Deep Representation Learning with Genetic Programming

Lino A. Rodríguez-Coayahuitl, Hugo Jair Escalante,
Alicia Morales-Reyes

Technical Report No. CCC-17-009
January 17, 2018

Department of Computational Sciences
National Institute for Astrophysics Optics and Electronics
Tonantzintla, Puebla

Deep Representation Learning with Genetic Programming

Lino Alberto Rodríguez Coayahuitl, Hugo Jair Escalante, Alicia Morales Reyes

Computer Science Department
National Institute of Astrophysics, Optics and Electronics
Luis Enrique Erro # 1, Santa María Tonantzintla, Puebla, 72840, México

Abstract

A representation can be seen as a set of variables, known as features, that describe a phenomenon. Machine learning (ML) algorithms make use of these representations to achieve the task they are designed for, such as classification, clustering or sequential decision making. ML algorithms require compact, yet expressive, representations; otherwise they might take too much time to return an output, or not have enough information to correctly discriminate the phenomena being presented with. Representation learning is the subfield of computer science that deals with the automatic generation of representations (as opposed to human engineered representations). Representation learning has achieved notoriously good results in the recent years thanks to the emergence of massive layered structures comprised of non-linear operations, known as deep architectures, such as Deep Neural Networks (DNN). DNN, and other deep architectures alike, work by gradually reducing and abstracting the input representation in each successive layer. In this technical report, we describe a research proposal to develop a new type of deep architecture for representation learning, based on Genetic Programming (GP). GP is a machine learning framework that belongs to evolutionary computation. GP has already been used in the past for representation learning; however, many of those approaches required of human experts knowledge from the representations' domain. In this proposal, we explore the pitfalls of developing representation learning systems with GP that do not require experts' knowledge, and propose a solution based on layered GP structures, similar to those found in DNN.

Keywords: machine learning, representation learning, feature extraction, genetic programming, evolutionary computation.

Contents

1. Introduction

Machine learning is the field of computer science concerned with the development of algorithms and software agents that master a task through the acquisition of experience from repeatedly executing the task or by observation of an expert (Mitchell, 1997). There are different classes of machine learning algorithms, such as supervised learning, unsupervised learning, reinforcement learning algorithms, among others (Ayodele, 2010; Alpaydin, 2014).

In supervised learning, an algorithm is presented with several examples of the task it must learn, i.e. many inputs-outputs pairs, and the algorithm then must discover a model that correctly maps all the inputs with their corresponding outputs (Littman & Isbell, 2015). This model can be an abstract mathematical function or a computer algorithm that memorizes and leverages on the examples so far known. The idea is that this model is robust enough so when new unlabeled instances of the task are presented, the algorithm is capable of correctly predicting their output. *Classification* is a an example of a supervised learning task. In image classification an algorithm is presented with several sets of images, each image belonging to a particular class (e.g., images of boats, trees, buildings, etc.). The algorithm is informed of the class each image belongs to, so it can build a model that correlates each image to the corresponding label, and that can be used later to match new unlabeled images. Regression is another supervised learning task, similar to classification, where outputs are continuous instead of discrete.

A very special type of supervised learning is *self-supervised* learning. In self-supervised learning, the labels for training are generated from the samples themselves (Chollet, 2016). Autoencoders are a prime example of self-supervised learning (Chollet, 2016; Plaut, 2016; Gogna & Majumdar, 2016). An autoencoder is an algorithm that both compresses and decompresses input data. Autoencoders are trained to achieve the highest fidelity in decompressed reconstructions, given some degree of compression required.

On the other hand, in unsupervised learning, the algorithms are not presented with labeled pairs of input-output, but instead, it is entirely up to the algorithm to find a model that describes structures or patterns found in the input data. An example of unsupervised learning is association rule mining (Agrawal *et al.*, 1993), which consist in finding interesting (given some criteria for interestingness) relationships between variables in databases (e.g. in a database of supermarket sales, the products that are likely to be bought together). Another common task of unsupervised learning is clustering, which consists in grouping the input samples in such a way that objects belonging to the same group are similar, in some way or another. Unsupervised learning can be a task in itself, like in the association rules example, but it is also commonly an intermediate step to preprocess data that later will be feed to supervised learning algorithms: unsupervised learning algorithms can remove redundancies in the sample data, making it more compact and easier to process for supervised learning algorithms.

Reinforcement learning (RL) algorithms concern with tasks involving sequential decision making (Sutton & Barto, 1998), for example, learning to play a board game, driving a car, etc. RL algorithms interact with an environment and, according to the actions they take, receive a feedback from the environment in the form of a reward, in every time step (in whatever form the time is discretized) or for every action they take; their objective is to discover a model that allows them to take actions -given any possible state of the world they interact with- such that the average accumulative reward they get is maximized over an hypothetical infinite time horizon.

A question that remains for all types of machine learning scenarios is, how exactly the input data -that is, the samples in supervised learning or the state of the world in RL- are presented to an algorithm? In what

form are the data fed to the machine learning algorithms? Because, if, for example, a classifier was to learn to distinguish cancer cells from healthy cells, it would be easier for the algorithm if just vectors of numbers describing shape, size and symmetry of cells were feed to it, instead of the raw images of cells. Similarly, a software agent designed to learn to play the game of Go, would have a head-on start if the world's state were a matrix that represented the current state of the board instead of a live video feed of the very board itself. On the other hand, what would happen if vectors describing cells did not have enough information to discern a healthy from a cancer cell? Then the algorithm would not learn to correctly classify cells.

The area that deals with these issues is *feature engineering*. A *feature* is an individual measurable property or characteristic of the phenomenon being observed (Bishop, 2006), e.g. an input variable from the samples that we might be trying to classify. The vector of features that entirely describe the data that a machine learning algorithm needs to process is known as *representation*. Classification, reinforcement learning, and clustering algorithms require compact, yet descriptive, representations, because they suffer from a phenomena known as *the curse of dimensionality* (Bellman, 1961), which means that execution time and/or the amount of necessary training data of these algorithms grows exponentially with respect to the representation size (Hughes, 1968; Bishop, 2006). This often require for representations to be carefully handcrafted by experts of the problem's domain.

On the other hand, *representation learning* is a set of methods that allow a machine to be fed with raw data and automatically discover representations needed for detection or classification (LeCun *et al.*, 2015). Representation learning methods can be broadly classified into two categories: feature selection and feature extraction. Feature selection methods attempt to reduce dimensionality of the representation by selecting a subset of features from the original feature space. They achieve this by discarding redundant or low variance features (unsupervised feature selection), or by heuristically improving the subset by measuring a classifier performance fed with it (supervised). In contrast, feature extraction methods attempt to reduce the feature space by generating a new set of features as a result of linear and non-linear transformations of the original ones. This new representation not only is more compact, but can also be more descriptive, in a sense that generalize better the sample data, clean it from noise, etc. Feature extraction methods are: matrix factorization (Lee & Seung, 1999, 2001), Linear discriminant analysis (Mika *et al.*, 1999), Principal component analysis (Wold *et al.*, 1987), artificial neural networks (ANN) (Williams & Hinton, 1986; Lecun *et al.*, 1998; Hinton & Salakhutdinov, 2006b), among others.

In recent years ANN have automatically produced representations that significantly boosted the accuracy of classification systems (Ciregan *et al.*, 2012; Krizhevsky *et al.*, 2012; Sermanet *et al.*, 2012). These ANN consist of several stacked layers of non-linear transformations. These so called Deep Neural Networks (DNN), were previously thought too slow to train or to converge to very poor solutions when consisted in more than two or three layers (Hinton & Salakhutdinov, 2006a); but recent advances in the area now allow to train DNN of several hundred layers (Huang *et al.*, 2016). Also, latest massively parallel general purpose graphics processing units (GPU) allowed to train these DNN with very large training datasets (Krizhevsky *et al.*, 2012). DNN work by generating a more compact and more abstract representation in each forward layer, thus at the final layer data should be clear enough for discrimination by a classification algorithm (LeCun *et al.*, 2015).

Another computational tool that has been used for representation learning is *Genetic Programming* (Koza, 1992). Genetic Programming (GP) belongs to the class of Evolutionary Elgorithms (EA) that search for a solution, in this case a mathematical model or very simple computer programs, through the use of processes that mimic natural evolution such as, small mutations of candidate solutions (*individuals*), crossover of good performance individuals among a population, and discarding poor performing candidate solutions (survival

evolutionary pressure).

Even though GP has been used in the past for representation learning (Trujillo & Olague, 2006; Shao *et al.*, 2014; Liu *et al.*, 2015), many of those approaches require the designer of the system to indirectly provide some high level information, i.e. experts' knowledge of the problem domain is still required. Attempts to produce GP representation learning systems that do not require experts knowledge have not yield results competitive with other state of the art representation learning methods when faced with high dimensionality problems (Parkins & Nandi, 2004; Limón *et al.*, 2015).

In this research proposal we formulate a framework for representation learning based on genetic programming, that aims at tackling the disadvantages of previous approaches based on GP (i.e., the need of experts knowledge when dealing with high dimensionality problems and efficiency issues). Our proposed framework is inspired by the deep architectures briefly discussed before, in the sense that we pose that successive layers of representations can be generated through GP that gradually reduce and abstract the initial input representation, up to the point where an useful representation for other ML tasks is obtained. We call this approach *Deep Genetic Programming*; to the authors' best knowledge, no similar approach has been documented before.

The remainder of this document is organized as follows: in Section 2 GP is introduced in detail, together with a new taxonomy to classify current GP works applied to representation learning; in Section 3 we briefly review, leveraging from our proposed taxonomy, some hallmark research done related to representation learning with GP; in Section 4 we formally state our research proposal; in Section 5 we present some early results to support our research hypothesis; and finally in Section 6 we provide some final thoughts and concluding remarks.

2. Background

In this section we briefly present deep learning and then offer a complete overview of the core concepts of Genetic Programming (GP). Unlike Genetic algorithms (GA), which are a parameter-only optimization method, GP allows to discover and optimize complete mathematical and computational models that best describe some desired phenomena. As a machine learning technique, GP has been successfully used in regression tasks (Koza, 1992; Vladislavleva, 2008), classification (Koza, 1994; Loveard & Ciesielski, 2001) and representation learning (see Sec. 3).

This section is divided as follows: first, we review some of the main ideas behind deep learning, then we present the typical architecture of genetic programs, and finally we introduce a new proposal for a taxonomy of *primitives* used in genetic programs developed for representation learning. This taxonomy is required to better understand the state of the art of representation learning through GP reviewed in Sec. 3.

2.1. Deep Learning

Deep learning is a set of representation learning methods known in general as *deep architectures* (Bengio *et al.*, 2009). DNN are an example of such class of algorithms. Deep architectures consist in several stacked layers, each layer consisting of non-linear operations (Bengio *et al.*, 2009), where these operations share connections between them, serving ones' outputs as others' inputs. Example of deep architectures, other than

DNN, are *Deep Belief Networks* (Hinton & Salakhutdinov, 2006b), *Deep Boltzmann Machines* (Salakhutdinov & Hinton, 2009), *Deep Gaussian Processes* (Damianou & Lawrence, 2013), among others (Tang, 2013). This research proposal centers around the idea of a new deep architecture, based on *Genetic Programming*.

2.1.1. Deep Learning with Neural Networks

ANN are function approximation algorithms represented by a collection of interconnected, simple units, called *artificial neurons*. ANN can be *trained* to perform non-linear transformations over the data they receive as input. ANN can be trained in a supervised learning fashion, where they are presented with thousands of labeled samples, that are used to calibrate the weights of interconnections of artificial neurons within them, until the output they generate correctly labels all samples from the training set. So, for example, an ANN can be trained to perform image classification by transforming a set of images it receives as input, into a vector that signals the corresponding label of each image.

The capabilities of ANN for transforming the input data are directly related to the amount of stacked layers of neurons within them.

Deep neural networks (DNN) are nothing but ANN with multiple stacked layers of neurons, which can be used to learn highly nonlinear representations. These DNN can receive as input raw data, such as pixels from an image, and generate a more abstract representation, that is manageable for classification or clustering algorithms. They work by transforming the representation at one level (starting with the raw input) into a representation at a higher, slightly more abstract level. With the composition of enough such transformations, very complex functions can be learned (LeCun *et al.*, 2015). The key aspect of deep learning is that these layers of features are not designed by human engineers: they are learned from data using a general-purpose learning procedure (LeCun *et al.*, 2015).

Motivated by the astounding results for representation learning obtained by deep networks (see Sec. 3.1), we were inspired to bring a methodology that just like DNNs is capable of learning highly nonlinear functions: we propose a structured, deeply layered, approach for representation learning based in GP.

2.2. Genetic Programming

GP is an evolutionary computation technique,such as Genetic Algorithms (GA), Evolutionary Strategies (ES) (Beyer & Schwefel, 2002) or Differential Evolution (DE) (Storn & Price, 1997), aimed at automatically solving problems, even if the user does not know the structure of the solution in advance (Poli *et al.*, 2008). It shares the main components of evolutionary algorithms: a population composed of individuals -that represent candidate solutions- and a set of operators that transform both the individuals and the population (Koza, 1994).

2.2.1. Individual Representation

An individual in GP is a candidate solution to the phenomenon being modeled or problem being tackled. Traditionally, individuals in GP take the form of a tree that represents a mathematical function. Fig. 1 shows an example of a tree structure that represents the function $f(x, y) = (2.2 - (\frac{x}{11})) + (7 * \cos(y))$ (Axelrod, 2007).

In a GP tree, every node represents itself a function. Functions represented by nodes are known as *primitives*. Internal nodes represent typical functions, in the sense that these are functions which take as

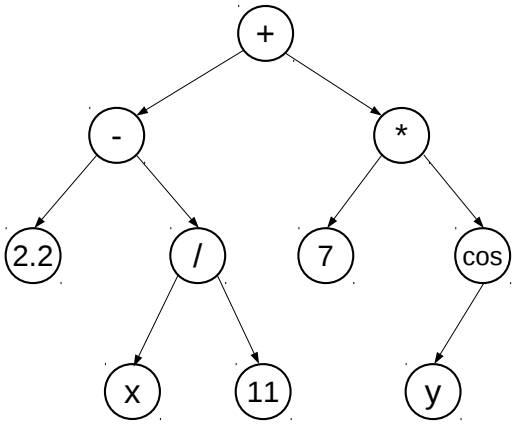

Figure 1: Tree structures like this are typically used in GP to represent individual candidate solutions. Nowdays modern GP approaches make use of multiple trees to represent a single individual solution. In the example pictured above, the direction of the arrows indicate the descendants of each node, but the data flows in the inverse direction.

input some arguments, apply some transformation over them, and return an output for further processing. Values returned by the children of a given node are input arguments for the function such node represents. On the other hand, leaf nodes are what it is known as *zero-argument functions*[1] . These nodes do not take any input, as they can be constant values or variables taken directly from the problem being modeled. In the context of machine learning and representation learning, these variables can be taken from the feature set.

Tree structures can be directly used for scalar function regression; however, they fall short in capabilities for other tasks. State-of-the-art GP makes use of forests, a set of trees, to represent a single candidate solution, that is, multiple trees acount for a single individual. Forest naturally extend the capabilities of GP to tackle problems as diverse as vector function regression, prototype generation, representation learning, among others.

Nevertheless, it is still important to understand the tree structure as the backbone of individual representation in GP. Many GP operators operate at a tree and at a node level, even when individuals are comprised of several trees.

2.2.2. GP Operators

GP operators are functions that transform individuals, and the population in general, in order to find better performing individuals for the desired task being tackled. Similarly to standard GAs, the typical operators in GP are *mutation*, *crossover*, and *selection* (Koza, 1994; Poli *et al.*, 2008).

GP operators are applied at an individual level implicitly modifying the entire population. Through evolution, better individuals are generated to improve performance of the tackled problem.

Mutation is a random, partial change, to an individual. In evolutionary algorithms mutation is associated to exploiting the search space and therefore increasing the selective pressure. When dealing with single tree

[1] They are also called *terminals* in many works, but we will stick with the term zero-argument functions.

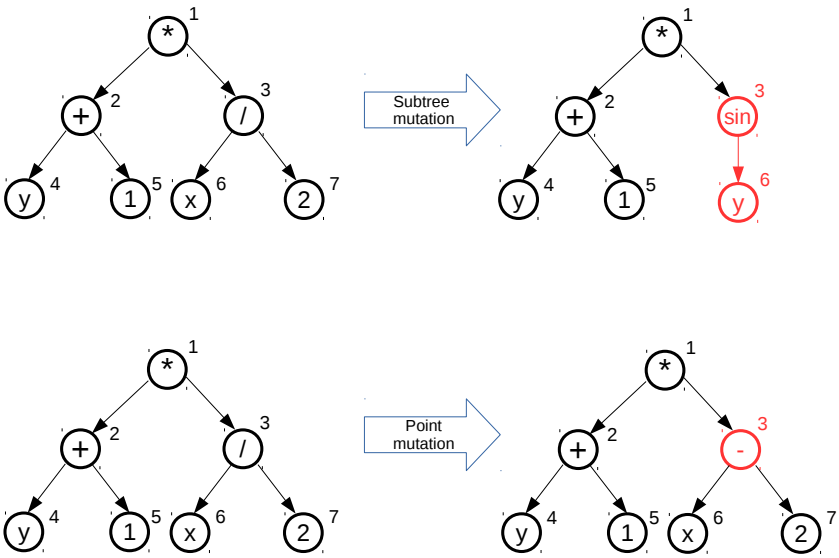

Figure 2: In subtree mutation, for example, node 3 is randomly selected for mutation. In the case of subtree mutation, the entire subtree rooted at node 3, along with such node, is replaced by a completely new random subtree. In the case of point mutation, only node 3 is replaced by a compatible (in terms of number of inputs) primitive.

individuals, there are two main types of mutation operators: subtree mutation, and point mutation. Subtree mutation consists in randomly choosing a node and replace the subtree rooted at such node with an entirely new random subtree. Similary, point mutation consists in choosing a random node and replace it with another, arity-compatible, primitive. Fig. 2 illustrate both type of mutations.

When individuals are forests, mutation can take other forms. Subtree and point mutation can still be used, but complete replacement of trees in forests with new random ones can also be a form of mutation in this case. Fig. 4a illustrates such a forest mutation.

In evolutionary algorithms, crossover is an exploratory genetic operator that promotes exploration of the landscape. In GP the most typical form of crossover, when dealing with single tree individuals, is subtree swap. In this form of crossover, two trees (candidate solutions) are picked among the current population. In each one of them, a node is randomly chosen, so subtrees rooted at such nodes are exchanged among them. The result of such crossover is two new trees. Fig. 3 shows an example of subtree swap crossover.

When individuals are forest, crossover can be the exchange of complete trees among forests. Fig. 4b and c illustrate two types of forest crossovers.

The main idea behind crossover is that it is possible to reach near-optimal solutions by combining quality chunks found among diverse individuals in the population. This approach has proven successful in GA, however it still not clear if techniques such as subtree swap is as effective in GP. In fact, some research suggest subtree swap does not help to achieve better results than subtree mutation (Parkins & Nandi, 2004).

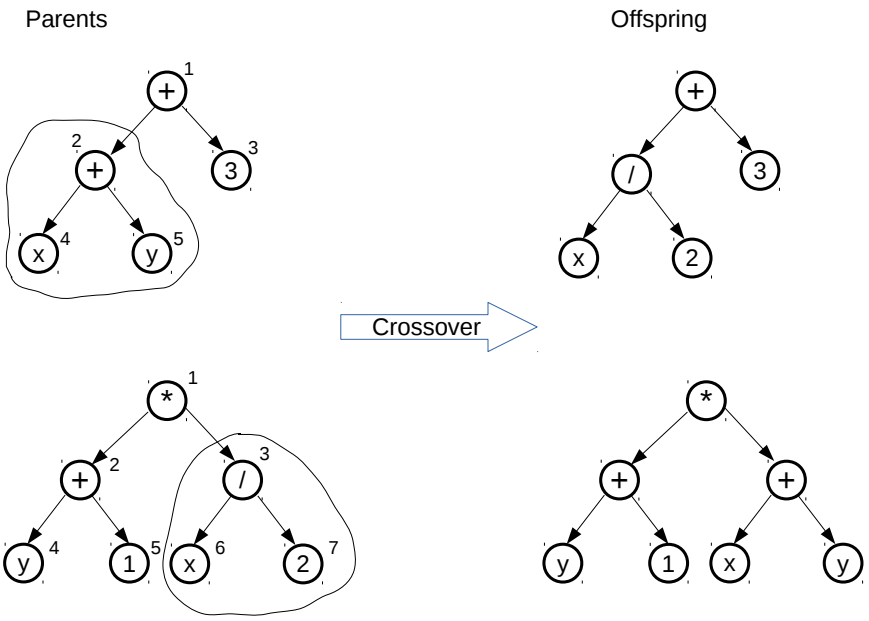

Figure 3: Two trees are used for subtree crossover. For each tree, a random node is selected as crossover point, and the subtrees rooted in such nodes are exchanged, resulting in two new trees.

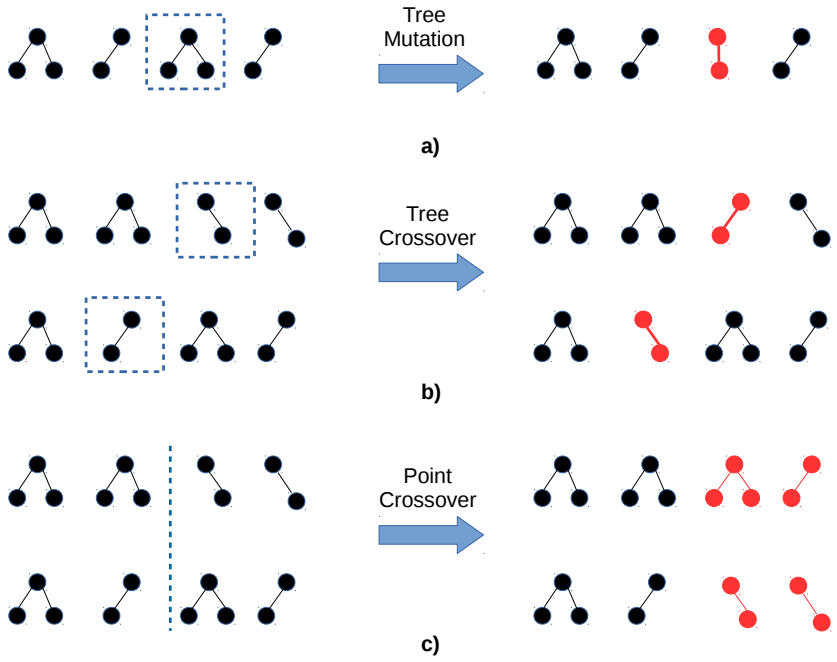

Figure 4: Different type of evolutionary operations when individuals are comprised of more than one tree: a) tree mutation, a tree in the individual is randomly selected and replaced with a new, random tree; b) tree crossover, two trees are exchanged among two individuals; c) point crossover, a random point is selected in the list of trees that comprises the forest, and all trees from that point onwards are exchanged with another individual.

2.2.3. Selection

The process of selecting individuals that will pass to the next generation in EA is sometimes considered also an operator, since some of these mechanisms take as input a number of individuals and return as output a new set of individuals, very much like mutation and crossover operators.

In GP the selection mechanism are the same of those found in GA, such as *roulette-wheel selection*, *tournament selection* and *truncation selection*, to name a few. To use selection operations we first need to define how many individuals in the current population will pass to the next generation and how many will be replaced with new individuals (generated through the operators). Once we have chosen this parameter, in the case of a basic roulette-wheel selection mechanism, we would have to order the individuals in the current population according to their performance (in whatever task we are trying to achieve) and randomly pick a position in the ordered list. This individual passes to the next generation. The random pick can be biased towards the top of the list. We repeat this procedure until we have picked up all the individuals that will survive to the next generation. In tournament selection we randomly choose n individuals from the current generation, where n is usually a small number ranging anywhere between 2 and 5, the best performing individual from this small group survives to the next generation. We repeat the procedure until we picked up all the individuals that will make it to the next population. In truncation selection we simply order the current individuals according to their performance and keep the amount of best performers necessary to make it up for the next generation.

2.2.4. GP Parameters

The purpose of GP is to find a model that describes, predicts or makes discriminations of a phenomenon we are interested in. To achieve this, GP generates an initial population of random candidate solutions (individuals). These initial solutions will probably be very bad at performing the task we are interested; however, through a process that mimics natural evolution with the help of crossover and mutation operators, and selection mechanisms, GP will gradually improve the performance of individuals that comprise the population, through this artificial evolutionary search. The available parameters that govern this evolutionary search vary depending on the architecture we set for the evolutionary search. Fig. 5a shows the most simple of the possible evolutionary architectures. In this architecture, we have to define three parameters: the size of the population (both initial and for every generation), the probability of crossover and the probability of mutation. The probability of crossover (mutation) refers to the proportion of individuals in the population that will undergo crossover (mutation). These individuals are picked up randomly, but the proportions must hold. The offspring generated are added up to the population to generate an extended population pool. Then, the selection mechanisms prunes the extended population to fit it to the population size and generate in that way the next population. Traditionally, the crossover and mutation probabilities are fixed to add up to one, so the extended population is twice as big as the population size.

A slightly more advance evolutionary architecture is depicted in Fig. 5b. In this setup the offspring are not added up to the population pool, but instead they form their own pool of individuals. There is an additional parameter that informs the selection mechanism of how many individuals must be chosen from the original population pool and how many from the offspring pool. This parameter allows to fine tune the degree of diversity introduced to each new generation, which in turns controls the exploration/exploitation preferences. When the selection mechanism is informed to replace only one (all but one) individual(s) from the original population with one (all but one) offspring, the setup is known as steady state (generational replacement).

A third possibility of evolutionary architecture is shown in Fig. 5c. In this setup, the selection mechanism not only selects individuals from the original population that will make it to the next, but in the process, it also selects the individuals that will be allowed to breed offspring, that is because individuals from the population that get selected to pass to the next one, are the only ones that will generate offspring through mutation and crossover operators. A parameter determines for individuals in the population the probability of being selected to pass to the next, i.e. the proportion of individuals from the next generation that are directly taken from the current generation. The rest of the next population is generated by crossover and mutation over these very same individuals, in amounts corresponding to their respective probabilities. These setup allows to calibrate the diversity forcedly introduced to the population, like in the previous setup, but also limits the amounts of individuals that need to be evaluated in each generation, by directly discarding poor performing individuals in the population without using them to breed offspring. This shortcut can be beneficial in machine learning contexts, where we need to evaluate the performance of all individuals, with massive amounts of training samples.

The final two parameters needed for a GP in all scenarios are the termination criteria and the maximum allowed depth of the candidate solutions (trees or trees in forests). The termination criterion typically being the maximum number of generations allowed, or when an individual hits the global optimum, if we know such value.

2.2.5. Fitness evaluation

Every individual in a GP is evaluated against an objective function. The result of this evaluation is assigned as a *fitness* value to the corresponding individual in the population. This evaluation process is repeated in every generation, and this fitness value is used by the selection mechanism to choose highly fitted (not the best, since it is not a fully greedy process) individuals in a population that will pass to the next generation or that will be allowed to breed offspring to generate new individuals. This fitness measure along with the selection mechanism generate the evolutionary pressure that pushes the overall population towards promising areas in the search space, and that eventually yield acceptable or near-optimal solutions to the problem at hand. The objective function works as the *high level question* we want the GP to answer (Poli *et al.*, 2008).

Some researchers hold that GP is quite ”resistant” to parameter selection (Poli *et al.*, 2008), and that GP will workout acceptable solutions one way or another given the wide range of possible parameters combinations (population sizes, operators probability, etc.). This is not the case with the objective function. The objective function is perhaps the most critical part of the GP. The wrong objective function will stagnate the evolutionary search, landing at very poor local optima. A slightly modified version of such an objective function might yield way better results, if defined or formulated in a way that is more informative to the evolutionary search. Objective function design is critical both to machine learning algorithms and evolutionary algorithms, GP being both of them, makes objective functions' design twice as important. There is no magic formula for designing or selecting the right objective functions, but some general guidelines might be followed, e.g. avoiding binary objective functions (one that returns simply right or wrong), converting equality to inequality constraints, etc.

It is important to remember that a GP candidate solution will actually be tested against a set of samples, whether in training or in testing phases. That means the actual objective function will typically be maximization or minimization of some aggregate function that combines results of applying every sample of the training or testing dataset to each individual in the population. These particular results are called *fitness cases* or *fitness instances*. The selection of this aggregate function is another pitfall in designing the objective

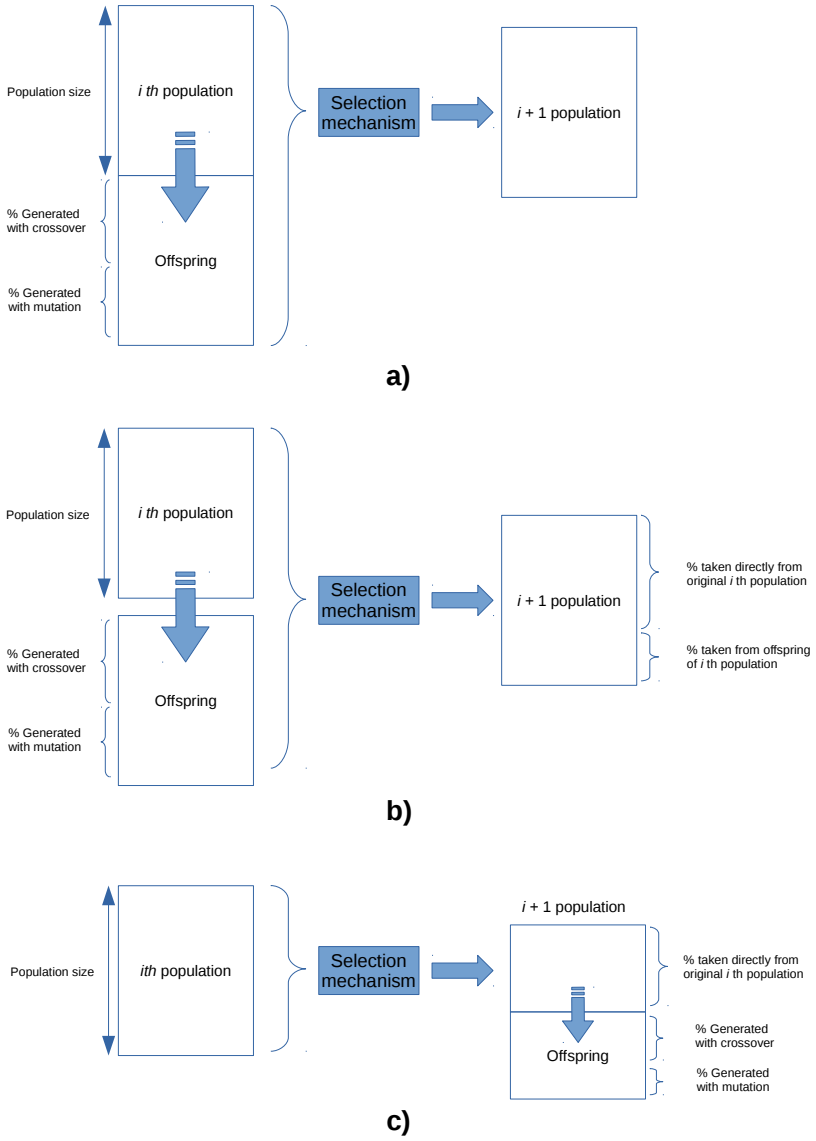

Figure 5: Different types of evolutionary architectures that govern the dynamics of individuals replacement in populations: a) offspring and parents share an extended population pool from which next generation is built; b) offspring do not compete with parents in order to make it to the next generation, which increases the diversity in the population; c) only a selected individuals can cross or mutate, drastically reducing computational resources needed.

function, although the mean is by far the most commonly utilized. For example, in a classification task, a typical objective function would be to minimize the mean error of classification across the entire training dataset or, conversely, the maximization of the mean classification accuracy.

2.3. Taxonomy of primitives

In this subsection we propose a new classification system for the types of primitives found in the GP research literature. As stated in previous sections, GP is a tool that has been used for a wide range of task such as regression, classification, feature extraction and prototype generation. The types of primitives used in each of these works vary so widely among them that we consider necessary to develop a classification system that could help us better understand the state of the art in GP, specifically for feature learning, that it is the concern of our research proposal. To the best of our knowledge, there is no similar taxonomy in the literature.

This classification system will also help us state our research hypothesis in a clearly differentiated way from the rest of the works found in the area, and its relevance.

There are two main classes of primitives: input based functions, and zero-argument functions. Input based functions are further divided into three groups, depending on the dimension of the inputs and the output of the function.

2.3.1. Low Level Functions

We call *low level functions* to those that take as input n scalars as argument, and return a single scalar as output. Examples of such kind of functions are all arithmetic operations such as, addition, subtraction, multiplication, etc. Trigonometric functions also fall in this category. Fig. 6 shows some examples of low level functions as tree nodes.

Genetic programs that use only low level functions are primarily used for scalar function regression, as well as for classification tasks.

The main characteristic of low level functions is their simplicity, and as such, they do not contribute any expert knowledge about the problem at hand being tackled.

2.3.2. Mezzanine Level Functions

We call *mezzanine level functions* to those that take as inputs some vector, array or tensor form of data, and return as output a single scalar. Examples of such kind of functions are statistical measures of data, such as the mean or the standard deviation, or vector and matrix operations, such as the *L2-norm*. These functions serve the purpose of connecting high level functions and complex forms of data, to lower level functions; but at the same time, they can contribute expert knowledge to the set of primitives used in a GP, and by doing so, they may be used as a feature extraction stage in a GP individual. Mezzanine level functions are commonly found in works related to times series prediction. Fig. 7 shows some examples of mezzanine level functions.

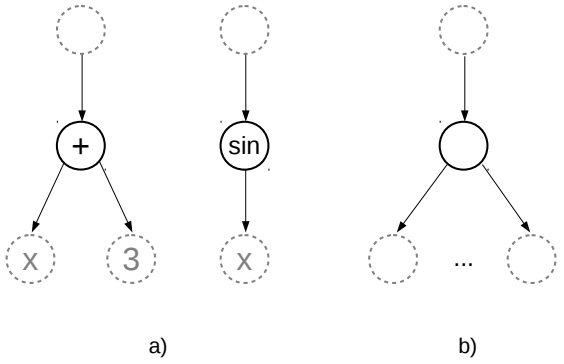

Figure 6: a. Two examples of low level functions: arithmetic addition and sine function; b. A generic depiction of low level functions. Low level function may have any fixed number of inputs, but all should be scalars, as well as their output.

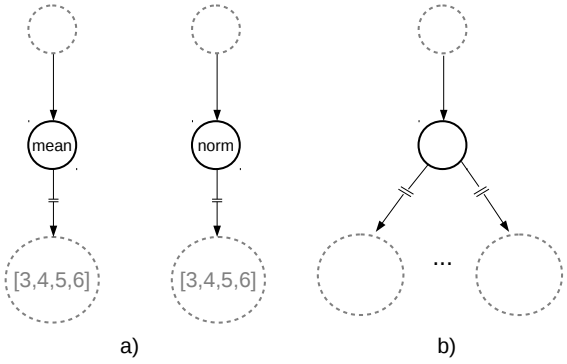

Figure 7: a. Two examples of mezzanine level functions: the mean over the elements of a vector, and the norm of a vector; b. A generic depiction of mezzanine level functions: mezzanine level function may have any fixed number of inputs, and they may be of any dimension > 0, but their output is always a single scalar. In the figure, the dashed arrows that serve as edges to the tree nodes represent the inverse flow of multidimensional data.

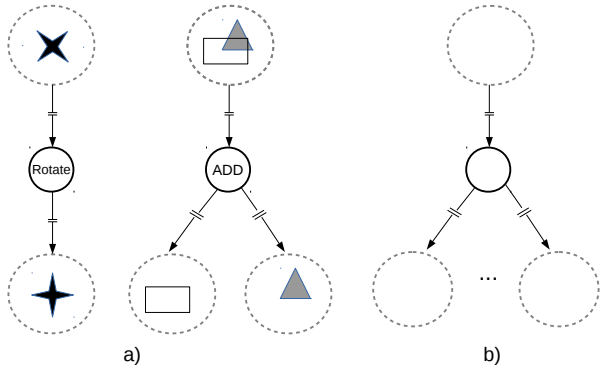

Figure 8: a. Two examples of high level functions: the rotation and addition of images; b. A generic depiction of high level functions: high level function may have any fixed number of inputs, and they may be of any dimension > 0, and they yield their output in an vector, array or tensor form of data. In the figure, the dashed arrows that serve as edges to the tree nodes represent the inverse flow of multidimensional data.

2.3.3. High Level Functions

We call *high level functions* to those that take as inputs vectors, arrays or tensors, and return as output either vectors, arrays or tensors, but not scalars. Examples of such kind of functions are image kernel filters, such as an edge detector, or matrix operations (addition, subtraction, etc.). High level operations can contribute the highest amount of expert knowledge to a GP, so it does not has to process data from scratch, nor discover on its own complex operations that are useful for the problem being treated.

Genetic programs that make use of high level functions have consistently achieved state of the art results in representation learning through the years, and even today in the era of Deep Learning. Some examples of these works are (Shao *et al.*, 2014; Liu *et al.*, 2015; Olague *et al.*, 2014). Fig. 8 shows some examples of high level functions.

2.3.4. Zero-argument Functions

Just as regular functions are classified according to data size they receive as input and return as output, zero-argument functions can also be classified according to the amount and type of data they deliver. We further classify zero-argument functions into four types:

- Type-1 (I_1) are scalar variables that describe input data of GP candidate solutions. These can be meaningful data, such as the total amount of red color in an image, or raw data, such as the individual value of a pixel. Type-1 zero-argument functions' output can be input arguments only to low level functions.
- Type-2 (I_2) are constant scalar values that modify the model being constructed by the GP. They remain constant across all sample data presented to GP individuals. Type-2 zero-argument functions output can be input arguments to low level functions only.
- Type-3 (I_3) are vector, array or tensor variables. These can be, for example, complete images, time series, etc. Type-3 zero-argument functions output can be input arguments to mezzanine or high level functions, but not to low level functions.

- Type-4 (I_3) are constant vectors, arrays or tensors. Examples of this kind of functions are prototypes, image masks, or any other constant signal. They are to Type-3 functions, what Type-2 are to Type-1. Type-4 functions' output can serve as input arguments to both mezzanine and high level functions.

2.3.5. Zero Level Functions

Zero level functions, completely unrelated to zero-argument functions, are a very special type of functions. These are the kind of functions that return as output a binary value, such as 1 and 0, or *true* and *false*. They can receive as input an equally binary value, or scalar values, but they always return as output a true or false signal. Examples of such kind of functions are logic and bitwise operators, such as *and*, *or*, *not*, and inequality test functions. Fig. 9 shows three examples of zero level functions.

Zero level functions make more sense when GP is used to synthesize digital circuits, but they are also quite used for binary or one-class classification machine learning problems. It still subject to further research whether they are strictly necessary for such kind of problems, since it could also be possible to use a less or greater than zero condition at the top of a low level function for discrimination in a binary classification problem. For this reason we set this family of functions at the end and aside of the rest of the proposed taxonomy.

3. Related Work

In this section we briefly mention some of the areas where Deep Learning has achieved great success, then we review the most recent and representative works related to feature learning using GP. We will focus our attention on works in the areas of image processing (classification), time series (prediction), and natural language processing (text classification), given that these areas usually present with high dimensionality ML problems, the aim of our proposed method. Moreover, there are relevant works of feature learning with GP on a diverse number of application arenas such as medicine, mechanical engineering, etc.

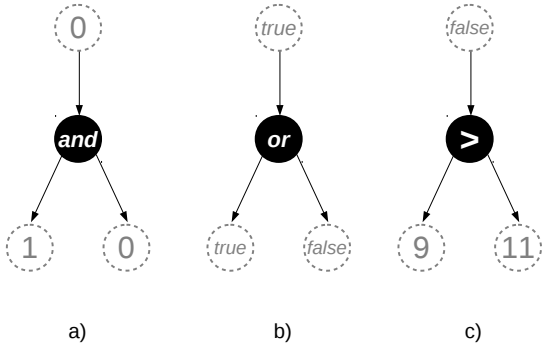

Figure 9: Three examples of Zero level functions, their inputs and the corresponding output: a) bitwise *and*; b) boolean *or*; c) inequality test.

3.1. Deep Learning

Deep learning has taken the lead in generating representations in contexts as diverse as image classification (Krizhevsky *et al.*, 2012), speech recognition (Hinton *et al.*, 2012), and natural language processing (Sutskever *et al.*, 2014; Collobert *et al.*, 2011).

The work done by Krizhevsky *et al.* (2012) sparked the interest in deep architectures, demonstrated again the reliability of ANN as representation learning method, but overall it also showed that multilayer ANN could be efficiently trained with large training datasets. In (Krizhevsky *et al.*, 2012), authors developed a deep neural network to classify the ImageNet LSVRC-2010 dataset (Russakovsky *et al.*, 2015) consisting in 1.2 million high-resolution images, belonging to 1000 classes. Krizhevsky *et al.* (2012) contribution was twofold: they developed an efficient GPU implementation of deep networks to make training faster, and implemented a regularization technique known *dropout* (Srivastava *et al.*, 2014) that prevented overfitting. This work was a breakthrough, since it almost halved the error rate for object recognition at the time, and precipitated the rapid adoption of deep learning by the computer vision community (LeCun *et al.*, 2015). Since then, a race to develop deeper architectures has been going on for researchers of the area, notable works in this regard are (Szegedy *et al.*, 2015) (22 layers), (He *et al.*, 2016) (up to 152 layers) and (Huang *et al.*, 2016) (more than 1200 layers).

Though powerful, deep architectures are far from perfect, and a lot of questions remain open regarding their true capabilities and inner operations. Wishful thinking by researchers of the area led them to believe that the abstract representations present in the inner layers of the deep networks bore a similarity to the way human beings process information. For example, in a face recognition DNN, researchers expected that the first layers dealt with recognizing borders or textures, further layers, shapes, and top layers described, for example, the eyes of a person. Recent research (Nguyen *et al.*, 2015; Szegedy *et al.*, 2013) shed light on the fact that deep networks do not abstract information at all like human beings, and not only that, but also these deep neural networks can be easily fooled into recognize objects in images unrecognizable to humans, and in images that are pretty much noise, thus raising serious questions about the generality of deep neural networks. All these evidence entice us to research more profoundly into hierarchically structured representation learning through other methods.

3.2. Representation Learning with GP

3.2.1. High Level Functions Only

Works relying only on high level functions go as far back as (Zhang *et al.*, 2003; Trujillo & Olague, 2006), and up to recent works such as (Shao *et al.*, 2014).

In (Trujillo & Olague, 2006), the authors developed a GP to automatically synthesize *Interest Point Detectors*. Interest points are pixels in images that are particularly useful for algorithms that perform object recognition (i.e. image classification), stereo matching, object tracking, among others tasks, so these algorithms do not have to operate on the massive amounts of pixels an entire image may consist of. Interest Point Detectors take as input a complete image and return as output a reduced set of pixels of it, performing in fact, a feature selection (reduction) task. The GP proposed by Trujillo and Olague utilized Type-3 zero-argument functions only, consisting in a variety of preprocessed versions of the complete image from which the interest points are to be extracted. The function set consisted only in high level functions, like complete images addition, extraction, square root, Gaussian filters, and histogram equalization. The GP combines all

these elements through the evolutionary process resulting in a mathematical formula that represents the Interest Point Detector. The objective function that guides the evolutionary process is a measure of robustness proposed for interest point detectors by Schmid *et al.* (2000).

More recently, Shao *et al.* (2014) proposed a GP for feature learning directly aimed at image classification problems. In a similar vein to Trujillo & Olague (2006), the zero-argument functions set consists mostly of Type-3 functions, more precisely these are: each RGB image channel and its gray-scale version. The input based functions are further divided into two groups. One group consist of filters (Gaussian, Laplacian, equalizers, etc.) and image arithmetic operators (e.g. sum or extraction of two images), which main characteristic is that the output image is the same size as the input one(s). The other group of input based functions consist of image resizing functions. Even though the objective of this last group of functions is to reduce the size of the image, none of these can be considered mezzanine level, since they all still render as output an array representing an smaller version of the image being processed. On top of every subtree rooted at tree's root generated by this GP approach, there is a flattening function that collapses a 2D image array into a 1D vector. Finally, at the root of all trees generated by the GP there is a concatenation function that ties together each one of these 1D vectors into a single 1D vector that is the new feature representation for the image being classified. Interestingly enough, this 1D vector still undergoes a PCA processing in order to reduce the dimensionality of the new feature representation, and only then is passed to a SVM for classification. The main objective function that guides the evolutionary process is to minimize the error in classification performed by the SVM. There is a secondary objective function that consists in minimizing the size of the trees generated by the GP. The goal of posing this problem as multiobjective is to control bloating in trees generated by the GP, and thus, the overfitting.

Both of these works briefly reviewed here yielded state of the art results at their time. The work of Shao *et al.* (2014) even outclassed two DNN against which is compared. This success led to an extended version of the approach applied to action recognition in video, presented in Liu *et al.* (2015).

Although competitive, the main disadvantage of works that rely in high level functions is that there is an implicit expert knowledge contributed by the designers of the algorithms to the feature extraction process precisely in the selection and definition of such high level functions. Trujillo and Olague explicitly acknowledge this fact, stating that the success of their proposed approach is thanks to their vast experience in image processing research, that allowed them to propose the set of specialist input-based and zero-input functions that fed their GP.

To the oblivious observer, Gaussian or Laplacian filters, finite-differences edge detectors, or histogram equalizers are completely unknown, and make little sense as why they are there in order to perform feature extraction. This reminds us one of the biggest achievements of Deep Neural Networks: they require almost null understanding of the problem at hand being solved in order to be utilized; in other words, they provide a good expert knowledge replacement, and can be applied to a wide range of problems without necessarily delving too deeply into new contexts each time the nature of the problem changes.

3.2.2. Mezzanine and Low Level Functions

There is a vast amount of works on GP that rely on Mezzanine and Low level functions in the literature related to financial time series analysis. Examples of such works are Potvin *et al.* (2004); Lohpetch & Corne (2009, 2010, 2011); Myszkowski & Bicz (2010); Esfahanipour & Mousavi (2011); Moscinski & Zakrzewska (2015); all these works based on the pioneering research done by Neely *et al.* (1997) and Allen & Karjalainen (1999). Financial technicians and analysts had long been using statistical measures of data, such as the mean

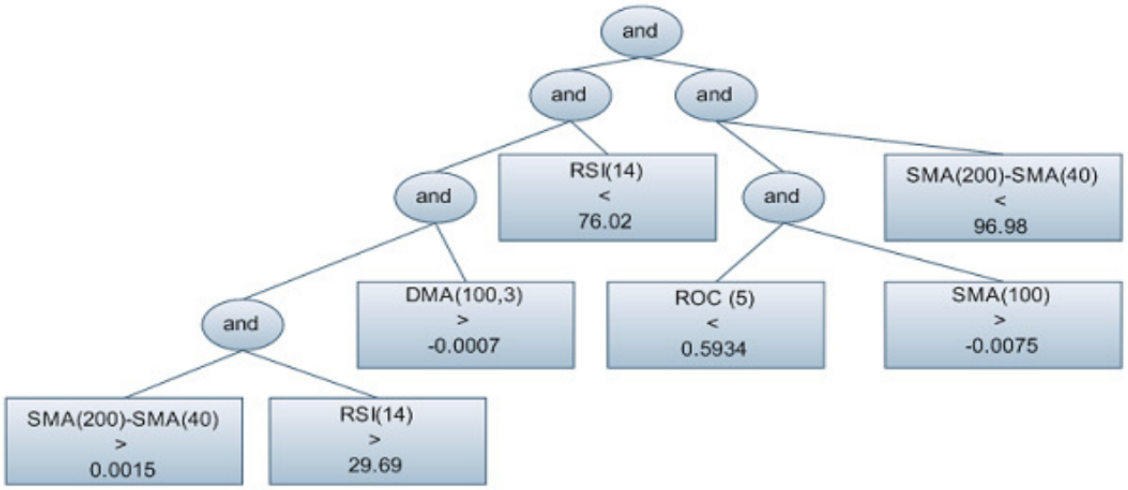

Figure 10: Example of a one-class classifier generated through GP for time series prediction. In this case, leaf nodes are composed of both mezzanine level functions and zero-argument functions, the last of which deliver chunks of time series to the mezzanine component of the leaf. Notice functions *SMA*, *RSI*, *ROC*, etc. these functions are highly specialized financial indicators, and make little sense to the untrained observer why they were chosen in the first place to form part of the set of primitives.

or the standard deviation, as well as many other linear and non-linear filters to extract more meaningful numbers from the crude financial time series. It was, and still is, an open question how to combine the output of such filters in order to generate effective buying and selling signals that translate into increased profits from trading; however, GP became a natural tool for automatically finding these combinations: the filters they already use are mezzanine level functions, that when coupled with low and zero level functions, it is possible to synthesize trading rules, that is, time series predictors that estimate when such a time series will go up or down. The conventional approach in these works consists in using GP to evolve two distinct individuals (GP trees), one will output a true signal when estimates that the time series will go up, and the other will attempt to predict when the series goes down, i.e. two one-class classification systems. The set of primitives used consist of many of these filters used in finance, that form a mezzanine layer of functions, as well as low and zero level functions that take the outputs of these filters and combine them to generate a true or false final output. The zero-argument functions consist of chunks of time series being analyzed; their size should make some sense given the context, e.g. a chuck of the time series corresponding to the last week, last day or last month. Fig. 10 shows an example of such kind of evolved individuals (Myszkowski & Bicz, 2010) and Fig. 11 shows the very same individual broke down in the terms of our proposed taxonomy. For an example of the application of this very same scheme to time series analysis but in other context, refer to Sotelo *et al.* (2013), where authors use a GP built with mezzanine and low level functions for the classification of epilepsy attack stages from signals obtained through Electrocortigrams. Al-Sahaf *et al.* (2012) presents a work for feature extraction and classification of images that relies on mezzanine and low level functions, akin to the approach used for time series analysis; in fact, they are the first authors, to the best of our knowledge, that acknowledge the existence of these different types of primitives as our taxonomy proposes. In their work, they call mezzanine functions ”Aggregation functions”, and low level functions, ”Classification functions”.

The problem with mezzanine level functions, as with high level functions, is that they bring a sort of expert knowledge to the system, and that they implicitly limit the search space of the GP as well, which can be detrimental to the tackled problem.

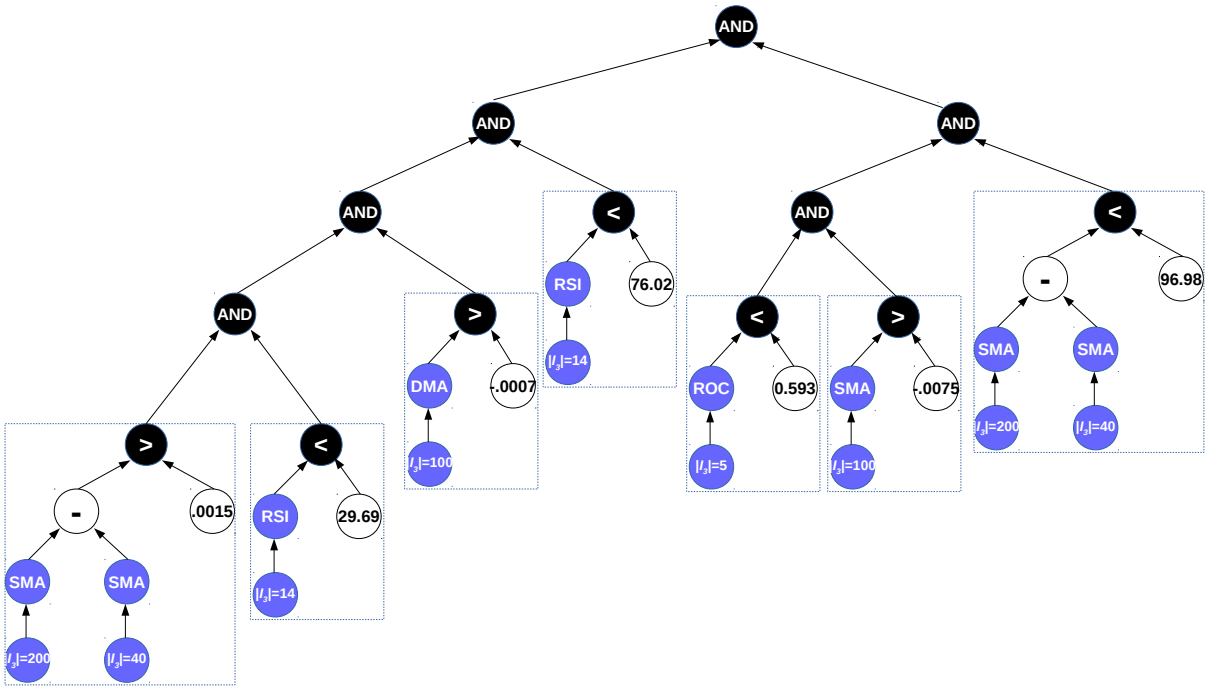

Figure 11: The same GP individual depicted in Fig 10 but broken down in order to illustrate it as an example of our proposed taxonomy. Blue nodes are either mezzanine level or zero-arg Type-3 functions; white nodes are either low level or zero-arg Type-2 functions; black nodes are all low level functions. The boxes encompass nodes that comprise a single leaf node in the original depiction. Mezzanine functions comprehend a feature extraction layer, while low and zero level functions perform the classification itself.

3.2.3. Low Level Functions Only

Low level functions have typically been the only component of many early works on GP; just as Al-Sahaf *et al.* (2012) proposes, these functions are more *ad hoc* for the construction of classification systems, rather than representation learning. The problem with low level function is that they operate at individual, or crude, feature level, and if the tackled problem presents a large initial feature space, the search space for a tree, or set of trees, that both, select or transform relevant features and performs the classification, becomes intractable.

It does not mean low level functions on their own are useless; if the problem's initial feature space is small enough, low level functions can generate high quality classification, regression and feature extraction systems. Nevertheless, low level functions alone have been used in large scale problems, with mixed results.

Parkins & Nandi (2004) developed a GP to perform classification of hand written digits. Their GP relies only in low level functions, and hence its zero-argument functions consisted of individual pixels of the images representing digits (as well as random scalar constants). They developed and fine-tuned their algorithm with a dataset (CEDAR, 1992) consisting of rather small images, only 16×16 pixels, because as they acknowledge, performing a complete study of their approach in a dataset as big (both in terms of number of samples and size of images) such as MNIST (LeCun, 1998), with 60,000 training samples and images of 28×28 pixels, was infeasible. However, once they tuned their GP (evolutionary parameters, types of GP operators used, etc.) on small images, they tested their algorithm with MNIST, achieving a classification accuracy of 79% on the testing set, well below the current state of the art. Nevertheless, to the best of our knowledge, this is the best result obtained in MNIST with GP using only low level functions. This result should not be confused with other works such as Teredesai *et al.* (2001), where they also develop a GP with only low level functions to classify digits of MNIST, but the GP is used as classification algorithm only, and it is fed with handcrafted features.

An important result was reported by Limón *et al.* (2015), where authors performed feature extraction with GP using almost only low level functions. They tested their approach in a variety of datasets, both with small and large number of features. The features their GP generated were fed to a 1-nearest neighbor (1NN) algorithm to perform classification. Their results showed that when datasets samples were small (up to 90 features), learned representations helped the 1NN algorithm to achieve state of the art classification accuracies, but when the datasets consisted in large feature spaces (in the order of thousands), their approach fell behind to the rest of feature extraction techniques. This result suggests that GP is a very competitive tool for representation learning, but that still needs to be adapted in order to be capable of dealing with large input feature spaces. Cano *et al.* (2017) reported a similar result; they developed a multi-objective GP approach for both feature extraction and enhancement of data visualization. Although their approach admittedly tried to balance between these two apparently contradictory objectives, instead of focusing on the feature extraction process, they found out they approach could not generate representations that enhanced classification accuracies when the input representations were composed of 60 or more features.

Lin *et al.* (2008) developed a multilayered GP approach for feature *construction* (Liu & Motoda, 1998) and classification that resembles the most to our proposed approach. Their method consisted in using GP to evolve a binary classifier, the resultant classifier (a GP tree) returned a real valued scalar a for every instance, such that $a \geq 0$ for one of the classes and vice versa. Then, this output is added to the set of features in order to act as a new feature, and a whole new GP learning process would start with the enhanced representation. This process can be repeated in a iterative manner until no increase in classification accuracy is obtained. It is important to remark that their method was aimed at binary classification and feature construction, whereas our method is designed for feature extraction and representation learning.

4. Research Proposal

In this section we state the core components of our research proposal: the hypothesis, research questions, objectives of our research and the methodology.

4.1. Statement of the Problem

We approach the problem of feature extraction using GP with low level functions only. From a dataset with an arbitrary number of samples, each sample j represented by feature vector o_j that belongs to $\mathbb{R}^n$, we wish to learn a new, more compact and abstract, representation $\mathbf{M}$, such that each sample j is now represented by feature vector p_j that belongs to $\mathbb{R}^m$, and $m \ll n$.

4.1.1. A straightforward approach for representation learning with GP and low level functions only

In this proposed research, for every feature to be learned $f_i^m \in \mathbf{M}$, we will build a GP tree (that is, we are using forest-like individuals) we denote as t_i. Tree t_i receives as input the n original features, and returns as output feature f_i^m. Let us suppose, without loss of generality, that each tree $t_i, \forall f_i^m$ is composed of (low level only) input-based functions of arity 2, i.e. each t_i is a complete binary tree. Since tree t_i could require, conceivably, all n original feature to generate f_i^m, t_i is a perfect binary tree, and input features can only go in leaf nodes of GP trees, then the height of tree t_i is, approximately at least, $\lceil log_2(n) \rceil$, and the number of internal nodes is, approximately, $2^{\lceil log_2(n) \rceil}$. For simplicity, let us assume for now on that n is a power of 2, therefore the number of internal nodes of t_i is n. Now let us suppose that we will use a set of K low level functions; each internal node can take the form of any of these K functions, then the total size of the search space the GP needs to explore is $\mathcal{O}(mK^n)$. Fig 12a shows a depiction of this simple approach.

This is an optimistic, lower bound estimate, since we are not yet taking into account that Type-2 zero argument functions (constants) are probably needed as leaf nodes as well; but then again, this estimate shows us the complexity of the problem we are dealing with. Although evolutionary algorithms are ideally suited to explore search spaces of such exponential growth, we propose that there might exist additional steps to the standard GP that can be taken to improve its efficiency.

Example 1. *Suppose we wish to process a set of images to convert them from a original feature space of 64x64 gray scale pixels to a vector of 32 new features. Hence,* $n = 4096$ *and* $m = 32$*. We are set to search for a GP individual composed of 32 trees; each tree, potentially, of height 12. Suppose we are considering the following set of low level functions* $\{+, -, \times, /\}$*. The GP needs to search for an optimal individual among, at least,* 32×4^{4096} *distinct possible solutions.*

The problem of representation learning defines intractable search spaces. Using a standard GP approach to tackle representation learning would be computationally expensive. Moreover, an important number of poor solutions could be generated and the GP would need to deal with them. Considering the structural layered processing of Deep Learning allows to significantly improve GP performance to tackle representation learning. A layered GP scheme will gradually reduce the search space while reducing the computational burden.

The straightforward approach for representation learning with GP is computationally inefficient, because the search space for the optimal solution includes massive amounts of very poor candidate solutions the GP has to filter on its own.

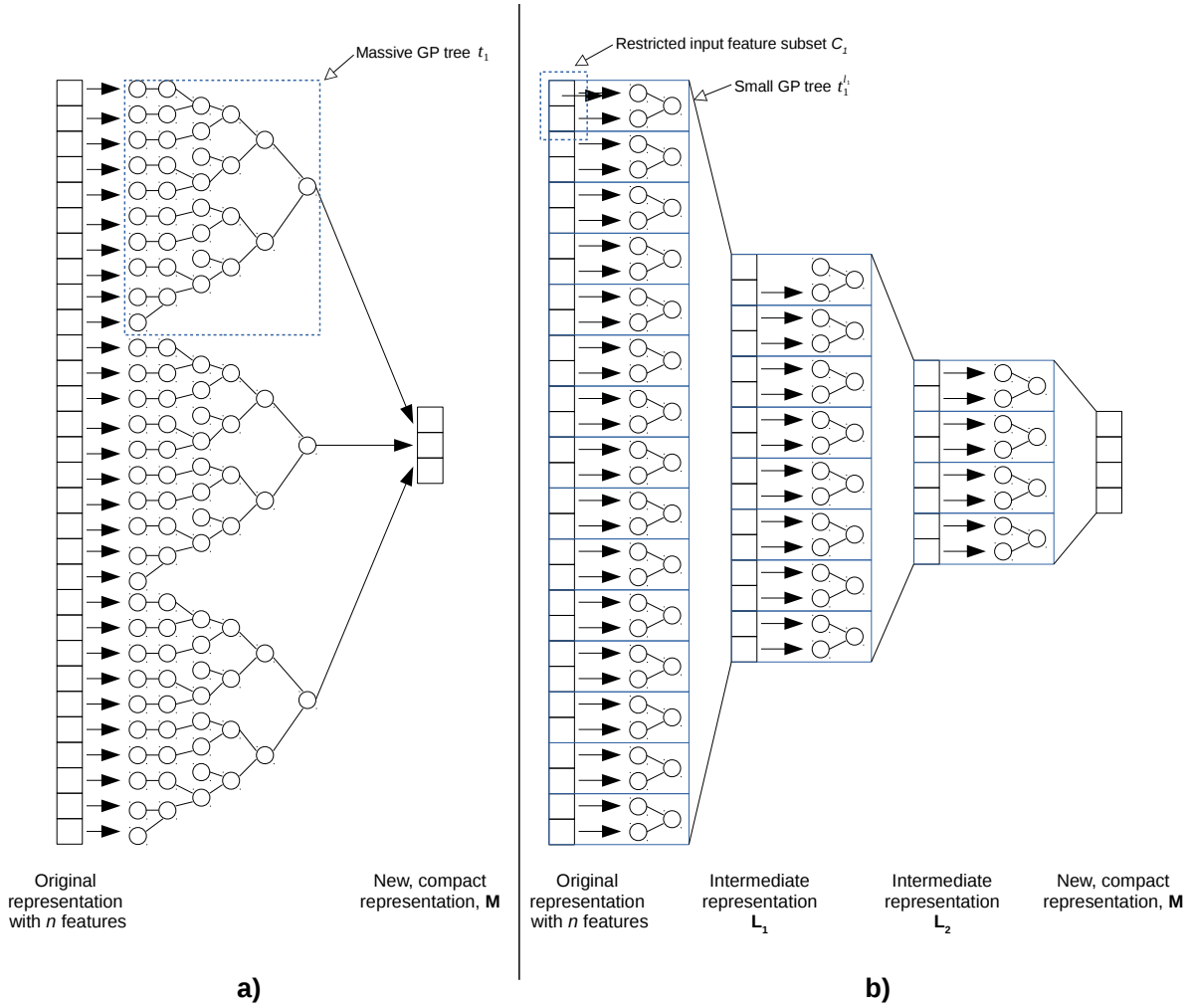

Figure 12: Graphical depictions to contrast between (a) a straightforward approach to representation learning with GP and low level functions, and (b) our hypothesis for the type of models the GP must search for. In the depictions, squares represent features; in the in straightforward approach, GP tree may use any feature as variable for their zero-argument functions, whether in our proposed approach, the smaller GP trees can only *see* a small subset of available features at any given representation layer.

4.2. Hypothesis

It is possible to achieve a higher efficiency[2] than the straightforward approach in the evolutionary search for representation learning if we direct the search towards individuals that mimic the models of Deep Learning, i.e. models based on multiple GP layers that generate intermediate representations that gradually reduce the feature space.

4.2.1. Proposed approach

We propose to partition the search for the optimal solution in the following way. The starting n features are partitioned into c subsets C_i, such that $|C_i| \ll n, \forall i$, and we set to discover an intermediate representation

[2]Better quality solutions in the same amount of time, given the same amount of computational resources.

$\mathbf{L_1}$, composed of l_1 features, that we will refer to as a *layer*, such that $n > l_1 \gg m$. For every feature $f_i^{l_1} \in \mathbf{L_1}$ we build a GP tree $t_i^{l_1}$, that will generate such feature; the set of zero-argument functions that can take $t_i^{l_1}$ is C_i, plus constant values in some arbitrary range. Once we have found layer $\mathbf{L_1}$, we iterate the procedure in order to find next layers $\mathbf{L_2}, \mathbf{L_3}, ...\mathbf{L_z}$, composed of $l_2, l_3, ...l_z$ features, respectively, such that $l_z = m$. Fig. 12b illustrates this proposed approach.

To the best of our knowledge, there is not a similar approach in the literature. The advantage of the proposed method is threefold: (1) by limiting our approach to the use of low level functions only, we guarantee that no human expert knowledge is required by the system, putting in on par with DNN in this regard; (2) low level functions offer a higher degree of flexibility to the GP to transform the input representation in any way needed, contrary to the use of high and mezzanine level functions; and (3) our approach proposes a more efficient search of the of candidate solutions space than the straightforward GP approach.

The intuitive idea behind the hypothesis is that, if the subsets C_i are chosen to group features that share common traits (such as neighboring pixels in images, or contiguous timestamps in time series), then the GP is forced to first search for, and exploit local properties present in the original feature spaces that is to be transformed.

4.3. Research Questions

Main Research Question

How can we design and adapt a Genetic Program to efficiently generate compact, descriptive and manageable representations of high dimensionality data sets associated to machine learning problems in a way that does not require any human expert knowledge?

Research subquestions

1. *How can we evaluate the quality of a learned representation?* That is, *what is the fitness of a GP individual for representation transformation?*, *What objective functions can drive the evolutionary search for representation learning with GP?*
   These are actually open questions in the overall representation learning research field (Bengio *et al.*, 2013), and we do not expect to provide a definitive answers to them; our concern is more related with answering how to tailor our algorihtm to generate compact representations useful for classification and clustering tasks.
2. *What relationship exists between GP search efficiency and, both the quantity of layers z, and the cardinality of subsets C_i?*
   We hypothesize that layered and local segmentations will make the GP search more efficient. But does stacking up more and more layers can improve the performance of the proposed approach or might exist a sweet spot of the right amount of layers, beyond of which the efficiency of the algorithm begins to degrade? Similar questions can be raised regarding the amount and size of C_i subsets.
3. *What mechanisms can be implemented so trees $t_i^{l_x}, \forall i$ share information so do not evolve in complete isolation?*,
   The tessellation initially proposed considers that trees $t_i^{l_x}, \forall i$ can only take zero-argument functions from set C_i, hence relationships among features belonging to different subsets C_i are only implicitly exploited at further layers. But what gains in performance can be obtained if the family of subsets C_i is not a partition? And if not a partition, then how can the subsets C_i be defined?

4. *How can we implement online or semi online forms of training/evolution to further increase computational efficiency?*
   One criticism to evolutionary algorithms is their low speed of convergence, compared to other methods such as stochastic gradient descent. Would it be possible for our proposed approach to evolve solutions using small subsets of samples (minibatches), instead of sweeping through all the training dataset in each generation and for every individual in the population?
5. *How the method would perform if all z layers are evolved simultaneously?*
   Our hypothesis centers around the idea of evolving layers of intermediate representations in a sequential manner; but defining layers size, along with C_i subsets in each layer, might be enough to channel the GP search toward promising areas, further increasing the efficiency of the GP search (since we do not have to wait until one layer is fully evolved in order to start evolving the next one).
6. *How the proposed method compares with other state-of-the-art representation learning methods, such as PCA, matrix factorization, Deep Learning, and high/mezzanine level functions based GPs?*
   In what kind of problems our proposed method could improve the state-of-the-art results? Deep Learning has proven itself a successful approach specially in image processing task. But other problems, such as those dealing with time series, or when training datasets are small, are difficult for deep architectures. Could our approach take the lead in such scenarios? Could our evolutionary approach be competitive (both in terms of speed and quality of solutions) in those scenarios where Deep Learning is already very good?

4.4. Objectives

Main Objective

Design, develop, implement and evaluate a multilayered representation learning framework based on Genetic Programming, that can obtain competitive performance with state of the art solutions.

Specific Objectives

1. To define a set of objective functions to assess solutions or individuals at a genotypic level in terms of their compactness and in relation to classification and clustering tasks.

2. To provide conclusive experimental evidence on the behavior of the proposed GP framework performance as a function over its two most distinctive parameters: cardinality sizes of C_i subsets and number of layers.

3. To implement a multi-population inspired scheme that fragments the GP individuals and allow them share information at same-individual level and determine if this approach further enhances the efficiency of the proposed framework against when using regular single population approach.

4. To implement or, if necessary, develop different incremental learning methods in order to avoid full training datasets evaluations every generation.

5. To develop a method that attempts to evolve all layers of the proposed GP approach simultaneously.

6. To evaluate the proposed framework on typical benchmarking datasets of for machine learning problems and compare its performance with state of the art deep architectures.

4.5. Justification

So far, state of the art results in representation learning with Genetic Programming have been achieved only through heavy use of high level functions, or mezzanine level, at most. The problem with this approach is that: (a) high level functions are a form of human expert knowledge brought to the system by the designer and, (b) the resulting systems are somewhat limited to the capabilities of such hand-crafted high level functions. Nowadays, machine learning algorithms for representation learning that do not require any expert knowledge input and that can transform input features in any way needed are more desirable. Deep neural networks have succeeded in this regard in certain scenarios, and hence their importance. Genetic programs that are restricted to the use of low-level functions promise to solve both issues since they do not contain any implicit nor explicit expert knowledge, and have more flexibility in the process of transforming the original features; however they pose a new problem: depending on specific application domains (image processing, natural language, time series analysis, etc.) solutions or individuals in the form of a single or a set of trees (forests) can be unmanageable in terms of representation; the vast number of features that define the landscape can make the problem computationally intractable.

Neural networks and Deep neural networks have stood out as a machine learning tool among many others thanks to their ability to represent any function as needed. This property has make them to be considered *universal function approximators* (Hornik *et al.*, 1989). Although there are not equivalent theorems for GP, as far as we are aware of, the authors of this research proposal suspect that GP can also perform as an universal function approximator. For this reason, we consider GP a relevant computational tool that should be further explored and expanded. In our literature review we presented a wide variety of works that prove the usefulness of GP for representation learning. However, we also brought light to the fact that GP based approaches for representation learning hit a wall when encountered with problems with massive amounts of features.

To solve this issue, we propose to attempt an approach partially inspired by the deep architectures, that learn multiple layers of representations that gradually abstract the resulting feature space. Whether layers in the deep architectures and the GP layers that will be generated by the proposed framework maintain some relationship, in terms of features present in them, is still to be answered.

4.6. Methodology

The basic steps that will be taken to achieve our objectives are a permanent literature review on the general field of representation learning, but emphasizing on representation learning through GP, as well as the development and constant improvement of a GP software library to perform representation learning. Additionally, the following steps are considered:

1. A literature review on evaluation criteria of learned representations. We will select a set of such criteria to be used, directly or modified, as objective functions in our evolutionary approach. The purpose of our framework is to generate compact representation ultimately useful for classification and clustering tasks. A straightforward approach would be to use, for example, the classification error. However, using this approach might result cumbersome for intermediate representations layers that are not yet compact enough. We propose to use a *self-supervised* machine learning approach to gradually reduce each successive representation layer. The typical self-supervised technique is the autoencoder (Chollet, 2016). We plan on building autoencoder-like individuals with GP that minimize a distance metrics such as: the euclidean distance, the Manhattan distance, the mean squared error, etc.

Once the representations generated by the GP autoencoders are compact enough, we will use these representations to feed classification algorithms such as k-nearest neighbors, perceptron or multilayer perceptron, or even a GP classification layer itself. The objective is to test if more compact representations translate directly into higher classification accuracies.

2. We will perform a complete study varying the amount of layers and size of subsets C_i, to generate a diverse set of GP autoencoders, and study the relationship between number of layers, size of subsets C_i, quality of found solutions, and approach efficiency. We are also considering a multiobjective optimization approach that help us perform this study. Under this approach, the framework proposes different sizes for the next layer, and sticks with the one that balances individuals with the highest reconstruction (least distance to original samples) and dimensionality reduction. All layers are generated iteratively in this manner. The downside of this approach would be the computational cost involved.

3. To improve the proposed GP framework performance, several population topologies will be explored. Initially, solutions will interact by a mesh grid array where trees $t_i^{l_x}$ of C_i with neighboring (spatial or temporal) features share some of them. Also, an hypercubic (Seitz, 1985) population topology will be explored.

4. Implement different incremental learning mechanisms to avoid evaluation of the entire training dataset in every generation, and thus speed up the execution time of the algorithms. We plan to implement and compare at least three mechanisms:
   - A fully online learning approach, where only one sample of the dataset is used for fitness evaluation in each generation.
   - A minibatch learning approach, where a small, variable, subset of samples from the training dataset are used for fitness evaluation in each generation.
   - An incremental learning approach, inspired in the research done by Morse & Stanley (2016), where individuals inherit a portion of the fitness to their offspring.

5. Extend the framework to assess the possibility of evolving all the GP layers simultaneously using the minimization of the classification error as the objective function, equally trying different classification algorithms for such study. At this point we will also attempt to perform unsupervised learning in the form of clustering. Optimization of metrics for clustering can be used as objective functions, such as the Dunn's Index (Dunn, 1974), the Davies-Bouldin index (Davies & Bouldin, 1979), or the Chou-Su index (Chou *et al.*, 2004).
   If training all the GP layers turns out unfeasible, but we know the optimal amount of layers (from the study of point 2 of the methodology), we can also try to evolve more than one layer at the same time.

6. Select a set of hallmark datasets for machine learning training and evaluation, for the tasks of image classification, text classification and time series prediction. For image classification, we are considering the following machine learning dataset, in the given order:
   - MNIST handwritten digit labeled database (LeCun, 1998). It is a standard benchmarking dataset for machine learning applications such as classification and clustering. It has the advantage of being relatively simple, almost monochromatic, and as a such, can be considered as an entry level test for new algorithms.
   - LFWcrop (Sanderson, 2014) is a modified version of Labeled Faces in the Wild, face recognition database (Huang *et al.*, 2007), it is supposedly more challenging than the original, since it

removes the background that could be used as a context to help face recognition (classification) algorithms. The images of this dataset are gray scale.

- and the CIFAR-10 image classification dataset (Krizhevsky & Hinton, 2010). This another typical image classification algorithm. Consist of full RGB color images of ten different classes of objects.

We can notice a trend to gradually increase the difficulty presented to our proposed framework. First we propose to try monochromatic images, then gray scale, and finally full color images.
The final step will be to try the calibrated algorithm (size of subsets C_i, amount of layers, size of mini-batches, evolutionary parameters) that resulted from the study applied to image datasets, to datasets of other nature, such as text classification and time series prediction. For text classification and time series prediction, datasets are still to be decided.

5. Preliminary results

In this section we present some experimental results that back our hypothesis and research proposal overall. We develop an autoencoder with GP using different experimental setups. Our initial findings suggest that the partitioning of the original input features, as proposed in the hypothesis, as well as an online form of training using minibatches, are key elements that dramatically boost the performance of a GP in large scale problems. To the best of our knowledge, this is the first time an autoencoder is synthesized using only evolutionary computation, unlike other approaches where deep learning architectures for autoencoders are developed using GAs or EC in general, such as in (Fernando *et al.*, 2016).

5.1. Problem Description

Given a dataset comprised of an arbitrary number of samples, each one described by the same n features, we wish to reduce the n features to $l = \frac{3}{4}n$, losing the least possible amount of information. That is, from new l features, it should be possible to reconstruct samples to the original n features dataset, given some measure of similarity. The mechanism tasked with generating (reconstructing) the compact (original) representation from the original (compact) representation is the *encoder* (*decoder*). Both mechanisms together are known as an *autoencoder* (Hinton & Salakhutdinov, 2006b; Goodfellow *et al.*, 2016). Autoencoders are typically built with ANN, but here we will use a GP approach.

5.1.1. Individual Representation

The encoder will be comprised of a forest of l GP trees $t_i^{l_1}$; the decoder will be comprised of a forest of n GP trees $t_i^{l_2}$. Each tree $t_i^{l_1}$ will generate feature i of the compact representation $\mathbf{L}$, and each tree $t_i^{l_2}$ will generate feature i of reconstructed representation $\mathbf{M}$. The GP individual comprehends both encoder and decoder forests. Figure 13 shows a depiction of the GP autoencoder individual. Zero-argument functions of each tree $t_i^{l_2}, \forall i$ can only be taken from features of representation $\mathbf{L}$ (as well as constant values within some range), i.e. none of these trees can see any of the original features. Similarly, variables for zero-argument functions of each tree $t_i^{l_1}, \forall i$ can be only be taken from the original representation, and they cannot look ahead for features from representation $\mathbf{L}$ they are constructing.

We construct a set of randomly generated GP autoencoders individuals, that will constitute an initial population, and through GP evolutionary process, we will search for an autoencoder that maximizes the average similarity between each sample and its reconstruction, across the entire dataset.

5.2. Experimental Setup

We kept constant the following parameters across all the experiments performed: population size: 60; encoder (decoder) trees max allowed depth: 4 (4); set of primitives other than zero-argument consisted in binary arithmetic functions $\{+, -, \times, \div\}$, square root, and two trigonometric functions, $\sin$ and $\cos$. The division function is protected, meaning that any attempt to divide between zero returns as output 1×10^6, instead of an error that could potentially crash the algorithm's execution, and since the input values from the samples of any dataset will be set in the range between $[0, 1]$, and thus reconstruction output is set to be in the same range, this protection guarantees that any invalid division will return values out of the range that makes any sense from the tackled problem. We expect that the evolutionary search will clean up the individuals in the population from attempting to perform any invalid division.

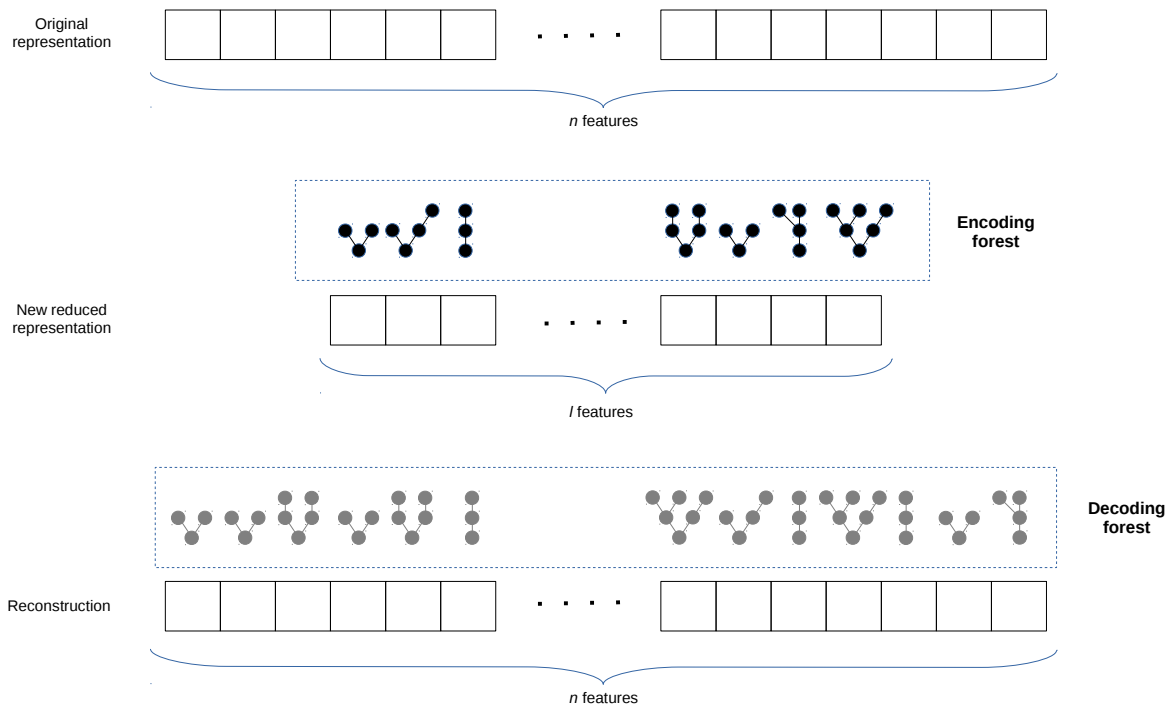

Figure 13: A GP individual representation of an autoencoder. A GP tree is used to generate each new feature of the compact representation, as well as to reconstruct each original feature. Thus the individual is comprised of an encoder forest and a decoder forest.

We also used an evolutionary architecture as shown in Fig. 5c to guide the evolutionary search in all our experiments. In each generation, only half of the population is chosen through binary tournament (Goldberg & Deb, 1991) to make it to the next generation. From this half of the population, new individuals are generated with an 0.6 probability of crossover and a .3 probability of mutation. Thus, given that population size is fixed to 60 individuals, 30 of them are directly taken from the previous generation, 18 are generated from crossover and 9 are generated through mutation. The remaining 3 individuals are chosen directly from previous generation, through a mechanism known as *elitism*, that consists in selecting the top performers to always make it to each new generation. Elitism is necessary to guarantee a convergence and to generate additional evolutionary pressure; otherwise, binary tournament might miss to choose top individuals, losing promising solutions and thus generating an oscillating behavior in the overall population performance.

The genetic operations used consist of variants of trees mutation and crossover, as illustrated in Fig. 4a and 4b, respectively. Since individuals are comprised of two forest, it is important to note that crossover operator works among only the same types of forest, i.e. when two individuals are chosen for crossover, first the encoder forests from each individual exchange trees between them, and then decoder forests do the same, but at no point an encoder and a decoder exchange trees. When applying mutation, one tree in the encoder and one tree in the decoder are replaced with new, different, randomly generated trees.

To determine similarity between an original sample and the reconstructed output from the autoencoder, we used the mean square error (MSE), defined in Eq. 1. MSE receives as input original sample x and reconstructed y vectors, and compares them feature by feature, averaging the difference across all of features.

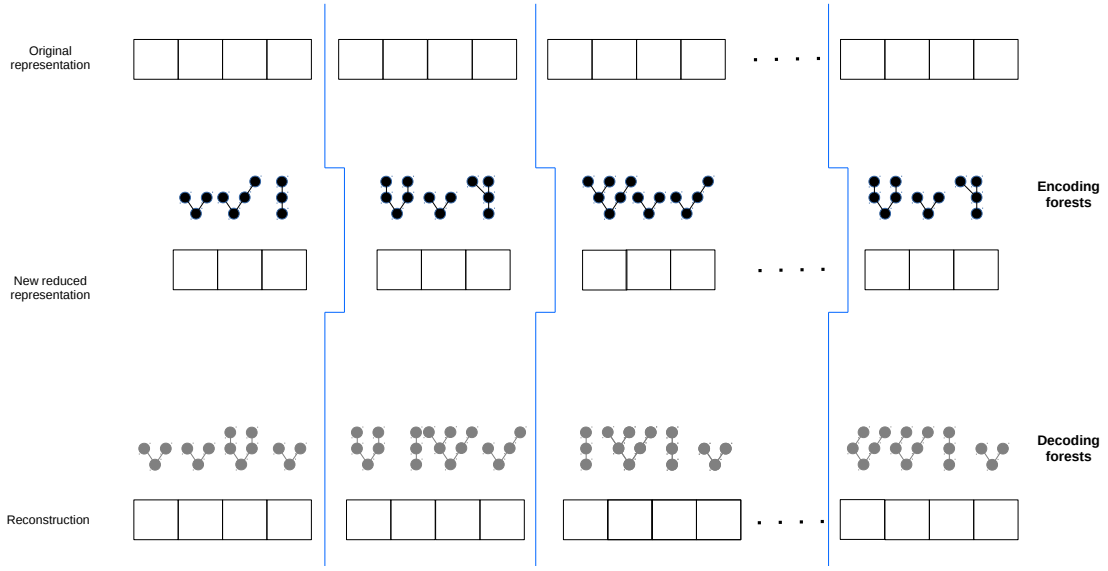

Figure 14: The partitioning of the autoencoder as proposed by our research hypothesis.

MSE output can be thought as a *distance* between a sample and its reconstruction.

$$d_{\mathbf{MSE}}(x, y) = \frac{1}{n}\sum_{i=1}^{n}(x_i - y_i)^2 \tag{1}$$

All parameters described so far remain constant across the three different experimental setups we tested. Now we will describe how each of these setups vary among them. The first setup consist in what we can call a *vanilla* GP. In this setup, zero-argument functions of each tree $t_i^{l_1}, \forall i$ can be taken from the entire universe of n original features. Analogously, zero-argument functions of each tree $t_i^{l_2}, \forall i$ consist of any of the l features from the compact representation $\mathbf{L}$. This is our control setup, and its result will serve as a baseline for comparison purposes with the proposed approach.

In the second setup we perform a partitioning of: original n features, encoding forests, l features of the compact representation, and decoding forests. We divide the l encoding forests in $\frac{l}{3}$ subsets C_i consisting of three trees. The zero-argument functions for the trees in each C_i are also limited such that, each C_i can only see a subset of four different features from the total n original features. Mirroring this configuration, the decoding forests are also partitioned in subsets of four trees, such that trees in each subset can only choose zero-argument functions from a subset of three different features from the total l features of the compact representation. All these subsets are coupled together, to form a set of miniautoencoders that comprise the complete autoencoder. Fig. 14 illustrates this partitioning.

Since we will be dealing first with grayscale image datasets in our experiments, and will use each individual pixel as an input feature, the partitions are built considering neighboring pixels.

Table 1: Average MSEs and execution times for each experimental setup. For the third setup, "Training MSE" refers to the average MSE over the entire training dataset, which differs from the fitness of the final top performer individual. Training MSE and the fitness of the the final top performer are the same in the case of the first two experimental setups.

Setup	**Training MSE**	**Testing MSE**	**Exec. Time**
Vanilla GP	0.3816712	0.3820767	40:00
Partitioned GP	0.0273235	0.0273373	48:00
Partitioned GP + Minibatch	**0.0158897**	**0.0158592**	01:43

The objective function in both of these setups is to minimize the average MSE across all pairs sample-reconstruction from some given dataset. The evolutionary process is executed for 40 generations in both setups. In every generation, each individual of the population is tested against the entire sample dataset, the resulting MSEs for every instance in the dataset are averaged, and this result is assigned as the fitness for a given individual.

In out third experimental setup we use a *mini-batched* form of training. Instead of presenting the entire dataset to each individual every generation, only a very small, variable, subset of samples (minibatch) of the dataset are presented to each individual in every generation. In other words, the objective function is minimizing MSE average for all samples in a minibatch. The number of generations is chosen such that, *no. of generations* $\times$ *size of minibatches* $=$ *size of complete data set.* The minibatches conform a partition, in the mathematical sense, of the entire dataset. In this way, each sample of the dataset is seen only once by 60 indivuals (the size of the population). Notice that this does not mean that all individuals ever see all samples once, because many individuals will not make it to further generations, and will never be tested against most of the samples in the dataset. Even more interesting, is the fact that the final top performer individual (the solution returned by the GP) might actually be tested against only the final minibatch, given that this individual is the result from a crossover or mutation from individuals of the penultimate generation. The purpose of this setup, is to dramatically reduce the computational cost of the GP algorithm.

5.3. Results

We used MNIST (LeCun, 1998) digits dataset to test our experimental setups. MNIST is a dataset composed of $70,000$ samples. Each sample consist of an image of 28×28 gray scale pixels. Each sample is an image of a handwritten digit. The values of the pixels are in the range of $[0, 1]$. Each one of the 784 pixels that compose the images are used as features, i.e. these are the variables that GP individuals can use as zero-argument functions for their encoding forests. The compact representation $\mathbf{L}$ consist in 588 new features potentially distinct from the original 784 pixels. This compact representation is then decoded back into 784 features to represent reconstructions; 588 features from the compact representation are variables that GP individuals can use as zero-argument functions for their decoding forests.

Table 1 shows resulting average MSEs for different setups tested, as well as their running time. MNIST is split in training and a testing sets, consisting of $60,000$ and $10,000$ samples, respectively. For each setup, GP population is evolved using only half of the training dataset. This is done because of time and computational constrains. The top performer individual that results from the evolution process is tested with the testing set, composed of images never before seen during the evolution process. The first and second columns of Table 1 indicate the average MSE over half training and full testing set respectively. The third column refers to the total time elapsed during the evolution process for each setup, in hh:mm format.

The Partitioned GP + Minibatch setup was configured as follows: minibatches were composed of 100

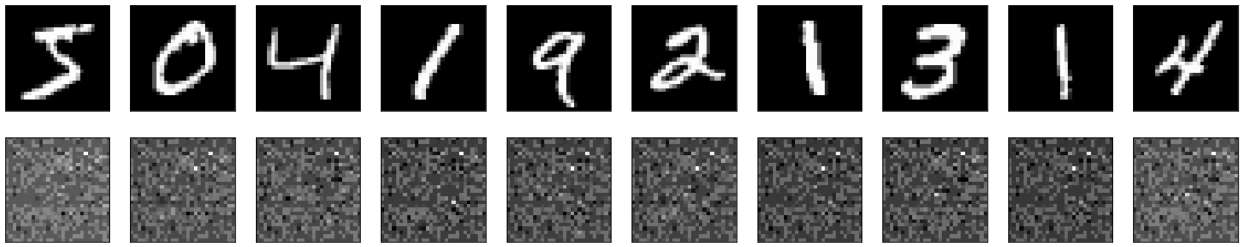

Figure 15: Top row: first ten images of the training dataset. Bottom row: their resulting reconstruction generated by the top performer individual returned by the Vanilla GP after the entire evolutionary process of 40 generations.

samples, and since the training data set is limited to $30,000$ samples, the algorithm runs for 300 generations.

All experiments were done in a workstation with an Intel Xeon CPU with 10 physical cores at 2.9 GHz, with two virtual cores per each physical core, to amount for a total of 20 processing threads, 16 GB of RAM, running Ubuntu Linux 16.04. Algorithms implementation and setups were done by using an in-house software library developed in Python version 3.6. Accelerated NumPy library is used only in the final step of fitness evaluation (averaging the MSE of all sample-reconstruction pairs) of each individual. The Vanilla GP can make use of the multiple processing cores by parallelizing the evaluation of sample-reconstructions pairs. On the other hand, partitioned versions of GP distribute evolution of multiple miniautoencoders across most (but not all) processing threads available, and in this case the evaluations of the sample-reconstructions pairs is done sequentially for each miniautoencoder.

A visual depiction of the performance of the synthesized autoencoders is shown in Figures 15 through 17. Fig. 15 shows the reconstruction by the best autoencoder after the entire evolutionary process of the Vanilla GP, for the first ten images in the training set, along with the original images. Fig. 16 shows the gradual increase in performance obtained from the Partitioned GP through the evolutionary process. Fig. 17 compares the reconstruction for the first ten images in the training set, as obtained by the best autoencoders generated by the three different experimental setups. It is important to notice, that these ten images are the first used for training in the Partitioned GP + Minibatch approach, they are only seen in the first generation, and never used again in further generations, in this setup.

5.3.1. Further Studies

We carried away a more complete experimental study of the Partitioned GP + Minibatch setup. We performed runs of this approach varying the size of the minibatches as well as allowing the algorithm to process more than just one time each sample. Since this setup is far more computationally efficient than the first two, we could also perform the same study using the complete training set of MNIST. Tables 2 and 3 shows results in these experiments. Table 2 shows results of using minibatches of size 30, 60, 100, 300, and 600; as well as allowing the algorithm to give one, two and five forward passes to the training data, for the case when we use only half of the training set. The number of generations is determined by those two parameters. For example, if minibatches are of size 30 (600), and the algorithm only gives one forward pass over data, that means 1000 (50) generations; but if the algorithm has to execute two forward passes over the training samples, then the number of generations doubles, and so on. Table 3 shows these same results but when using the complete training dataset. In this case, if minibatches are of size 30 (600), and the algorithm only gives one forward pass over the data, that means 2000 (100) generations.

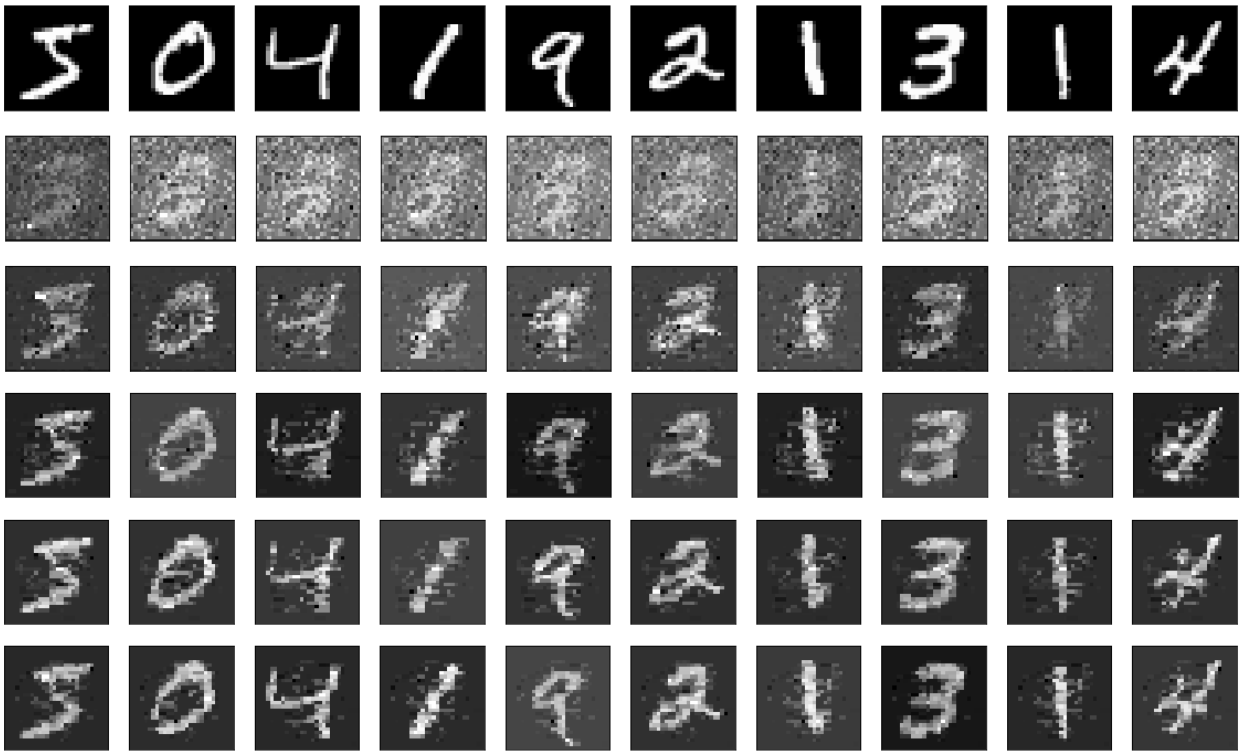

Figure 16: A depiction of the reconstruction generated by the top performer individuals during the evolution process of the Partitioned GP. From top row to bottom:the original first ten images of the training set, their best reconstruction obtained after 0, 10, 20, 30 and 40 generations, respectively.

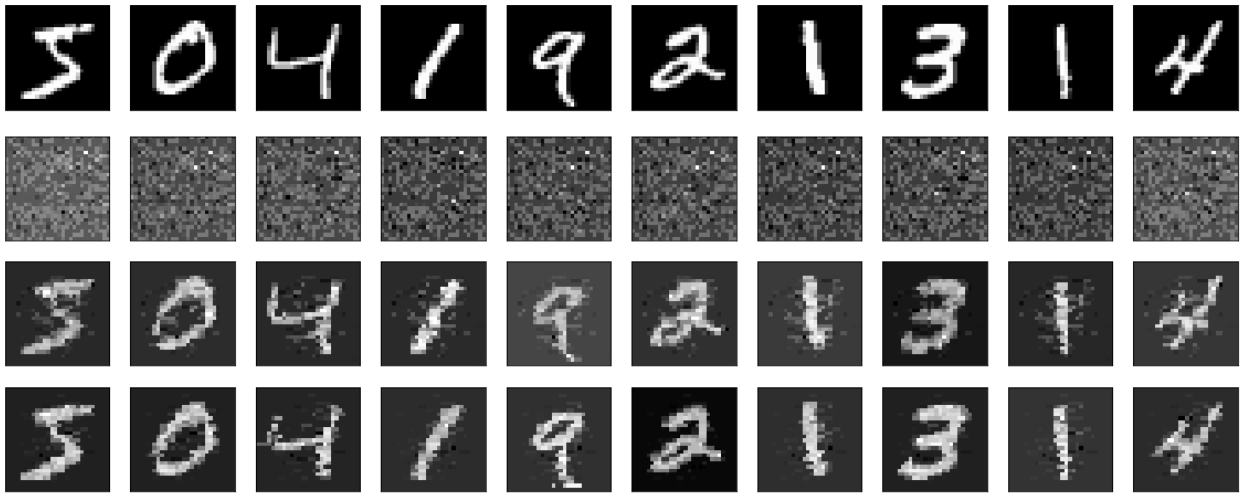

Figure 17: Comparisson of the reconstruction of the three experimental setups. From top to bottom: original first 10 images from the training set, best Vanilla GP reconstruction, best Partitioned GP reconstruction, and best Partitioned GP + minibatch approach reconstruction.

Table 2: Average MSEs and execution times for a Partitioned GP + Minibatch approach varying the size of the minibatches to 30, 60, 100, 300, and 600 samples; and giving 1, 2 and 5 forward passes over the sample set. These results are using the half, 30000 samples, MNIST training dataset for training.

	Mini Batch Size														
	30			60			100			300			600		
Passes	Training	Testing	Time	Training	Testing	Time	Training	Testing	Time	Training	Testing	Time	Training	Testing	Time
1	0.01463	0.01464	02:31	0.01561	0.015568	01:51	0.01588	0.015859	01:43	0.02052	0.02044	01:26	0.02615	0.02616	01:24
2	0.01218	0.01218	04:52	0.01309	0.013029	03:35	0.01277	0.012751	03:15	0.01802	0.01800	02:46	0.02185	0.02183	02:48
5	0.01014	0.01010	11:54	**0.01002**	**0.009988**	09:06	0.01009	0.010077	07:57	0.01338	0.01337	06:55	0.01557	0.01558	06:30

Table 3: Average MSEs and execution times for a Partitioned GP + Minibatch approach varying the size of the minibatches to 30, 60, 100, 300, and 600 samples; and giving 1, 2 and 5 forward passes over the sample set. These results are using the entire, 60000 samples, MNIST training dataset for training.

	Mini Batch Size														
	30			60			100			300			600		
Passes	Training	Testing	Time	Training	Testing	Time	Training	Testing	Time	Training	Testing	Time	Training	Testing	Time
1	0.01338	0.01334	04:44	0.01279	0.012737	03:36	0.01345	0.013442	03:15	0.01764	0.01764	02:49	0.02023	0.02028	02:39
2	0.01049	0.01045	09:31	0.01127	0.011232	07:24	0.01153	0.011486	06:32	0.01457	0.01449	05:30	0.01742	0.01738	05:31
5	0.00928	0.00921	23:27	**0.00883**	**0.008763**	18:04	0.00951	0.009502	16:27	0.01116	0.01115	14:26	0.01418	0.01414	13:23

5.3.2. Comparison with Neural Networks

We picked up the best performing setup from our complete study (minibatches of size 60) and compared its performance against two different ANN based autoencoders. The first architecture consisted in a fully connected multilayer Perceptron (MLP) with one hidden layer; its architecture is as follows: the input is of size 784 (the no. of pixels of MNIST images), the input is fully connected to a hidden layer of 588 (same compression ratio as our GP autoencoder) rectified linear units (ReLU) (Nair & Hinton, 2010), and the hidden layer is fully connected to the output layer of 784 (same size as input) ReLUs. The purpose of this setup is to perform a layer-to-layer (as in GP layers vs ANN layers) and epoch-to-epoch (as in passes over training data) comparison. Nevertheless, this is an admittedly disadvantageous comparison for the ANN autoencoder, first off because the GP autoencoder can perform multiple non-linear transformations in cascade per layer, whereas the ANN can only perform 1 per layer, and second, because the GP approach levers from a population of multiple individuals to explore the solution space, whereas the ANN optimizes over a single set of parameters. Nonetheless we still believe this comparison is useful to contrast the behavior of the two approaches. We also set the size of the minibatches of the MLP to 60, just as in our GP approach. We implemented the MLP in TensorFlow Deep Learning library (Abadi *et al.*, 2016) using the Keras frontend (Chollet, 2017).

The second architecture consisted in a deep autoencoder (DA) as proposed in (Hinton & Salakhutdinov, 2006b). We rearranged the layers of the network in the following way: the input remains consisting of 784 variables, followed by encoding hidden layers of size 1000-800-650-588 for the encoder part of the network; everything else (activation units, training method, batch sizes, etc.) remained the same as proposed by Hinton & Salakhutdinov (2006b), including the fact that the decoder part of the network is a mirror of the encoder, and then follows a structure of four layers of 650-800-1000-784 units each. The purpose of this setup is to compare our GP approach against a DA capable of performing as much non-linear transformations as our GP autoencoder (remember that our encoding and decoding GP forest consists in trees of depth 4). However, we pre-trained this DA as specified in (Hinton & Salakhutdinov, 2006b); this confers it an advantage over our GP autoencoder. Still, this DA is helpful to compare the performance of our GP approach against the performance of a far more modern ANN approach than the simple one-hidden layer MLP.

We performed the comparison using the full training set only. Table 4 shows the results of each approach, for the different amounts of forward passes/epochs.

Table 4: An MSE comparison between the proposed GP approach, a simple MLP autoencoder and a Deep autoencoder (DA) . The column passes refers to the no. of times each algorithm is allowed to process each single sample for training purposes. In the case of the MLP and the DA, this is equivalent to the parameter known as *epochs*.

Passes	**GP**		**MLP**		**DA**	
	Testing	Time	Testing	Time	Testing	Time
1	0.01273	03:36:00	0.0467	00:00:12	0.00641	00:09:00
2	0.01123	07:24:00	0.035	00:00:24	0.00594	00:12:00
5	0.00876	18:04:00	0.0215	00:01:04	0.00529	00:20:00
50	-	-	0.0038	00:11:03	0.00367	02:25:00
200	-	-	-	-	0.00285	08:00:00

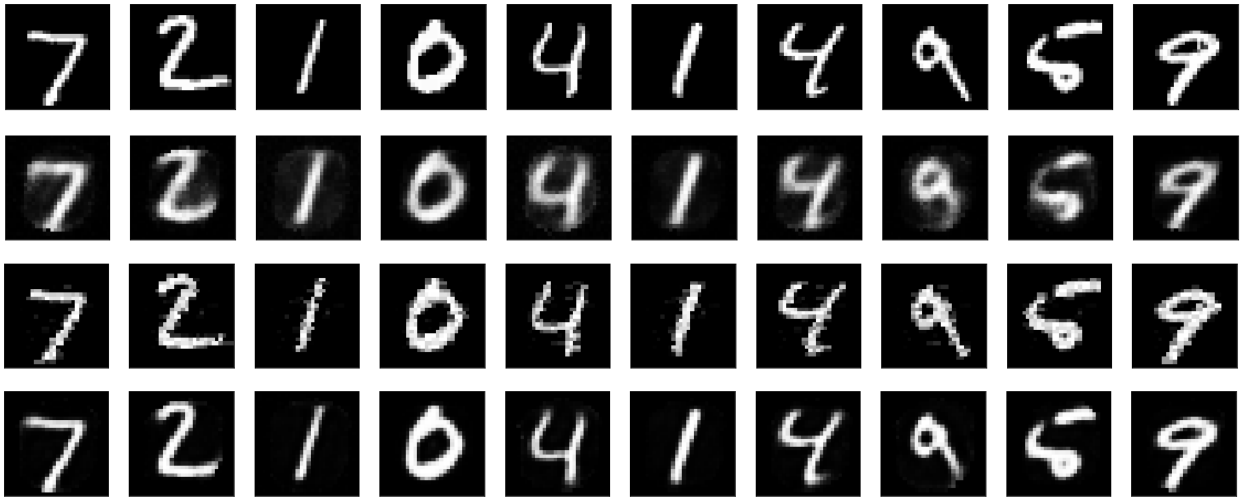

Figure 18: Reconstruction comparison of first ten images of tMNIST testing set, as generated by different autoencoders. From top row to bottom: original images, reconstructions generated by the autoencoder obtained by training an MLP for 5 epochs, by an autoencoder synthesized with a Partitioned GP evolved with minibatches of size 60 and 5 forward passes, and by the autoencoder obtained by training a MLP for 50 epochs. The full training set was used for training/evolving in all cases.

Fig. 18 shows a visual comparison of the reconstructions generated by the different autoencoders obtained with both the GP and the MLP, for the first ten images of MNIST testing set.

5.4. Discussion and Follow-up Work

From the results presented in the previous subsection we can observe that part of our hypothesis holds, since the GP partitioned versions achieve performances one order of magnitude higher than the Vanilla GP. A closer examination of the forests and trees generated by the bests individuals of the Vanilla GP confirms our suspicions: the encoding trees generate new features that then are used by the deconding trees to reconstruct features that are completely unrelated to the original features used by the encoding trees. That is, the search space for zero-argument functions is too big to be efficiently explored by the evolutionary process given the limited amount of time (generations) and computational resources (population size) assigned to it. The Vanilla GP never gets to correctly align original features-compact representation features-reconstructed features in the encoding-decoding trees. In comparison, the partitioned GP approaches are far more efficient, since they do not have to perform this task at all.

Even more interesing, is the fact that the minibatch version of the partitioned GP is better in solution quality than the full batch partitioned GP, as well as being considerably faster. This makes this approach even more efficient than any of the other two. We cannot yet fully explain the reasons for this phenomenon, but we suspect that the semi online form of evolutionary training contributes a form regularization to the GP autoencoder. It also noteworthy, from the results of the further study on the minibatch version, that there seems to exist a sweet spot regarding to minibatch size. If the minibatches are too small, it requires far more generations to sweep over the entire training dataset, which translates to more GP evolutionary operations (crossover, mutation, selection), which are computarionally expensive; on the other hand, if the minibatches are too big, the quality of the obtained solutions decays, and approximates to that of the full batch version. Further research is needed in this regard.

When compared against a one-hidden layer ANN autoencoder, results show that GP behaves quite different from ANN. GP can quickly (in terms of passes over the training data) build acceptable encoding-decoding models, while an ANN that attempts to generate a representation of 588 features from 785 initial features has just too many parameters to adjust. This could be due the way the GP explore the solution search space (through a population), nonetheless this still means the GP is simply a more efficient approach in terms of data usage. When compared against a deep autoencoder, results shows a clear advantage for the DA. However it is important to remember that we are comparing a single layer GP against a multilayer ANN, and that the ANN was granted a pre-training process.

Nevertheless, our GP approach is still quite behind ANN in terms of total execution time. However this has more to do with the current state of the software implementations of each. While the GP was implemented in pure Python and is prototype grade, TensorFlow is a highly optimized, mature, enterprise grade library.

The inmediate steps to take from this point in our research, is to test our framework, in its current state, with other datasets to verify the results found so far hold. MNIST is fairly simple, it consist of almost monochrome images. We also wish to test our approach limiting the number of training samples available, and compare its performance for such scenario against the MLP.

6. Conclusions

In this section we summarize the present document, enumerate the expected contributions of this research proposal, and give some final thoughts regarding the current state of our framework and the preliminary results so far.

In this research proposal we approached the area of representation learning, we reviewed the current state of the art of representation learning with Genetic Programming and proposed a new taxonomy to classify the efforts done so far in this regard. Our proposed taxonomy allowed us to clearly discern the shortcomings of the state of the art works in representation learning with GP and signaled us a research direction. We proposed a research hypothesis that attempts to capture and bring the goodnesses of conventional Deep Learning to representation learning with genetic programming. And finally, we also provided a set of preliminary results that suggest our hypothesis might hold if future research and development is carried out.

6.1. Expected Contributions

The following contributions to the computer science field in general can be achieved:

1. A brand new framework for representation learning based on evolutionary computation. A framework that does not require the contribution of human expert knowledge, and that can potentially learn any representation needed (universal function approximation), possibly tying with the current state of the art that relies on deep neural networks.
2. A new taxonomy to better understand the state of the art in representation learning with Genetic Programming.
3. A software library that implements the proposed framework.
4. A new research field, as well as new directions for long term future research and innovation.

6.2. Preliminary Results Remarks

Although the execution time of our proposed approach is still far from being competitive with the conventional algorithms based on neural network, the current state of our research already represent a contribution to the fields of evolutionary computation, machine learning and computer science in general. We believe that if we can expand and confirm our results with other datasets, our work done so far is subject for publishing, and we wish to pursue such endeavor.

As already discussed in Sec. 5, our preliminary results suggest that an efficient implementation in software of a Genetic Programming algorithm, one can make use of vectorial instructions and cache level memory found in current CPUs (just as neural networks implementations do), or the massive parallel architectures of graphics processing units, could level the field for our framework. Unfortunately, this task is not considered part of our research proposal, but can be considered already a technology research direction worthy of consideration, pointed out by our research.

6.3. Timetable of Activities

Table 5 shows schedule of activities necessary to achieve the goals of this research proposal, in accordance with the methodology proposed in Sec. 4.6.

At the present time we are working in the continuous development of the software library and testing our framework in image datasets under a self-supervised approach for GP of only one layer. The next steps after the defense of this research proposal will be the experimental study related to number of layers and subsets C_i size, as well as the implementation of a mesh grid topology for interconnection of subsets C_i.

Table 5: Thesis project Gantt diagram

Task	Quarters															
	2017				2018				2019				2020			
	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4
Literature review		x	x	x	x	x	x	x	x	x	x	x				
Development of software library		x	x	x	x	x	x	x	x	x	x	x				
Test framework in self-supervised learning		x	x	x	x											
Perform study related to numbers of layers				x	x	x	x	x								
Assess connection topologies for subsets C_i				x	x	x	x	x								
Implement incremental learning mechanisms			x	x	x	x	x	x								
Journal article on Autoencoders with DeepGP						x										
Tests framework in supervised learning						x	x	x	x							
Extend framework for multiple layer evolution							x	x	x							
Journal article on Classification with DeepGP										x						
Experimental study with images datasets		x	x	x	x	x	x	x	x							
Experimental study with other datasets										x	x	x				
Thesis writing											x	x	x			
Thesis dissertation													x			

References

Abadi, Martín, Agarwal, Ashish, Barham, Paul, Brevdo, Eugene, Chen, Zhifeng, Citro, Craig, Corrado, Greg S, Davis, Andy, Dean, Jeffrey, Devin, Matthieu, *et al.* 2016. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*.

Agrawal, Rakesh, Imieliński, Tomasz, & Swami, Arun. 1993. Mining association rules between sets of items in large databases. *Pages 207–216 of: Acm sigmod record*, vol. 22. ACM.

Al-Sahaf, Harith, Song, Andy, Neshatian, Kourosh, & Zhang, Mengjie. 2012. Two-tier genetic programming: Towards raw pixel-based image classification. *Expert Systems with Applications*, **39**(16), 12291–12301.

Allen, Franklin, & Karjalainen, Risto. 1999. Using genetic algorithms to find technical trading rules. *Journal of financial Economics*, **51**(2), 245–271.

Alpaydin, Ethem. 2014. *Introduction to machine learning*. MIT press.

Axelrod, Ben. 2007. *Genetic Programming*. https://en.wikipedia.org/wiki/File:Genetic_Program_Tree.png. Accessed 05/05/17.

Ayodele, Taiwo Oladipupo. 2010. Types of machine learning algorithms. *In: New advances in machine learning*. InTech.

Bellman, Richard. 1961. *Adaptive control process: a guided tour*.

Bengio, Yoshua, *et al.* 2009. Learning deep architectures for AI. *Foundations and trends® in Machine Learning*, **2**(1), 1–127.

Bengio, Yoshua, Courville, Aaron, & Vincent, Pascal. 2013. Representation learning: A review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence*, **35**(8), 1798–1828.

Beyer, Hans-Georg, & Schwefel, Hans-Paul. 2002. Evolution strategies–A comprehensive introduction. *Natural computing*, **1**(1), 3–52.

Bishop, Christopher M. 2006. *Pattern recognition and machine learning*. springer.

Cano, Alberto, Ventura, Sebastián, & Cios, Krzysztof J. 2017. Multi-objective genetic programming for feature extraction and data visualization. *Soft Computing*, **21**(8), 2069–2089.

CEDAR. 1992. *1, USPS Office of Advanced Technology*.

Chollet, Francois. 2016. *Building Autoencoders in Keras*.

Chollet, François. 2017. Keras (2015). *URL http://keras. io*.

Chou, C-H, Su, M-C, & Lai, Eugene. 2004. A new cluster validity measure and its application to image compression. *Pattern Analysis and Applications*, **7**(2), 205–220.

Ciregan, Dan, Meier, Ueli, & Schmidhuber, Jürgen. 2012. Multi-column deep neural networks for image classification. *Pages 3642–3649 of: Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*. IEEE.

Collobert, Ronan, Weston, Jason, Bottou, Léon, Karlen, Michael, Kavukcuoglu, Koray, & Kuksa, Pavel. 2011. Natural language processing (almost) from scratch. *Journal of Machine Learning Research*, **12**(Aug), 2493–2537.

Damianou, Andreas, & Lawrence, Neil. 2013. Deep gaussian processes. *Pages 207–215 of: Artificial Intelligence and Statistics*.

Davies, David L, & Bouldin, Donald W. 1979. A cluster separation measure. *IEEE transactions on pattern analysis and machine intelligence*, 224–227.

Dunn, Joseph C. 1974. Well-separated clusters and optimal fuzzy partitions. *Journal of cybernetics*, **4**(1), 95–104.

Esfahanipour, Akbar, & Mousavi, Somayeh. 2011. A genetic programming model to generate risk-adjusted technical trading rules in stock markets. *Expert Systems with Applications*, **38**(7), 8438–8445.

Fernando, Chrisantha, Banarse, Dylan, Reynolds, Malcolm, Besse, Frederic, Pfau, David, Jaderberg, Max, Lanctot, Marc, & Wierstra, Daan. 2016. Convolution by evolution: Differentiable pattern producing networks. *Pages 109–116 of: Proceedings of the 2016 on Genetic and Evolutionary Computation Conference*. ACM.

Gogna, Anupriya, & Majumdar, Angshul. 2016. Semi Supervised Autoencoder. *Pages 82–89 of: International Conference on Neural Information Processing*. Springer.

Goldberg, David E, & Deb, Kalyanmoy. 1991. A comparative analysis of selection schemes used in genetic algorithms. *Foundations of genetic algorithms*, **1**, 69–93.

Goodfellow, Ian, Bengio, Yoshua, & Courville, Aaron. 2016. *Deep learning*. MIT press.

He, Kaiming, Zhang, Xiangyu, Ren, Shaoqing, & Sun, Jian. 2016. Deep residual learning for image recognition. *Pages 770–778 of: Proceedings of the IEEE conference on computer vision and pattern recognition*.

Hinton, G. E., & Salakhutdinov, R. R. 2006a. Reducing the Dimensionality of Data with Neural Networks. *Science*, **313**(5786), 504–507.

Hinton, Geoffrey, Deng, Li, Yu, Dong, Dahl, George E, Mohamed, Abdel-rahman, Jaitly, Navdeep, Senior, Andrew, Vanhoucke, Vincent, Nguyen, Patrick, Sainath, Tara N, *et al.* 2012. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, **29**(6), 82–97.

Hinton, Geoffrey E, & Salakhutdinov, Ruslan R. 2006b. Reducing the dimensionality of data with neural networks. *science*, **313**(5786), 504–507.

Hornik, Kurt, Stinchcombe, Maxwell, & White, Halbert. 1989. Multilayer feedforward networks are universal approximators. *Neural networks*, **2**(5), 359–366.

Huang, Gao, Sun, Yu, Liu, Zhuang, Sedra, Daniel, & Weinberger, Kilian Q. 2016. Deep networks with stochastic depth. *Pages 646–661 of: European Conference on Computer Vision*. Springer.

Huang, Gary B., Ramesh, Manu, Berg, Tamara, & Learned-Miller, Erik. 2007 (October). *Labeled Faces in the Wild: A Database for Studying Face Recognition in Unconstrained Environments*. Tech. rept. 07-49. University of Massachusetts, Amherst.

Hughes, Gordon. 1968. On the mean accuracy of statistical pattern recognizers. *IEEE transactions on information theory*, **14**(1), 55–63.

Koza, John R. 1992. *Genetic programming: on the programming of computers by means of natural selection*. Vol. 1. MIT press.

Koza, John R. 1994. Genetic programming II: Automatic discovery of reusable subprograms. *Cambridge, MA, USA*.

Krizhevsky, Alex, & Hinton, G. 2010. Convolutional deep belief networks on cifar-10. *Unpublished manuscript*, **40**.

Krizhevsky, Alex, Sutskever, Ilya, & Hinton, Geoffrey E. 2012. Imagenet classification with deep convolutional neural networks. *Pages 1097–1105 of: Advances in neural information processing systems*.

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

LeCun, Yann. 1998. The MNIST database of handwritten digits. *http://yann. lecun. com/exdb/mnist/*.

LeCun, Yann, Bengio, Yoshua, & Hinton, Geoffrey. 2015. Deep learning. *Nature*, **521**(7553), 436–444.

Lee, Daniel D, & Seung, H Sebastian. 1999. Learning the parts of objects by non-negative matrix factorization. *Nature*, **401**(6755), 788.

Lee, Daniel D, & Seung, H Sebastian. 2001. Algorithms for non-negative matrix factorization. *Pages 556–562 of: Advances in neural information processing systems*.

Limón, Mauricio García, Escalante, Hugo Jair, Morales, Eduardo, & Pineda, Luis Villaseñor. 2015. Class-specific feature generation for 1NN through genetic programming. *Pages 1–6 of: Power, Electronics and Computing (ROPEC), 2015 IEEE International Autumn Meeting on*. IEEE.

Lin, Jung-Yi, Ke, Hao-Ren, Chien, Been-Chian, & Yang, Wei-Pang. 2008. Classifier design with feature selection and feature extraction using layered genetic programming. *Expert Systems with Applications*, **34**(2), 1384–1393.

Littman, Michael, & Isbell, Charles. 2015. *Machine Learning - Supervised Learning*. https://www.youtube.com/watch?v=Ki2iHgKxRBo. Accessed 11/02/16.

Liu, Huan, & Motoda, Hiroshi. 1998. *Feature extraction, construction and selection: A data mining perspective*. Vol. 453. Springer Science & Business Media.

Liu, Li, Shao, Ling, Li, Xuelong, & Lu, Ke. 2015. Learning Spatio-Temporal Representations for Action Recognition: A Genetic Programming Approach. *IEEE Transactions on Cybernetics*.

Lohpetch, Dome, & Corne, David. 2009. Discovering effective technical trading rules with genetic programming: Towards robustly outperforming buy-and-hold. *Pages 439–444 of: Nature & Biologically Inspired Computing, 2009. NaBIC 2009. World Congress on*. IEEE.

Lohpetch, Dome, & Corne, David. 2010. Outperforming buy-and-hold with evolved technical trading rules: Daily, weekly and monthly trading. *Pages 171–181 of: European Conference on the Applications of Evolutionary Computation*. Springer.

Lohpetch, Dome, & Corne, David. 2011. Multiobjective algorithms for financial trading: Multiobjective out-trades single-objective. *Pages 192–199 of: Evolutionary Computation (CEC), 2011 IEEE Congress on*. IEEE.

Loveard, Thomas, & Ciesielski, Victor. 2001. Representing classification problems in genetic programming. *Pages 1070–1077 of: Evolutionary Computation, 2001. Proceedings of the 2001 Congress on*, vol. 2. IEEE.

Mika, Sebastian, Ratsch, Gunnar, Weston, Jason, Scholkopf, Bernhard, & Mullers, Klaus-Robert. 1999. Fisher discriminant analysis with kernels. *Pages 41–48 of: Neural Networks for Signal Processing IX, 1999. Proceedings of the 1999 IEEE Signal Processing Society Workshop.* IEEE.

Mitchell, Tom. 1997. *Machine learning. WCB*.

Morse, Gregory, & Stanley, Kenneth O. 2016. Simple Evolutionary Optimization Can Rival Stochastic Gradient Descent in Neural Networks. *Pages 477–484 of: Proceedings of the 2016 on Genetic and Evolutionary Computation Conference*. ACM.

Moscinski, Rafal, & Zakrzewska, Danuta. 2015. Building an Efficient Evolutionary Algorithm for Forex Market Predictions. *Pages 352–360 of: International Conference on Intelligent Data Engineering and Automated Learning*. Springer.

Myszkowski, Paweł B, & Bicz, Adam. 2010. Evolutionary algorithm in forex trade strategy generation. *Pages 81–88 of: Computer Science and Information Technology (IMCSIT), Proceedings of the 2010 International Multiconference on*. IEEE.

Nair, Vinod, & Hinton, Geoffrey E. 2010. Rectified linear units improve restricted boltzmann machines. *Pages 807–814 of: Proceedings of the 27th international conference on machine learning (ICML-10)*.

Neely, Christopher, Weller, Paul, & Dittmar, Rob. 1997. Is technical analysis in the foreign exchange market profitable? A genetic programming approach. *Journal of financial and Quantitative Analysis*, **32**(4), 405–426.

Nguyen, Anh, Yosinski, Jason, & Clune, Jeff. 2015. Deep neural networks are easily fooled: High confidence predictions for unrecognizable images. *Pages 427–436 of: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*.

Olague, Gustavo, Clemente, Eddie, Dozal, León, & Hernández, Daniel E. 2014. Evolving an artificial visual cortex for object recognition with brain programming. *Pages 97–119 of: EVOLVE-A Bridge between Probability, Set Oriented Numerics, and Evolutionary Computation III*. Springer.

Parkins, AD, & Nandi, Asoke K. 2004. Genetic programming techniques for hand written digit recognition. *Signal Processing*, **84**(12), 2345–2365.

Plaut, David C. 2016. *Unsupervised Learning*.

Poli, R, Langdon, WB, & McPhee, NF. 2008. A field guide to genetic programming (With contributions by JR Koza)(2008). *Published via http://lulu. com*.

Potvin, Jean-Yves, Soriano, Patrick, & Vallée, Maxime. 2004. Generating trading rules on the stock markets with genetic programming. *Computers & Operations Research*, **31**(7), 1033–1047.

Russakovsky, Olga, Deng, Jia, Su, Hao, Krause, Jonathan, Satheesh, Sanjeev, Ma, Sean, Huang, Zhiheng, Karpathy, Andrej, Khosla, Aditya, Bernstein, Michael, Berg, Alexander C., & Fei-Fei, Li. 2015. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, **115**(3), 211–252.

Salakhutdinov, Ruslan, & Hinton, Geoffrey. 2009. Deep boltzmann machines. *Pages 448–455 of: Artificial Intelligence and Statistics*.

Sanderson, C. 2014. *LFWcrop Face Dataset*.

Schmid, Cordelia, Mohr, Roger, & Bauckhage, Christian. 2000. Evaluation of interest point detectors. *International Journal of computer vision*, **37**(2), 151–172.

Seitz, Charles L. 1985. The cosmic cube. *Communications of the ACM*, **28**(1), 22–33.

Sermanet, Pierre, Chintala, Soumith, & LeCun, Yann. 2012. Convolutional neural networks applied to house numbers digit classification. *Pages 3288–3291 of: Pattern Recognition (ICPR), 2012 21st International Conference on*. IEEE.

Shao, Ling, Liu, Li, & Li, Xuelong. 2014. Feature learning for image classification via multiobjective genetic programming. *IEEE Transactions on Neural Networks and Learning Systems*, **25**(7), 1359–1371.

Sotelo, Arturo, Guijarro, Enrique, Trujillo, Leonardo, Coria, Luis N, & Martínez, Yuliana. 2013. Identification of epilepsy stages from ECoG using genetic programming classifiers. *Computers in biology and medicine*, **43**(11), 1713–1723.

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

Storn, Rainer, & Price, Kenneth. 1997. Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces. *Journal of global optimization*, **11**(4), 341–359.

Sutskever, Ilya, Vinyals, Oriol, & Le, Quoc V. 2014. Sequence to sequence learning with neural networks. *Pages 3104–3112 of: Advances in neural information processing systems*.

Sutton, Richard S, & Barto, Andrew G. 1998. *Reinforcement learning: An introduction*. Vol. 1. MIT press Cambridge.

Szegedy, Christian, Zaremba, Wojciech, Sutskever, Ilya, Bruna, Joan, Erhan, Dumitru, Goodfellow, Ian, & Fergus, Rob. 2013. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*.

Szegedy, Christian, Liu, Wei, Jia, Yangqing, Sermanet, Pierre, Reed, Scott, Anguelov, Dragomir, Erhan, Dumitru, Vanhoucke, Vincent, & Rabinovich, Andrew. 2015. Going deeper with convolutions. *Pages 1–9 of: Proceedings of the IEEE conference on computer vision and pattern recognition*.

Tang, Yichuan. 2013. Deep learning using linear support vector machines. *arXiv preprint arXiv:1306.0239*.

Teredesai, Ankur, Park, Jaehwa, Govindaraju, Venu, *et al.* 2001. Active handwritten character recognition using genetic programming. *Lecture notes in computer science*, 371–379.

Trujillo, Leonardo, & Olague, Gustavo. 2006. Synthesis of interest point detectors through genetic programming. *Pages 887–894 of: Proceedings of the 8th annual conference on Genetic and evolutionary computation*. ACM.

Vladislavleva, Ekaterina Yurievna. 2008. *Model-based problem solving through symbolic regression via pareto genetic programming*. Ph.D. thesis, CentER, Tilburg University.

Williams, DRGHR, & Hinton, Geoffrey. 1986. Learning representations by back-propagating errors. *Nature*, **323**(6088), 533–538.

Wold, Svante, Esbensen, Kim, & Geladi, Paul. 1987. Principal component analysis. *Chemometrics and intelligent laboratory systems*, **2**(1-3), 37–52.

Zhang, Mengjie, Ciesielski, Victor B, & Andreae, Peter. 2003. A domain-independent window approach to multiclass object detection using genetic programming. *EURASIP Journal on Advances in Signal Processing*, **2003**(8), 206791.