NOVA IMS

Information Management School

MGI

Mestrado em Gestão de Informação

Master Program in Information Management

Improving Malware Detection with Neuroevolution: A Study with the Semantic Learning Machine

Mário José Santos Teixeira

Project presented as partial requirement for obtaining the Master's degree in Business Intelligence and Knowledge Management

NOVA Information Management School
Instituto Superior de Estatística e Gestão de Informação
Universidade Nova de Lisboa

NOVA Information Management School
Instituto Superior de Estatística e Gestão de Informação
Universidade Nova de Lisboa

IMPROVING MALWARE DETECTION WITH NEUROEVOLUTION: A STUDY WITH THE SEMANTIC LEARNING MACHINE

by

Mário José Santos Teixeira

Project presented as partial requirement for obtaining the Master's degree in Business Intelligence, with a specialization in Knowledge Management and Business Intelligence

Advisor: Professor Ivo Gonçalves, PhD

Co-Advisor: Professor Mauro Castelli, PhD

February, 2019

ACKNOWLEDGMENTS

I would like to thank my supervisor Ivo Gonçalves for his guidance and support during this though challenge. Thank you also to my co-supervisor Mauro Castelli for his contribution, support and constant availability. For both of you to have properly introduced me into the world of Artificial Intelligence, with such a good and recent algorithm to work with, it was truly a great experience.

A big thanks also to my mother, father and grandmother, who have always supported me throughout my whole academic life, pushing me to always do my best and really helped relieve me from other tasks so I could focus on my work.

A thank you to my work colleagues, that everyday incentivize me to be a better me and have always been very supportive to this project.

Last but definitively not least, to Bruna Ribeiro, that has been by my side through every single step of this hard journey, not letting me off the track for even a little bit, always giving me motivation and strength to continue, being my role model and a big part of this projects' final result. A huge thank you.

ABSTRACT

Machine learning has become more attractive over the years due to its remarkable adaptation and problem-solving abilities. Algorithms compete amongst each other to claim the best possible results for every problem, being one of the most valued characteristics their generalization ability.

A recently proposed methodology of Genetic Programming (GP), called Geometric Semantic Genetic Programming (GSGP), has seen its popularity rise over the last few years, achieving great results compared to other state-of-the-art algorithms, due to its remarkable feature of inducing a fitness landscape with no local optima solutions. To any supervised learning problem, where a metric is used as an error function, GSGP's landscape will be unimodal, therefore allowing for genetic algorithms to behave much more efficiently and effectively.

Inspired by GSGP's features, Gonçalves developed a new mutation operator to be applied to the Neural Networks (NN) domain, creating the Semantic Learning Machine (SLM). Despite GSGP's good results already proven, there are still research opportunities for improvement, that need to be performed to empirically prove GSGP as a state-of-the-art framework.

In this case, the study focused on applying SLM to NNs with multiple hidden layers and compare its outputs to a very popular algorithm, Multilayer Perceptron (MLP), on a considerably large classification dataset about Android malware. Findings proved that SLM, sharing common parametrization with MLP, in order to have a fair comparison, is able to outperform it, with statistical significance.

KEYWORDS

Geometric semantic genetic programming; Artificial Neural Networks; Genetic Programming; Supervised Learning; Semantic Learning Machine; Multilayer Neural Networks

INDEX

1. Introduction 1
1.1. Background and Problem Definition 1
1.2. Study Objectives 2
1.3. Study Relevance and Importance 2
1.4. Structure 2
2. Literature Revision 3
2.1. Supervised Learning 3
2.2. Genetic programming 5
2.2.1. Individual representation 5
2.2.2. Initial population 7
2.2.3. Diversity operators 7
2.2.4. Fitness 9
2.2.5. Selection criteria 9
2.2.6. Implementation 10
2.3. Geometric Semantic Genetic Programming 11
2.3.1. Geometric semantic mutation 12
2.3.2. Geometric semantic crossover 13
2.4. Artificial Neural Networks 15
2.4.1. Biological neural networks 15
2.4.2. Learning 16
2.4.3. Network initialization 16
2.4.4. Hyper parameters 17
2.4.5. Other considerations 18
2.4.6. Gradient Descent 19
2.4.7. Backpropagation 19
2.4.8. Multilayer Perceptron 21
2.4.9. Evolutionary Neural Networks 24
2.5. Semantic Learning Machine 26
3. Methodology 30
3.1. Data 30
3.2. Algorithms 30
3.2.1. SLM configuration 31
3.2.2. MLP configuration 33

4. Results and analysis 35
4.1. First test 35
4.1.1. SLM variants 35
4.1.2. MLP variants 37
4.1.3. Overall comparison 41
4.2. Second Test 42
4.2.1. SLM variants 42
4.2.2. MLP variants 44
4.2.3. Overall comparison 47
5. Conclusions 48
6. Bibliography 49

LIST OF TABLES

Table 1 - SLM parameters 31
Table 2 - MLP parameters 34
Table 3 - Validation AUROC for each SLM variant considered 35
Table 4 - Best SLM configuration by variant 35
Table 5 - Learning step for SLM-BLS 36
Table 6 - Number of iterations for each SLM variant considered 36
Table 7 - AUROC values for each MLP variant considered 37
Table 8 - Best MLP configuration by variant 37
Table 9 - Number of iterations for each MLP variant considered 37
Table 10 - Learning rate by MLP variant 38
Table 11 - L2 penalty by MLP variant 38
Table 12 - Activation functions use by MLP variant 38
Table 13 - Batch size by MLP variant 38
Table 14 - Batch shuffle use by MLP variant 39
Table 15 - Number of layers for each MLP variant considered 39
Table 16 - Total number of hidden neurons for each MLP variant considered 39
Table 17 - Momentum in MLP SGD 39
Table 18 – Nesterov's momentum use in MLP SGD 40
Table 19 - Beta 1 in MLP Adam 40
Table 20 - Beta 2 in MLP Adam 40

Table 21 - Validation AUROC for each SLM variant considered 42
Table 22 - Best SLM configuration by variant 42
Table 23 - Learning step for SLM-BLS 43
Table 24 - Number of iterations for each SLM variant considered 43
Table 25 - AUROC values for each MLP variant considered 44
Table 26 - Best MLP configuration by variant 44
Table 27 - Number of iterations for each MLP variant considered 44
Table 28 - Learning rate by MLP variant 44
Table 29 - L2 penalty by MLP variant 45
Table 30 - Activation functions use by MLP variant 45
Table 31 - Batch size by MLP variant 45
Table 32 - Batch shuffle use by MLP variant 45
Table 33 - Number of layers for each MLP variant considered 46
Table 34 - Total number of hidden neurons for each MLP variant considered 46
Table 35 - Momentum in MLP SGD 46
Table 36 – Nesterov's momentum use in MLP SGD 46
Table 37 - Beta 1 in MLP Adam 46
Table 38 - Beta 2 in MLP Adam 46

LIST OF FIGURES

Figure 1 - Stoppage of training before starting to overfit (Henriques, Correia, & Tibúrcio, 2013) 4
Figure 2 - Simple GP individual represented as a syntax tree 6
Figure 3 - GP crossover diversity operator example 8
Figure 4 - GP mutation diversity operator example 8
Figure 5 - Genotype to semantic space mapping (Vanneschi, 2017) 12
Figure 6 – Biological Neuron (Karpathy, n.d.) 15
Figure 7 – Perceptron (Dukor, 2018) 16
Figure 8 - Simplest form of a multilayered artificial neural network 26
Figure 9 - SLM Mutation (Gonçalves, 2016) 27
Figure 10 - Multilayer SLM mutation operator example 33
Figure 11 - Boxplots for AUROC values of SLM and MLP algorithms respectively 41

Figure 12 - Boxplots respective to the second test for the AUROC values of SLM and MLP algorithms respectively...47

LIST OF ACRONYMS AND ABBREVIATIONS

GSGP Geometric Semantic Genetic Programming

GP Genetic Programming

ANN Artificial Neural Network

EANN Evolutionary Artificial Neural Network

SLM Semantic Learning Machine

MLP Multilayer Perceptron

1. INTRODUCTION

This chapter's goal is to provide the reader an introduction of this study's work, clarifying its background and the motivation and relevance that led to its development, as well as the objectives and structure of this work.

1.1. BACKGROUND AND PROBLEM DEFINITION

Machine Learning statistical methods and algorithms go as early as the 1950's decade, with a continual growth of evolution throughout the years. A particular goal for these systems is to output intelligence comparable to the output a human being would be able to give, being therefore new human-competitive systems (Gonçalves, 2016; Koza, Keane, & Streeter, 2003; Turing, 1950).

A big focus on this topic has been the development of systems capable of self-sustainability and continual improvement, such as evolutionary artificial neural networks (Yao, 1993, 1999). The continuous aim on this topic has been to replicate as closely as possible the behavior of biological neural networks that, based on several distinct biological sensorial inputs, generate specific knowledge from it. On this particular subject, machine learning refers to the task of generating knowledge (predictive patterns) based on a set of examples of unseen data. The maintenance for these systems is to keep being fed data in order to have more distinct training cases, to better assess and perfect the evaluation model that generates the predictive output, avoiding the typical challenge of overfitting in order to be successful.

A specific topic on this subject is Genetic Programming (GP) algorithms that are a class of computational methods inspired by natural evolution, having the great characteristic of flexibility on model's evolution, having no *a priori* constrains that could restrict the set of functions or structure to be used. The downside of GP is the constant concern on overfitting (Gonçalves, 2016; Gonçalves, Silva, & Fonseca, 2015a).

The system in focus on this study, Geometric Semantic Genetic Programming (GSGP), is a recently proposed strand of GP focused on smoothing the fitness landscape as unimodal (no local optima solutions) with a linear slope, which allows for an easier search of the semantic space. This characteristic can be applied to any variation of supervised learning task, with great results when compared to other learning algorithms while still having already evidence of not falling into the overfitting problems of standard GP (Gonçalves et al., 2015a; Moraglio, Krawiec, & Johnson, 2012; Vanneschi, 2017).

Gonçalves studied the application of GSGP to artificial neural networks, creating the Semantic Learning Machine (Gonçalves, 2016; Gonçalves, Silva, & Fonseca, 2015b). All recent studies to test SLM have proven that it can outperform state of the art evolutionary algorithms as well as classic algorithms on several distinct benchmark tasks (Gonçalves et al., 2015b; Jagusch, Gonçalves, & Castelli, 2018).

The aim of this project is to expand the knowledge on SLM's competitive ability, implementing it on multilayer neural networks and testing how it will perform against the well-established Multilayer Perceptron (MLP) algorithms on large datasets.

1.2. Study Objectives

The scope for this project is to implement the SLM algorithm on neural networks with multiple hidden layers. Afterwards, it will be put to the test on a large dataset and compare its results against MLP's results on the same dataset.

In pursuance of this project's vision, the following objectives have been defined:

1. Evaluate the state of art on artificial neural network algorithms, focusing on MLP
2. Define common parameters for SLM and MLP for a fair comparison
3. Implement the SLM model containing GSGP algorithm and expand it to neural networks with multiple hidden layers
4. Implement a popular MLP algorithm
5. Test both algorithm's performance against the same dataset
6. Compare results and analyze conclusions

1.3. Study Relevance and Importance

The proposed project implementation has impact on the fields of Genetic Programming, Geometric Semantic Genetic Programming, Artificial Neural Networks and Semantic Learning Machine. GSGP and SLM has already proven themselves with some interesting results on other study applications (Castelli, Vanneschi, & Silva, 2014; Gonçalves et al., 2015b; Jagusch et al., 2018; Vanneschi, 2017) by outperforming traditional genetic programming algorithms. There are still areas of study that need to be assessed for GSGP to be considered a state of art. One of them is the lack of evidence on multi-layered neural networks (Gonçalves, 2016).

The expected outcome is for SLM to have consistently better accuracy on the results versus MLP.

If successful, the results would reinforce that GSGP is capable of being applied to any artificial learning system of any kind of complexity, proving its ability of generalization. This would also mean that even for more complex artificial learning systems, there would not be a need for the concept of population and convergence operators such as crossover. This will also allow for more opportunities for proof of concepts for this approach in comparison to other fields of study.

1.4. Structure

This chapter intends to guide the reader, detailing the structure for this document.

Alongside this chapter, the document is composed of 6 chapters. Chapter 2 contains the literature background required for the reader to be contextualized and integrated with previously existing studies. Chapter 3 details the methodology used to obtain the results. Chapter 4 presents and analyzes the results obtained. Chapter 5 summarizes the key points to retain from this study. Chapter 6 presents the bibliography used.

2. LITERATURE REVISION

2.1. SUPERVISED LEARNING

Supervised learning comprehends a machine learning task in which, for a given set of instances, a model should be able to build an algorithm capable of understanding the pattern that the input instances represent, in order to be able to predict any future unseen instance as precisely and accurately as possible (Gonçalves, 2016). These features not only aim to complement knowledge where humans cannot extract alone from direct data analysis, but also provide other means of comparison, validation of results, and a more accurate version of outcomes when referring to prediction tasks based on relation between multiple variables (Kotsiantis, 2007).

The training is supervised, meaning that all input data is represented by the same features and also that every instance has an assigned target value. The dataset is a collection of labelled vectors $\{(x_i, y_i)\}_{i=1}^{N}$, in which each dimension $j = 1, \dots, D$ contains a value that is part of the description of the example, also named as feature. All features x^j should translate the same knowledge about the x_k, for all $x_k, k = 1, \dots, N$ on the dataset. Label y_i can either be an element of a set of classes $\{1,2, \dots, C\}$ or a real number (Burkov, 2018). The model's algorithm aims at being capable of inferring a function that reflects the underlying pattern behind the variables to obtain the predicted outcome, with the least deviation possible from the actual target. The relation pattern of the features in comparison to the target variable is the learning goal of the algorithm (Cunningham, Cord, & Delany, 2016).

Depending on the type of prediction, different learning tasks can be defined. For a numerical value problem, a regression task is assigned, while when trying to label the instance to a given set, a classification task is preferred. The case when the target varies between two values is referred to as binary classification (Gonçalves, 2016; Kotsiantis, 2007).

Since the goal of the model will be to predict unidentified individuals, another crucial aspect to consider when producing the model is its generalization ability. The generalization of a model reflects its accuracy when applied to unseen data. If the full dataset was provided for the model to train upon, the end result would be an algorithm perfectly shaped to that particular input dataset, but with a poor prediction power for new unseen instances. The main goal of a model is to have the best result possible on the prediction of unseen data, in other words, having a good generalization ability. Therefore, a supervised learning model should have the best generalization ability possible (Gonçalves, 2016).

Since supervised learning can be implemented to countless scopes, each has its own metrics and performance priority preferences. Due to this, sometimes trade-offs are inevitable, and the error metrics also vary according to the applied domain, being possibly even biased by it (Cunningham et al., 2016). For this reason, it has been proven that the same algorithm performs differently according to its parametrization, which can lead to models excelling on some metrics but failing the threshold of others. Some of the best overall performing predictors, according to benchmark tests, are Bagged Trees, Random Forests and Neural Networks (Caruana & Niculescu-Mizil, 2006).

In order to improve the generalization ability of the model, the provided dataset is usually split into two different sets. The first one, the training dataset, will be used to train and evolve the model. From here, the algorithm should already be able to infer a relation between the input features. The second

set, referred to as the testing dataset, will be used to evaluate the performance of the model on unseen data and therefore understand the generalization ability of the algorithm. A model should aim to perform well in test data as it translates on good performance upon future unseen data. However, models may output better results on training data than on unseen data. Such models are said to be overfitting, translated to an exaggerated modulation to the pattern of the training data. Learning the algorithm to predict the training data is useless on the overall scheme since, as previously mentioned, the aim is to fit the model to unseen data (Gonçalves, 2016). Figure 1 shows how the testing set can be used to stop training before it starts overfitting.

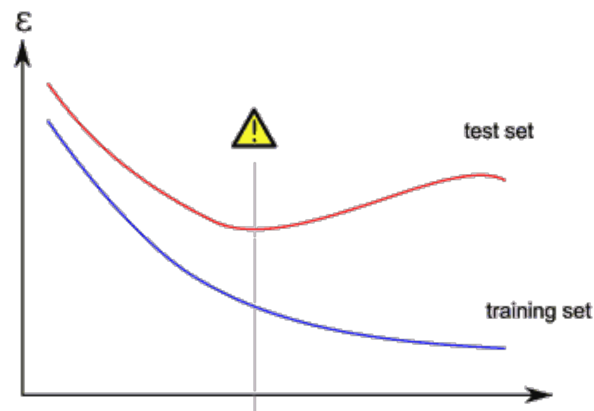

Figure 1 - Stoppage of training before starting to overfit (Henriques, Correia, & Tibúrcio, 2013)

2.2. GENETIC PROGRAMMING

Genetic programming (GP) (Koza, 1992) is an extension of the genetic algorithms (GA) theoretical approaches, dealing with GA's representation limitations by applying them to dynamically flexible structures of computer programs. Therefore, as well as GA, GP mimics its premises on the principles of biological processes that nature has perfected over the years. Nature's structures have evolved over long periods of time as a consequence of Darwin's natural selection as well as of extremely rare random genetic mutations (Banzhaf, Martín-Palma, & Banzhaf, 2013). In nature, those same structures would be put to test up against the environment, on which the fittest to it would be the ones to survive for further generations, creating a new population. Keeping the same mindset, originated the drive to understand how could computer programs evolve by themselves and assure a better result on each generation (Angeline, 1994; Krawiec, Moraglio, Hu, Etaner-Uyar, & Hu, 2013).

The most frequently used GP representation explains the computer programs as trees, known as standard GP, but there also exist other representations such as linear GP and graph GP. Standard GP consists on nodes, that can be either functions or input, further explained below, connected amongst themselves by branches (Fan, Gordon, & Pathak, 2005; Gonçalves, 2016; Krawiec et al., 2013).

The process itself remains fairly similar to nature's approach and also to the genetic algorithm's one, which in the case of computer programs, the goal is to evolve an algorithm up to a solution that has the best results possible. Let a computer program solution be named onwards as individual. The best individual would be the one that best represents the solution that is closest to the optimal one.

All the individuals pass through an evaluation as a way to understand how well each one performs solving the task at hand, being then assigned a correspondent value. In order for an individual to be considered better than another, a performance measure must be defined, related to the problem under analysis, being that measure named fitness. In GP, due to the selective pressure, the fittest individuals will be the ones to reproduce into the next generation. The individual with the highest fitness is the one with the least distance from the optimal solution and therefore is considered to be the best individual (Fan et al., 2005; Krawiec et al., 2013).

2.2.1. Individual representation

In GP, an individual represents a computer program. There have been several distinct representations proposed by researchers, such as the cartesian (Miller, 1999), linear (Banzhaf, 1993) and, the one that fits the purpose of this project, syntax trees (Koza, 1992). Figure 2 displays a simple GP individual represented as a syntax tree.

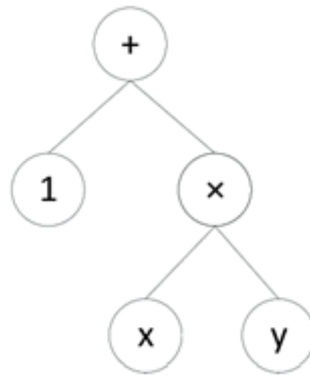

Figure 2 - Simple GP individual represented as a syntax tree

There are two sets of components on a syntax tree: nodes and leaves. Nodes are populated from a pre-existing function set, while the leaves, also known as terminals, can be set based on an also pre-existing terminal set. All individuals will be built based on the same function set and terminal set, meaning that the creation of these two defines the boundaries on how GP programs can be generated (Koza, 1992; Poli, Langdon, McPhee, & Koza, 2008).

The function set $F = \{f_1, f_2, \dots, f_n\}$ contains all possible operational functions to be used by the program. These can be logical functions (AND, OR, NOT, IF, etc.) or mathematical (+, -, *, sin, cos, etc.) (Koza, 1992). Koza also defined an important property, named closure, meant to keep the programs' integrity. This assures the programs' safety by avoiding forbidden operations, such as the division by 0, that was replaced by safe division, which returns 1 when encountering a division by 0. It also maintains the program's functionally by ensuring a type consistency, for instance, avoiding mixing the sum of a mathematical terminal with a Boolean result (Koza, 1992).

The terminal set $T = \{T1, T2, \dots, TN\}$ contains the program's available arguments. These can be the program's input variable, possible constant values or even random values.

Taking into consideration the individual in Figure 2, one can understand it was built with, at least, function set $F = \{+,*\}$ and $T = \{x, y, 1\}$.

2.2.2. Initial population

At the start of the evolution process, a new population of individuals is created. The number of individuals created depends on the parametrization given to the specific GP program. Upon creation of the population, the following parametrization is shared by all individuals:

- Function set (F) - which are the available functions for individuals to use;
- Terminal set (T) - all the input variables and constants that are added when there is enough knowledge on the problem to do so;
- Depth (d)- determines how many levels shall an individual have at maximum.

There are three possible methods to create new individuals. The first, the full growth method, sees an individual to be created with maximum depth level on all the possible branches of the tree, generating a tree with $2d - 1$ nodes total. The second way to create individuals is the growth method, that, at each tree level, randomly selects if the node will be a terminal or a function, until depth $d - 1$, to which afterwards all nodes are considered as terminals. The third and final growth method is called ramped half-and-half, which initializes half of the population by using the full method and the other half with the growth method (Koza, 1992).

2.2.3. Diversity operators

GP has at its disposal two operators to be applied to individuals, to create some genetic diversity amongst the population, and hopefully create better and more robust individuals. The operators are crossover and mutation, both needing individuals from an existing population, denominated as parents, producing the same number of new individuals, called children or offsprings. Both operators may be applied depending on a probability assigned for each one, but not at the same time (Koza, 1992; Krawiec et al., 2013; Poli et al., 2008). Commonly, crossover is applied with higher frequency than mutation (Gonçalves, 2016).

2.2.3.1. Crossover

The crossover operator requires two parents from the existing population and produces two children. It randomly selects two different nodes (one from each parent) known as crossover points and, subsequently, it swaps the subtrees rooted at the selected crossover points. Subtree from parent A is added to parent B at its crossover point, producing the first children. The symmetrical process is then applied again to get the second children. These new structures will be the two new children of the current generation (Gonçalves, 2016; Koza, 1992). Figure 3 illustrates this procedure.

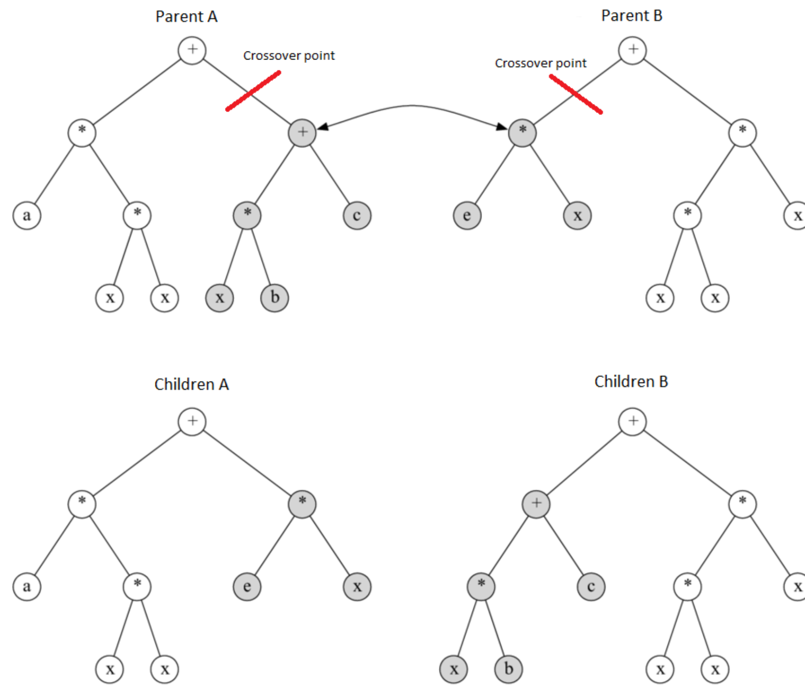

Figure 3 - GP crossover diversity operator example

2.2.3.2. Mutation

The mutation operator requires only one parent. It is intended to allow for random diversity to be induced in the population (Koza, 1992). A random point on the tree is selected, called mutation point. A random tree is then generated and the subtree from the parent is cut off, to be replaced by the new random tree. The new structure will be the offspring to be included in the current generation (Gonçalves, 2016). Figure 4 illustrates the mutation procedure.

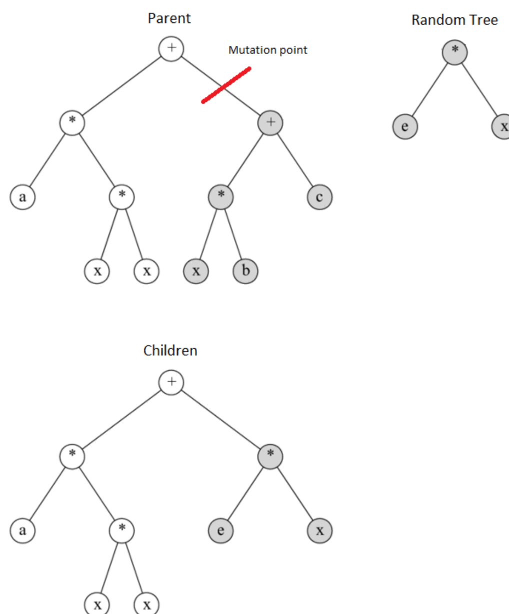

Figure 4 - GP mutation diversity operator example

2.2.4. Fitness

The fitness of an individual is not directly tied to it. It is merely a representation of that individual's quality over a specific set of data. This allows different individuals to be compared amongst each other, when presented to the same set of data, under the same fitness evaluation measure, named the fitness function. The closest an individual gets from correctly evaluating a set of input instances according to their respective targets, the higher fitness value that individual has (Koza, 1992).

A very common fitness function is the Root Mean Square Error (RMSE), also applied to this study's tests. Its goal is to evaluate the overall distance from all the computed outputs upon comparison to their targets. Being y_i the computed output for observation i and $\hat{y}_i$ its actual target, RMSE is:

$$RMSE = \sqrt{\frac{\sum_{i=1}^{n}(\hat{y}_i - y_i)^2}{n}}$$

2.2.5. Selection criteria

Through fitness evaluation of individuals, it is possible to compare them and rank them by fitness value. The best-ranked individuals will be used to evolve the population, while the worst ones are discarded. All individuals have an attached probability of being selected for future generations, falling the choice on the selection criterion applied.

Different approaches can be taken when considering, first, which individuals will be chosen as parents to apply the diversity operators to and, second, which population should carry on evolving into the next generations. One method of selection is using Tournament Selection, that randomly groups a certain number of individuals with uniform probability and the best one is selected according to its fitness output. The survivor selection is used as well, being the most common method the elitist survival, to which only the best individuals are kept going further into next generations. Several precautions need to be taken into consideration when parametrizing the selection criteria. Keeping only the best individual might convert the solution into a local optima one. Keeping too many individuals and the search process is heavier and more sparse space, to which usually new offsprings get better results on. Parametrizing on a balanced search is a key aspect to be taken into consideration (Gonçalves, 2016; Krawiec et al., 2013).

2.2.6. Implementation

To implement GP, one must define distinct parameters as Koza suggested (Koza, 1992):

- Function set
- Terminal set
- Fitness evaluation function
- Population initialization method
- Maximum tree depth (optional)
- Selection operator
- Crossover probability
- Mutation probability
- Stopping criterion

It should be noted that some parameters require additional parametrization. For instance, if Tournament Selection is the chosen selection operator, the number of individuals to be considered for the Tournament should also be defined.

After defining these, the algorithm works by:

1. Initializing the population
2. While the stopping criterion is not met:
    1. Evaluate all the individual's fitness
    2. Apply selection operator
    3. For each new individual to be created:
        1. Calculate what diversity operator will be applied
        2. Choose the parent
        3. Apply the diversity operator
    4. Replace the population with a new one
3. The best individual of all will be the final solution

It should also be noted that, due to GP's main goal of freely exploring the search space, it is possible that even with the same parametrization, two GP programs might produce different end results.

2.3. Geometric Semantic Genetic Programming

In 2012, genetic operators for GP were introduced, denominated as geometric semantic operators, whose main feature of interest was the induction of a fitness landscape characterized by the absence of locally sub-optimal solutions (unimodal fitness landscape), for any supervised learning problem where the program's fitness is a distance between a set of target values and the corresponding set of calculated outputs (Gonçalves, Silva, Fonseca, & Castelli, 2016; Moraglio et al., 2012; Vanneschi, 2017).

The semantics of a program is defined as the mathematical function to which afterwards corresponds a respective fitness output, when applied to an input vector of data. The n-dimensional space to which the vector belongs to is denominated as semantic space. The target vector itself is a point in the semantic space (Figure 5) (Vanneschi, 2017). Traditional GP operators act on the syntactic representation of individuals, their genotype, ignoring their actual significance in terms of semantic value. Moraglio et. al. question if such a semantically apathetic evolution would fit well across problems from different domains, since the meaning of the problems determines their search success (Moraglio et al., 2012).

Although these new semantically aware methods outputted with greater performance than traditional methods, they had the drawbacks of being very wasteful since they were focused on a trial-and-error approach and also that they do not provide insight on how syntactic and semantic searches relate to each other (Moraglio et al., 2012).

Moraglio et al. proposed a formal implementation of these semantically aware operators of GP, named Geometric Semantic Genetic Programming (GSGP) (Moraglio et al., 2012). GSGP has a great advantage in terms of evolvability, given by a unimodal fitness landscape. One limitation that persists is the task of visualizing and understanding the final solution generated. Not having any locally sub-optimal solution makes any symbolic regression easy to optimize for GP, regardless of the size of data. This eliminates one of the GP limitations of being extremely difficult to study in terms of fitness landscapes, due to the extreme complexity of the genotype/phenotype mapping and also the complexity of the neighbourhoods induced by GP crossover and mutation. The characteristic of having a unimodal error surface is quite rare on machine learning and should give GP a clear advantage compared to other systems, at least in terms of evolvability (Gonçalves, 2016; Moraglio et al., 2012; Vanneschi, 2017; Vanneschi, Castelli, Manzoni, & Silva, 2013).

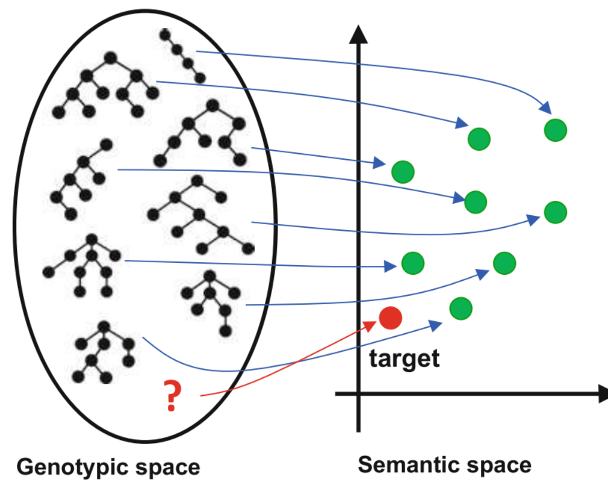

Figure 5 - Genotype to semantic space mapping (Vanneschi, 2017)

2.3.1. Geometric semantic mutation

The objective of the Geometric Semantic Mutation (GSM) is to generate a transformation on the syntax of GP individuals that has the same effect as the box mutation (Gonçalves et al., 2015a; Vanneschi, 2017). GSM always has the possibility of creating an individual whose semantics is closer to the target than before. The definition (Moraglio et al., 2012) is:

Definition – Given a parent function $P: R^n \rightarrow R$, the geometric semantic mutation with mutation step *ms* returns the real function $PM = P + ms\,.(T_{R1} - T_{R2})$, where T_{R1} and T_{R2} are random real functions.

Each element of the semantic vector of PM is a slight variation of the corresponding element in P's semantics. The variation is of small proportion since it is given by a random expression centered in zero. By changing the mutation step *ms* we are able to tune the impact of the mutation.

In order to make the perturbation even weaker, it is useful to limit the codomain of the possible outputs of the random trees into a given predefined range. This allows for a better control of the mutation. To guarantee that the output of the random trees assume values in [0,1], it is possible to use the output of a random tree as the input of a logistic function (Vanneschi, 2017).

The importance of having random individuals included in the mutation instead of random numbers is that when an individual is mutated, the aim is to perturb each one of its coordinates by a different amount. The perturbation must have the following properties (1) it has to be random; (2) it has to be likely to be different for each coordinate; (3) it does not have to use any information from the dataset. A random expression is likely to have different output values for different fitness cases. The difference between two expressions is used instead of just one random expression because, it may happen that, especially in the final part of a GP run, some of the coordinates have already been approximated in a satisfactory way, while others have not and in that case, it would be useful to have the possibility of

modifying some coordinates and not modifying others. The difference between two random expressions is a random expression centered in zero. By using the difference between two random expressions we are imposing that some coordinates may have a perturbation likely to be equal, or at least as close as possible, to zero (Gonçalves et al., 2015a; Moraglio et al., 2012; Vanneschi, 2017; Vanneschi et al., 2013).

2.3.2. Geometric semantic crossover

Geometric crossover generates one offspring that, for each coordinate I, a linear combination of the corresponding coordinates of the parents p and q, with coefficients included in [0,1], whose sum is equal to 1.

$$o_i = (a_i * p_i) + (b_i * q_i)$$

where $a_i \in [0,1]$, $b_i \in [0,1]$ and $a_i + b_i = 1$

The offspring can geometrically be represented as a point that stands in the segment joining the parents. The objective is to generate a tree structure that stands in the segment joining the semantics of the parents. The definition states (Moraglio et al., 2012):

Definition – Given two parent functions $T_1, T_2 : \mathbb{R}^n \rightarrow \mathbb{R}$, the geometric semantic crossover returns the real function $T_{xo} = (T_1 . T_R) + ((1 - T_R) . T_2)$, where T_R is a random real function whose output values range in the interval [0,1].

Using a random expression T_R instead of a random number can be interpreted analogously as the same as to Geometric Semantic Mutation. For the geometric crossover, the fitness function is supposed to be measured with the Manhattan distance; if the Euclidean distance is used instead, then TR should be a random constant instead (Moraglio et al., 2012; Vanneschi, 2017).

Since the offspring T_{xo} stands between the semantics of both parents T_1 and T_2, it cannot be worse than the worst of its parents.

Geometric semantic operators create offsprings that contain the complete structure of parents plus one or more random trees, therefore the size of the offspring is always much larger than the size of their parents. To respond to this issue, an automatic simplification after each generation, on which the individuals are replaced by semantically equivalent ones, was suggested (Moraglio et al., 2012). This, however, increases the computational cost of GP and is only a partial solution to the growth. Automatic simplification can also be a very hard task.

Another implementation was originally presented assuming a tree-based representation (Castelli, Silva, & Vanneschi, 2015; Vanneschi et al., 2013) but can actually fit any type of representation. This algorithm is based on the idea that an individual can be fully described by its semantics. At every

generation, a main table containing the best individuals is updated with the semantics of new individuals and the information needed to build new ones is saved. In terms of computational time, the process of updating the table is very efficient as it does not require the evaluation of the entire trees. Evaluating each individual requires constant time, independent from the size of the individual itself. There is also a linear space and time complexity with respect to population size and number of generations. The final step of this implementation is the reconstruction of the individuals, focusing only on the final best one. Disregarding the time to build and simplify the best individual, which can be made offline after the algorithm runs, the proposed implementation allowed to evolve populations for thousands of iterations with a considerable speed up with respect to standard GP (Gonçalves, 2016; Gonçalves et al., 2015a; Vanneschi, 2017).

2.4. Artificial Neural Networks

With the continuous developments on artificial intelligence, scientists/investigators decided to study and find a way to mimic the human brain, dating the first experiences as far as 1943 (McCulloch & Pitts, 1943). Artificial Neural Networks (ANN) are machine learning models that intend to replicate the human brain's operational behaviour. They are extremely powerful due to their non-linear interpolation ability as well as the fact that they are universal approximators, all of this while being completely data driven. Three elements are particularly important in any model of artificial neural networks: structure of the nodes; topology of the network and the learning algorithm used to find the weights (Buscema et al., 2018; Haykin, 2004; Raul Rojas, 1996). ANNs are usually used for regression and classification problems such as pattern recognition, forecasting, and data comprehension (Gershenson, 2003).

2.4.1. Biological neural networks

The information flow of the human brain happens via form of neurons, approximately 100 billions of them (Herculano-Houzel, 2009), forming extremely complex neural networks. Each neuron, as the one represented on Figure 6, is composed by the dendrites that receive ions through electrical stimuli called synapses. All the synapses flow into the neuron cell body, denominated as *nucleus*, responsible for processing the input signal. Via the neuron's *axon*, all the synapses' 'information' is propagated to its neighbours, depending only on the strength of the connection and sensitiveness to the signal. The *axon* terminals of a neuron are connected to its neighbours' *dendrites*. Researchers conservatively estimate there are more than 500 trillion connections between neurons in the human brain. Even the largest artificial neural networks today don't even come close to approaching this number (Patterson & Gibson, 2017).

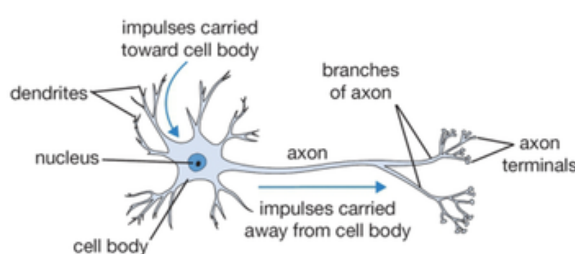

Figure 6 – Biological Neuron (Karpathy, n.d.)

ANNs reflect the exact same behaviour. Each ANN is represented by neurons and their connections to each other. To the connection between neurons are associated weights. ANNs are also split by layers, in order to dictate an information flow. Neurons of one layer communicate only with the neurons of the next layer, and to each connection a weight is assigned. There is no communication between neurons that belong to the same layer. The minimum structure of an ANN must contain one input layer and one output layer. In between, more layers can be added to the structure, being those layers denominated as hidden layers. Each ANN can contain between zero and infinite hidden layers.

The simplest form of an ANN is called the perceptron (Figure 7), which is represented by containing merely one layer of input neurons connected to a single output neuron. The number of neurons of the first layer, denominated as input layer, will be the same as the number of variables of the problem to which the neural network is being applied to and the value for each one of these neurons is the value of each variable. The value of each input neuron is then multiplied to the assigned weight between that neuron and the neuron of the next layer. The result of the output neuron is the sum of all input neurons times their weights. To the calculated value, a transfer function called activation function is applied, making all results that are brought into the network have an output that can be split by a linear function. Therefore, a limitation of the Perceptron is that it can only be applied to linearly separable problems (Gershenson, 2003; Gonçalves, 2016; Raul Rojas, 1996; Yao, 1999).

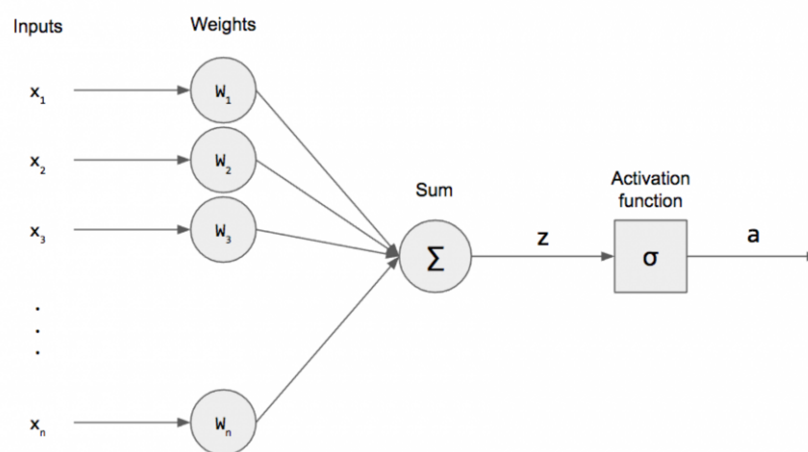

Figure 7 – Perceptron (Dukor, 2018)

2.4.2. Learning

The process of implementation of a neural network includes two phases: learning and testing. To define how can the neural network learn, a definition of learning must be set. Everyday learning can be translated into an increase of knowledge on a subject by means of studying, experience or being taught. Comparing this concept with what was described above as Supervised Learning, the learning process of neural networks is the ability to optimize either its weight values or structure, so that with the input values it takes, based on the difference of the network's output compared to the actual value, it can optimize itself to produce the lesser error possible, which translates to better results. The testing refers to the application of the model built (Gonçalves, 2016)**.**

2.4.3. Network initialization

The topology of an ANN references its external structure, stating the layers, the neurons and the connections between them. Its configuration can determine the success or failure of the model. However, there are no pre-determined rules to define it. Different types of ANNs will have different

topologies, however best the specialists responsible for the experiment find fit to be applied to it. It can have between a single layer up to a large number of them. For instance, the multilayer perceptron (MLP) is a standard multilayer feed forward network (the calculation between neurons is feed forward from the input nodes to the output nodes) with one input layer, one output layer, and one or multiple hidden layers. Because of the importance of the topology, it would make sense to use a GA to determine it, but researches showed it does not give better results (Yao, 1993).

All of these decisions influence the ANNs performance. However, the most efficient way is still to have an expert on both ANNs as well as the problem at hand to initialize them, as there is no scientific way of automatic optimization. These choices involve not only all of the ANNs hyperparameters, but others relevant to the learning process such as population size, number of generations, probability of crossover and mutation operations and what the best selection method is if an evolutionary training approach is considered (Buscema et al., 2018; Gershenson, 2003; Yao, 1999).

2.4.4. Hyper parameters

ANNs, as mentioned on the chapter above, are distinguished from each other based on their parametrization. Topology wise, there have already been mentions to the number of hidden layers of a network as well as the number of hidden neurons on each layer, again, not existing a scientific approach to assign values to these two.

Apart from those parameters, there are other tuning parameters that heavily affect the final outcome. First of those is the decision when to stop training the neural network and to those we name the stopping criterion. These can vary between knowledge specific suggestions, such as to stop training when the output reaches a certain threshold, to mathematical as the error deviation variation criteria, that takes into account how much the error differs from one training epoch to the other. There are also other options more generic, as stopping training after a certain amount of time, after a certain number of training epochs or if there is no improvement after a defined number of training epochs.

Another very important parameter is the learning rate (or also known as learning step), that controls the “navigation” of the neural network within the fitness landscape. A high learning step sets a high variability on the results, since it will always try to compensate the gap between the target values and the current output with a high correction. A low learning step might make the ANN stuck on a local minimum on the fitness landscape as well as increase the processing time. A general good approach is to have a relatively high learning step at first and decrease it after each training epoch. Other experiments have tried setting different rates for different layers (Haykin, 2004; Srinivas, Subramanya, & Babu, 2016).

Another part of Neural Network’s optimization decisions is the choice of the activation functions. These are particularly useful to induce non-linearity to the model, a key aspect on their universal approximator ability. Apart from this, different activation functions come with different perks. The vastly most used are the sigmoid function, squashing values between 0 and 1, tangent function that tends to center the output to 0 which has an effect of better learning on the subsequent layers, also with negative and positive output values considered as 0 and 1 respectively and finally one that is drawing a lot of interest which is the ReLU function, which removes the problem of vanishing gradient

faced by the above two (i.e. gradient tends to 0 as x tends to +infinity or -infinity). ReLU has the disadvantage of having dead neurons which result in larger NN's without an associated benefit. All the neurons on the network except the input layer one's should have an activation function being applied to them (Gershenson, 2003).

The number of neurons on the output layer is dependent on the specific case being studied. In case the final output is binary, either one value or another, or the target is a continuous number, a single output neuron is required. In case there are three or more output classes, an output neuron must be created for each target class and each instance should have only one of those output neurons set to 1 (Buscema et al., 2018).

2.4.5. Other considerations

Even if they are to be considered irrelevant when compared to their trade-off, a few drawbacks still need to be considered on ANNs. If not properly parametrized or when data contains noise, the resulting ANN can become excessively complex compared with statistical techniques in some scenarios. Another big common critique relates to its interpretability, since they work like black-boxes. Some intermediary determinations are meaningless to humans and the final result is in most of the cases unable to be explained in human terms. In cases of improper parameter tuning can also make the ANN unable to achieve reasonable results, as, for instance, a learning rate too high can result a high fluctuation on the search space, making the ANN unable to land on a local or global optimum. Some of these downsides can be solved by making an adjustment in the ANN using gradient descent or genetic algorithms, applying them to determine the connection weights, the network topology or even defining the learning rules. These ANNs are considered innovative and the literature about them is still relatively poor (Priddy & Keller, 2009; Srinivas et al., 2016; Yao, 1999).

2.4.6. Gradient Descent

A simple iteration of any kind of neural network bases itself on taking the data through the input layers, propagating it onward to the neurons on the hidden layers and getting a final output, which is then compared to the actual target to measure the accuracy of the model. The behavior of the network from the moment the first iteration ends until the end of training is what usually characterizes it (Rumelhart, Hinton, & Williams, 2013).

The most common, and with better results shown, ANNs are those whose weights are randomly generated on the beginning of the training process and are then iteratively updated at the end of each learning iteration. The most common methods to achieve this are gradient descent methods, such as backpropagation, simulated annealing or genetic algorithms. Gradient descent refers to the use of the gradient of the error function on the error surface. The gradient is the slope of the error surface, indicating the sensitivity of the error to weight changes (Buscema et al., 2018).

To avoid getting stuck on local optima solutions, algorithms have included the possibility of accepting a worse error or to perform jumps within the search space, if the result is not found to be good enough (Bottou, 2012; Wanto, Zarlis, Sawaluddin, & Hartama, 2017).

2.4.7. Backpropagation

Backpropagation (BP) (Rumelhart et al., 2013) is a very well-known learning algorithm for multi-layer neural networks and perhaps the most commonly applied gradient descent algorithm.

BP's goal is to iteratively adjust the weights in the network to produce the desired output, by minimizing the output error. It works by randomly initializing weights and optimizing all of the network's weights based on feedback from the output neurons. The input is received, propagated through the network, getting an output from it and then the loss function value is calculated for it, this usually being target output minus actual output. For each neuron, BP calibrates the weight by multiplying its value with the backpropagated error and afterwards, re-iterate by forward propagating the inputs again. The process is repeated until any stopping criteria is met.

BP, however, is very susceptible to local optimas in the search space of the fitness landscape. This means the starting point of the local search influences the final input of the algorithm, therefore several attempts of random initial networks should be tried out in order to try to avoid local optimas.

For BP to generate competitive results, a precise tuning of its parameter is required (Wanto et al., 2017). When applied to classification problems, its generalization ability can be subpart of its optimum result, due to the evaluation criteria difference between pattern classification and generalization ability (Tomomura & Nakayama, 2004).

2.4.7.1. Weight updating process

Given an input vector propagated through the network, its error output can be measured by a distance metric, such as the squared error. BP's objective is to minimize the error:

$$E_i = \sum_{i=1}^{n} |\hat{y}_i - y_i|$$

Where $\hat{y}_i$ is the known target value and y_i was the resulting output of the network.

Backpropagation updates the weights by calculating:

$$\Delta w_i = \eta \frac{\delta E}{\delta w_i}$$

Where η is positive if the output was lower than the target value, or negative otherwise. The reader is referred to (Priddy & Keller, 2009) for a detailed breakdown on how to reach the weight updating formulae.

2.4.8. Multilayer Perceptron

The Multilayer Perceptron (MLP) are layered feedforward networks typically trained with basic backpropagation (Baldi & Hornik, 1995). MLPs allow signals to travel only from input layer to output layer, having each layer composed of a large and highly interconnected neurons. The size of the input layer (the size of a layer is always correspondent to the number of neurons it contains) is determined by the number of features of the dataset. The number of output neurons is also defined based on the problem at hand. However, the number of hidden layers and hidden neurons are free parametrization, chosen by mere user experience and problem knowledge. These two parameters highly influence the balance between underfitting and overfitting (Phaisangittisagul, 2016).

The MLP is a very general model. One popular deployment of the MLP with possibility of customizing several of its parameters is Scikit-Learn's implementation of MLP (Pedregosa, Weiss, & Brucher, 2011). This will be the version implemented in this study and used for all future references of MLP.

MLP's algorithm is the merging of multi-layered neural networks with Backpropagation algorithm. It revolves around feeding the network with a set of training data, calculating its output values and based on the error produced, iteratively optimize its internal weights until it starts overfitting. Backpropagation's goal is to minimize the cost function used to calculate MLP's error. On neural networks with multiple hidden layers, the gradients are the partial derivative of the cost function on every weight, multiplied by the learning rate and subtracted from the current weight value. The gradients point to a reduction of the cost function surface space which, by maintaining a fixed topology from the beginning, only revolves around the network's weights. Therefore, Backpropagation, by iteratively computing the weights, is in theory blindly moving towards a minimum on the fitness landscape.

More than the two basic parameters referred above that control the network's topology, MLP allows for other types of customization to improve its results. The first parameter to be considered is the Backpropagation variation algorithm used to train MLP. The two most popular available solvers are Stochastic Gradient Descent (SGD) (Bottou, 2012) and Adam (Kingma & Ba, 2015). SGD is an iterative approach to discriminative learning of linear classifiers under convex loss functions. SGD is considered very efficient and easy to implement. Considering a pair (x, y) represented as z of arbitrary input x and scalar output y, a loss function $\ell(\hat{y}, y)$ that measures the cost of predicting $\hat{y}$ when facing the real output y and a family $\mathcal{F}$ of functions $f_w(x)$ parametrized by weight ω. The goal is to seek the function $f \in \mathcal{F}$ that minimizes the loss $Q(z, \omega) = \ell(f_w(x), y)$. Gradient descent approximates it throughout all the available examples. Each iteration updates the networks weight ω based on the gradient of the error function:

$$w_{t+1} = w_t - \alpha \frac{1}{n} \sum_{i=1}^{n} \nabla_w Q(z_i, w_t)$$

Where α represents the learning step parameter. SGD simplifies this expression by, instead of computing the whole gradient, each iteration estimates the gradient on a single randomly picked example z_t :

$$w_{t+1} = w_t - \alpha \nabla_w Q(z_t, w_t)$$

While SGD maintains a single learning rate for all weight updates and the learning rate does not change during training, in Adam, a learning rate is maintained for each network weight (parameter) and separately adapted as learning unfolds. Kingma & Ba state that the algorithm takes advantage of the benefits of both Adaptive Gradient Algorithm, that maintains a parameter specific learning rate which improves the performance on problems with sparse gradients, and also Root Mean Square Propagation, that also maintains parameter specific learning rates that are adapted based on the average of recent magnitudes of the gradients for the weight, as in how quickly it is changing (Kingma & Ba, 2015).

Since most problems require complex networks to solve them, MLPs naturally tend to overfitting (Phaisangittisagul, 2016). Researchers proposed the application of a penalty term to control the magnitude of the model parameters. One very popular and to be considered on this particular study is the L_2 regularization, derived from Tikhonov's regularization (Tikhonov, 1943). This technique discourages the complexity of the model, penalizing the loss function. Regularization works under the assumption that smaller weights generate simpler models and thus help avoid overfitting. Having a loss function represented as $L(x,y) = \sum_{i=1}^{n}(y_i - h_\theta(x_i))^2$, by applying L_2 regularization it will be then represented as:

$$L(x,y) = \sum_{i=1}^{n}(y_i - h_\theta(x_i))^2 + \lambda \sum_{i=1}^{n} \theta_i^2$$

Where λ is the regularization parameter that determines how severely should the weights be penalized. L_2 is considered to not being robust to outliers, delivers a better prediction when the output variable is a function of all input features and is also able to learn complex data patterns.

Another possibility on MLP's parametrization is the batch size parameter, as well as the possibility of batch shuffle. The known advantages of using batch sizes smaller than the number of available samples are first, for computing efficiency, as it takes less memory to train the network with fewer samples, and also that the MLP is usually able to train faster. Since the weights are updated at the end of the processing of samples, by reducing the available samples, the error is backpropagated quicker. The obvious drawback to this technique is that, the smaller the batch size is, the less accurate the estimate of the gradient will be. The option to have a shuffle every time a batch is generated aims on reducing the discrepancy of samples, especially on the last batch, theoretically providing the algorithm a batch that is more reflective of the dataset (Bello, 1992).

Another parameter that, for the SGD variant of the framework, allows to configure how it should explore the solution landscape is called momentum. This method allows to accelerate gradient vectors, theoretically leading to a faster conversion and avoiding of some local optima solution. It also helps to ignore noise on the data, focusing on getting an approximation of the end function by the calculating weighted averages. Let the momentum parameter be represented as β, then the new sequence follows the equation:

$$V_t = \beta V_{t-1} + \alpha \nabla_w L(W, X, y) \quad , \quad \beta \in [0,1[$$

$$W = W - V_t$$

Where L is the loss function, ∇ is the gradient and α is the learning rate.

Upon expanding the expression for three consecutive elements, it results in:

$$V_t = \beta\beta(1-\beta)S_{t-2} + \ldots + \beta(1-\beta)S_{t-1} + \ldots + (1-\beta)S_t$$

Where S_t are the gradients for a particular element. The older values of S get much smaller weight and therefore contribute less for overall value of the current point of V. The weight is eventually going to be so small that it can be forgotten since its contribution becomes too small to notice.

Within the same scope, SGD has also the possibility of using a momentum variant, named Nesterov's Momentum (Nesterov, 2004). It slightly differs from the momentum detailed above, by updating is formula as:

$$V_t = \beta V_{t-1} + \alpha \nabla_w L(W - \beta V_{t-1}, X, y) \quad , \quad \beta \in [0,1[$$

$$W = W - V_t$$

The main difference is in classical momentum, the velocity is corrected first and then take a considerable step on the solution space according to that velocity (and then repeat). In Nesterov's momentum, the algorithm first performs a step into the velocity direction and then corrects to a velocity vector based on new location (and then repeat) (Sutskever, Martens, Dahl, & Hinton, 2013).

For the Adam algorithm variance, it contains two parameters, named beta1 and beta2, that represent a similar notion to the concept of momentum. Adam calculates an exponential moving average of the gradient and the squared gradient, and the parameters beta1 and beta2 control the decay rates of these moving averages. The initial values of beta1 and beta2 close to 1 (not inclusive), result in a bias of moment estimates towards zero. Beta1 reflects the exponential decay rate for the first moment estimates while Beta2 reflects the exponential decay rate for the second moment estimates.

Wilson et al. observed that the solutions found by adaptive methods generalize worse than SGD, even when these solutions have better training performance. The results suggest that practitioners should reconsider the use of adaptive methods to train neural networks (Wilson, Roelofs, Stern, Srebro, & Recht, 2017).

2.4.9. Evolutionary Neural Networks

Evolutionary Neural Networks (EANN) combine the field of ANN with the evolutionary search procedures of GP. General EANN evolution practices are applied to a scope of three common ANN features: the connection weights, the architectures and the learning rules, being the latest the least explored. A stand out feature of EANNs is that they are able to evolve towards the fittest solution if set on an environment without interference and given enough time to explore the solution space (Yao, 1993).

Supervised learning is mostly considered a weight training optimization, finding the best set of connection weights for a network according to a certain criterion, with its most popular algorithm being the detailed on chapter 2.4.7, Backpropagation, that tries to minimize the total mean square error between the produced output and the actual target. This error is the one used to search the weight space. Although successful in many areas, BP has the drawbacks of getting stuck on local optima solutions and it is considered inefficient when searching on a vast, multimodal and nondifferentiable function. Yao refers that a way to overcome these gradient descent search algorithms is to consider them an evolution towards an optimal set of weights defined by a fitness function. This would allow for global search procedures to be used to train EANNs. The author breaks down the approach into first deciding the representation of the connection weights and second evolve it through Genetic Algorithms (GA), in this case by means of GP due to their representation. For this evolution, the topology and learning rule would be pre-defined and fixed. The procedure can be described as:

1. Decode individuals into sets of connection weights
2. Calculate their total mean square error and define the fitness of the individuals
3. Reproduce children with a probability associated to their fitness
4. Apply crossover, mutation or inversion

Results showed that the evolutionary approach was faster, the larger the EANN was, which implies the scalability of evolutionary training is better than that of BP training. Simple sets of genetic operators can be equally used in evolutionary training, such as the Gaussian random mutation, Cauchy random mutation, etc. The evolutionary training approach can also have the ANN's complexity and improve its generalization ability by adding a penalty to the error function, with the tradeoff of the computational cost for such measures. Evolutionary training is usually more computationally expensive and slower than gradient descent training. GAs are often outperformed by fast gradient descent algorithm techniques on networks with a single hidden layer, however evolutionary training is more efficient than BP on multilayer neural networks. The author also suggested a hybrid approach that incorporated local search procedures on evolutionary training exploratory ability, combining GA's global sampling with a local search algorithm to find the fittest of solutions on the neighborhood of the GA's result .
Yao also explored the evolution of EANNs architectures and results showed that EANNs trained by evolutionary procedures had a better generalization ability than EANNs trained only by BP. Some referred approaches impacted feature extraction by encoding only the most important features. This accelerated training as resulted on more compact representations of EANNs. The procedure of the topology evolution followed these steps:

1. Decode each individual into an architecture with the necessary details
2. Train each EANN with fixed learning rules and random initial weights
3. Calculate the fitness of the individual, that could be based on the training error, generalization ability, training tie, etc.
4. Reproduce children with probability according to their fitness
5. Apply crossover, mutation or inversion

The author reflected that the architecture representation of EANNs depended heavily on the application and prior knowledge. Also, the fitness evaluation for encoded architectures was very noisy since an individual's first fitness evaluation is based on random initialization of weight values. The crossover performed the best between groups of neurons rather than individual neurons, still not answering the question of the boundary to be considered a large group.
Finally, Yao explored the evolution of learning rules, where their evolution cycle consisted on:

1. Decode each individual into a learning rule
2. Randomly generate EANNs architecture and weights
3. Calculate the fitness of each individual
4. Reproduce children with a probability according to fitness
5. Apply crossover, mutation or inversion

Similar to the architecture's results, the fitness evaluation turned out to be noisy once more due to the randomness of the topology creation.

2.5. Semantic Learning Machine

Having already assessed the advantages of GSPS and its ability to be implemented on any supervised learning task, Gonçalves et al. proposed a new algorithm that utilizes GSGP's concept of the unimodal fitness landscape with the world of ANNs, the Semantic Learning Machine (Gonçalves, 2016; Gonçalves et al., 2015b).

SLM bases its premises on GSGP's mutation operator, developing a mutation operator that performs a linear combination on two individuals, one from the existing population (I_1) and a randomly generated individual (I_2). The mutation will be performed by including I_2 on a branch of I_1. In this scenario, I_2 will be the simplest form of a multilayered neural network, one with a single hidden layer containing a single hidden neuron, as shown in Figure 8. This neural network is the one that is going to be merged through mutation to the original ANN.

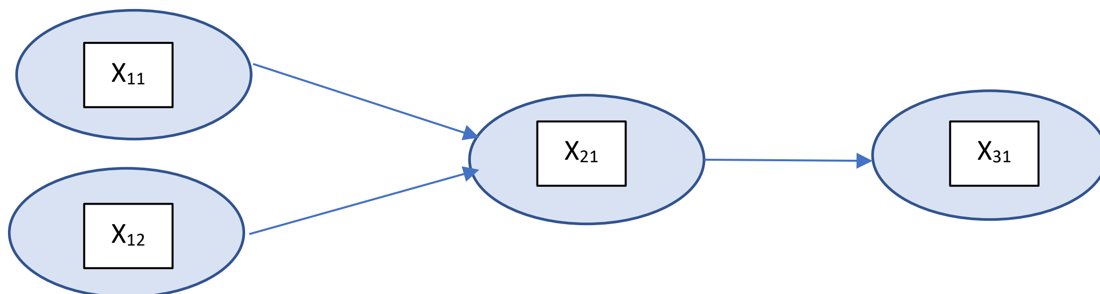

Figure 8 - Simplest form of a multilayered artificial neural network

A single neuron is added to each hidden layer. Having a number of layers $N \in [3, +\infty[$, the new neuron added to layer L_n, takes as input layer $L_n - 1$'s data and gives output to new neurons of the layer $L_n + 1$, or to the output layer for the case of the last hidden layer (Figure 9). The degree of impact of the mutation will be controlled by the mutation step parameter ((Gonçalves, 2016; Gonçalves et al., 2015b). Any activation function can be assigned to the generated neurons, although it should be non-linear to be able learn non-linearities from the data. The semantic impact of this mutation is weighted by the connection between the new neuron and the output layer neurons that is the learning step parameter, interchangeable for all connections. The input weights from the layer previous to the added neuron are all randomly generated. There are no restrictions applied to the weights' initialization.

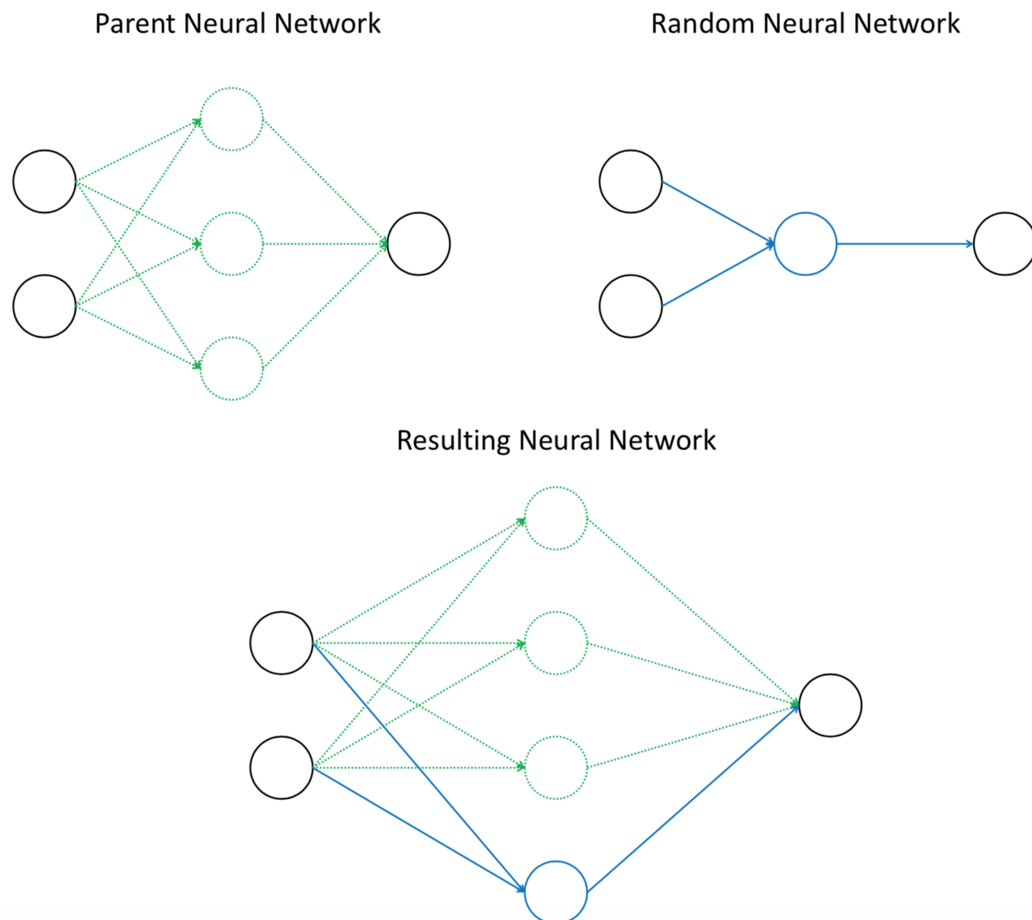

Figure 9 - SLM Mutation (Gonçalves, 2016)

This mutation can be applied to Neural Networks taking into consideration that there must be at least one hidden layer on the ANN and that the output neurons must have a linear activation function. All of the remaining topology of the ANN, meaning the layers, neurons and respective weights, remain unchanged from iteration one of training until a stopping criterion is found. There is no restriction on the activation function of the output neurons. In order to prevent overfitting, Gonçalves et al. recommended a small codomain function on the last hidden layer and a small mutation step (Gonçalves, 2016; Gonçalves et al., 2015a, 2015b).

Taking advantage of GSGP's properties of unimodal fitness landscape ability (Bosman, Engelbrecht, & Helbig, 2017), it is possible to evolve an ANN with a constantly positive fitness increase. SLM is a stochastic ANN construction algorithm originally derived from GSGP, that searches over unimodal error landscapes. As the issue of local optima does not apply, the evolutionary search can be performed by simple hill climbing (Jagusch et al., 2018). The concept can be applied to ANNs of any size, both in terms of number of neurons and number of hidden layers.

One of the main strengths of this mutation operator is also that it does not affect at all the ANN's structure (Gonçalves, 2016). There is absolutely no change on the existing network topology, meaning no structure changes and no weight changes from the original generated values. In terms of

computational cost, there is no need to perform the heavy task of processing the values for the remaining of the network every time. Only the new added neuron's values are calculated, and the result is directly computed to the output neurons that allows for a fast fitness re-evaluation directly proportional to the size of the input data.

There are no restrictions attached to the choice of the activation function on the output neurons. Gonçalves et. al. empirically proven that applying a linear activation function influenced GSGP to overfit to training data (Gonçalves, 2016). Activation functions with low codomain perform better on terms of generalization for the output neurons (Gonçalves et al., 2015a). To obtain even better generalization results, it is advised to assign a small value to the learning step parameter

The following parameters need to be defined in order to properly run an SLM:

- Learning/Mutation step (LS) – influence of the mutation operator on the ANN, influencing the weight of the neurons added through each mutation to the output neurons. It can be a fixed constant, or it can be variable for each application of the operator
- Initial population size (IPS) – number of individuals belonging to the initial randomly generated ANN's pool
- Mutation Limit (ML) - number of different individuals per generation
- Maximum number of layers (MNL) – control parameter for the maximum allowed number of layers for each random ANN
- Maximum number of neuros for each hidden layer (MNN) – control parameter for the maximum allowed number of neurons for each layer of a random ANN
- Minimum and Maximum value for weight of neuron connections – sets the bound limits from which the weighted connections between neurons can be randomly set
- Stopping criterions – limits the SLM's run upon achieving a certain state, such as maximum allowed fitness or maximum number of generations

The algorithm starts by generating a population of the size defined by the respective parameter. Each individual that belongs to the initial population will be randomly generated, using the MNL and MNN parameters. These parameters are used merely to control the size of the individuals, so that no over complex individual is created. There can be an opportunity to apply some problem knowledge when defining these two parameters. It is taken as an assumption that there will always be an input layer of size equal to the input variables size, as well as an output layer with pre-defined number of output neurons, depending on the supervised learning task.

From the initial generated population, the fitness of all individuals is calculated. Only one solution is kept at each step along the run. The individual with the highest fitness is kept as the best individual, to which the mutations will be applied to.

The algorithm will loop until one of the stopping criterions is met. On each iteration, there will be new ML number of copies from the best individual to which the mutation is then applied. The only difference between all of these children individuals is the weights that connect the newly added neuron to the previous layer. After all the mutations are performed, the best individual from the iteration is kept, to continue the cycle, or the original one is, if by any chance there was no individual which's fitness was greatest than the current best, from this iteration mutations (Gonçalves et al., 2015b).

SLM has proven to surpass GSGP with statistical significance on training data while maintaining a similar result in terms of generalization ability, on regression datasets (Gonçalves et al., 2015b, 2016).

Gonçalves et al. Implemented and tested the inclusion of two variants for SLM, both aiming to reduce the risk of the model overfitting (Gonçalves, Silva, Fonseca, & Castelli, 2018). One, called optimal learning step (OLS), which is equivalent to the optimal mutation step computation in GSGP (Gonçalves et al., 2015a) and the second being a fixed/best learning step (BLS) version, that has a constant value of 0.01 for the learning step parameter. Both variants resulted with competitive results while still keeping a low number of hidden neurons and fast evolution time.

Jagusch, Gonçalves, & Castelli have studied the comparison between the two SLM variants of OLS and BLS described above against Neuro-Evolution of Augmenting Topologies (NEAT) (Stanley & Miikkulainen, 2002), Multilayer Perceptron (MLP) and Random Forests (RF) (Breiman, 2001) on 9 real world classification and regression datasets. It also implemented variants to include information from the semantic neighborhood to decide when to stop the search before overfitting occurs. The best SLM variants achieve superior generalizations in two thirds of the comparisons (24 out of 36). In the remaining comparisons, no method is found to be superior to the best SLM variant. The best SLM variant was always superior on training. SLM-OLS was able to evolve significantly smaller NNs, while also requiring a significantly smaller computational effort. In a comparison of classification datasets, SLM was able to outperform RF.

3. METHODOLOGY

This work studies the algorithm designed by Gonçalves (Gonçalves, 2016), described in section 5. This particular study intends to perform a fair comparison between the recently emerging SLM algorithm with more traditional neural network methodologies on large datasets. To achieve this, we will compare both algorithms with by using parametrizations that are as similar as possible. So far, all SLM's studies have had benchmarks of datasets with a small number of instances. On this study, SLM's performance will be tested for the first time against an extensive dataset, whose subject is malware detection.

The data for this study is presented in Section 6.1. Section 6.2 will focus on the algorithms used and respective parametrization. Subsection 6.2.1 will go over the SLM algorithm and its parametrization while subsection 6.2.2 will focus on Multi-Layered Perceptron, from now on MLP, parametrization.

3.1. DATA

For this work, a dataset about Android malware was studied. The dataset was built by first collecting traces from 1000 of the most popular applications between January and April of 2014, from different categories, belonging to Google Play and analyzing them through an antivirus software service, confirming that this data was safe from malicious software. Then, from the Drebin dataset (Arp, Spreitzenbarth, Malte, Gascon, & Rieck, 2014), a malware dataset was appended, containing traces from 1000 random applications (Canfora, Medvet, Mercaldo, & Visaggio, 2015).

The final dataset used on this study contains 70990 instances and 113 features, reflecting metrics on CPU, memory, I/O and network (Canfora, Medvet, Mercaldo, & Visaggio, 2016) about the device's performance in the presence of the applications referred above.

The target value describes if the specific instance has malware or not, being the value of 1 representative of a device containing malware and 0 the opposite. The dataset's percentage of class 1 instances is around 50%.

3.2. ALGORITHMS

Both SLM and MLP algorithms share common procedures. At first, a Stratified K-fold is applied to the dataset, splitting it into 30 stratified parts. For each partition of data, each one of the algorithm's variants will be tested. The error evaluation metric is the Root Mean Squared Error, between the outputs of the model and the target values of the dataset. Both algorithms were executed 30 times for ensuring statistical robustness.

3.2.1. SLM configuration

Gonçalves' prior work focused solely on multi-layered neural networks with a single hidden layer, thus making SLM's mutation operator adding a single neuron on this hidden layer per mutation (Gonçalves, 2016).

On this work, SLM's scope is extended to multi-layered neural networks with multiple hidden layers. Different mutation operators could be applied, such as keeping adding new neurons only to the last hidden layer, adding a single neuron to all hidden layers, adding multiple new neurons to all hidden layers, mutating the number of hidden layers, etc. For this work, the focus will be solely on the mutation operator that adds a random number of neurons to each hidden layer, distinct for each layer. This was not contemplated in the original study conducted by Gonçalves, having however been slightly discussed by Jagusch et. al on different problems (Jagusch et al., 2018). This particular mutation operator will be now tested for its performance the first time on both a malware detection environment and an extensive dataset.

Two variants of the algorithm will be tested, as designed by Gonçalves et al. (Gonçalves et al., 2018), denominated as Best Learning Step (representative of the previously studied Optimized Learning Step) and Best Learning Step, from now on referred as SLM-OLS and SLM-BLS. Both variants share the same configuration parameters with the exception of the learning step. SLM-BLS generates a learning step at the beginning, randomly chosen between 0.001 and 1. SLM-OLS computes the optimal learning step for each iteration's mutation, using the Moore-Penrose inverse.

Table 1 - SLM parameters

Parameter	**Value**
Maximum number of combinations	30
Stopping criterion	Maximum of 200 iterations
Maximum number of layers	5
Initial maximum number of neurons per layer	5
Maximum neuron connection weight	0.5
Maximum bias connection weight	1.0
Mutation maximum new neurons per layer	3
Population size	10

SLM starts by generating an initial population of ANNs, controlled by the parameter “*Population size”*. Each NN is then generated, built with a random number of hidden layers, between 1 and the value chosen on “*Maximum number of layers”*, in this study, equal to 5. Each hidden layer will contain a random number of neurons, between one and the value of the parameter “*Initial maximum number of neurons per layer”*, set to 5 for this study. At this stage, the properties of bias and maximum neuron connection weights are set. Both will be instantiated with a random value between 0.1 and the value of the parameters “*Maximum bias connection weight*” and “*Maximum neuron connection weight”*, which in this study are 1.0 and 0.5 respectively. All of the network’s neurons are connected to others with a connection weight between negative “*Maximum neuron connection weight”* and positive “*Maximum neuron connection weight”*.

From the initial population, the best individual in terms of fitness is kept for the remainder of the evolution process. Afterwards, until the stopping criterion is met, in this case 200 iterations, SLM will run its evolution procedure. Its evolution consists on, at each iteration, generate a number of offsprings, controlled again by the parameter “*Population size”*. Each offspring will be a clone of the previous champion (individual with the best fitness evaluation), to which the mutation operator described above will be applied.

To each new neuron, a bias defined at the creation of the ANN is attached. The activation function of the new neuron is randomly chosen from a list containing Sigmoid, Tahn and ReLU. The new neurons will randomly choose which neurons from the previous layer they will be connected to. The output connection weight of the new neurons, as well as the bias connection weight, will have a value of between negative learning step and positive learning step. An illustration of a network before and after its mutation is shown in Figure 10.

Then, once more, only the best offspring is compared to the previous champion, and from the two, the one with the best fitness evaluation is kept for the next iteration, repeated until a stopping criterion is met.

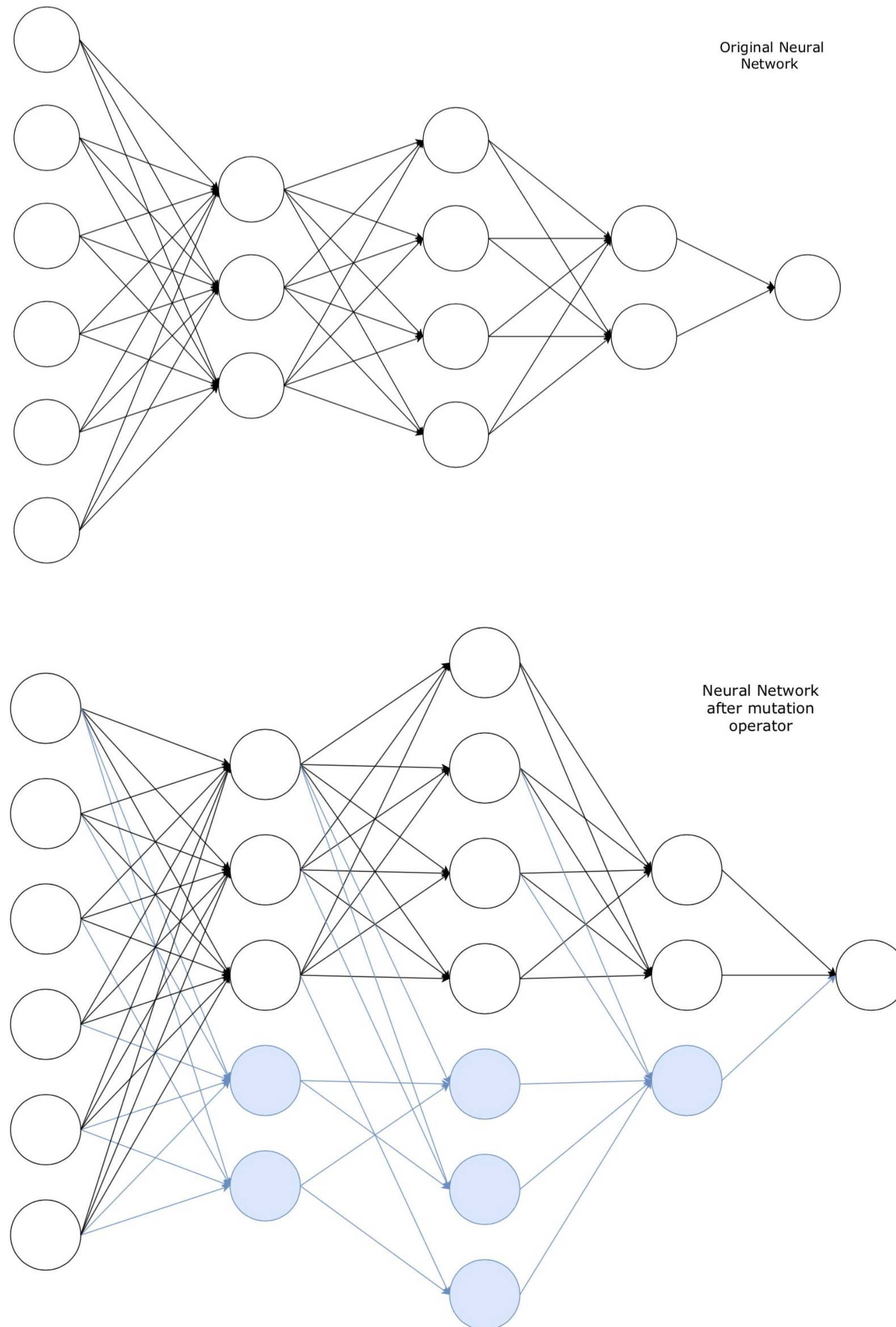

Figure 10 - Multilayer SLM mutation operator example

3.2.2. MLP configuration

Two MLP variants were also considered for this study, those being the two stochastic gradient descent, SGD and ADAM. The two will be referred from here forward as MLP-SGD and MLP-ADAM. Similarly, to SLM, both of these versions share almost all the same parameters. The difference between them relies on the MLP-SGD version to have a configuration that randomly turns on or off the parameter called "*Nesterov's momentum*", while MLP-ADAM contains the parameters *"beta_1"* and *"beta_2"*.

Table 2 - MLP parameters

Parameter	Value
Maximum number of combinations	30
Stopping criterion	Maximum of 200 iterations
Maximum number of layers	5
Maximum number of neurons per layer	200
Maximum neuron connection weight	0.5
Max Alpha	10
Minimum batch size	50
Minimum initial learning rate	0.001
Maximum initial learning rate	1

4. RESULTS AND ANALYSIS

4.1. FIRST TEST

4.1.1. SLM variants

All of the values were obtained from nested cross-validation, which means that they represent the mean and standard deviation achieved by the considered best models from the inner cross-validation.

The first analysis performed is the Area Under Receiver Operating Characteristic (AUROC) values produced by both variants, represented in Table 3, where one can verify that the OLS variant outperformed BLS. Taking into consideration the results of Table 4, that represents each variant's performance before the same subset of the K-Fold split data, the SLM-OLS variant's is clearly the best performer, having outperformed on every subset of the dataset.

Table 3 - Validation AUROC for each SLM variant considered

Variant	**OLS**	**BLS**
Value	0.784 +- 0.012	0.678 +- 0.022

Table 4 - Best SLM configuration by variant

Variant	**OLS**	**BLS**
Configuration wins	30	0

The results for the average number of iterations for each SLM variant, whose results are reported in Table 6, show that the SLM-BLS variant is able to converge to its optimal solution requiring fewer iterations than OLS. The boundaries of the learning step parameter seem to have been properly defined as the learning step for SLM-BLS rests almost in between the minimum and maximum allowed values, as shown in Table 5.

The overall results for this test confirm that even with a substantially larger dataset, SLM-OLS keeps outperforming other SLM variants.

Table 5 - Learning step for SLM-BLS

Variant	**BLS**
Learning step	0.581 +- 0.229

Table 6 - Number of iterations for each SLM variant considered

Variant	**OLS**	**BLS**
Iterations	160.900 +- 32.359	130.233 +- 47.319

4.1.2. MLP variants

Analyzing MLP's variants, following the same order as on SLM, starting by the AUROC analysis of the results, as present on Table 7, one can state that the best performer was the Adam variant, by a small difference between averages but retaining a higher variance. Table 8 shows that the SGD variant managed to outperform Adam one third of the times.

Table 7 - AUROC values for each MLP variant considered

Variant	**Adam**	**SGD**
Value	0.716 +- 0.076	0.690 +- 0.030

Table 8 - Best MLP configuration by variant

Variant	**Adam**	**SGD**
Configuration wins	20	10

Taking into consideration the number of iterations required to reach its best performance, Table 9 shows us that the Adam variant was able to reach it faster, with an average of 31 iterations sooner. The variance is very similar for both variants.

Table 9 - Number of iterations for each MLP variant considered

Variant	**Adam**	**SGD**
Iterations	104.967 +- 51.145	135.967 +- 47.482

Interestingly, there is a high discrepancy on the average learning step of both solutions. Adam kept a learning rate relatively close to the minimum learning step value, with a low variance as well, while SGD converged to an average value in between the minimum and the maximum allowed learning step, as Table 10 shows.

Table 10 - Learning rate by MLP variant

Variant	**Adam**	**SGD**
Value	0.147 +- 0.161	0.473 +- 0.222

The L2 penalty was relatively similar for both variants, valuing at approximately halfway point between the minimum and maximum values allowed for the parameter, keeping a moderate relation between weights and bias (not punishing either), as Table 11 shows.

Table 11 - L2 penalty by MLP variant

Variant	**Adam**	**SGD**
Value	4.954 +- 2.967	4.262 +- 2.638

Interestingly, the Adam variant had a higher preference for mainly the ReLU activation function, while SGD's results showed a much lower discrepancy between ReLU and Tahn, never considering however the Logistic function, Table 12 reports.

Table 12 - Activation functions use by MLP variant

Variant	**Adam**			**SGD**		
	Logistic	**ReLU**	**Than**	**Logistic**	**ReLU**	**Tahn**
Value	3	21	6	0	17	13

Having freedom to choose randomly between 50 and all the available dataset instances, on average Adam opted to use only about 25% of the dataset and SGD used even less, needing only 15% of it to generate its best results (Table 13).

Table 13 - Batch size by MLP variant

Variant	**Adam**	**SGD**
Value	17676.233 +- 774.139	11215.567 +- 9369.088

Comparing the option to shuffle or not, there is no considerable difference between any of the two options on both variants, having Adam opted to use batch shuffle only 40% of the times and SGD opted to use it 53%, Table 14 shows.

Table 14 - Batch shuffle use by MLP variant

Variant	Adam		SGD	
	With shuffle	**Without shuffle**	**With shuffle**	**Without shuffle**
Value	12	18	16	14

Interestingly, having the possibility of generating networks with the minimum of 1 and a maximum of 5 hidden layers, both variants opted for smaller topologies, with Adam having between 1 and 2 hidden layers on average while SGD had only 2 to 3 hidden layers on average as well (Table 15).

Table 15 - Number of layers for each MLP variant considered

Variant	**Adam**	**SGD**
Value	1.600 +- 0.855	2.367 +- 1.402

Taking a look at Table 16, another interesting metric is that, having the freedom of every hidden layer being composed of between 1 and 200 neurons, that reflects on a possible maximum number of hidden neurons of 1000, both variants again performed better with small topologies. Using the metric above, dividing the average maximum number of neurons by the average number of hidden layers, both Adam and SGD uses around 95 neurons per hidden layer, sitting closely between the minimum and maximum allowed values.

Table 16 - Total number of hidden neurons for each MLP variant considered

Variant	**Adam**	**SGD**
Value	151.667 +- 96.271	223.900 +- 152.134

Finally, reflecting on the exploratory option of the algorithm itself, SGD's momentum, that was limited between 0 (non inclusive) and 1, the algorithm averaged 0.679 with a close to 30% variance, meaning that the algorithm is more prone to using momentum than not (Table 17). Out of the 30 folds, SGD also opted more often not to use Nesterov's Momentum, using it only approximately 27% of the times as Table 18 reports.
On the other hand, Adam's beta values laid in between the allowed parameters of inclusive 0 to exclusive 1, showing no particular peculiarity, as represented in Table 19 and Table 20.

Table 17 - Momentum in MLP SGD

Variant	**SGD**
Momentum	0.679 +- 0.295

Table 18 – Nesterov’s momentum use in MLP SGD

Variant	**With**	**Without**
Value	8	22

Table 19 - Beta 1 in MLP Adam

Variant	**Adam**
Value	0.538 +- 0.245

Table 20 - Beta 2 in MLP Adam

Variant	**Adam**
Value	0.564 +- 0.262

4.1.3. Overall comparison

Now comparing the best results obtained by the SLM and MLP algorithms for their performance on generalization (test set performance over all split folds), their values are as represented on the boxplots of Figure 11. Each of the two boxplots represents the median as the most central line, the 25th and 75th percentile being the edges and the furthest data points as small diamonds.

The plots show that the overall results are very similar, with a small positive discrepancy from SLM's median over MLP's. The plot also shows that SLM, apart from the two far distant results, is much more consistent on its results, having the 2nd and 3rd percentile closer than the respective from MLP, as well as a smaller distance from top percentile to bottom percentile compared to MLP's.

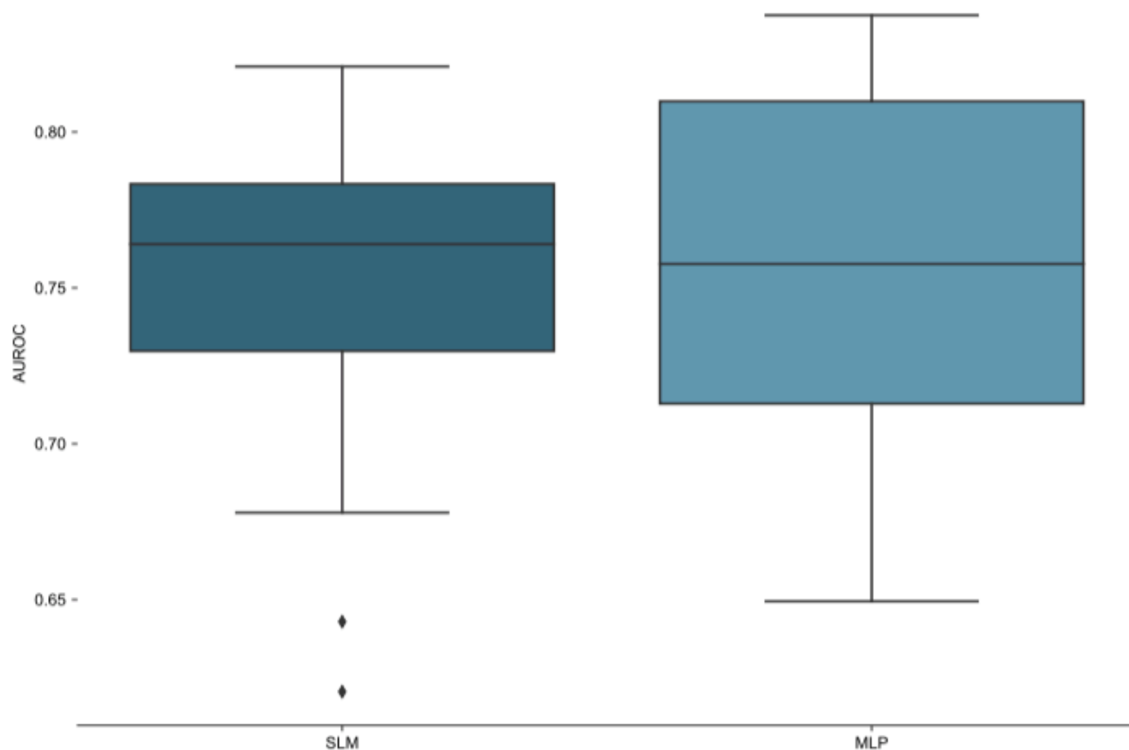

Figure 11 - Boxplots for AUROC values of SLM and MLP algorithms respectively

Next, in order to assess the statistical significance of the result, the Kolmogorov-Smirnov test was applied to confirm that the data was normally distributed, which was in fact the outcome. This result means that it is in possible to use Student's T-test to analyze the statistical difference between the results, being the null hypothesis that both medians are equal. Computing the best SLM runs to the best MLP runs, the *p—value* was 0.6690, meaning that the difference between both medians is not statistically different.

4.2. SECOND TEST

Although the results obtained on the first test already show that SLM is competitive with state-of-the-art algorithms, over a generalization comparison, with slightly better results while remaining more consistent, the question arises if by trying to be fair on the parametrization between the algorithms, enough time was given for SLM to evolve. To address this question, a new test was performed, being the only difference the change on the Stopping Criterion parameter, updated to 500 iterations for both SLM and MLP algorithms.

4.2.1. SLM variants

Performing the same analysis done on test number one, as expected, the SLM-OLS variant still maintains a higher AUROC value than BLS, as shown in Table 21. Increasing the number of epochs to evolve, led also to a positive 4.08% increase of its generalization ability. There has been no difference in SLM-OLS's domination on every fold of the dataset, still considered the best performer on 30 out of 30 folds, shows Table 22.

Table 21 - Validation AUROC for each SLM variant considered

Variant	**OLS**	**BLS**
Value	0.816 +- 0.010	0.733 +- 0.026

Table 22 - Best SLM configuration by variant

Variant	**OLS**	**BLS**
Configuration wins	30	0

On the first test, SLM-OLS averaged around 80% of the maximum number of allowed iterations with a 16% variance, while SLM-BLS recorded 65% with an approximate 24% variance. On this second test, SLM-OLS still maintained an 82% with a 12% variance of the maximum allowed iterations, while SLM-BLS had 78% and 18% variance. These values indicate that possibly the maximum number of allowed iterations could still be increased more to generate even better AUROC results.

The average learning step has increased from 58% of the maximum allowed parameter value to 62%.

Table 23 - Learning step for SLM-BLS

Variant	**BLS**
Learning step	0.623 +- 0.230

Table 24 - Number of iterations for each SLM variant considered

Variant	**OLS**	**BLS**
Iterations	406.800 +- 60.659	391.367 +- 89.245

4.2.2. MLP variants

By analyzing the new AUROC values, represented in Table 25, one can see that the values remain almost the same as on the previous test, with only a mere 0.016 values increased on Adam while SGD have an even smaller increment, of only 0.002 values on average. The variant preference remained exactly the same as on the first test, shown in Table 26.

Table 25 - AUROC values for each MLP variant considered

Variant	**Adam**	**SGD**
Value	0.732 +- 0.085	0.692 +- 0.027

Table 26 - Best MLP configuration by variant

Variant	**Adam**	**SGD**
Configuration wins	20	10

While on test one, the Adam variant only trained for 52% of the allowed maximum iterations before overfitting, on the second test it has only seen a 5% increase of the maximum number of allowed iterations. In fact, the SGD variant had 67,5% and actually decreased the percentage of maximum allowed iterations down to 58%, shows Table 27.

Table 27 - Number of iterations for each MLP variant considered

Variant	**Adam**	**SGD**
Iterations	287.867 +- 154.817	290.700 +- 120.811

Regarding the learning rate for both variants, the largest difference came from the SGD variant, more than doubling its first value, meaning that its best results came from taking a more exploratory approach, as stated in Table 28.

Table 28 - Learning rate by MLP variant

Variant	**Adam**	**SGD**
Value	0.124 +- 0.160	0.365 +- 0.280

On this test, the L2 penalty lowered for the Adam variant, meaning that larger weights were encouraged during training. On the SGD variant there was no significant difference (Table 29).

Table 29 - L2 penalty by MLP variant

Variant	**Adam**	**SGD**
Value	3.362 +- 2.724	4.759 +- 2.852

This time, regarding activation functions, the Adam variance balanced the Tahn activation function, preferring it over ReLU more often than on test one. The SGD variant shows exactly the same output of the previous test, shows Table 30.

Table 30 - Activation functions use by MLP variant

Variant	**Adam**			**SGD**		
	Logistic	**ReLU**	**Tahn**	**Logistic**	**ReLU**	**Tahn**
Value	2	15	13	0	17	13

The batch size results do not show much of a difference, having the Adam variant increased its batch size by a mere approximate 1% of the dataset. The SGD variant increasing the batch size as well by slightly over 2% of the dataset (Table 31).

Table 31 - Batch size by MLP variant

Variant	**Adam**	**SGD**
Value	18106.867 +- 8813.754	12790.167 +- 9397.114

The option to use batch shuffle or not remains fairly similar to the last test, this time the values approaching an almost equal distribution between both options Table 32.

Table 32 - Batch shuffle use by MLP variant

Variant	**Adam**		**SGD**	
	With shuffle	**Without shuffle**	**With shuffle**	**Without shuffle**
Value	14	16	14	16

On this test, the Adam variant leaned more into a slightly higher average number of hidden layers, still under the halfway of the minimum and maximum allowed values for it. SGD variant had an even smaller average increase, remaining just above the halfway point between the parameter limits Table 33).

Table 33 - Number of layers for each MLP variant considered

Variant	**Adam**	**SGD**
Value	2.233 +- 1.165	2.600 +- 1.567

Both variants increased their overall topology, having a higher number of overall neurons. The Adam variant reaching now approximately 105 neurons and the SGD variant approximately 100 neurons per hidden layer, on average Table 34.

Table 34 - Total number of hidden neurons for each MLP variant considered

Variant	**Adam**	**SGD**
Value	233.967 +- 140.452	268.400 +- 174.939

Regarding momentum on the SGD variant, the value remains fairly similar to the one obtained on the first test (Table 35). However, this time, the best performing runs had implemented more times the Nesterov's momentum, an increase of 13% of the number of times (Table 36).

Table 35 - Momentum in MLP SGD

Variant	**SGD**
Momentum	0.688 +- 0.260

Table 36 – Nesterov's momentum use in MLP SGD

Variant	**With**	**Without**
Value	12	18

Finally, evaluating the beta values of the Adam variant, their average is fairly similar to the one obtained on test one, as seen in Table 37 and Table 38.

Table 37 - Beta 1 in MLP Adam

Variant	**Adam**
Value	0.575 +- 0.265

Table 38 - Beta 2 in MLP Adam

Variant	**Adam**
Value	0.555 +- 0.256

4.2.3. Overall comparison

Following the same order of analysis of the first test, the first point to consider is the generalization performance of the best SLM outputs versus the best MLP's, represented on the boxplots of Figure 12.

This time, with more epochs of evolution, the discrepancy between SLM's results versus MLP's has increased. SLM's median value difference to MLP's one increased. This time, the consistency of the results is even more evident, with a very small difference between values of the 2nd and 3rd quartile, as well as a small difference between top and bottom quartile values on the SLM algorithm. On MLP however, the results appear to be fairly similar to the last tests results.

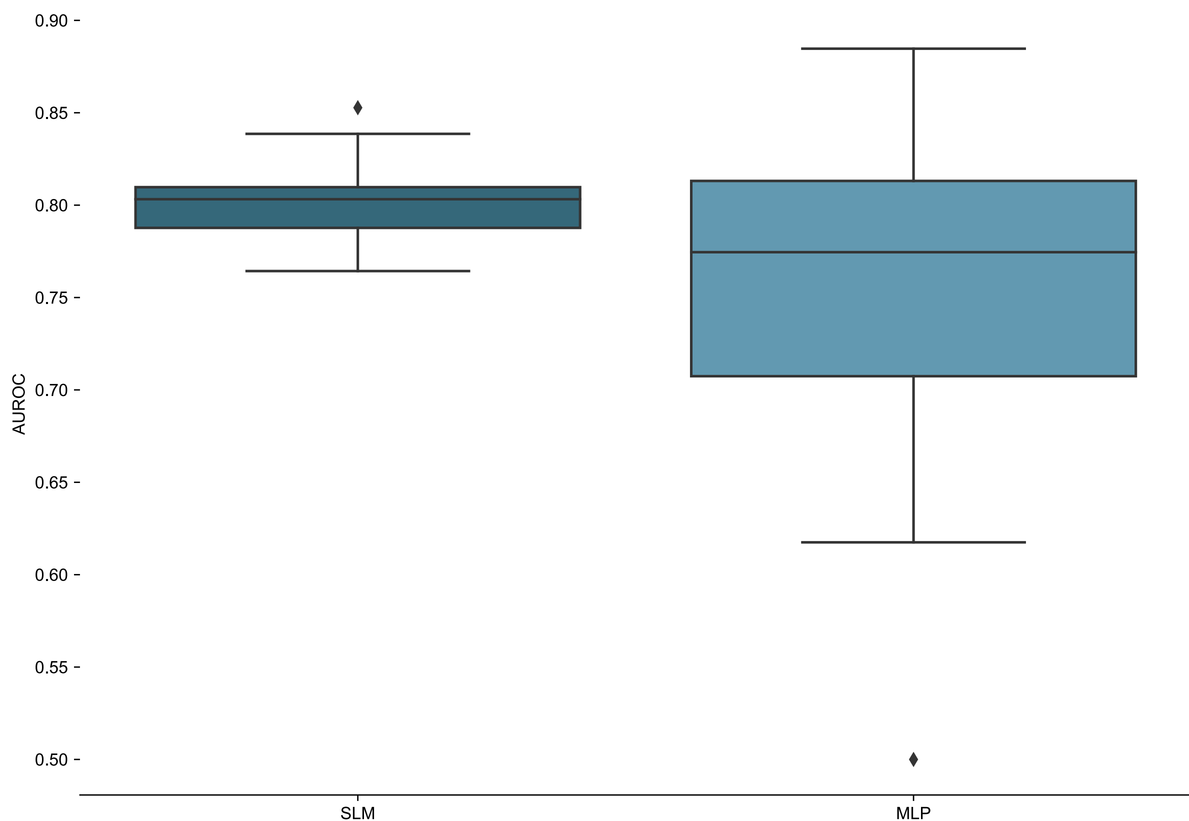

Figure 12 - Boxplots respective to the second test for the AUROC values of SLM and MLP algorithms respectively

The Student's T-test was again performed to assess statistical difference between the two algorithms. This time, the *p-value* calculated was 0.0074, which, by conventional criteria, is considered to be very statistically significant.

5. CONCLUSIONS

The main purpose of this project had two main points of focus:

- Expand and assess the viability of SLM to multilayer neural networks
- Evaluate SLM's competitiveness and generalization ability when facing a large dataset

Both of these objectives were met. Firstly, SLM was successfully applied to multilayer neural networks by adapting the mutation operator. The initial SLM design was to work only on neural networks with a single layer, adding neurons only to the single hidden layer. On this study, the mutation operator now handles multiple hidden layers, adding a random number of neurons to each one, controlled by prior parametrization.

The second objective was also accomplished, and actually produced great results for SLM. Under the same parametrization as MLP, a state-of-the-art competitor, SLM was able to outperform it on a large dataset, by having greater AUROC values, with a difference that was statistically significant. This translates as SLM being considered competitive while still being able to maintain a good generalization ability.

In order to have a significant and well performed set of tests, to produce credible and meaningful results, a stratified k-fold of 30 splits, with the cross validation of 2 splits was implemented. Both algorithms had the same parametrization, meaning that both were set to explore with the same conditions, guarantying a fair comparison between them. These features were, however, conditionings of the practical study, specifically on the computational cost associated with them.

Test results also showed, as the analysis of the number of iterations detailed, that SLM was still restricted in terms of training epochs. As MLP's results were fairly identical, this opens up the possibility of achieving SLM possibly achieving even greater results if the number of training iterations was raised.

The work performed on this project provided good results and serves as a solid basis for future work. It would be a nice exploratory option to expand the subject of this study.

Some exploratory options to be considered are:

- Perform the same test using a higher number of training iterations. Previous work on GSPG had it tested against other algorithms on runs of up to 5000 iterations
- Explore the results of setting specific activation functions on the neurons (neurons from the initial individual, neurons from the mutation, neurons of just a specific layer, etc.)
- Apply SLM to multilayer neural networks with multiple output neurons
- Assess SLM's competitive viability against datasets of other subjects

6. BIBLIOGRAPHY

Angeline, P. J. (1994). *Genetic programming: On the programming of computers by means of natural selection,*. *Biosystems* (Vol. 33). https://doi.org/10.1016/0303-2647(94)90062-0

Arp, D., Spreitzenbarth, M., Malte, H., Gascon, H., & Rieck, K. (2014). DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket. *Choice Reviews Online*. https://doi.org/10.5860/choice.45-0765

Baldi, P. F., & Hornik, K. (1995). Learning in Linear Neural Networks: A Survey. *IEEE Transactions on Neural Networks*. https://doi.org/10.1109/72.392248

Banzhaf, W. (1993). Genetic Programming for Pedestrians. In *Proceedings of the 5th International Conference on Genetic Algorithms, ICGA-93*.

Banzhaf, W., Martín-Palma, R. J., & Banzhaf, W. (2013). Evolutionary Computation and Genetic Programming. *Engineered Biomimicry*, (July 2012), 429–447. https://doi.org/10.1016/B978-0-12-415995-2.00017-9

Bello, M. G. (1992). Enhanced Training Algorithms , and Integrated Training / Architecture Selection for Multilayer Perceptron Networks, *3*(6), 864–875.

Bosman, A., Engelbrecht, A., & Helbig, M. (2017). Fitness Landscape Analysis of Weight-Elimination Neural Networks. *Neural Processing Letters*. https://doi.org/10.1007/s11063-017-9729-9

Bottou, L. (2012). Stochastic gradient descent tricks. *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. https://doi.org/10.1007/978-3-642-35289-8-25

Breiman, L. E. O. (2001). Random Forests, 5–32. https://doi.org/10.1023/A:1010933404324

Burkov, A. (2018). *Machine learning*. *Expert Systems* (Vol. 1). https://doi.org/10.1111/j.1468-0394.1988.tb00341.x

Buscema, P. M., Massini, G., Breda, M., Lodwick, W. A., Newman, F., & Asadi-Zeydabadi, M. (2018). Artificial neural networks. In *Studies in Systems, Decision and Control*. https://doi.org/10.1007/978-3-319-75049-1_2

Canfora, G., Medvet, E., Mercaldo, F., & Visaggio, C. A. (2015). Detecting Android malware using sequences of system calls, 13–20. https://doi.org/10.1145/2804345.2804349

Canfora, G., Medvet, E., Mercaldo, F., & Visaggio, C. A. (2016). Acquiring and Analyzing App Metrics for Effective Mobile Malware Detection, 50–57. https://doi.org/10.1145/2875475.2875481

Caruana, R., & Niculescu-Mizil, A. (2006). An empirical comparison of supervised learning algorithms. *Proceedings of the 23rd International Conference on Machine Learning*, *C*(1), 161–168. https://doi.org/10.1145/1143844.1143865

Castelli, M., Silva, S., & Vanneschi, L. (2015). A C++ framework for geometric semantic genetic programming. *Genetic Programming and Evolvable Machines*, *16*(1), 73–81. https://doi.org/10.1007/s10710-014-9218-0

Castelli, M., Vanneschi, L., & Silva, S. (2014). Prediction of the Unified Parkinson's Disease Rating Scale assessment using a genetic programming system with geometric semantic genetic operators. *Expert Systems with Applications*, *41*(10), 4608–4616.

https://doi.org/10.1016/j.eswa.2014.01.018

Cunningham, P., Cord, M., & Delany, S. J. (2016). Supervised learning, (May).

Fan, W., Gordon, M. D., & Pathak, P. (2005). Genetic Programming-Based Discovery of Ranking Functions for Effective Web Search. *Journal of Management Information Systems*, *21*(4), 37–56. https://doi.org/10.1080/07421222.2005.11045828

Gershenson, C. (2003). Artificial Neural Networks for Beginners. In *Artificial Neural Networks for Beginners*. https://doi.org/10.1093/icesjms/fsp009

Gonçalves, I. (2016). *An Exploration of Generalization and Overfitting in Genetic Programming : Standard and Geometric Semantic Approaches*. Coimbra.

Gonçalves, I., Silva, S., & Fonseca, C. M. (2015a). On the Generalization Ability of Geometric Semantic Genetic Programming. *Springer International Publishing Switzerland 2015*, *9025*, 41–52. https://doi.org/10.1007/978-3-319-16501-1

Gonçalves, I., Silva, S., & Fonseca, C. M. (2015b). Semantic Learning Machine: A Feedforward Neural Network Construction Algorithm Inspired by Geometric Semantic Genetic Programming. *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, *9273*, 280–285. https://doi.org/10.1007/978-3-319-23485-4

Gonçalves, I., Silva, S., Fonseca, C. M., & Castelli, M. (2016). Arbitrarily Close Alignments in the Error Space: a Geometric Semantic Genetic Programming Approach. *GECCO '16 Companion: Proceedings of the Companion Publication of the 2016 Annual Conference on Genetic and Evolutionary Computation*, 99–100. https://doi.org/doi:10.1145/2908961.2908988

Gonçalves, I., Silva, S., Fonseca, C. M., & Castelli, M. (2018). Unsure When to Stop ? Ask Your Semantic Neighbors. *GECCO '18, Berlin, Germany, July 15-19, 2017*. https://doi.org/10.1145/3071178.3071328

Haykin, S. (2004). Neural Networks: A comprehensive foundation. *Neural Networks*. https://doi.org/0-13-273350-1

Herculano-Houzel, S. (2009). The human brain in numbers: a linearly scaled-up primate brain. *Frontiers in Human Neuroscience*, *3*(November), 1–11. https://doi.org/10.3389/neuro.09.031.2009

Jagusch, J., Gonçalves, I., & Castelli, M. (2018). Neuroevolution under Unimodal Error Landscapes : An Exploration of the Semantic Learning Machine Algorithm. *Gecco*, 159–160. https://doi.org/10.1145/3205651.3205778

Kingma, D. P., & Ba, J. L. (2015). Adam : A Method For Stochastic Optimization, 1–15.

Kotsiantis, S. B. (2007). Supervised Machine Learning: A Review of Classification Techniques. *Informatica*, *31*, 249–268. https://doi.org/10.1115/1.1559160

Koza, J. R. (1992). Genetic programming: on the programming of computers by natural selection. *Cambridge,MA:MITPress*, *4*(2), 87–112. Retrieved from http://refhub.elsevier.com/S0019-0578(14)00059-7/sbref32

Koza, J. R., Keane, M. A., & Streeter, M. J. (2003). What's Al Done for Me Lately? Genetic Programming's Human-Competitive Results. *IEEE Intelligent Systems*. https://doi.org/10.1109/MIS.2003.1200724

Krawiec, K., Moraglio, A., Hu, T., Etaner-Uyar, A. S., & Hu, B. (2013). *Genetic Programming*.

McCulloch, W. S., & Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics*. https://doi.org/10.1007/BF02478259

Miller, J. F. (1999). An empirical study of the efficiency of learning boolean functions using a Cartesian Genetic Programming approach. *Proceedings of the Genetic and Evolutionary Computation Conference*.

Moraglio, A., Krawiec, K., & Johnson, C. G. (2012). Geometric semantic genetic programming. *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, *7491 LNCS*(PART 1), 21–31. https://doi.org/10.1007/978-3-642-32937-1_3

Nesterov, Y. (2004). *Introductory lectures on convex optimization: A basic course*. *Operations Research*. https://doi.org/10.1007/978-1-4419-8853-9

Patterson, J., & Gibson, A. (2017). *Deep Learning*. (M. Loukides & T. McGovern, Eds.), *Data Science* (1st ed.). O'Reilly Media. https://doi.org/10.1016/b978-0-12-814761-0.00010-1

Pedregosa, F., Weiss, R., & Brucher, M. (2011). Scikit-learn : Machine Learning in Python, *12*.

Phaisangittisagul, E. (2016). An Analysis of the Regularization between L 2 and Dropout in Single Hidden Layer Neural Network. https://doi.org/10.1109/ISMS.2016.14

Poli, R., Langdon, W. B., McPhee, N. F., & Koza, J. R. (2008). *Field Guide to Genetic Programing*. *Wyvern*. https://doi.org/10.1016/S0021-9991(03)00029-9

Priddy, K. L., & Keller, P. E. (2009). *Artificial Neural Networks: An Introduction*. *Artificial Neural Networks: An Introduction*. https://doi.org/10.1117/3.633187

Raul Rojas. (1996). Neural Networks: A Systematic introduction. *Neural Networks*. https://doi.org/10.1016/0893-6080(94)90051-5

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (2013). Learning Internal Representations by Error Propagation. In *Readings in Cognitive Science: A Perspective from Psychology and Artificial Intelligence*. https://doi.org/10.1016/B978-1-4832-1446-7.50035-2

Srinivas, S., Subramanya, A., & Babu, R. V. (2016). Training Sparse Neural Networks. Retrieved from http://arxiv.org/abs/1611.06694

Stanley, K. O., & Miikkulainen, R. (2002). Evolving Neural Networks through Augmenting Topologies. *Evolutionary Computation*, *10*(2), 99–127. https://doi.org/10.1162/106365602320169811

Sutskever, I., Martens, J., Dahl, G. E., & Hinton, G. E. (2013). On the importance of initialization and momentum in deep learning. *Proceedings of the 30th International Conference on Machine Learning Jmlr W&Cp*. https://doi.org/10.1109/ICASSP.2013.6639346

Tikhonov, A. N. (1943). On the stability of inverse problems. *Dokl. Akad. Nauk SSSR*.

Tomomura, M., & Nakayama, K. (2004). An internal information optimum algorithm for a multilayer perceptron and its generalization analysis. *Electronics and Communications in Japan, Part II: Electronics (English Translation of Denshi Tsushin Gakkai Ronbunshi)*, *87*(5), 67–80. https://doi.org/10.1002/ecjb.20077

Turing, A. (1950). Computing machinery and intelligence.

Vanneschi, L. (2017). An Introduction to Geometric Semantic Genetic Programming. https://doi.org/10.1007/978-3-319-44003-3_1

Vanneschi, L., Castelli, M., Manzoni, L., & Silva, S. (2013). A new implementation of geometric semantic GP and its application to problems in pharmacokinetics. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* (Vol. 7831 LNCS, pp. 205–216). https://doi.org/10.1007/978-3-642-37207-0_18

Wanto, A., Zarlis, M., Sawaluddin, & Hartama, D. (2017). Analysis of Artificial Neural Network Backpropagation Using Conjugate Gradient Fletcher Reeves in the Predicting Process. *Journal of Physics: Conference Series*, *930*(1). https://doi.org/10.1088/1742-6596/930/1/012018

Wilson, A. C., Roelofs, R., Stern, M., Srebro, N., & Recht, B. (2017). The Marginal Value of Adaptive Gradient Methods in Machine Learning, (Nips). https://doi.org/10.1016/j.expthermflusci.2014.11.005

Yao, X. (1993). A Review of Evolutionary Artificial Neural. *International Journal of Intelligent Systems*, 40.

Yao, X. (1999). Evolving artificial neural networks. *Proceedings of the IEEE*, *87*(9), 1423–1447. https://doi.org/10.1109/5.784219