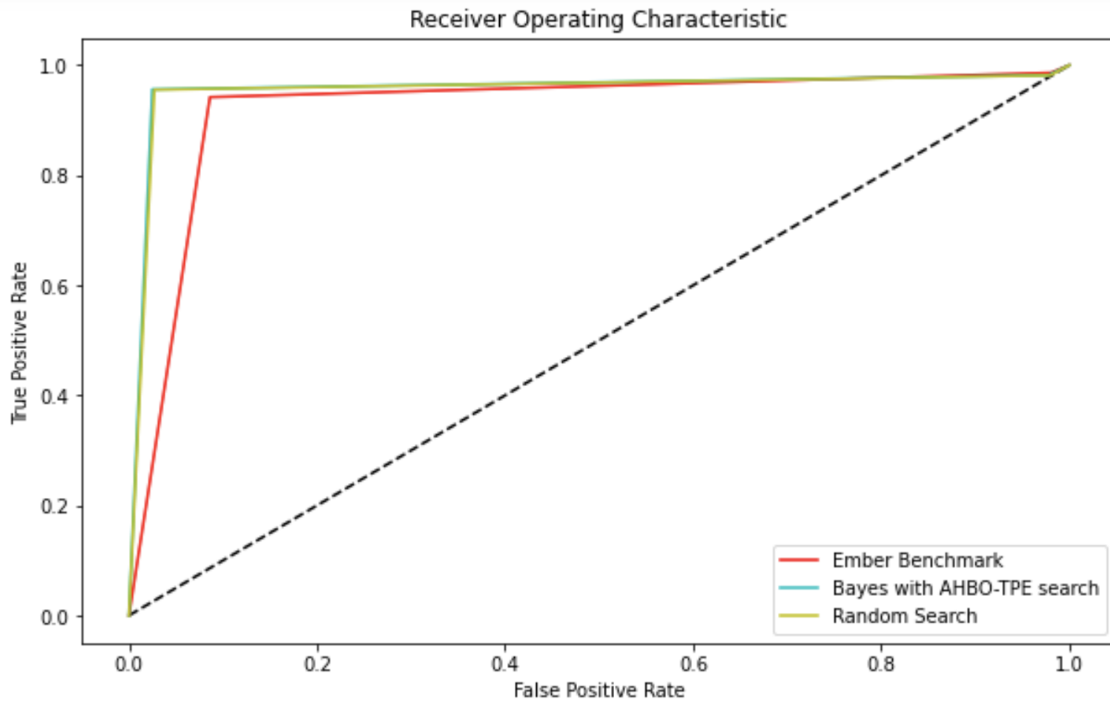

Figure 3.3: FPR and TPR Comparison for AHBO-TPE (Cyan), Random Search (Yellow), and Default Benchmark Model (Red) applied to the Ember Dataset.

Table 3.13: Confusion matrix for three classification models

	Actual Benign	Actual Malicious
AHBO-TPE	97886	1904
	2114	98096
Random Search	97888	1748
	2112	98252
Benchmark model	99000	3498
	1000	96492

Summary of the performance of each model based on the confusion matrix 3.13:

- AHBO-TPE: The model correctly classified 97886 (out of a total of 100,000) benign samples as benign and 98096 malicious samples (out of a total of 100,000) as malicious. However, it misclassified 2114 benign samples as malicious (false positives) and 1904 malicious samples as benign (false negatives).

- Random Search: The model correctly classified 97888 (out of a total of 100,000) benign samples as benign and 98252 malicious samples (out of a total of 100,000) as malicious. However, it misclassified 2112 benign samples as malicious (false positives) and 1748 malicious samples as benign (false negatives).

- Benchmark model: The model correctly classified 99000 (out of a total of 100,000) benign samples as benign and 96492 malicious samples (out of a total of 100,000) as malicious. However, it misclassified 1000 benign samples as malicious (false positives) and 3498 malicious samples as benign (false negatives)

Table 3.14: False positive rate (FPR) and false negative rate (FNR) for each classification model

Model	FPR (%)	FNR (%)
AHBO-TPE	2.114	1.904
Random Search	2.148	1.773
Benchmark model	1.000	3.498

Table3.14 shows the false positive rate (FPR) and false negative rate (FNR) which provides useful information about the performance of a classification model. In this case, the AHBO-TPE and Random Search models have similar FPRs and FNRs, while the Benchmark model has a lower FPR but a higher FNR.

It's important to note that the acceptable values for FPR and FNR depend on the application's requirements and the consequences of misclassification. For example, in a security-related application, a low FNR may be more critical than a low FPR since misclassifying a malicious sample as benign could have severe consequences. Conversely, in a healthcare-related application, a low FPR may be more critical than a low FNR since misdiagnosing a healthy individual as sick could lead to unnecessary treatment and costs.

Table 3.15 illustrates the highest performing parameters obtained using AHBO-TPE for the kaggle.com dataset.

Table 3.15: ML Models Hyper-parameter results using AHBO-TPE (Kaggle Dataset).

ML Models	Hyper-Parameter	Range	Best Hyper-Parameter Results
LightGBM	num_leaves	1:512	10
	Min_child samples	20:500	90
	n_estimators	10:100	19
	boosting_type	gbdt	Gbdt
	learning_rate	0.01:0.5	0.4418140187193226
	Subsample_for_bin	2000: 200,000	80,000
	Colsample_bytree	0.6:1.0	0.7307181013749854
	feature_fraction	0.5:1.0	0.6726481091942302
	Bagging_fraction	0.5:1.0	0.5893616201923844
	Reg_alpha	0.0:1.0	0.195989486417426
	Reg_lambda	0.0:1.0	0.1939778453324642
	Is_unbalance	True, False	False
	objective	Binary	Binary
RF	n_estimators	10:200	100
	max_depth	10:50	15
	max_features	auto, sqrt	sqrt
	min_samples_split	10:50	19
	min_samples_leaf	10:50	10
	criterion	entropy, gini	entropy
KNN	n_neighbors	1:100	3
GNB	N/A	N/A	N/A
SGD	penalty	none, l1, l2, elasticnet	L2
	loss	hinge, log, squared_hinge	log
	max_iter	20:1000	790
	alpha	0.0001:0.2	0.0001
LR	Max_iter	20:500	155
	C	1.0:50.0	7
	solver	lbfgs, sag, saga	sag

3.5 Discussion

The results show how the default values of parameters generally give suboptimal results and how optimal choices of the parameter values for various models can vary significantly from their defaults. The results also show that applying HPO to malware detection can be computationally practical. Where there are a great number of hyper-parameters (for example, LightGBM has more than one hundred), some *efficient* automated means of determining effective choices are essential. Credible manual tuning will not be feasible, and many HPO approaches may be computationally impractical. The work has shown the utility of using proxy evaluation functions for determining hyper-parameter values. In particular, AHBO-TPE has been shown to be a very effective and efficient informed approach. Other forms of surrogates may bring benefits.

For practical purposes, we informally identified plausible parameters that should be subject to variation and allowed the remaining ones to be set at the defaults. It is possible that improvements in results could be obtained by allowing variation in the parameters that were fixed at their default values. It also suggests the possibility of adopting a sequential approach to optimising over the full range of parameters, i.e., once investigated parameters have been subject to variation and evaluation, they could be fixed at their optimal values and previously fixed parameters then are allowed to vary. The focus of our work has been Windows PE files. Similar investigations of other malware types are now needed to determine how well our approach generalises.

3.6 Conclusions

We have shown that HPO matters a great deal for ML-based malware detection. The use of default parameters will generally not be optimal, and the results overall would suggest researchers in malware and ML are missing a significant opportunity to use HPO to improve results attained by specific techniques of interest. Every improvement matters to the security of the protected systems and reduces costs in one form or another: getting the best out of malware detectors matters a great deal, and HPO has much to offer. We have also shown that a specific informed technique (AHBO-TPE) has particular potential for application to malware detection.

Using HPO to provide Bergstra et al.'s 'formal outer loop' should be normal practice to ensure any targeted technique is exploited fully. Adopting HPO in this way brings methodological benefits. For the development of the field, we need to be able to compare competing techniques at their best, and HPO can provide a principled and repeatable way to get the best (or close to it) from all competing techniques. We propose that HPO be an essential element of the ML process for malware detection applications, i.e., that Bergstra et al.'s 'formal outer loop' be adopted, and recommend further research into the use of HPO for tuning malware detectors.

3.7 Summary

In this chapter, we investigate the use of various ML algorithms for building malware classifiers and how best to tune the parameters of those algorithms – generally known as hyper-parameter

optimisation. We examine the effects of some simple (model-free) parameter tuning methods and a state-of-the-art Bayesian model-building approach. We argue that HPO should be the norm to ensure that any technique is fully explored. We need to be able to compare competing techniques at their best for the field's development, and HPO can provide a principled and repeatable way to get the best (or close to it) from all competing techniques. Our work is carried out using EMBER, a major published malware benchmark dataset on Windows Portable Execution (PE) metadata samples and another dataset from kaggle.com.

Chapter 4

Covering Arrays ML HPO for Static Malware Detection

This chapter shows the use of covering arrays as a design of experiments approach to the application of HPO for classical ML techniques of interest.

4.1 Introduction

4.1.1 Covering Arrays: Dealing with the Curse of Dimensionality

Grid Search applies full combinatoric evaluation of the cross-product of discretised domains. The total number of combinations is the product of the cardinalities of the individual domains D_i.

$$totalCombinations = \prod_{i=1}^{n} card(D_i)$$

Grid Search can obviously give a thorough exploration of the parameter space, assuming the individual domains are suitably discretised. However, in some areas of engineering, it is found that full combinatorial evaluation can be wasteful. For example, in software testing, particular sub-combination coverage of parameter values can provide a very high fault detection capability. But we may not know in advance the specific sub-combinations that will be most revealing. Some effective means of exploring the combinatorial space is needed that do not incur the costs of a full Grid Search.

Covering arrays provide one such mechanism. Furthermore, the concept can be applied to different 'strengths' allowing flexibility in the thoroughness of the exploration of the search space at hand. At a basic level, a covering array is defined by the number n of its domains (parameters) and its strength t. We will denote strength here using $t = 1$, $t = 2$, $t = 3$ etc. A covering array with n domains and with strength t can be referred to as a CA_t^n.

Consider a combinatorial search space with parameter domains $A = \{0,1\}$, $B = \{0,1\}$, and $C = \{0,1\}$. A CA_1^3 provides a suite of cases where each value of each domain occurs at least once. Here, we are considering sub-combinations involving only single ($t = 1$) domains. This is easily achieved

by an array with just two case rows as shown below.

A	B	C
0	0	0
1	1	1

If we had, say, 26 binary domains $A,B,...,Z$, then a similar covering array, i.e. a CA_1^{26}, with two rows would satisfy the $t = 1$ strength requirement, i.e. with rows as shown below.

A	B	C	..	X	Y	Z
0	0	0	..	0	0	0
1	1	1	..	1	1	1

A CA_1^n clearly gives a rather weak coverage (exploration) of the domain space for most purposes. In the $A,B,...,Z$ example, only 2 from 2^{26} possible row values are sampled.

For a CA_2^n each combination of values from any two ($t = 2$) domains is present in the array. A CA_2^3 for the A, B, and C example is given below.

A	B	C
0	0	0
0	1	1
1	0	1
1	1	0

We can see that the four possible values of (A, B) are present, i.e. A, B=$(0,0)$ in row 0, $(0,1)$ in row 1,$(1,0)$ in row 2, and $(1,1)$ in row 3. Similarly, we can see that four possible values of (A, C) and the four values of (B, C) are also present. Thus, all pairs of values from any two domains from A, B and C are present and so the given array is indeed a CA_2^3. The simplest strength CA_3^3 array would give full combinatorial coverage (all 8 (A, B, C) combinations) with the usual binary enumeration of 0 to 7 for the rows, i.e. $[0,0,0]$ through to $[1,1,1]$.

4.1.2 Generating Covering Arrays

The actual generation of arrays is not our focus. A good deal of theoretical and practical work has been carried out in algorithms to do so. Our motivation for using covering arrays was inspired by their use in software testing. A method for generating CA_2^n arrays for software test suites is given in [126]. As they state, "For a system with two or more input parameters, the IPO strategy generates a pairwise test set for the first two parameters, extends the test set to generate a pairwise test set for the first three parameters, and continues to do so for each additional parameter."

CAs are widely used in the combinatorial testing field. Their use reduces the number of tests needed in comparison to exhaustive combinatorial testing. This led to an increased usage of a specific instance of the IPO strategy called In-Parameter-Order-General (IPOG). IPOG can be used to generate covering arrays of arbitrary strengths [125]. It is a form of greedy algorithm and might not yield test

suites of minimal size. It has been noted that providing an optimal covering array is an NP-complete problem [139].

The IPOG strategy has gained traction in the software testing field. This is due to the competitive test suites that are yielded by such covering arrays in comparison with other extant approaches for generating test suites. Additionally, it exhibits a lower generation time of test suites compared with other algorithms. The main goal of IPOG is to minimize the generated test suite size. This is a significant area to explore especially when the cost of testing is very high. The duration of test cycles will be reduced with fewer tests. However, there are some cases when the test execution is very fast and does not impact the overall testing time. Instead, optimizing test suites can be very costly as test generation time can become dominating [140], [141]. Optimisation of the IPOG family was introduced by [130].

In this chapter, we use an implementation of FIPOG (an IPOG variant) provided by the cAgen tool. cAgen is freely available as indicated below. We show that the use of FIPOG's covering arrays can achieve excellent results (and better than using the default parameters) far more quickly than using full Grid Search. In our approach, an index in a row of a covering array can select a value from a discrete set or a *subrange* of values from a given range (and then randomly sample from within the selected subrange). We believe this to be original.

4.2 Research Question

In this chapter, we investigate a simple research question:

RQ: Can covering arrays provide a highly efficient and effective means of hyper-parameter optimisation for machine learning-based Windows PE malware detectors?

4.3 The cAgen Array Generator

4.3.1 Parameter Specification and Array Generation

The cAgen tool set is available online [83]. It allows the user to specify parameters and sets of associated values. For technical reasons that are concerned with our specific approach to the use of covering arrays, we will assume that a parameter domain with R elements is indexed by values 0, 1, (R-1). Figure 4.1 shows a completed specification for the (A,B,C) example above.

Input Parameter Model

Export IPM...

Name	Values	Cardinality
A	0,1	2
B	0,1	2
C	0,1	2

Add Type Name

Constraints

Figure 4.1: Full Parameter Specification for ABC Example

Array Generation

Algorithm: FIPOG t 2 λ 1 Generate

TEST SET

t=2 4 rows Randomize Don't-Care Values Show model values Export...

A	B	C
0	0	0
0	1	1
1	0	1
1	1	0

Showing rows 1-4

Figure 4.2: Array Generation for ABC example above with t=2

Having specified the parameters we can invoke the generation capability of cAgen. Figure 4.1 shows the array generation stage for the A, B, and C examples, where a value of t=2 has been selected. If we wanted each pair to occur multiple times, we could specify a larger value of *lambda*. Several generation algorithms are available. Figure 4.2 shows that we have chosen FIPOG, for better performance and fast generation [83]. The array can then be stored in a variety of formats. We have chosen to use CSV format throughout for compatibility with our general approach in this thesis.

4.3.2 Array Indexing

We will use lists to represent parameter spaces. A list's elements will be either actual parameter values or else a list representing a subdomain. The values $0, 1, \ldots, (R-1)$ are interpreted as indices to the corresponding elements in the domain list. For example, $MAX_DEPTH = [5, 10, 15, 20, None]$ would be a simple list with 4 specific integer values and a 'None' value. $LEARNING_RATE =$

$[0.001, .01, 0.1, 0.2]$ is a simple list of four real values. $MAX_LEAVES = [[1,2,3],[4,5,6],[7,8,9]]$ is a list of lists of values. Here, the list cardinalities are given by $card(MAX_DEPTH) = 5$, $card(LEARNING_RATE) = 4$ and $card(MAX_LEAVES) = 3$. Thus, for the list of lists the cardinality is the cardinality of the highest-level list. A value in a row of a covering array is an index into the top-level list for the corresponding parameter.

Python lists allow us to include different types of elements. Thus, in MAX_DEPTH we see that integer values, as well as a 'None' parameter value, can be specified. 'None' typically means that the algorithm can proceed as it sees fit, with no direction from the user for this parameter. $Scikit-learn's$ ML algorithms often have such parameters as defaults. Where a parameter is represented by a simple list then the covering array value for the parameter is used to index the specific element of the list. Thus, an array value of 2 in the covering array column corresponding to MAX_DEPTH corresponds to a parameter value of 15, i.e. $MAX_DEPTH[2] = 15$. The indexed array element may be a list. Thus, a covering array value of 2 for MAX_LEAVES, gives $MAX_LEAVES[2] = [7,8,9]$. In such a case, a value is randomly selected from the list. Thus, each of the values 7, 8, and 9 are now selected with a probability of 0.333. In practice, we represent ranges of integer values more compactly, via the use of low, high, and increment indicators. Thus, we will typically represent the list $[1,2,3,4,5]$ by $[low, high, incr] = [1,5,1]$. We adopt the convention of both low and high being included in the denoted range.

4.4 Methodology

We will apply a variety of ML techniques and specify suitable parameter domains for the parameters we wish to experiment with. We will evaluate the full combinatorial domain and over all rows of the covering arrays of interest (for t=2,3,4). A full combinatorial evaluation or a full covering array evaluation (i.e. all rows evaluated) will be referred to as a 'run' or 'iteration'. We will carry out 30 runs for each array of interest and for the full combinatorial case. We do this to gain insight into the *distribution* of outcomes from the technique. Some runs may give better results than others, even if the same array has been used as the basis for the run. This is due to the stochastic selection of elements within selected ranges as indicated above. Pooling the results from the 30 runs provides a means of determining an accurate and useful distribution for the approach. In practice, a user may simply use one run of a covering array search, if he or she is confident it will give good enough results. Our evaluation activities aim to determine whether such confidence is justified.

4.5 Experiments

Here we outline the experiments carried out and provide sample data and execution environment details. Discussion of the results is given in Section 4.6.

4.5.1 Execution Environment

Our work uses two powerful freely available toolkits: Scikit-learn [22] and the cAgen tool [83]. The the experiments were carried out using the Windows OS 11, with 11 Gen Intel Core i7-11800H, with 2.30 GHz processor, and 16 GB RAM.

4.5.2 Experiments Settings

This section shows the results when various ML techniques are applied with the specific parameter settings of t values. The techniques include well-established approaches: Decision Trees (DT) [22], xgboost [38] , and Random Forest (RF) [22, 137]. A state-of-the-art approach—LightGBM [39]—is also used. For initial experiments, we adopted the default parameter settings by cAgen toolkit for RF, DT, LightGBM and Xgboost. Then we added our own setting as seen in table 4.1. The results will be obtained using the evaluation metric of scikit-learn's Accuracy [142].

4.5.3 Implementation Details

Table 4.1 shows the implementation details for all four ML models using cAgen tool.

Table 4.1: ML Models cAgen configurations

ML Algorithms	Hyper-parameters	Hyper-parameter IPM values	T-Strengths values	IPM values	Number of Iterations
RF	n_estimators, max_depth, criterion, min_samples_split, min_samples_leaf, max_features	[[100, 300, 50],[350, 550, 50],[600, 800, 50]] [[1, 10, 1],[11, 15, 1],[16, 20, 1],None] ['entropy', 'gini'] [[5, 25, 5],[30, 50, 5]] [[5, 25, 5],[30, 50, 5]] ['auto', 'sqrt', 'log2', 'None']	T-2, 3, 4	0,1,2 0,1,2,3,4 0,1 0,1 0,1 0,1,2,3	30
LightGBM	num_leaves, boosting_type, Subsample_for_bin, Is_unbalance, max_depth	[[20,80,20],[100,160,20]] ['GBDT','GOSS'] [[1000,5000,1000],[6000,10000,1000],[11000,15000,1000]] [True, False] [1,5,10,15,20,25]	T-2,3,4	0,1 0,1 0,1,2 0,1 0,1,2,3,4,5	30
Xgboost	Min_child_weight gamma max_leaves reg_alpha max_depth	[1, 2, 4, 6, 8, 10, 12, 14] [[1, 4, 1],[5, 8, 1]] [2, 4, 6, 8, 10, 12] [0.01,0.1,0.2,0.3,0.4,0.5] [1,5,10,15,20,25]	T-2,3,4	0,1,2,3,4,5,6,7 0,1 0, 1, 2, 3,4,5 0, 1, 2, 3, 4,5 0, 1, 2, 3, 4, 5	30
DT	max_depth, criterion, min_samples_split, min_samples_leaf, max_features	[[1,10,1],[11, 15, 1],[16, 20, 1], None] ['entropy', 'gini'] [[5, 25, 5],[30, 50, 5]] [[5, 25, 5], [30, 50, 5]] ['auto', 'sqrt', 'log2', 'None']	T-2,3,4	0, 1, 2, 3 0, 1 0, 1 0, 1 0, 1, 2, 3	30

There are three target ranges in our implementation: a) Simple lists, e.g. [0,1,2,3,4 or None], where the covering array index indicates the specific value. b) List of lists, (e.g. [[30,50,1],[350,50,1]]), where the covering array index indicates which inner list (subdomain) should randomly be chosen from. c) Boolean values, then either one or another depending on the values (e.g Entropy or Gini).

4.6 Results

Results of hyper-parameter optimisation based on covering arrays (with strengths of 2, 3, or 4) and Grid Search are shown in the following tables. The best-performing parameter values are given, together with the time taken to complete the corresponding search, coverage (number of evaluations) and summary accuracy data. Tables 4.2, 4.3, 4.5 and 4.4 show results for RF, LightGBM, Xgboost, and DT respectively. The results for Grid Search (over the same discretised parameter ranges) are also shown in each table. In the tables "No. of evaluations" is equal to the number of rows (i.e. combinations) in the covering array multiplied by the number of iterations (30).

Table 4.2: RF Model cAgen Results Comparison

ML Algorithms	Optimal Values	T-values/ Grid Search	Time to complete	No. of evaluations searched	Score (accuracy)
RF	400 14 entropy 5 10 None	T2	2h 35min 21s	480	0.9904
	700 None entropy 5 10 None	T3	7h 31min 17s	1500	0.9902
	150 18 entropy 5 10 None	T4	11h 58min 4s	2880	0.9902
	650 19 entropy 5 5 None	Full Grid Search	2 Days, 23 hrs, 58Min and 18s	11520	**0.9906**

Table 4.3: LightGBM Model cAgen Results Comparison

ML Algorithms	Optimal Values	T-values/ Grid Search	Time to complete	No. of evaluations searched	Score (accuracy)
LightGBM	gbdt 80 2000 False 20	T2	1h 25min 13s	540	**0.9910**
	goss 140 16000 False 25	T3	2h 34min 30s	1080	0.9906
	gbdt 40 1000 False 25	T4	4h 35min 39s	2160	0.9910
	goss 80 2000 False 20	Full Grid Search	6h 48min 5s	4320	0.9910

We can see that the DT classifier in Table 4.4 is the fastest of all ML models. Even with Grid Search it is still efficient with this particular technique taking only 5 minutes and 43 seconds to

Table 4.4: DT Model cAgen Results Comparison

ML Algorithms	Optimal Values	T-values/ Grid Search	Time to complete	No. of evaluations searched	Score (accuracy)
DT	entropy 13 None 5 20	T2	27.2s	480	0.9843
	entropy 14 None 5 10	T3	1min 29s	1500	0.9845
	gini 18 None 10 5	T4	2min 46s	2880	0.9855
	gini None None 10 20	Full Grid Search	3min 39s	3840	**0.9859**

Table 4.5: Xgboost Model cAgen Results Comparison

ML Algorithms	Optimal Values	T-values/ Grid Search	Time to complete	No. of evaluations searched	Score (accuracy)
Xgboost	1 1 10 0.4 20	T2	1h 4min 41s	1440	0.9902
	1 1 4 0.4 25	T3	5h 20min 55s	8640	0.9902
	1 1 4 0.01 15	T4	22h 15min 38s	51840	0.9906
	1 1 2 0.01 15	Full Grid Search	1d 10h 46min 52s	103680	**0.9906**

finish 3840 evaluations. However, cAgen is much more efficient with only 42.5 seconds to finish. Although only 480 evaluations with $t = 2$ are made it achieves the same accuracy as Grid Search but with less time and effort. The second fastest ML model after DT was LightGBM, which highlights covering array capability even more. Table 4.3 shows a huge disparity in time between Grid Search and cAgen runs. The cAgen approach is faster than the Grid Search with only 1 hour, 41 minutes and 18 seconds taken to complete the search, while the latter took 7 hours, 57 minutes and 14 seconds to complete. Both have reached excellent values for finding hyper-parameter choices while having higher accuracy. Strength values $t = 2$ and $t = 3$ in LightGBM, even though they have almost the same results obtained but both have reached that score with different hyper-parameter values. cAgen is more efficient than Grid Search, using less time. The third ML model was RF, where cAgen runs have reached the highest performing choices for $t = 2$ with 2 hours and 35 minutes. In contrast, Grid Search took 2 days 23 hours and 58 minutes to complete the search. The difference between cAgen and Grid Search in Table 4.2 is significant evidence of the usefulness of covering arrays for hyper-parameter optimisation. Xgboost was the slowest of all models to achieve the best values. It took more computational time than the other techniques to achieve the best values for strengths t3 and t4, and even Grid Search. The figures below 4.3, 4.4 and 4.5 compare the accuracy results between the selected models ($t = 2$, $t = 3$ and $t = 4$) in a histogram. (These histograms are not normalised between techniques, i.e. the total counts may vary between techniques. However, the general distributions can be compared.)

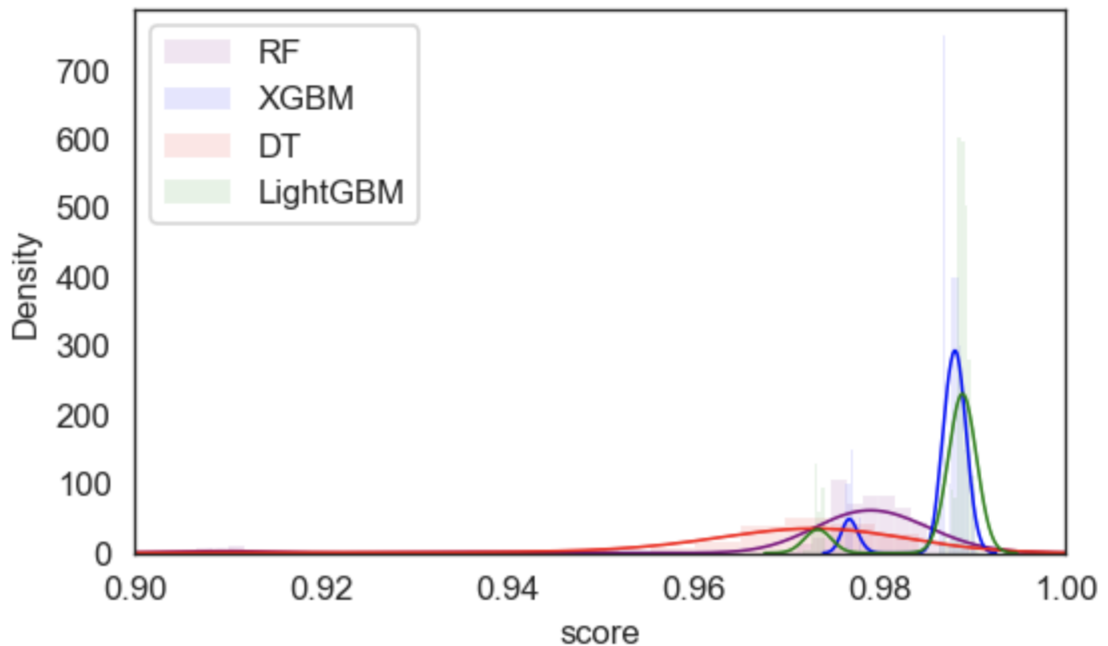

Figure 4.3: cAgen ML Models Results Comparison for Strength $t = 2$

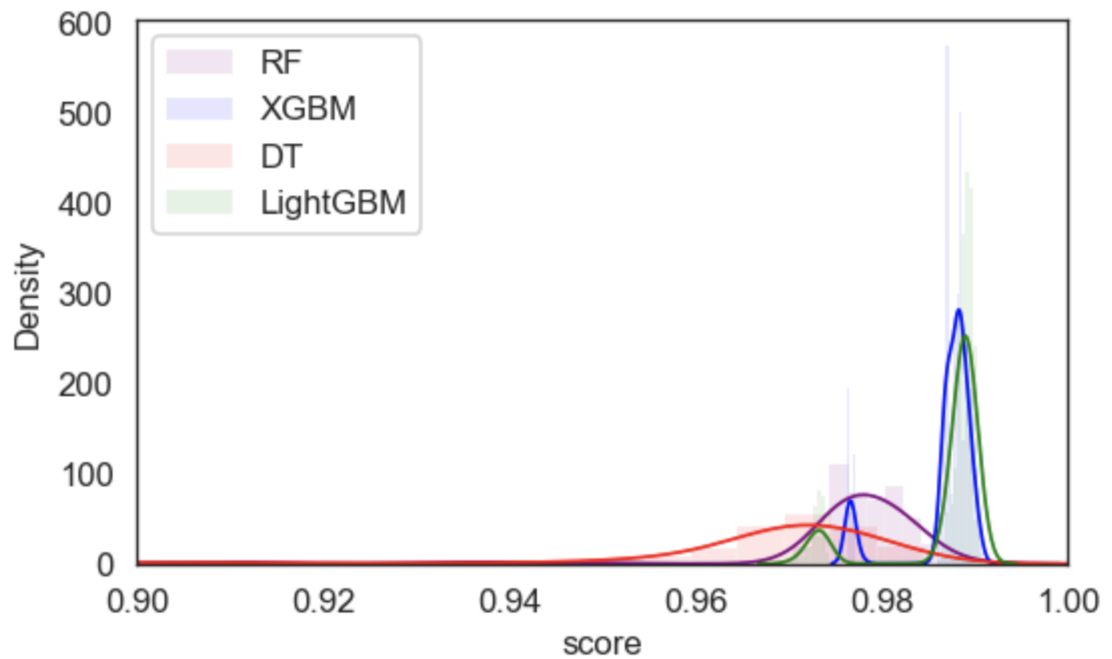

Figure 4.4: cAgen ML Models Results Comparison for Strength $t = 3$

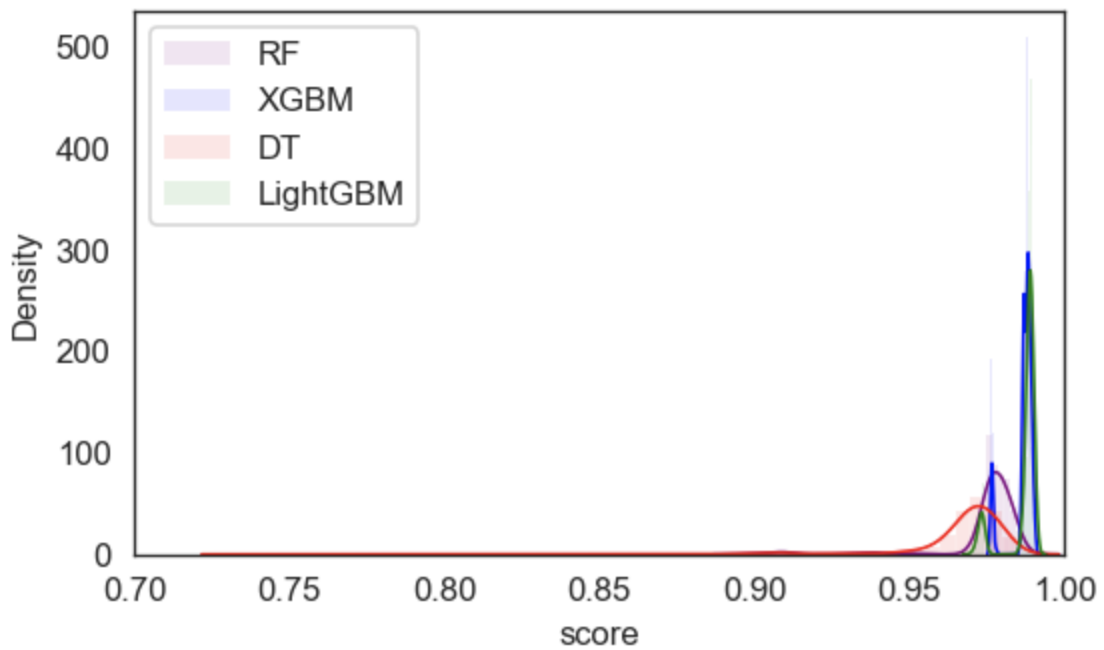

Figure 4.5: cAgen ML Models Results Comparison for Strength $t = 4$

4.6.1 Comparing the Results of CA against another Benchmark

The authors in [136] have benchmarked several ML models' performances using Ensemble learning with 10-fold cross-validation. For DT and RF the accuracy results were 0.989 and 0.984 respectively. Our model achieved 0.9849 and 0.9904. However, the main aim of our paper is to evaluate various coverage strategies, not necessarily to achieve an optimal value for each ML technique application. If explicit optima are the target then further optimisations should be considered.

4.7 Conclusions

cAgen, a covering array approach with various strengths, was used to find high-performing hyper-parameters for targeted ML models. It was compared to Grid Search. Our results show that the systematic coverage offered by covering arrays can be both highly effective and efficient. The covering arrays produced by cAgen produced superior results to Grid Search across all four ML models. We highly recommend the covering arrays approach for ML researchers and the community overall. Although our work has focused on improving attained accuracy of malware classification, other tasks in security may well benefit from such an approach (particularly ML-based classification tasks).

4.8 Summary

Four machine learning techniques were tuned using cAgen, a tool for generating covering arrays. The results showed that cAgen is an efficient approach to achieving optimal parameter choices for ML techniques. Furthermore, we recommend this unique approach of optimising parameters and choosing optimal hyper-parameters of a given ML technique. This research will help the development of better classifiers for static PE malware detection. Using Covering Arrays may be of significant benefit to the ML hyper-parameter optimisation community, malware detection community and overall security testing community. Furthermore, the cAgen toolkit has proved readily usable for our own malware detection purposes. Its flexibility will also serve the various communities well.

Chapter 5

HPO for Evolutionary Algorithms

This chapter demonstrates the use of two EA methods to enhance and optimise the hyper-parameters of six ML models.

5.1 Introduction

In this chapter, our focus will be on meta-heuristic algorithms, specifically Evolutionary Algorithms (EAs). This chapter 1) investigates the effectiveness of hyper-parameter tuning using the algorithms of two significant optimisation frameworks: Deap [98, 97] and TPOT [111]; 2) develops understanding to help new users of ML to get best value of applying ML; and 3) Provides specific recommendations for hyper-parameter choices for our chosen deployment area. Section 5.2 gives the aim of the chapter. Section 5.3 presents our research question. Section 5.4 describes the experimental setups. Section 5.5 gives the results and a discussion of our work. Section 5.6 provides conclusions. Section 5.7 summarises the chapter.

5.2 Aim of this chapter

In this chapter, we use two optimisation frameworks, DEAP and TPOT, described in section 2 for hyper-parameter optimisation of 6 ML techniques. DEAP chromosomes encode for parameters of an identified ML technique. Evolution proceeds to develop fitter (higher performing) parameters. TPOT is free to choose *which ML model* to use as part of its operation.

5.3 Research Question

We investigate the following research question:

RQ: Can Evolutionary Algorithms, specifically TPOT and Deap, find hyper-parameter vectors that produce better performance than the defaults for machine learning-based Windows PE malware detectors?

5.4 Experiment Setup

ML models were based on Scikit-learn [22]. The experiments were carried out on an iMac (Mac operating system Big Sur, version 11.0.1, Retina 5K, 27-inch, 2017,4.2 GHz Quad-Core Intel Core i7,16 GB 2400 MHz DDR4 RAM, and a Radeon Pro 580 8 GB graphics card), Jupyter Notebook version 6.1.0, and Python Version 3.7.0. The work used the Ember dataset.

5.4.1 Initial Experiments and Evaluation Metrics

This section describes the proposed Deap and TPOT models. Six traditional ML models will be fine-tuned with the help of Deap. Automatic model selection will be performed using TPOT. Deap selection criteria are default hyper-parameters and manual tuning for some of them. This experiment aims to compare the accuracy [142] gained after finding the optimal hyper-parameters from TPOT and Deap. We have taken the default ML model's accuracy for all six models, and the results are shown in Table 5.3. Our benchmark model LightGBM had the highest accuracy.

5.4.2 Deap Setup

For our experiment, we have chosen three-fold validation. Several other parameters must also be initialised at random before beginning any experiment. We will focus on a specific set of parameters. For more information about the complete set of parameters, please refer to the documentation in [109]. Ten parameters are used, excluding the ML model's parameters. Namely, the estimator (that is the ML model to be used and in our case the 6 models we selected) params (the specific hyper-parameters that we want to initialise beforehand), scoring (accuracy) for our experiments, population size (population size of the GA, it has an integer value and the default size is 50), gene mutation probability (chromosome mutation probability, a float value with default of 0.1), gene crossover probability (the probability of exchanging genes between different chromosomes, a float value with default value of 0.5), tournament size (the size of tournament selection stage in the GA, an integer value with default of 3), generation number (is the number of generation for GA, an integer number with default size of 10), CV (the cross validation iterator to split the data set, with default value of 3) and the last one is verbose (verbosity, this one controls the messages that is generated throughout the search; the higher the number, the more messages shown and it has an integer value). The experiment took about three months to complete. Deap hyper-parameters configurations are below; all of these are initialised manually beforehand:

Table 5.1: Pre-initialised Deap Hyper-parameter Configuration Ranges

Hyperparameter	Hyper-parameter Range
Population size	1 : 50
Gene-mutation-prob	0.1 : 0.90
Gene-crossover-prob	0.1 :1.0
Tournament-size	1 : 10
Generations-number	5: 15

5.4.3 TPOT Setup

TPOT is easier to use since it is automated. However, we need to select the right setting beforehand. The first is the built-in configurations (it is for classification and regression tasks; however, we are focusing on classification only). The second setting is a customised configuration, however, we will not address it in our experiment. We note that the search space taken from documentation needs a little bit of improvement, and we hope this will also aid TPOT in the future. There are six built-in configurations in TPOT, default TPOT, TPOT Light, TPOT MDR, TPOT Sparse, TPOT NN, and TPOT cuML. For more information about each configuration please refer to the documentation in [108]. The custom configuration with search parameters already predefined from the same ML model existing parameters inside TPOT. Some hyper-parameters are introduced as well from the GA, and we need to randomly initialise [108]. Again, we will include only the parameters we used in our experiment. Namely, config-dict (the name of the built-in configuration that we want to use, e.g. TPOT NN), generations (the number of generations TPOT will run, and it must be positive), population size (the number of individuals to retain the population for each generation), early stopping rounds (TPOT will check if there is no improvement in the current pipeline. If so, it will terminate), and the last hyper-parameter is the score (accuracy in our experiment). Below we have TPOT GA hyper-parameter configurations; all of these are initialised manually beforehand for Config-dict TPOT hyper-parameters:

Table 5.2: Pre-initialised TPOT hyper-parameter configurations (Config-dict)

Hyperparameter	Hyper-parameter Range / Metrics
Population size	5 : 30
Generation	3 : 30
Early Stopping Rounds	3 : 10
Score	Accuracy

5.4.4 TPOT Pre-built Configurations

We ran the experiment 30 times for each built-in configuration that worked with the dataset. There were different config-dict initialisations, and for each, there were different values for the parameters. Out of the six configurations we are interested only in three (Default TPOT NONE, TPOT Light and TPOT NN).

5.4.5 TPOT Custom Configuration

TPOT custom configuration experiment with the predefined search space (within TPOT) for each ML classifier, we note the following:

- Due to conflicting hyper-parameters there will always be a run-time error with our dataset and only one-Pareto optimisation generation will be out. For this specific reason, we decided not to proceed with the customized configuration.

5.4.6 Target ML Algorithms

In our experiments, we investigated several widely used supervised ML classifiers: Logistic Regression (LR), Light Gradient Boosted Machines (LightGBM), Random Forest (RF), Stochastic Gradient Descent (SGD), Gaussian NaiveBase (GNB), and K-Nearest Neighbour (KN). For more information about each technique's default parameters please refer to [22].

5.5 Results and Discussion

This section presents the results of our experiments. Table 5.3 displays both the model's results with the default parameters and what is achieved via DEAP optimisation.

Table 5.3: ML Model Accuracy: Comparison using Default and Deap Optimization

Selected ML model	Default ML Models Accuracy Score	Deap Accuracy score
RF	89%	94%
KNN	69%	75%
LR	59%	62%
SGD	56%	59%
LightGBM	92%	94%
GNB	60%	60%

5.5.1 Deap Results

Optimizing Models with Deap: Out of the six selected classifiers, RF and LightGBM scored a tie in the model's accuracy. The original time to train LightGBM was reduced by 15 minutes (originally 26 minutes), and it took 11 mins 25 seconds to train. Moreover, RF's original time to train was decreased by 26 minutes (originally 52 minutes), and it took 26 minutes to finish training. RF and KNN significantly increased the model accuracy after being optimised with Deap. Table 5.3 compares defaults and Deap after optimizing the models. RF with the default parameters scored 89%, and with the newly found hyper-parameters, it is 94%, which is about a 5% increase in model performance. KNN default parameters scored an accuracy of 69%, and with the new hyper-parameter, the score jumped to 75%, which is about a 6% increase in the model performance. The rest of the models either stayed the same (GNB) or the other had a slight increase in accuracy LR (2% increase), LightGBM (2% increase) and SGD(3%). **Importance of Tuning GA Parameters:** tuning the predefined GA parameters is also important. For example, in LightGBM, values below 1.0 in gene-mutation-crossover and higher values than 0.5 in gene-mutation would lower the accuracy of the model. Also, a population size lower than 50 would cause the same issue. This shows how important it is to notice these small changes. **Effect of Different Parameters on Model Accuracy:** The accuracy of different ML models would change accordingly with different parameters. RF and KNN showed a significant increase in accuracy after optimizing hyper-parameters, while GNB remained the same (due to no hyperparameters).

Table 5.4: ML Model and Deap Search Space / Hyper-parameter results

ML Model	Default hyperparameter	Search Space	Best hyperparameter values	Deap model pre-defined hyperparameters	Deap pre-defined Search Space values
RF	n_estimators Max_features Max Depth min-samples-split Min_samples_leaf criterion	1:100 Auto 5:50 2:11 1:11 Entropy or Gini	84 auto 20 8 8 Entropy	scoring cv(cross validation) verbose Population-size gene-mutation-prob gene-crossover-prob tournament-size generations-number	Accuracy 3 1 35 0.50 0.90 6 10
SGD	penalty loss max_iter l1_ratio	L2 Hinge, squared hinge or modified huber 1:500 0:1	L2 Modified huber 366 0	scoring cv(cross validation) verbose Population-size gene-mutation-prob gene-crossover-prob tournament-size generations-number	Accuracy 3 3 10 0.10 0.5 5 5
GNV	N/A	N/A	N/A	N/A	N/A
LightGBM	Num_leaves Min_child_samples Boosting-type Learning_rate subsample_for_bin feature_fraction bagging_fraction Is_unbalance reg_alpha reg_lambda objective	31:800 20:400 GBDT 0.01:0.5 2000:200000 0.5:1.0 0.5:1.0 True or False 0.0:1.0 0.0:1.0 Binary	608 113 GBDT 0.01 106461 0.5 0.5 False 1.0 0.25 Binary	scoring cv(cross validation) verbose Population-size gene-mutation-prob gene-crossover-prob tournament-size generations-number	Accuracy 3 1 50 0.3 1.2 5 15
LR	max_iter C solver	1:500 0:100 Liblinear ,lbfgs, sag, saga	441 33.48 lbfgs	scoring cv(cross validation) verbose Population-size gene-mutation-prob gene-crossover-prob tournament-size generations-number	Accuracy 3 3 15 0.5 0.6 5 10
KNN	N-neighbours	1:20	16	scoring cv(cross validation) verbose Population-size gene-mutation-prob gene-crossover-prob tournament-size generations-number	Accuracy 3 1 10 0.10 0.5 3 5

5.5.2 TPOT Results

Issues with Hyper-parameters in TPOT Experiment: The hyper-parameters introduced by TPOT caused low accuracy for the problem due to conflicting parameters. **Results of Automated Search with TPOT:** BernoulliNB and Decision Trees were chosen multiple times by TPOT. The final results were the decision trees classifier in Table 5.5. **TPOT Classifiers:** TPOT chose BernoulliNB as the best classifier in all pre-built configurations (None, NN, and Light) with low accuracy. The Decision Trees classifier had the best accuracy out of all the classifiers chosen by TPOT.

Table 5.5: TPOT built-in Configuration Search Space Results.

TPOT built-in configuration value	Chosen ML Model and hyperparameters	TPOT predefined-Hyperparameter search space	TPOT predefined-search Range	Chosen Hyperparameter value	Accuracy score
NONE	BernoulliNB (alpha=0.1, fit_prior=True)	generations population_size Verbosity early_stop cv scoring	3:10 5:30 3 3:10 3 accuracy	3 15 3 4 3 accuracy	51%
Light	BernoulliNB (alpha=1.0, fit_prior=True)	Generations Population-size Verbosity early-stop cv scoring	3:10 5:30 3 3:10 3 accuracy	3 15 3 4 3 accuracy	61%
NN	Decision Trees Classifier (criterion=gini, max-depth=1, min-samples-leaf=10, min-samples-split=3)	Generations Population-size Verbosity early-stop cv scoring	3:10 5:30 3 3:10 3 accuracy	5 20 3 3 3 accuracy	61%

5.6 Conclusion

Here we used two modern evolutionary algorithm frameworks: TPOT and Deap. The default hyper-parameters of six different ML models were determined and then optimised using Deap. Deap can improve 5 of the six models, giving percentage improvements of (+5, +6, +3, +3, +2, 0). Notably, the default usage performances were improved even for the best two performing ML approaches (RF: 89% to 94 %) and (LGBM: 92% to 94 %). Models discovered by Deap were compared to those found by TPOT. Then, the best of each is compared to one another to determine which hyper-parameter setting is most effective. Deap returned better hyper-parameters for each ML model. When it comes to finding the optimal hyper-parameters and optimising ML models for classifying and detecting malware samples, Deap is the best EA.

5.7 Summary

The baseline accuracy of six traditional machine learning techniques was established under default parameter choices. The tuning capability of two evolutionary algorithms, the Tree-Based Pipeline Optimization Tool (TPOT) and the Distributed Evolutionary Algorithm in Python (Deap), were compared. The results show that Deap is an effective evolutionary search technique to optimise and increase machine learning classifier accuracy for static PE malware detection.

Chapter 6

Using CA and Grid Search with Deep Learning

This chapter illustrates the use of HPO approaches for Deep Neural Networks (specifically, the use of Covering Arrays).

6.1 Introduction

Deep Learning (DL) is a subset of Machine Learning (ML) that is widely used in different fields, with particular successes in Computer Vision, Natural Language Processing and Machine Translation. DL is a part of the Artificial Neural Network (ANN) theory. There are several kinds of DL models: Deep Neural Networks (DNNs), Feed Forward Neural Networks (FFNNs), Deep Belief Networks (DBNs), and others [40]. Most DL models have similar traits and hyper-parameters. The performance of the trained model depends significantly on the values of the parameters of the model [143, 144, 145, 146, 147, 148].

6.1.1 Aim of this Chapter

The main objective of the work of this chapter is to investigate specific hyper-parameter tuning of deep learning-based Windows PE malware detectors. We illustrate the significant effects of parameter choices on performance and show the power of hyper-parametrisation. Specifically, we build on the encouraging performance of covering arrays explored in section 4.3 for traditional ML approaches.

6.1.2 Chapter Contribution

The contributions of this chapter are:

1. We demonstrate the performance improvements that systematic hyper-parameter search can bring, working from a highly plausible baseline model for comparison.

2. We demonstrate that covering arrays from the cAgen tool can be used to efficiently find excellent parameter values with less time and fewer iterations compared to Grid hyper-parameter search.

The targeted performance metric used in this work is Accuracy. This follows many malware detection research papers and is very common practice for many deep learning applications. It is one of the major accepted performance criteria. To the best of our knowledge, no one has attempted to study the effects of different Deep learning hyper-parameters for malware detection.

Below, section 6.2 gives the research question investigated in this chapter. Section 6.3 details the experiment setup and gives preliminary observations. Section 6.4 compares the use of Grid Search and Covering Arrays provided by cAgen for NN hyper-parameter optimization tasks. Section 6.5 documents the results. Section 6.6 discusses issues raised and section 6.7 provides conclusions. Section 6.8 provides a summary of the chapter.

6.2 Research Question

In this chapter, we investigate the following research question:

RQ: Can Covering Arrays and Grid Search provide a highly efficient and effective means of hyper-parameter optimisation for machine learning-based Windows PE malware detectors?

6.3 Experiment Setup and Early Notes

6.3.1 Experiment Setup

DL models were based on Scikit-learn [138]. The experiments were carried out on an iMac (Retina 5K, 27-inch, 2017,4.2 GHz Quad-Core Intel Core i7,16 GB 2400 MHz DDR4 RAM, and a Radeon Pro 580 8 GB graphics card) and Windows OS 11, with 11 Gen Intel Core i7-11800H, with 2.30 GHz processor, and 16 GB RAM. Inputs were subject to normalisation via the scikit-learn library's StandardScaler function [149]. 3-fold validation was used throughout.

6.3.2 Baseline Model

We started with a baseline model in order to optimise it. Ember's model before optimisation had 12 neurons in the first layer, 8 neurons in the second layer and 1 in the last layer. It achieved 81.2% accuracy. In contrast, the Kaggle dataset model had 12 neurons in the first layer, 8 neurons in the second layer and 1 in the last layer. It achieved 94%. Given the established baselines, we started our approaches of optimising through different hyper-parameters.

6.4 Grid Search versus cAgen for NN Hyper-parameter Optimization Tasks

Below we explain the methods that we use for both cAgen and Grid Search.

6.4.1 Grid Search

The four main phases of Grid Search involve iterating over the set of defined values in the search space, incorporating the training set, cross-validating (here, using 3 folds) it during training, and finally, employing the validation set to output the best results in order to optimise performance with the optimal values. In Grid Search, that is expected behaviour.

6.4.2 cAgen

cAgen has its own parameters to be defined beforehand. First, we have to define a workspace to prepare our model. Then we choose the Input Parameter (IPM) to alter the parameters of our model. The last step is to set the value of t to specify the strength. The picture below shows the cAgen tool implementation details for NN. Due to the large size of the Ember dataset, we will only investigate covering arrays of strengths 1 and 2. In contrast, for the Kaggle dataset, we will investigate covering arrays of strengths 1, 2 and 3.

Input Parameter Model

Export IPM...

Name	Values	Cardinality
HIDDEN	0,1,2,3,4,5	6
OPTIMISERS	0,1	2
ACTIVATIONS	0,1,2,3,4,5,6,7	8
DROPOUT_PROBS	0,1,2,3,4,5,6,7,8,9	10
EPOCHS	0,1,2,3,4	5
BATCHES	0,1,2,3,4	5
LEARNING	0,1,2,3,4,5,6,7	8

+ Add Type Name

Constraints

Figure 6.1: cAgen Implementation Detail for both Ember and the Kaggle Datasets

6.4.3 Hyper-parameter Grid Search Configurations

We have selected the following hyper-parameters to assess their performance using Ember and Kaggle datasets. They are the Number of Epochs, Batch Size, Number of Neurons, Optimisation Method (either SGD or Adams), Dropout Relay, Type of Activation Function and Learning Rate (LR). We will examine the effects of different parameter choices and their performance. The following tables 6.1 and 6.2 show the hyper-parameters configurations for each DL model parameter.

Table 6.1: DL Model Grid Search Hyper-parameter Configurations (Ember Dataset)

Hyper-parameter	Grid Search Space	Best Hyper-parameter results
Number of Neurons(per layer)	[1200,1400,1800,2000,2200,2400]	2400,1200,1200,1
Number of Epochs	[20,40,60,80,100]	40
Batch Size	[16,32,48,64,80]	64
Optimiser	Adam or SGD	Adam
Drop out Relay	[0.0,0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9]	0.1
Learning_rate	[0.001, 0.01, 0.1, 0.11,0.12, 0.113,0.114,0.2]	0.114
Activation Function(per layer)	softmax, softplus, softsign, relu, tanh, sigmoid, hard_sigmoid, linear	hard_sigmoid , relu , relu , sigmoid

Table 6.2: DL Model Grid Search Configuration Space (Kaggle Dataset)

Hyper-parameter	Grid Search Space	Best Hyper-parameter results
Number of Neurons(per layer)	[55,60,65,70,75,80]	80,75,80,1
Batch Size	[48,64,80,100,128]	60
Number of Epochs	[20,40,60,80,100]	40
Optimiser	Adam or SGD	SGD
Drop out Relay	[0.0,0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9]	0.1
Learning_rate	[0.001, 0.01, 0.1, 0.11,0.12, 0.113,0.114,0.2]	0.2
Activation Functions(per layer)	softmax, softplus, softsign, relu, tanh, sigmoid, hard_sigmoid,linear	sigmoid,tanh,relu,sigmoid

6.5 Results

Below are the results from cAgen tool for both Kaggle 6.3 and Ember dataset 6.4

Table 6.3: DL Model cAgen Results Comparison (Kaggle Dataset)

ML Algorithm	Optimal Values Found	T-Strengths values/ Grid Search	Time to complete	Number of Combination searched	Score (Accuracy)
DL	75,150,1 SGD tanh,sigmoid,relu 0.0 20 128 0.0112	T2	7m 36s	60	0.9814
	75,150,1 Adam tanh,sigmoid,relu 0.0 20 80 0.0001	T3	49m 1s	361	0.9839
	75,37,1 SGD softsign,sigmoid,relu 0.1 20 80 0.0112	T4	3h 17min 14s	1452	0.9842
	75,80,80 Adam softsign,tanh,relu 0.2 60 80 0.0112	Full Grid Search	∼4 days 3hrs 6 m	all	**0.9862**

Table 6.4: DL Model cAgen Results Comparison (Ember Dataset)

ML Algorithm	Optimal Values Found	T-Strengths values/ Grid Search	Time to complete	Number of Combination searched	Score (Accuracy)
DL	2400,1200,1200 Adam softplus,relu,sigmoid 0.0 20 64 0.00001	T1	15 hrs 31min	10	0.9563
	1200,1200,1200 Adam sofplus,relu,sigmoid 0.5 40 128 0.0001	T2	4d 23h 5min 47s	60	**0.9575**
	2400,1200,1200 Adam hard_sigmoid,relu 0.1 40 64 0.0	Full Grid Search	∼29 days 15 hrs	all	0.9542

6.6 Discussion

Below we describe the effects of different hyper-parameter choices throughout our experiment.

6.6.1 Tuning the Number of Neurons

In [145], the authors state that number of neurons greatly affects the performance of a neural network. We have set up a Grid Search space with varying values from 1200 up to 2400 neurons. See table 6.1 and 6.2. The results showed a different level of performance starting from 81% (Kaggle 90%) accuracy of the models until it stagnated around 84.5% (Kaggle 92%). The best results were 2400 (Kaggle was 80) (which is slightly higher than the number of features we have in our Ember (2381) and Kaggle (75) features. That is an approximate 2 to 4 percent increase just from tuning the number of neurons with our model. The chosen activation function also had an impact.

6.6.2 Tuning the Number of Layers

Since we do not know what exact number of layers we should have, experimentation is all that we can do right now [146]. Both cAgen and Grid search aided in identifying the best results that we can gain through the specified search space. We started with 1 dense layer and 1 activation function (linear) to give the output. The accuracy of the model was not of concern at earlier stages, now we just wanted to make sure that it had a plausible result. We started iterating through our space with different varying results from 35% to 45% accuracy of the model, until we reached a high-performing configuration around one dense layer for the input with a vector size of 2381 (80 for Kaggle), a second dense layer with a victor size of 1200 (75 for Kaggle), a third dense layer with a vector size of 1200 (80 for Kaggle) and the last dense layer to return the binary output.

6.6.3 Tuning of Activation Functions

Activation functions are a crucial part of NN design. Different activation functions showed different results in our models. We have selected 8 activation functions for our problem: Softmax, Softplus, Softsign, Relu, Tanh, Sigmoid, Hard_sigmoid, Linear. We have set up a Grid Search using scikit-learn Grid Search CV. We noticed the following accuracy results across the selected functions for both datasets. The results are: Softplus (0.928%), Softmax (0.795%), Softsign (0.937%), Relu (0.926%), Tanh (0.932%), Sigmoid (0.939%), Hard_sigmoid (0.94%) and Linear (0.896%). This clearly shows how important choosing appropriate activation functions is to the performance of the model [150]. In the end, we chose 4 activation functions based upon our search results: for Ember we decided to use the Hard_sigmoid function in the first layer, Relu in the second layer, and Sigmoid in the third and fourth layers. Our experiment showed that the use of these different activation functions gave higher/better accuracy for our model. In contrast the activations used in the four layers for processing the Kaggle dataset were Sigmoid, Tanh, Relu and Sigmoid.

6.6.4 Tuning Dropout relay

Dropout relay is another function to prevent making the NN more complex by regularizing it. Basically, it is a way to approximate the model training within NN with different architectures in parallel. It is also a way to improve the model generalisability of the NN. This function aided in approximately adding a 1 to 3% increase to the model accuracy. Two dropout relay functions were experimented upon in our model within a range of values between 0.1 to 0.9. The best results were between 0.1 and 0.5; anything above that value would start to decrease the accuracy of our model. For example, once we specify the value for the function to be 0.6 the model accuracy would drop from 93% accuracy to approximately 88%. This shows how important dropout relay is in our model accuracy. This idea of adding a dropout relay in the hidden layers came from the following authors [143, 144]. One thing to add, it is very well established that the placement of dropout relays in the neural network might as well affect the performance of the NN.

6.6.5 Choice of Optimisers

The choice of selecting the specific optimiser is of importance as well. We decided to go with two optimisation techniques and compare the results from both SGD and Adam optimisers. However, different optimisers have different dependent parameters. There are a lot of optimisation techniques and each comes with its own issues. For example, with the SGD optimisation method, we have to pre-initialise the learning rate and experiment on it; because it is not easy to know which learning rate would work [148]. After experimenting for a while the results showed that both Adam and SGD would be a good choice for our model accuracy. The experiments with SGD had an impact on the learning rate (LR) hyper-parameter. The LR experiment focused on the exact value to provide for our model's optimal learning. So we set up a Grid Search space with values ranging from 0.0001 to 0.5. We noticed that values above 0.2 in the learning rate would sharply decrease our model accuracy score. For example, a learning rate with a 0.4 value would have an accuracy score of 65%. This is an indication that anything above a value of 0.2 would decrease the model score. For that, we selected the new search space for learning rate as indicated in 6.1. The model accuracy results provided by the Adam optimiser were affected by the number of epochs and batch size.

6.6.6 Tuning the Number of Epochs and Batch Size

The batch size is the sample before it is updated in the model (while training) and the number of epochs is the whole complete cycle through the entire training of the data set. The number of epochs can be said to be the budget that you would allow the model to be trained for. Both of them are important parameters that need to be set before the training. Again here there is no one rule for all, we have to mix and match until the model finds better values from the search. We have set up a Grid Search through which we have learned the best outcome for our specific problem. The results demonstrate that a higher number of epochs would start over-fitting the model (which in turn decreases the model accuracy). Also, a low batch size would decrease the accuracy of our model. So with both those in mind, we started experimenting with different values. Ember on the one hand we have a number of epochs with values from 20 to 100 (the best was 40). On the other hand, we have

batch sizes with values from 16 to 80 (the best was 64). Similarly, Kaggle had the number of epochs with values from 20 to 100 (the best was 40), however, batch sizes with values from 48 to 128 (the best was 60). Our findings approve the claim of [147], in which tuning of batch size as well as the number of epochs would surely affect the model performance [147]. Getting the desired set of values would go a long way in improving any NN model. Please refer to table 6.1 and 6.3 for epochs and batch size values.

6.7 Conclusion

The optimisation of hyper-parameters is fundamental to the design of Neural networks. There is no "one size fits all" choice. Several hyper-parameters were subject to experimentation to determine their effects on the model's efficiency. This experiment used both the cAgen tool and Grid Search. The cAgen tool shows great promise as a method for quickly searching for the best parameter settings for neural networks.

6.8 Summary

Hyper-parameter optimisation is vital to improving any neural network (NN) model. Several hyper-parameters were studied: the Number of Epochs, Batch Size, Number of Layers, Number of Neurons, Optimization Method, Dropout Relay, Type of Activation Function, and Learning Rate(LR). We used the cAgen tool and Grid Search optimisation from the scikit-learn Python library to find the best hyper-parameters of Keras deep learning models. The time and efficiency of cAgen are by far better than those of Grid Search in finding the best parameters. Finally, we showed that our hyper-parameter optimisation choices would significantly improve the performance of the NN model for static malware detection for Ember (from 81.2% to 95.7%) and for Kaggle (from 94% to 98.6%) in terms of accuracy.

Chapter 7

Conclusions and Future Work

This chapter summarises the motivation for the work and evaluates it. It also provides conclusions and identifies future work.

7.1 Context and Motivation

Chapters 1 and 2 provide the motivational context for the work presented in this thesis. The major components are summarised below:

- **Malware is a problem.** Malware is one of the biggest threats that Internet users (business owners, corporate organisations, hospitals, etc.) face today. Windows PE files are very common and essential to Windows OS. Their compromise, giving rise to PE malware, is a major security problem.
- **Issues for modern malware detection.** Detection must be effective, i.e. FPs and FNs must be minimised. Academia generally proceeds as though these were of equal importance. The anti-malware industry, however, places a much higher cost on FPs (considered unacceptable), than on FNs (considered a limitation). From that perspective, the goal of malware detectors is not to minimize both errors but to minimize FNs while keeping FPs at zero. However, mangement for specific system may make nuanced choices. Whatever trade-offs are to be made, the performance of any underlying ML-based approach will depend on the hyper-parameters chosen. Making high-performing hyper-parameter choices therefore matters. However, malware increasingly evades detection techniques, and detectors need to be updated or adapt as the threat landscape changes. Methods are needed to detect both seen malware and unseen malware. Traditionally, signature-based detectors handle seen malware well, but anomaly detection has a greater chance of detecting unseen malware. The scale of modern systems (e.g. the size of cloud storage) poses a major challenge for detection: detection must be efficient to allow its deployment in areas such as forensics or threat hunting where vast file storage may need to be scanned for malware. Overall, there is a pressing need for fast and re-trainable malware classifiers. Furthermore, it is beneficial to detect malware before execution, i.e. static detection is generally preferred.

- **Potential for Machine Learning.** Machine Learning (ML) has significant potential for improving static malware detection: it can play a crucial role in extracting insight from malware samples; ML-based detectors can detect some previously unseen malware, such as previously unseen malware with properties similar to those of recognised malware; a huge array of machine learning-based classification algorithms can be brought to bear on the malware detection problem; and an ML approach can also significantly reduce the manual effort involved in developing detectors, giving more rapid deployment.

- **Focus and Extrapolation.** Here, Windows PE files are a means to an end; the same issues apply to detecting other malware. Although malware is our major interest, our work also seeks to motivate consideration of HPO and the use of state-of-the-art approaches, in particular, more widely in the application of ML in cybersecurity.

- **The problem of hyper-parameter optimisation.** This is addressed in chapter 2. Many ML techniques are parameterised, and the choice of parameters may significantly affect performance. In modern, widely used ML tool-kits, algorithms often have many tens of parameters (and sometimes more). This leads to the thorny issue of how such parameters may be best set; a problem generally referred to as hyper-parameter optimisation (HPO).

- **What HPO can do for ML-based malware detection.** Suitable HPO has the potential to improve on the results obtained by a specific detection approach but also to enable fair comparison of techniques with techniques that are not the specific focus of researchers' investigation. (Comparisons are often made between their new technique (for which significant effort may have been expended on its optimisation) and 'vanilla' (unoptimised) variants of extant techniques.) Manual tuning is often simply impossible. (As an aside, we observe that many commercial ML users spend a great deal of time tuning for their specific needs).

- **What currently happens with HPO?** Existing ML toolkits address this problem to some extent by adopting *default values* for parameters; these values have been shown to work plausibly over many problems. However, for any specific problem, it is far from clear that the default values will be the best, or even good, choices. We have significant domain incentives to gain the best possible results for malware detection. Where HPO is adopted (to some degree) it is rarely done systematically. This is indicated in chapter 2.

- **The computational cost of HPO.** Although HPO has a great deal to offer, it comes at a computational price. We must train the model for every hyper-parameter evaluation, make predictions on the validation set, and then calculate the validation metrics. Developing a robust ML-based classifier for Windows PE with a credibly sized and diverse dataset such as Ember is a significant computational undertaking.

- **The need for efficient hyper-parameter exploration.** The computational costs involved are a disincentive to implementing Bergstra et al.'s 'formal outer loop'. There is a pressing need for traversing the hyper-parameter space efficiently to deliver high-performing hyper-parameter choices.

7.2 Recap: the Research Hypotheses

Our main research hypothesis is:

- ***Main Hypothesis**:* HPO can significantly improve the performance of static malware detectors based on machine learning.

The detailed research hypotheses are:

- ***Hypothesis1**:*AHBO-TPE is an efficient and effective technique for hyper-parameter optimisation of ML-based malware detectors. It can find high-performance hyper-parameter vectors more quickly than comparable techniques.
- ***Hypothesis2**:*A Covering Array is an efficient technique for hyper-parameter optimisation of ML-based malware detectors. It can find significantly better hyper-parameters in comparison to Grid Search and do so with reduced computation.
- ***Hypothesis3**:*An Evolutionary Algorithm can find new or improved hyper-parameter vectors that give better performance than the defaults of ML-based malware detectors.
- ***Hypothesis4**:*Grid Search and Covering Arrays can be used to efficiently achieve high-performing parameter choices for Deep Neural Network based malware detectors.

The first three hypotheses are evaluated using various underpinning ML classification approaches. The fourth hypothesis extends our investigation into HPO for optimising the model parameters of neural networks. All assume that the context is static malware detection of Windows PE files.

7.3 Evaluating the Evidence for the Hypotheses

The first hypothesis was identified in chapter 1.

- AHBO-TPE is an efficient and effective technique for hyper-parameter optimisation of ML-based malware detectors. It can find high-performance hyper-parameter vectors more quickly than comparable techniques.

We show that a specific technique in chapter 3 is highly promising and that HPO still significantly affects its malware detection performance. We argue that HPO should play an important part in ML-based malware detection research and development and in security applications more widely. Using HPO to provide Bergstra et al.'s 'formal outer loop' should be normal practice to ensure any targeted technique is exploited fully. Adopting HPO in this way brings methodological benefits. For the development of the field, we need to be able to compare competing techniques at their best, and HPO can provide a principled and repeatable way to get the best (or close to it) from all competing techniques.

- A Covering Array is an efficient technique for hyper-parameter optimisation of ML-based malware detectors. It can find significantly better hyper-parameters in comparison to Grid Search and do so with reduced computation.

We have shown that cAgen in chapter 4 (a combinatorial testing tool) is an efficient method to find optimal parameters that increase the model performance with less iteration and time. There is a huge difference in the time taken to finish iterations between our Covering Array (with cAgen) approach and Grid Search. cAgen outperformed Grid Search in all four ML models and produced better parameter values. The results show significant promise for adding Covering Arrays, especially cAgen to the ML hyper-parameter optimisation community, malware detectors community, and the security testing community.

- An Evolutionary Algorithm can find new or improved hyper-parameter vectors that give better performance than the defaults of ML-based malware detectors.

Here, we show that Deap, in chapter 5, is a promising instrument for optimising the six ML models of choice. The Deap experiment provided us with good parameter choices, superior to those provided by TPOT.

- Grid Search and Covering Arrays can be used to efficiently achieve high-performing parameter choices for Deep Neural Network-based malware detectors.

We demonstrate the improvements in chapter 6 that systematic hyper-parameter search can bring to achieve better performance, working from highly plausible baseline models for comparison. We demonstrate that using covering arrays generated by the cAgen tool is a highly efficient means to find excellent parameter values with less time and fewer iterations than with Grid Search.

7.4 Limitations and Future Work

We now consider limitations, means of addressing them, and other future work.

7.4.1 Dataset Issues

In our experiments, we have made use of two high-profile datasets: Ember2018 and a dataset from kaggle.com (created in 2014). Inevitably, these datasets may suffer from biases, e.g. spatial and temporal biases. **Spatial Bias** is introduced by the distribution of samples across different malware families. In both datasets, the malicious samples are drawn from diverse families, but some families might be over-represented while others are under-represented. This could affect the performance of machine learning models trained on this dataset, as they may become biased towards more prevalent families and perform poorly on underrepresented families. **Temporal bias** is introduced during the dataset's creation period. Malware evolves over time, and a dataset created at a specific time may not represent current malware trends. The Ember 2018 dataset was created using samples from 2012 to

2018, so it may not capture more recent developments in malware. This also applies to the Kaggle dataset since it is made with samples from 2012-2014.

Any research that seeks effective malware detection in a contemporary setting must avail itself of datasets that facilitate that task. Training and evaluating on out-of-date or otherwise narrowly defined datasets will not serve such research well.

- **Widening the range of datasets used.** In our work, our primary goal was to show how, given a dataset, ML-model hyper-parameters could be found by HPO techniques to maximise performance (measured in some plausible way). However, the nature of the datasets could affect overall results and we adopted only two datasets in our work. Evaluation over a much wider set of datasets, e.g. over non-Windows PE datasets, more contemporary datasets, or datasets with a more extended timeframe (and so with a wider range of malware present) would seem a plausible goal for further research. We can also use techniques such as oversampling or undersampling to balance the representation of different families of malware in the datasets used.

7.4.2 Other Limitations and Future Work

We have identified various further limitations and avenues of future work, as indicated below.

- **Widening the Scope of Informed Approaches.** We have used in chapter 3 a single (albeit highly effective) 'informed' hyper-parameterisation approach. The use of other informed hyper-parametrisation approaches could provide further insight and possible improvements.

- **Varying the ML Framework and its Elements.** We have embraced supervised learning in our work. But there is a clear role for unsupervised approaches and semi-supervised approaches. Furthermore, our ML model pipelines have generally been quite basic. More sophisticated elements, such as dimensionality reduction, importance sampling, and other feature engineering practices, could be beneficially adopted.

- **Alternative Evaluation Criteria.** We have been limited in the evaluation criteria we have used, e.g. accuracy and *ROC_AUC* were chosen largely for comparison with extant research results. Varying the sought balance between FPs and FNs should now be addressed. In particular, the anti-malware industry's preference for the lowest number of FNs consistent with zero FPs should be recognised. Operationally, this would usually mean that FPs are punished very highly, e.g. in an extreme form of F_β scoring. (In much academic research, the F_1 score is used, reflecting equal weighting.) It is, however, far from clear that current approaches would be effective for extremal cases. Strictly speaking, this would be problematic for the underlying ML technique, but there may indeed be induced difficulties with hyper-parameterisation too. For example, if the underlying loss function landscape becomes highly erratic (highly discontinuous), this might also induce a highly erratic hyper-parametrisation landscape.

- **Embracing Further Parameter Subsets.** For computational complexity reasons, we have had to make choices as to which parameters were subject to variation and which were left as defaults

in our experiments. However, there remains the possibility of adopting a sequential approach to optimising over the full range of parameters, i.e., once investigated parameters have been subject to variation and evaluation, they could be fixed at their optimal values and previously fixed or defaulted parameters then be allowed to vary. Thus, one subset of parameters would be allowed to vary (i.e. be subject to HPO techniques) at any one time. This is related to what is often called One-Factor-At-a-Time (OFAT) optimisation (with subset replacing factor).

- **Progressive Hyper-parameter Range Refinement.** Our HPO experiments assumed fixed hyper-parameter ranges or discretisations. But optimal choices may be under- or over-approximated in such set-ups. For example, a discrete set [0, 0.2, 0.4, 0.6, 0.8, 1.0] may be defined to 'cover' the continuous range [0,1], but optimal results might actually be obtained with a value of, say, 0.3, which is only approximately represented in our discretised set. However, it would seem possible to run HPO on our discrete domain and then redefine the domain to be investigated. For example, if a value of 0.4 gave rise to the best results, then a domain centred on that value might be chosen for a subsequent run of HPO, e.g. we might adopt a new domain of [0.3, 0.35, 0.4, 0.45, 0.5]. For a categorical data range [A, B, C, D, E] a reduced range might be chosen comprising the top two performing categorical data values in the first stage application of HPO, e.g. [A,D]. The same HPO approaches could be applied using the revised parameter ranges.

- **Generalising Choices for Covering Arrays.** The work in chapter 4 made specific choices regarding the covering arrays. We need not be so restricted. The cAgen toolkit provides the means for further experimental flexibility: a variety of approaches for generating arrays, e.g. FIPOG-F and FIPOG-2F); allowing constraints to be imposed; and allowing higher t-values (strengths) than adopted in this thesis.

- **Using Proxy Evaluation Functions.** Evaluating a model over a large dataset is computationally very expensive (as we have found in this thesis). This means that practical compromises have to be found, e.g. the adoption of coarser granularity of discretisation (with increased chances of not encompassing truly optimal values of parameters). However, if the underlying evaluation landscape can be modelled and allow efficient evaluation, this could be used to search for approximations to optimal parameter values. In a sense, Bayesian approaches use a form of this, but the modelling approach used is very specific. An alternative is to use neural networks as *function approximators*. That is, use the results of real hyper-parameter evaluations to train a neural network to act as a predictor function (predicting the evaluation result when supplied with a vector of hyper-parameter values). The search could revert to using the real evaluation function starting from the best vector of hyper-parameters obtained by hyper-parameter search using the neural network as an evaluation function.

7.5 Conclusions

From the work carried out in this thesis we conclude:

- Systematic HPO is rarely performed in ML-based malware detection research generally. This is also the case for the Windows PE detection.
- HPO has great potential to improve the results of applying ML approaches to PE malware detection. We see no obvious reason why other types of malware detection would not benefit similarly.
- Application of HPO is essential if we are to carry out principled research. Widespread adoption of HPO would lead to optimised evaluations of techniques and so make for fairer comparisons.
- Computational efficiency is a major challenge for HPO. We have shown that the usefulness of a variety of approaches for improving efficiency. Bayesian approaches are worthy of further investigation. Design of experiments approaches have excellent potential; in particular, our adoption of covering arrays seems very promising. A huge range of evolutionary approaches can also be brought to bear on the problem.
- Our investigation has demonstrated the usefulness of HPO approaches and has shown excellent results. Above we have identified numerous limitations, many of which are not intrinsic, i.e. they simply suggest avenues for improving our results.
- Overall, there are many approaches that can be taken for HPO. The ML-based malware detection community should further investigate them!

7.6 Acknowledgments of the Use of Freely Available Software

In our investigations, we have made significant use of freely available software toolkits. We would like to express our thanks to the developers of these toolkits. Specifically, we acknowledge the use of the following:

- Scikit-learn. This is a well-known and robust machine learning library with a large number of algorithms and tools for ML visualizations, preprocessing, model fitting, selection, and assessment [138]. This is used throughout the thesis.
- AHBO-TPE (Bayesian Hyperparameter optimisation using Tree Parzen Estimators). This forms part of the Hyperopt Python Library [78].
- cAgen. This is a high-performance t-way test generation tool[83] which we use (in chapter 4 and chapter 6) to generate covering arrays.
- TPOT (Tree-based Pipeline Optimization Tool). This is a Python-based automated machine learning tool that uses genetic programming to improve machine learning pipelines [111]. It is used in chapter 5.
- DEAP (Distributed Evolutionary Algorithms in Python). This is an open-source Python library designed for the rapid prototyping of evolutionary algorithms, including genetic algorithms (GA) [109]. It is used in chapter 5.

7.7 And Finally

As far as we are aware, this is the first thesis to focus on HPO in the context of ML-based malware detection. The area of HPO seems ripe for exploitation by the malware detection community and we recommend this area to them for future research.

Bibliography

[1] corner freecode. Executablefile. https://sites.google.com/site/freecodecorner/technologies/process-info/executable-file, June 2022.

[2] Alexios Koutsoukas, Keith J Monaghan, Xiaoli Li, and Jun Huan. Deep-learning: investigating deep neural networks hyper-parameters and comparison of performance to shallow methods for modeling bioactivity data. *Journal of cheminformatics*, 9(1):1–13, 2017.

[3] Af2. https://towardsdatascience.com/deep-study-of-a-not-very-deep-neural-network-part-2-a 2022.

[4] Af1. https://www.v7labs.com/blog/neural-networks-activation-functions#:~:text=The%20linear%20activation%20function%2C%20also,the%20value%20it%20was%20given., 2022.

[5] Li Yang and Abdallah Shami. On hyperparameter optimization of machine learning algorithms: Theory and practice. *Neurocomputing*, 415:295–316, 2020.

[6] Hyrum S Anderson and Phil Roth. Ember: an open dataset for training static pe malware machine learning models. *arXiv preprint arXiv:1804.04637*, 2018.

[7] E Carrera. pefile. https://github.com/erocarrera/pefile, 2022. Accessed : 2022-01-15.

[8] Abhishek Kumar Pandey, Ashutosh Kumar Tripathi, Gayatri Kapil, Virendra Singh, Mohd Waris Khan, Alka Agrawal, Rajeev Kumar, and Raees Ahmad Khan. Trends in malware attacks: Identification and mitigation strategies. In *Critical Concepts, Standards, and Techniques in Cyber Forensics*, pages 47–60. IGI Global, 2020.

[9] Weijie Han, Jingfeng Xue, Yong Wang, Lu Huang, Zixiao Kong, and Limin Mao. Maldae: Detecting and explaining malware based on correlation and fusion of static and dynamic characteristics. *Computers & Security*, 83:208–233, 2019.

[10] Pete Burnap, Richard French, Frederick Turner, and Kevin Jones. Malware classification using self organising feature maps and machine activity data. *computers & security*, 73:399–410, 2018.

[11] Aiman Al-Sabaawi, Khamael Al-Dulaimi, Ernest Foo, and Mamoun Alazab. Addressing malware attacks on connected and autonomous vehicles: Recent techniques and challenges. In *Malware Analysis Using Artificial Intelligence and Deep Learning*, pages 97–119. Springer, 2021.

[12] Mauricio. Benign malicious. `https://www.kaggle.com/amauricio/pe-files-malwares`, 2021. Accessed: 2021-11-10.

[13] H S Anderson and P Roth. elastic/ember. `https://github.com/elastic/ember/blob/master/README.md`, Feb 2021.

[14] Xufang Li, Peter K.K. Loh, and Freddy Tan. Mechanisms of polymorphic and metamorphic viruses. In *2011 European Intelligence and Security Informatics Conference*, pages 149–154, 2011.

[15] Philip O'Kane, Sakir Sezer, and Kieran McLaughlin. Obfuscation: The hidden malware. *IEEE Security & Privacy*, 9(5):41–47, 2011.

[16] M. Weber, M. Schmid, M. Schatz, and D. Geyer. A toolkit for detecting and analyzing malicious software. In *18th Annual Computer Security Applications Conference, 2002. Proceedings.*, pages 423–431, 2002.

[17] Matt Pietrek. An in-depth look into the win32 portable executable file format, part 2. *MSDN Magazine, March*, 2002.

[18] M Zubair Shafiq, S Momina Tabish, Fauzan Mirza, and Muddassar Farooq. Pe-miner: Mining structural information to detect malicious executables in realtime. In *International workshop on recent advances in intrusion detection*, pages 121–141. Springer, 2009.

[19] Murray Brand, Craig Valli, and Andrew Woodward. Malware forensics: Discovery of the intent of deception. *Journal of Digital Forensics, Security and Law*, 5(4):2, 2010.

[20] Cokrami. cokrami/docs. `https://github.com/corkami/docs/blob/master/PE/PE.md`, July 2022.

[21] Claudio Gambella, Bissan Ghaddar, and Joe Naoum-Sawaya. Optimization problems for machine learning: A survey. *European Journal of Operational Research*, 290(3):807–828, 2021.

[22] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *the Journal of machine Learning research*, 12:2825–2830, 2011.

[23] David W Hosmer Jr, Stanley Lemeshow, and Rodney X Sturdivant. *Applied logistic regression*, volume 398. John Wiley & Sons, 2013.

[24] Joseph O Ogutu, Torben Schulz-Streeck, and Hans-Peter Piepho. Genomic selection using regularized linear regression models: ridge regression, lasso, elastic net and their extensions. In *BMC proceedings*, volume 6, pages 1–6. Springer, 2012.

[25] Wangmeng Zuo, David Zhang, and Kuanquan Wang. On kernel difference-weighted k-nearest neighbor classification. *Pattern Analysis and Applications*, 11(3):247–257, 2008.

[26] Irina Rish et al. An empirical study of the naive bayes classifier. In *IJCAI 2001 workshop on empirical methods in artificial intelligence*, volume 3, pages 41–46, 2001.

[27] Ashraf M Kibriya, Eibe Frank, Bernhard Pfahringer, and Geoffrey Holmes. Multinomial naive bayes for text categorization revisited. In *Australasian Joint Conference on Artificial Intelligence*, pages 488–499. Springer, 2004.

[28] Carlos Bustamante, Leonardo Garrido, and Rogelio Soto. Comparing fuzzy naive bayes and gaussian naive bayes for decision making in robocup 3d. In *Mexican International Conference on Artificial Intelligence*, pages 237–247. Springer, 2006.

[29] Jason DM RENNIE. Tackling the poor assumptions of naive bayes text classification. machine learning. *ICML-2003, Washington DC*, 2003.

[30] Xiangyu Duan, Jun Zhao, and Bo Xu. Probabilistic models for action-based chinese dependency parsing. In *European Conference on Machine Learning*, pages 559–566. Springer, 2007.

[31] S Rasoul Safavian and David Landgrebe. A survey of decision tree classifier methodology. *IEEE transactions on systems, man, and cybernetics*, 21(3):660–674, 1991.

[32] Dimitrios Michael Manias, Manar Jammal, Hassan Hawilo, Abdallah Shami, Parisa Heidari, Adel Larabi, and Richard Brunner. Machine learning for performance-aware virtual network function placement. In *2019 IEEE Global Communications Conference (GLOBECOM)*, pages 1–6. IEEE, 2019.

[33] Li Yang, Abdallah Moubayed, Ismail Hamieh, and Abdallah Shami. Tree-based intelligent intrusion detection system in internet of vehicles. In *2019 IEEE global communications conference (GLOBECOM)*, pages 1–6. IEEE, 2019.

[34] Samantha Sanders and Christophe Giraud-Carrier. Informing the use of hyperparameter optimization through metalearning. In *2017 IEEE International Conference on Data Mining (ICDM)*, pages 1051–1056. IEEE, 2017.

[35] MohammadNoor Injadat, Fadi Salo, Ali Bou Nassif, Aleksander Essex, and Abdallah Shami. Bayesian optimization with machine learning algorithms towards anomaly detection. In *2018 IEEE global communications conference (GLOBECOM)*, pages 1–6. IEEE, 2018.

[36] Fadi Salo, MohammadNoor Injadat, Abdallah Moubayed, Ali Bou Nassif, and Aleksander Essex. Clustering enabled classification using ensemble feature selection for intrusion detection. In *2019 International Conference on Computing, Networking and Communications (ICNC)*, pages 276–281. IEEE, 2019.

[37] Kamatchi Arjunan and Chirag N Modi. An enhanced intrusion detection framework for securing network layer of cloud computing. In *2017 ISEA Asia Security and Privacy (ISEASP)*, pages 1–10. IEEE, 2017.

[38] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794, 2016.

[39] Lightgbm documentation. `https://lightgbm.readthedocs.io/en/latest`, 2021. Accessed : 2021-8-20.

[40] Wenpeng Yin, Katharina Kann, Mo Yu, and Hinrich Schütze. Comparative study of cnn and rnn for natural language processing. *arXiv preprint arXiv:1702.01923*, 2017.

[41] Eric W Bell and Yang Zhang. Dockrmsd: an open-source tool for atom mapping and rmsd calculation of symmetric molecules through graph isomorphism. *Journal of Cheminformatics*, 11(1):1–9, 2019.

[42] Tobias Domhan, Jost Tobias Springenberg, and Frank Hutter. Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves. In *Twenty-fourth international joint conference on artificial intelligence*, 2015.

[43] Yoshihiko Ozaki, Masaki Yano, and Masaki Onishi. Effective hyperparameter optimization using nelder-mead method in deep learning. *IPSJ Transactions on Computer Vision and Applications*, 9(1):1–12, 2017.

[44] Foo Chong Soon, Hui Ying Khaw, Joon Huang Chuah, and Jeevan Kanesan. Hyper-parameters optimisation of deep cnn architecture for vehicle logo recognition. *IET Intelligent Transport Systems*, 12(8):939–946, 2018.

[45] Hugo Larochelle, Yoshua Bengio, Jérôme Louradour, and Pascal Lamblin. Exploring strategies for training deep neural networks. *Journal of machine learning research*, 10(1), 2009.

[46] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[47] Christian Darken, Joseph Chang, John Moody, et al. Learning rate schedules for faster stochastic gradient search. In *Neural networks for signal processing*, volume 2. Citeseer, 1992.

[48] Matthew G Schultz, Eleazar Eskin, F Zadok, and Salvatore J Stolfo. Data mining methods for detection of new malicious executables. In *Proceedings 2001 IEEE Symposium on Security and Privacy. S&P 2001*, pages 38–49. IEEE, 2000.

[49] J Zico Kolter and Marcus A Maloof. Learning to detect and classify malicious executables in the wild. *Journal of Machine Learning Research*, 7(12), 2006.

[50] Edward Raff, Jon Barker, Jared Sylvester, Robert Brandon, Bryan Catanzaro, and Charles K Nicholas. Malware detection by eating a whole exe. In *Workshops at the Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[51] Huu-Danh Pham, Tuan Dinh Le, and Thanh Nguyen Vu. Static pe malware detection using gradient boosting decision trees algorithm. In *International Conference on Future Data and Security Engineering*, pages 228–236. Springer, 2018.

[52] Chris Fawcett and Holger H Hoos. Analysing differences between algorithm configurations through ablation. *Journal of Heuristics*, 22(4):431–458, 2016.

[53] KARTIK MALIK, MANISH KUMAR, MEHUL KUMAR SONY, RADHA MUKHRAIYA, PALAK GIRDHAR, and BHARTI SHARMA. Static malware detection and analysis using machine learning methods. 2022.

[54] Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P Adams, and Nando De Freitas. Taking the human out of the loop: A review of bayesian optimization. *Proceedings of the IEEE*, 104(1):148–175, 2015.

[55] Matthias Feurer and Frank Hutter. Hyperparameter optimization. In *Automated Machine Learning*, pages 3–33. Springer, Cham, 2019.

[56] Bernd Bischl, Olaf Mersmann, Heike Trautmann, and Claus Weihs. Resampling methods for meta-model validation with recommendations for evolutionary computation. *Evolutionary computation*, 20(2):249–275, 2012.

[57] Chris Thornton, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Auto-weka: Combined selection and hyperparameter optimization of classification algorithms. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 847–855, 2013.

[58] Aaron Klein, Stefan Falkner, Simon Bartels, Philipp Hennig, Frank Hutter, et al. Fast bayesian hyperparameter optimization on large datasets. *Electronic Journal of Statistics*, 11(2):4945–4968, 2017.

[59] Oden Maron and Andrew W Moore. The racing algorithm: Model selection for lazy learners. *Artificial Intelligence Review*, 11(1):193–225, 1997.

[60] Gang Luo. A review of automatic selection methods for machine learning algorithms and hyper-parameter values. *Network Modeling Analysis in Health Informatics and Bioinformatics*, 5(1):1–16, 2016.

[61] Evan R Sparks, Ameet Talwalkar, Daniel Haas, Michael J Franklin, Michael I Jordan, and Tim Kraska. Automating model search for large scale machine learning. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pages 368–380, 2015.

[62] Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren. *Automated machine learning: methods, systems, challenges*. Springer Nature, 2019.

[63] Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *International conference on learning and intelligent optimization*, pages 507–523. Springer, 2011.

[64] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. *Advances in neural information processing systems*, 24, 2011.

[65] Anupriya Gogna and Akash Tayal. Metaheuristics: review and application. *Journal of Experimental & Theoretical Artificial Intelligence*, 25(4):503–526, 2013.

[66] Yoshua Bengio. Gradient-based optimization of hyperparameters. *Neural computation*, 12(8):1889–1900, 2000.

[67] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. *arXiv preprint arXiv:1206.2944*, 2012.

[68] R Bellman. Dynamic programming princeton university press princeton. *New Jersey Google Scholar*, 1957.

[69] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of machine learning research*, 13(2), 2012.

[70] Frank Hutter, Holger Hoos, and Kevin Leyton-Brown. An efficient approach for assessing hyperparameter importance. In *International conference on machine learning*, pages 754–762. PMLR, 2014.

[71] Frank Hutter, Holger Hoos, and Kevin Leyton-Brown. An evaluation of sequential model-based optimization for expensive blackbox functions. In *Proceedings of the 15th annual conference companion on Genetic and evolutionary computation*, pages 1209–1216, 2013.

[72] James Bergstra, Daniel Yamins, and David Cox. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. In *International conference on machine learning*, pages 115–123. PMLR, 2013.

[73] Stefan Falkner, Aaron Klein, and Frank Hutter. Bohb: Robust and efficient hyperparameter optimization at scale. In *International Conference on Machine Learning*, pages 1437–1446. PMLR, 2018.

[74] Eric Brochu, Vlad M Cora, and Nando De Freitas. A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. *arXiv preprint arXiv:1012.2599*, 2010.

[75] I Dewancker, M McCourt, and S Clark. Bayesian optimization primer. https://static.sigopt.com/b/20a144d208ef255d3b981ce419667ec25d8412e2/static/pdf/SigOpt_Bayesian_Optimization_Primer.pdf, 2015.

[76] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Tobias Springenberg, Manuel Blum, and Frank Hutter. Auto-sklearn: efficient and robust automated machine learning. In *Automated Machine Learning*, pages 113–134. Springer, Cham, 2019.

[77] R Jones Donald. Efficient global optimization of expensive black-box function. *J. Global Optim.*, 13:455–492, 1998.

[78] James Bergstra, Brent Komer, Chris Eliasmith, Dan Yamins, and David D Cox. Hyperopt: a python library for model selection and hyperparameter optimization. *Computational Science & Discovery*, 8(1):014008, 2015.

[79] Katharina Eggensperger, Matthias Feurer, Frank Hutter, James Bergstra, Jasper Snoek, Holger Hoos, and Kevin Leyton-Brown. Towards an empirical foundation for assessing bayesian optimization of hyperparameters. In *NIPS workshop on Bayesian Optimization in Theory and Practice*, volume 10, page 3, 2013.

[80] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[81] Zohar Karnin, Tomer Koren, and Oren Somekh. Almost optimal exploration in multi-armed bandits. In *International Conference on Machine Learning*, pages 1238–1246. PMLR, 2013.

[82] MohammadNoor Injadat, Abdallah Moubayed, Ali Bou Nassif, and Abdallah Shami. Systematic ensemble model selection approach for educational data mining. *Knowledge-Based Systems*, 200:105992, 2020.

[83] (MaTRIS) Research Group. Covering array generation. https://matris.sba-research.org/tools/cagen/#/about, 2022. Accessed: 21 July 2022.

[84] Marc Claesen, Jaak Simm, Dusan Popovic, Yves Moreau, and Bart De Moor. Easy hyperparameter search using optunity. *arXiv preprint arXiv:1412.1114*, 2014.

[85] Mohammad Galety, Firas Husham Al Mukthar, Rebaz Jamal Maaroof, and Fanar Rofoo. Deep neural network concepts for classification using convolutional neural network: A systematic review and evaluation. 2021.

[86] Quanming Yao, Mengshuo Wang, Yuqiang Chen, Wenyuan Dai, Yu-Feng Li, Wei-Wei Tu, Qiang Yang, and Yang Yu. Taking human out of learning applications: A survey on automated machine learning. *arXiv preprint arXiv:1810.13306*, 2018.

[87] Ron Kohavi and George H John. Automatic parameter selection by minimizing estimated error. In *Machine Learning Proceedings 1995*, pages 304–312. Elsevier, 1995.

[88] Stefan Lessmann, Robert Stahlbock, and Sven F Crone. Optimizing hyperparameters of support vector machines by genetic algorithms. In *IC-AI*, pages 74–82, 2005.

[89] Pablo Ribalta Lorenzo, Jakub Nalepa, Michal Kawulok, Luciano Sanchez Ramos, and José Ranilla Pastor. Particle swarm optimization for hyper-parameter selection in deep neural networks. In *Proceedings of the genetic and evolutionary computation conference*, pages 481–488, 2017.

[90] Fernando Itano, Miguel Angelo de Abreu de Sousa, and Emilio Del-Moral-Hernandez. Extending mlp ann hyper-parameters optimization by using genetic algorithm. In *2018 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2018.

[91] Radwa Elshawi, Mohamed Maher, and Sherif Sakr. Automated machine learning: State-of-the-art and open challenges. *arXiv preprint arXiv:1906.02287*, 2019.

[92] Shahryar Rahnamayan, Hamid R Tizhoosh, and Magdy MA Salama. A novel population initialization method for accelerating evolutionary algorithms. *Computers & Mathematics with Applications*, 53(10):1605–1614, 2007.

[93] Borhan Kazimipour, Xiaodong Li, and A Kai Qin. A review of population initialization techniques for evolutionary algorithms. In *2014 IEEE Congress on Evolutionary Computation (CEC)*, pages 2585–2592. IEEE, 2014.

[94] Marc Claesen and Bart De Moor. Hyperparameter search in machine learning. *arXiv preprint arXiv:1502.02127*, 2015.

[95] Yuhui Shi and Russell C Eberhart. Parameter selection in particle swarm optimization. In *International conference on evolutionary programming*, pages 591–600. Springer, 1998.

[96] Min-Yuan Cheng, Kuo-Yu Huang, and Merciawati Hutomo. Multiobjective dynamic-guiding pso for optimizing work shift schedules. *Journal of Construction Engineering and Management*, 144(9):04018089, 2018.

[97] François-Michel De Rainville, Félix-Antoine Fortin, Marc-André Gardner, Marc Parizeau, and Christian Gagné. Deap: A python framework for evolutionary algorithms. In *Proceedings of the 14th annual conference companion on Genetic and evolutionary computation*, pages 85–92, 2012.

[98] Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner Gardner, Marc Parizeau, and Christian Gagné. Deap: Evolutionary algorithms made easy. *The Journal of Machine Learning Research*, 13(1):2171–2175, 2012.

[99] deap. `https://github.com/DEAP/deap`, 2021.

[100] Yann Collette, Nikolaus Hansen, Gilles Pujol, Daniel Salazar Aponte, and Rodolphe Le Riche. Object-oriented programming of optimizers–examples in scilab. *Multidisciplinary Design Optimization in Computational Mechanics*, pages 499–538, 2013.

[101] Tim Ribaric and Sheridan Houghten. Genetic programming for improved cryptanalysis of elliptic curve cryptosystems. In *2017 IEEE Congress on Evolutionary Computation (CEC)*, pages 419–426. IEEE, 2017.

[102] Serge Chardon, Boris Brangeon, Emmanuel Bozonnet, and Christian Inard. Construction cost and energy performance of single family houses: From integrated design to automated optimization. *Automation in Construction*, 70:1–13, 2016.

[103] Randal S Olson, Ryan J Urbanowicz, Peter C Andrews, Nicole A Lavender, Jason H Moore, et al. Automating biomedical data science through tree-based pipeline optimization. In *European conference on the applications of evolutionary computation*, pages 123–137. Springer, 2016.

[104] Matthieu Michel Jean Macret. *Automatic tuning of the OP-1 synthesizer using a multi-objective genetic algorithm*. PhD thesis, Communication, Art & Technology: School of Interactive Arts and Technology, 2013.

[105] Félix-Antoine Fortin, Simon Grenier, and Marc Parizeau. Generalizing the improved run-time complexity algorithm for non-dominated sorting. In *Proceedings of the 15th annual conference on Genetic and evolutionary computation*, pages 615–622, 2013.

[106] Vahab Akbarzadeh, Albert Hung-Ren Ko, Christian Gagné, and Marc Parizeau. Topography-aware sensor deployment optimization with cma-es. In *International Conference on Parallel Problem Solving from Nature*, pages 141–150. Springer, 2010.

[107] Randal S Olson, Nathan Bartley, Ryan J Urbanowicz, and Jason H Moore. Evaluation of a tree-based pipeline optimization tool for automating data science. In *Proceedings of the genetic and evolutionary computation conference 2016*, pages 485–492, 2016.

[108] R Olson. Using tpot. `http://epistasislab.github.io/tpot/using/`, 2021.

[109] rsteca. sklearn-deap. `https://github.com/rsteca/sklearn-deap/blob/master/evolutionary_search/cv.py`, 2021.

[110] Trang T Le, Weixuan Fu, and Jason H Moore. Scaling tree-based automated machine learning to biomedical big data with a feature set selector. *Bioinformatics*, 36(1):250–256, 2020.

[111] Randal S Olson and Jason H Moore. Tpot: A tree-based pipeline optimization tool for automating machine learning. In *Workshop on automatic machine learning*, pages 66–74. PMLR, 2016.

[112] Rafael G Mantovani, Tomáš Horváth, Ricardo Cerri, Joaquin Vanschoren, and André CPLF de Carvalho. Hyper-parameter tuning of a decision tree induction algorithm. In *2016 5th Brazilian Conference on Intelligent Systems (BRACIS)*, pages 37–42. IEEE, 2016.

[113] Jan N Van Rijn and Frank Hutter. Hyperparameter importance across datasets. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2367–2376, 2018.

[114] André Biedenkapp, Marius Lindauer, Katharina Eggensperger, Frank Hutter, Chris Fawcett, and Holger Hoos. Efficient parameter importance analysis via ablation with surrogates. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 31, 2017.

[115] Katharina Eggensperger, Marius Lindauer, Holger H Hoos, Frank Hutter, and Kevin Leyton-Brown. Efficient benchmarking of algorithm configurators via model-based surrogates. *Machine Learning*, 107(1):15–41, 2018.

[116] Philipp Probst, Anne-Laure Boulesteix, and Bernd Bischl. Tunability: Importance of hyperparameters of machine learning algorithms. *J. Mach. Learn. Res.*, 20(53):1–32, 2019.

[117] Alan Hartman and Leonid Raskin. Problems and algorithms for covering arrays. *Discrete Mathematics*, 284(1-3):149–156, 2004.

[118] Charles J Colbourn and Jeffrey H Dinitz. Part vi: Other combinatorial designs. In *Handbook of Combinatorial Designs*, pages 349–350. Chapman and Hall/CRC, 2006.

[119] David M. Cohen, Siddhartha R. Dalal, Michael L. Fredman, and Gardner C. Patton. The aetg system: An approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering*, 23(7):437–444, 1997.

[120] Renée C Bryce and Charles J Colbourn. The density algorithm for pairwise interaction testing. *Software Testing, Verification and Reliability*, 17(3):159–182, 2007.

[121] Renée C Bryce and Charles J Colbourn. A density-based greedy algorithm for higher strength covering arrays. *Software Testing, Verification and Reliability*, 19(1):37–53, 2009.

[122] Yu Lei and Kuo-Chung Tai. In-parameter-order: A test generation strategy for pairwise testing. In *Proceedings Third IEEE International High-Assurance Systems Engineering Symposium (Cat. No. 98EX231)*, pages 254–261. IEEE, 1998.

[123] Linbin Yu, Yu Lei, Raghu N Kacker, and D Richard Kuhn. Acts: A combinatorial test generation tool. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 370–375. IEEE, 2013.

[124] Jose Torres-Jimenez and Idelfonso Izquierdo-Marquez. Survey of covering arrays. In *2013 15th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pages 20–27. IEEE, 2013.

[125] Yu Lei, Raghu Kacker, D Richard Kuhn, Vadim Okun, and James Lawrence. Ipog: A general strategy for t-way software testing. In *14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'07)*, pages 549–556. IEEE, 2007.

[126] Michael Forbes, Jim Lawrence, Yu Lei, Raghu N Kacker, and D Richard Kuhn. Refining the in-parameter-order strategy for constructing covering arrays. *Journal of Research of the National Institute of Standards and Technology*, 113(5):287, 2008.

[127] Yu Lei, Raghu Kacker, D Richard Kuhn, Vadim Okun, and James Lawrence. Ipog/ipog-d: efficient test generation for multi-way combinatorial testing. *Software Testing, Verification and Reliability*, 18(3):125–148, 2008.

[128] Raghu N Kacker, David R Kuhn, Yu Lei, et al. Improving ipog's vertical growth based on a graph coloring scheme. 2015.

[129] Mohammed I Younis and Kamal Z Zamli. Mipog-an efficient t-way minimization strategy for combinatorial testing. *International Journal of Computer Theory and Engineering*, 3(3):388, 2011.

[130] Kristoffer Kleine and Dimitris E Simos. An efficient design and implementation of the in-parameter-order algorithm. *Mathematics in Computer Science*, 12(1):51–67, 2018.

[131] Fahad ALGorain and John Clark. Bayesian hyper parameter optimization for malware detection, 2021.

[132] Shao-Huai Zhang, Cheng-Chung Kuo, and Chu-Sing Yang. Static pe malware type classification using machine learning techniques. In *2019 International Conference on Intelligent Computing and its Emerging Applications (ICEA)*, pages 81–86. IEEE, 2019.

[133] Subhojeet Pramanik and Hemanth Teja. Ember-analysis of malware dataset using convolutional neural networks. In *2019 Third International Conference on Inventive Systems and Control (ICISC)*, pages 286–291. IEEE, 2019.

[134] R Vinayakumar and KP Soman. Deepmalnet: evaluating shallow and deep networks for static pe malware detection. *ICT express*, 4(4):255–258, 2018.

[135] Colin Galen and Robert Steele. Evaluating performance maintenance and deterioration over time of machine learning-based malware detection models on the ember pe dataset. In *2020 Seventh International Conference on Social Networks Analysis, Management and Security (SNAMS)*, pages 1–7, 2020.

[136] Nureni Ayofe Azeez, Oluwanifise Ebunoluwa Odufuwa, Sanjay Misra, Jonathan Oluranti, and Robertas Damaševičius. Windows pe malware detection using ensemble learning. *Informatics*, 8(1), 2021.

[137] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, et al. Api design for machine learning software: experiences from the scikit-learn project. *arXiv preprint arXiv:1309.0238*, 2013.

[138] Roc auc. https://scikit-learn.org/stable/modules/generated/sklearn.metrics.roc_auc_score.html. Accessed : 2022-4-28.

[139] Gadiel Seroussi and Nader H Bshouty. Vector sets for exhaustive testing of logic circuits. *IEEE Transactions on Information Theory*, 34(3):513–522, 1988.

[140] Paris Kitsos, Dimitris E Simos, Jose Torres-Jimenez, and Artemios G Voyiatzis. Exciting fpga cryptographic trojans using combinatorial testing. In *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, pages 69–76. IEEE, 2015.

[141] Kristoffer Kleine and Dimitris E Simos. Coveringcerts: Combinatorial methods for x. 509 certificate testing. In *2017 IEEE International conference on software testing, verification and validation (ICST)*, pages 69–79. IEEE, 2017.

[142] sklearn. sklearn-accuracy-metrics. https://scikit-learn.org/stable/modules/generated/sklearn.metrics.accuracy_score.html, 2022. Accessed: 2022-07-21.

[143] George E Dahl, Tara N Sainath, and Geoffrey E Hinton. Improving deep neural networks for lvcsr using rectified linear units and dropout. In *2013 IEEE international conference on acoustics, speech and signal processing*, pages 8609–8613. IEEE, 2013.

[144] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[145] Imran Shafi, Jamil Ahmad, Syed Ismail Shah, and Faisal M Kashif. Impact of varying neurons and hidden layers in neural network architecture for a time frequency application. *2006 IEEE International Multitopic Conference*, 2006.

[146] Brownlee Jason. How to configure the number of layers and nodes in a neural network.

[147] Ibrahem Kandel and Mauro Castelli. The effect of batch size on the generalizability of the convolutional neural networks on a histopathology dataset. *ICT express*, 6(4):312–315, 2020.

[148] Shiliang Sun, Zehui Cao, Han Zhu, and Jing Zhao. A survey of optimization methods from a machine learning perspective. *IEEE transactions on cybernetics*, 50(8):3668–3681, 2019.

[149] sklearn. sklearn-standardscaler. https://scikit-learn.org/stable/modules/generated/skle, 2022. Accessed: 2022-07-06.

[150] Soufiane Hayou, Arnaud Doucet, and Judith Rousseau. On the impact of the activation function on deep neural networks training. In *International conference on machine learning*, pages 2672–2680. PMLR, 2019.