Deep Representation Learning with Genetic Programming

por

Lino Alberto Rodríguez Coayahuitl

Tesis sometida como requerimiento parcial para obtener el grado de

Doctor en Ciencias, en el área de Ciencias de la Computación

por el

Instituto Nacional de Astrofísica, Óptica y Electrónica

Julio, 2020

Sta. Ma. Tonantzintla,
Puebla, México.

Director:

Dr. Hugo Jair Escalante Balderas

Co-Directora:

Dra. Alicia Morales Reyes

PROGRAMA DE POSGRADO EN CIENCIAS

EN CIENCIAS DE LA COMPUTACIÓN

Aprendizaje Profundo de Representaciones mediante Programación Genética

Tesis

para cubrir parcialmente los requisitos necesarios para obtener el grado de

Doctor en Ciencias

Presenta:

Lino Alberto Rodríguez Coayahuitl

Sta. Maria Tonantzintla, Puebla, México

2020

Tesis defendida por

Lino Alberto Rodríguez Coayahuitl

y aprobada por el siguiente comité

Dr. Hugo Jair Esalante Balderas

Codirector del Comité

Dr. Alicia Morales Reyes

Codirector del Comité

Dr.

Miembro del Comité

Dr.

Miembro del Comité

Dr.

Miembro del Comité

Dr.

Miembro del Comité

Dr.

Coordinador del Programa de Posgrado en Ciencias de la Computación

Dr.

Director de Estudios de Posgrado

Diciembre, 2019

Resumen de la tesis presentada como requisito parcial para la obtención del grado de Doctor en Ciencias en Ciencias de la Computación.

Aprendizaje Profundo de Representaciones mediante Programación Genética

En esta tesis se propone el desarrollo de un nuevo modelo de aprendizaje profundo basado en el entorno de trabajo de Programación Genéntica (PG). Aunque el aprendizaje profundo es normalmente considerado una área concerniente exclusivamente al estudio de cierto de tipo de redes neuronales artificiales, en esta tesis se aborda el aprendizaje profundo como un cambio de paradigma, en donde las dos etapas clásicas de una solución basada en aprendizaje maquina, i.e. la extracción de características y la predicción, se fusionan en un solo algoritmo de aprendizaje, representado por una estructura única compuesta por una secuencia de transformaciones no-lineales. La revisión de literatura que se presenta en esta tesis defiende esta nueva perspectiva, al presentar nuevos modelos de aprendizaje profundo propuestos recientemente en la literatura científica, que no basan su implementación en el uso de redes neuronales artificiales.

Como resultado de esta investigación, se proponen dos modelos de aprendizaje profundo basados en PG, uno para aprendizaje no supervisado y otro para aprendizaje supervisado. El modelo propuesto para aprendizaje no supervisado consiste en un método basado en PG para la evolución de algoritmos *autocodificantes*. De acuerdo a la experiencia del autor de esta tesis, ésta es la primera vez que algoritmos autocodificantes se obtienen mediante un modelo de aprendizaje máquina distinto al de las redes neuronales artificiales. El rendimiento obtenido por los algoritmos autocodificantes generados mediante PG es comparable a los que obtenian las redes neuronales profundas de hace diez años; sin embargo, nosotros consideramos que el marco de trabajo de la PG es más general que el de las redes neuronales artificiales, dado que no solamente se busca un vector de pesos, sino toda la estructura de la solución, y que ahí radica la relevancia de este resultado.

El modelo propuesto para aprendizaje supervisado consiste en una modificiación de la representación de la soluciones en PG. Este nuevo modelo, denominado FractalGP, permite la evolución / aprendizaje de manera eficiente de estructuras más *profundas* de lo que la representación clásica de PG permite. La evaluación experimental del modelo propuesto confirma un rendimiento superior con respecto a la versión clásica de PG. Adicionalmente, se propone una variante al FractalGP inspirada en las redes neuronales convolucionales, que perimite la evolución de arquitecturas de aprendizaje profundo donde todas las neuronas artificiales son reemplazadas por árboles sintácticos de PG.

Sin embargo, el resultado que consideramos más significativo de esta investigación está relacionado con un fenómeno que se observa después de un análisis minucioso de las soluciones generadas por el algoritmo FractalGP. Los resultados obtenidos muestran que en la arquitectura propuesta, en la que se reemplazan neuronas por árboles sintácticos, el PG hace que éstos árboles sintácticos evolucionen a algo que se asemeja a neuronas artificiales, es decir, la heurística evolutiva encuentra las piezas faltantes de la heurística de donde se importan el resto de los elementos, perdiendo la capacidad de aportar soluciones novedosas al problema tratado. La conclusión a la que se llega es que intentar importar ciertos aspectos heurísticos del aprendizaje profundo clásico a la PG podría ser un enfoque equivocado. Esto abre nuevas líneas de investigación que habrán de ser valoradas.

Palabras Clave: **Cómputo Evolutivo, aprendizaje máquina, aprendizaje profundo, aprendizaje de representaciones, procesamiento de imágenes.**

Abstract of the thesis presented in partial fulfillment of the requirements of the degree of PhD in Computer Science.

Deep Representation Learning with Genetic Programming

In this thesis, we propose the development of a new Deep Learning (DL) model based on the Genetic Programming (GP) framework. Although DL is typically considered a machine learning (ML) field solely concerned with certain class of artificial neural networks (ANNs), in this thesis we approach DL as a paradigm shift, where the two classical stages of a ML workflow, i.e. feature extraction and prediction, are fusioned in a single ML algorithm, represented by an unified pipeline composed of non-linear transformations. Our literature review, that revolves around recent efforts from the research community at developing new DL architectures that depart from the classical ANN-based models, supports our point of view.

As result from our research, we propose two GP-based DL models, one for unsupervised learning and another for supervised learning. The unsupervised learning model consists in a GP framework aimed at evolving *autoencoder* algorithms. To the best of the author's knowledge, this is the first time autoencoder algorithms are generated though a ML model other than ANNs. The performance obtained by the evolved autoencoders is comparable to the performance of deep networks proposed ten years ago; nevertheless, we consider the GP framework more general than that of ANNs, due to the fact that GP searchs for the entire solution's structure, whereas in the ANNs framework only the weights are optimized, hence the relevance of this result.

The proposed model for supervised learning consists in a modification of GP's individual representation. This new framework, we call Fractal GP, allows to efficiently evolve / learn deeper GP structures than a vanilla GP. Empirical assessment of the proposed model confirms its superior performance over the canonical version of GP. A further extension to the Fractal GP model inspired by Convolutional Neural Networks is proposed, that allows to evolve complete DL convolutional architectures, where all the artificial neurons are replaced by GP abstract syntax trees.

However, what we consider the most important result obtained from our research, concerns with a phenomenon we observed after a closer inspection of the individuals generated by the FractalGP. Results obtained showed that in the proposed model, where artificial neurons are replaced with syntax trees, the GP makes the syntax trees evolve towards structures that resemble artificial neurons, i.e. the artificial evolution heuristic finds the missing elements of the origin heuristic, thus losing the capacity of finding novel solutions to the problem tackled. The conclusion we arrive from this result, is that attempting to import certain heuristics aspects from classical DL into GP, might be a flawed approach. This opens new lines of research for serious consideration.

Keywords: **Evolutionary algorithms, machine learning, deep learning, representation learning, image processing.**

Agradecimientos

A mis directores de tesis, el Dr. Hugo Jair Escalante Balderas y la Dra. Alicia Morales Reyes, por haberme brindado la oportunidad de colaborar con ellos, y por todo su apoyo brindado durante estos tres años de trabajo realizado.

A los miembros del comité de tesis, la Dra. Pilar Gómez Gil, el Dr. Ariel Carrasco Ochoa, el Dr. Eduardo Morales Manzanares, el Dr. Jose Martinez Carranza, y el Dr. Mario Graff Guerrero, por sus valiosas aportaciones durante las revisiones de avance de tesis.

Al Dr. Enrique Muñoz de Cote, por todos sus conocimientos que compartió conmigo durante mi primer año de estudios doctorales.

Al Dr. Saúl Pomares Hernández, por su fundamental apoyo en momentos críticos durante mi permanencia como estudiante doctoral en el Instituto Nacional de Astrofísica, Óptica y Electrónica.

A todos los investigadores, estudiantes y personal del Instituto Nacional de Astrofísica, Óptica y Electrónica, pero en especial a los compañeros de la Coordinación de Ciencias Computacionales, por toda su enseñanza académica.

A CONACYT y proyecto de investigación No. 436184 por su apoyo económico.

“Esta investigación fue realizada gracias al apoyo del Consejo de Ciencia y Tecnología del Estado de Puebla”.

Table on Contents

Table on Contents (contd)

Table on Contents (contd)

List of Figures

List of Figures (contd)

List of Tables

List of acronyms

EA Evolutionary Algorithm

EC Evolutionary Computation

GA Genetic Algorithm

GP Genetic Programming

ML Machine Learning

DL Deep Learning

MLP Multi-layer Perceptron

ANN Artificial Neural Network

DNN Deep Neural Network

CNN Convolutional Neural Network

AE Autoencoder

GP-AE GP-generated Autoencoder

SGD Stochastic Gradient Descent

CFGP Convolutional Fractal Genetic Programming

Chapter 1. Introduction

Machine learning (ML) 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 ML algorithms, such as supervised learning, unsupervised learning, reinforcement learning algorithms, among others (Ayodele, 2010; Alpaydin, 2014). In this thesis, our focus is primarily on supervised, as well as unsupervised, learning.

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). *Classification* and *regression* are examples of tasks that algorithms can learn to perform through supervised learning. For example, 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, so it can be used later to match new unlabeled images. Regression is similar to classification, but where outputs are continuous instead of discrete.

On the other hand, in unsupervised learning, the algorithms are not presented with labeled pairs of input-output, instead, it is entirely up to the algorithm to find a model that describes structures or patterns found in the input data. A 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 (e.g. in a database of supermarket sales, products that are likely to be bought together). Unsupervised learning can be an intermediate step to preprocess data that later will be fed 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 their actions, receive feedback from the environment in a reward form; their objective is to discover a model that allows them to take actions such that the reward they get is maximized over an hypothetical infinite time horizon.

A question that remains for all types of ML scenarios is, how exactly the input data -*i.e.*, the samples in supervised learning or the state of the world in RL- are presented to an algorithm? In what form is data fed to ML algorithms? If, for example, a classifier were to learn to distinguish cancer cells from healthy cells, it would be easier for the algorithm if vectors of numbers describing shape, size and symmetry of cells were feed to it, instead of raw cells images. Similarly, a software agent designed to learn to play Go game, would have a head-on start if the world's state were a matrix that represented a 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. A features' vector that entirely describes data that an ML 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 requires for representations to be carefully handcrafted by experts of the problem's domain.

On the other hand, *representation learning* consists in automatically finding data representations such that classifiers or predictor systems can more easily extract useful information (Bengio *et al.*, 2013). Representation learning methods can be broadly classified into two categories: feature selection and feature extraction. Feature selection methods attempt to reduce dimensionality of 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 a features 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 it should also be more descriptive, in a sense that generalizes better the sample data, i.e. more abstract (Bengio *et al.*, 2013). Feature extraction methods include: 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 ANNs have automatically produced representations that significantly boosted classification systems accuracy (Ciregan *et al.*, 2012; Krizhevsky *et al.*, 2012; Sermanet *et al.*, 2012). Those ANNs 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 DNNs of several hundred layers (Huang *et al.*, 2016). DNNs 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 Algorithms (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, mutations of candidate solutions (*individuals*), crossover of good performance individuals among a population, and discarding poor performing candidate solutions. Even though GP has been used in the past for representation learning (Bot, 2001; Trujillo & Olague, 2006; 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 competitive results when compared against other state of the art representation learning methods in high dimensionality problems (Parkins & Nandi, 2004; Limón *et al.*, 2015); the kind of problems DNNs excel at.

In this thesis, we researched into the possibility of developing a representation learning framework based on GP, that aims at tackling 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 transform and abstract the initial input representation, up to the point where an useful representation for other ML tasks is obtained. From DL's perspective, the main contribution of this research is a set of new DL models where the fundamental processing units are not artificial neurons, but abstract syntax trees (the basic learning structure of GP). Therefore, this research is one of the first attempts to provide new DL models that depart from the traditional ANN-based approach.

1.1 Genetic Programming

GP is a framework for the evolution of simple computer programs or mathematical functions (Koza, 1992; Poli *et al.*, 2008); GP has been used for diverse set of tasks such as analog (Koza *et al.*, 1997) and

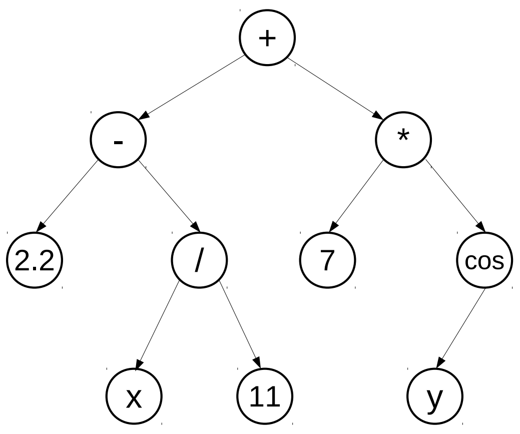


Figure 1.1: Tree structures like this are typically used in GP to represent individual candidate solutions.

digital (Sakanashi *et al.*, 1996) circuits design, hardware design (Comisky *et al.*, 2000), cellular automata rule discovery (Andre *et al.*, 1996), as well as for ML tasks. Being an EA, GP relies in a population of candidate solutions to the problem at hand, these candidate solutions are called individuals. In the context of ML tasks, each individual's performance is tested against a training dataset; the best individuals are selected to reproduce through the use of genetic operations, i.e. generate slightly modified versions of themselves; these new candidate solutions are also evaluated and the best performing replaces the worst original one, leading to a new generation of individuals, the process repeats until a stop criterion is met.

Individuals in GP are *abstract syntax trees* that represent a mathematical function or simple computer programs (Koza, 1992; Poli *et al.*, 2008). In the canonical form, internal nodes of GP trees can be arithmetic operations, such as $+, -, \times$, that operate over two scalar values or maybe trigonometric functions such as $\sin$, $\cos$, $\tan$, that operate over a single scalar value; and leaf nodes are single scalar variables taken from the feature set that describes the ML problem at hand. In this way, data flows from bottom nodes to the top root node where the final output is generated. Fig. 1.1 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 modern variants and advanced GP forms, specially some geared towards high dimensionality problems, leaf nodes may not be single variables, but vectors or even arrays extracted from a large representation space, and internal nodes may be complex functions that, either somehow transform these vector inputs to single scalars or directly operate over one or more of these multidimensional inputs and return as output another value of multiple dimensions. The problem with these higher forms of GP models, is that even though they do learn new data representations, they often carry expert's knowledge of a tackled problem (see Sec. 4.3 for discussion); and thus cannot be considered a representation learning method (because true *representation learning* must be performed automatically, i.e. in an agnostic fashion).

1.2 Problem Statement

High dimensional representation learning problems define intractable search spaces for GP. Using a standard GP approach to tackle representation learning would be computationally expensive, and inefficient, because the search space for the optimal solution includes massive amounts of very poor candidate solutions that a GP has to filter on its own.

We approach the problem of feature extraction using GP in its canonical form; i.e. in order to ensure that we perform representation learning through GP, we aim to process raw data without any human expert's knowledge of the problem domain embedded within GP individuals (in the form of advanced function nodes). From a dataset with an arbitrary number of samples, each sample j represented by a feature vector o_j that belongs to $\mathbb{R}^n$, we wish to learn a new, more compact and/or abstract, representation $\mathbf{M}$, such that each sample j is now represented by a feature vector p_j that belongs to $\mathbb{R}^m$, and such that $m \ll n$. The representation learned might be a by-product of another ML task, such as classification or regression, and not necessarily the main objective GP's evolutionary search or optimization.

1.2.1 A straightforward approach for representation learning with GP

Under a GP methodology based on canonical GP individuals, for every feature to be learned $f_i^m \in \mathbf{M}$, a GP tree is built and denoted as t_i. Tree t_i receives as input n original features, and returns as output the feature f_i^m. Let us suppose, without loss of generality, that each tree $t_i, \forall f_i^m$ is composed of functions of arity 2, i.e. each t_i is a 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 appear on leaf nodes, thus t_i height 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 would use a set of K functions; each internal node can take the form of any of those K functions, then the total size of the search space to explore is $\mathcal{O}(mK^n)$.

Example 1 *Suppose we wish to process a set of images to convert them from an 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 functions* $\{+, -, \times, /\}$*. The GP needs to search for an optimal individual among, at least,* 32×4^{4096} *distinct possible solutions.*

Example 2 *Suppose we wish to perform an image classification task, such that we need to convert a set of images from their raw representation of 64x64 gray scale pixels to an output vector of 10 variables, where each variable of the output vector signals the class to which the image belongs. In this case,* $n = 4096$ *and* $m = 10$*. Therefore, we are set to search for a GP individual composed of 10 trees; each tree, potentially, of height 12. Consider the same set of GP node functions of Example 1. Then, the GP needs to search for an optimal individual among, at least,* 10×4^{4096} *distinct possible solutions.*

These are optimistic, lower bound estimates, since we are not yet taking into account that constants are probably needed as leaf nodes as well; but then again, these estimates 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.

Considering the structural layered processing of Deep Learning could allow to significantly improve GP performance to tackle representation learning. A layered GP scheme that gradually transforms the search space while reducing the computational burden.

1.3 Research Questions

In this section we state the original research questions that led our investigation, as well as accompanying commentaries that correspond to the answers we reached after carrying out our research.

Main Research Question

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

Results obtained by some of our proposed models suggest that rather than completely mimicking the architectural designs of DNNs, we should revisit known, and develop new, ways for GP to evolve collections of subroutines. This might be the GP-oriented way of simultaneously learning to perform feature extraction while also evolving classification systems.

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 algorithms to generate useful representations for classification and clustering tasks.

In this, thesis we propose new GP models for both unsupervised and supervised learning. The unsupervised model aims to directly learn a new, more compact, representation; while the supervised approach tackles a regression problem and only generates intermediate representations as a by-product of the main objective; both approaches being feasible.

2. *What relationship exists between GP performance and the depth of a Deep GP approach?*

   When measuring the performance of an algorithm, we can consider the *efficacy* and *efficiency*. The efficacy in this context refers to the accuracy of a ML algorithm, such as the error obtained in a classification or regression task, independently of how much time the algorithm takes, or any other optimal conditions it may require, to converge to such result; i.e. the precision. The efficiency, considers the precision with respect to something else, such as the computational time invested in training (time efficiency), or the size of a training dataset required (data efficiency), to achieve a given accuracy. Here we refer to time efficiency.

   A trope in the deep learning field is that, in general, an increase in the depth of deep architectures can translate into an increased accuracy (see Sec. 4.1 and He *et al.* (2016)). Our hypothesis revolves more around the idea that a layered processing scheme could make the GP search more efficient (i.e., same accuracy in a shorter time span), but could this also mean that increasing the number of layers in an hypothetical deep GP model could also improve the accuracy of the proposed approach in complex ML tasks?

   Results suggest that efficacy is more related with the way raw input representations are decomposed in order to be subject to local processing (see Ch. 5), rather than to the 'depth' of a GP model. Due to time constraints, this research could not address the efficacy issue extensively; results obtained from our proposed approaches that explicitly attempt to increase the 'depth' of GP models (see Sec. 6.6) are inconclusive: at the moment we cannot determine if such proposed approaches achieve an increase in efficacy or only in efficiency.

3. *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?

The experimental evidence gathered during this research backs the feasibility of incremental or on-line forms of learning in GP. In Ch. 6 we also propose a radically different form of incremental learning more *ad hoc* for population based methods, i.e. evolutionary machine learning algorithms.

4. *How can we circumvent or solve one of the fundamental problems of deep learning, the credit assignment problem (Schmidhuber, 2015), in GP?*

   Our initial proposed approach centered around the idea of evolving layers of intermediate representations in a sequential manner, in a unsupervised learning fashion inspired by earlier deep networks (Hinton & Salakhutdinov, 2006b); but modern deep networks successfully optimize dozens of feature extraction layers simultaneously thanks to the *backpropagation* algorithm (under certain circumstances and in combination with other techniques), could a similar mechanism be implemented in GP?

   In Ch. 6 we establish the argument that GP is a mechanism that circumvents the credit assignment problem: notice how GP is a framework where complex, sequentially processing, systems are evolved. With this insight in mind we propose a modified version of GP that manages to evolve structures akin to deep networks, where artificial neurons are replaced by GP syntax trees, further proving evidence for this argument.

5. *How the proposed method compares with other state-of-the-art representation learning methods, such as PCA, Deep Learning, and expert's knowledge embedded GPs?*

   Our results show that GP still lacks behind state-of-the-art deep learning architectures. Results and comparisons performed for both unsupervised and supervised learning approaches proposed in chapters 5 and 6 respectively, show that GP might be a less precise ML method than DNN-based models, thus signaling that efficacy issues should be addressed before attempting increasing the efficiency of the GP framework.

1.4 Hypothesis

It is possible to achieve a higher efficiency[1] than the straightforward approach in the evolutionary search for representation learning if we direct the search towards individuals that mimic the models of Deep Learn-

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

ing, i.e. models based on multiple GP layers that generate intermediate representations that gradually transform the feature space. With these enhancements, GP can reach a competitive performance with modern deep learning networks.

1.5 Objectives

In this section we state the objectives that led our research.

1.5.1 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.

1.5.2 Specific Objectives

1. To define a set of objective functions to assess solutions, i.e. individuals, at a fitness level, to act as a proxy for their performance in learning useful representations for other ML tasks.
2. To provide conclusive experimental evidence on the behavior of the proposed GP framework performance as a function over the depth, i.e number of layers, of the proposed multi-layer GP architecture.
3. To implement or, if necessary, develop different incremental learning methods in order to avoid full training datasets evaluations every generation.
4. To develop a method that attempts to evolve simultaneously all layers of a multi-layer GP approach.
5. To evaluate the proposed framework on image datasets commonly used in ML research, and compare its performance with state of the art deep architectures.

1.6 Motivation

So far, state of the art results in feature extraction with GP have been achieved only through heavy use of high level functions nodes in GP individuals' representation (e.g. Trujillo & Olague (2006); Shao *et al.* (2013); Yan *et al.* (2014); see Sec. 4.3.1 for further discussion). The problem with this approach is that: (a) high level functions are a form of human expert knowledge brought to the system by a designer and, (b) resulting systems are somewhat limited to 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. DNNs have succeeded in this regard in certain scenarios, and hence their importance. GPs that are restricted to the use of basic function nodes (canonical representation) could provide a way to solve both issues, since they do not contain any implicit nor explicit expert knowledge, and have more flexibility in the process of transforming original features; however they pose a new challenge: in high dimensionality problems, candidate solutions can be of potentially unmanageable sizes; the vast number of features that define a 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 the author of this research is aware of, we suspect that GP can also produce universal function approximators. 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 GP's usefulness for feature extraction/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 an approach partially inspired by deep architectures, that learns multiple layers of representations that gradually abstract the initial feature space.

1.7 Contributions

The main contributions of this thesis research are:

1. A new deep learning architecture based on GP abstract syntax trees instead of artificial neurons.
2. A GP approach to unsupervised representation learning through the use of autoencoding structures.
3. A new classification system (taxonomy) to categorize and understand efforts from GP research community to representation learning/feature extraction.

1.7.1 Other Technical Contributions

Besides the scientific contributions product of our research, there is at least one non-scientific, technical contribution we consider significant:

- A new GP software library aimed at ML tasks, where all proposed models and experiments were implemented. The library is written in Python, and it is opensource, in line with the commitments of reproducibility and repeatability of modern research.

1.8 Thesis summary

The remainder of this document is organized as follows:

2. In Ch. 2 we analyze the concept of *deep learning*. We provide an overview on the main methods, algorithms, and architectures used in what are considered modern deep networks. We discuss what makes a network deep, rather than shallow, and what is the essence of deep learning.
3. In Ch. 3 we provide a thorough review on basic and advanced GP concepts. The contributions of our research are built upon all concepts reviewed in this chapter. We also propose a taxonomy to categorize different classes of GP methodologies found in literature.
4. Ch. 4 consists in a literature review where we cover from works related to representation learning with GP, to unconventional deep learning architectures recently proposed by other scientific teams. We also discuss how the idea of *deep learning* has also been somewhat present in the GP research community for a long time.
5. In Ch. 5 we propose an unsupervised learning method to representation learning based on the synthesis of autoencoding algorithms.
6. In Ch. 6 we present an extension to the GP frameworks that allows to evolve *deeper* tree structures than typically possible with the vanilla GP representation. We further extend this model to a new class of deep learning architecture based on GP, where all artificial neurons are replaced by GP's abstract syntax trees.
7. Finally in Ch 7 we provide a recapitulation on our research and discuss some possible future lines of research that emerge as a conclusion of the results found.

Chapter 2. Deep Learning

In this chapter we present the basic theoretical framework of Deep Learning (DL); this chapter is not meant to be a complete or detailed description on the mathematical foundations of DL, but rather a general introductory overview on: (i) the most important concepts of DL, (ii) modern techniques and methods used by DL practitioners, (iii) some unanswered and controversial issues in the field of DL to date, (iv) and how all these methods and concepts relate to the approach we are proposing in this thesis research.

2.1 Representation Learning

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). The key word in this definition is *automatically*. All representation learning methods are either feature extraction or feature selection methods, but not the other way around: not all feature extraction methods are representation learning techniques. It is possible to develop feature extraction, or even feature selection methods, that can leverage from human experts' knowledge of the problem's domain being tackled; in such scenario, automation in the process of generating a new representation is partially lost, and therefore cannot be considered a representation learning. It is important to make this clarification because many GP systems for feature extraction cannot be classified as representation learning methods, and the success of those that can be considered true representation learning is limited to low dimensional problems (for further discussion on this topic see Ch. 4), hence the importance of our research.

Formally, all DL methods are considered, at their core, representation learning methods (Bengio *et al.*, 2013; LeCun *et al.*, 2015); though this can be somewhat debatable. On one hand, it is certain to assert that DL methods are field agnostic: DL methods can process raw ML datasets without needing any human expertise or prior knowledge on the problem's nature. On other hand, unlike principal component analysis or matrix factorization, the aim of DL methods is rarely to simply come up with a new data representation; instead, most DL frameworks consist of complete ML pipelines that take as input samples in raw representation and return as output the classification or prediction, leaving generated representation somewhere in between nested layers of the DL model. Hence, DL can be considered more sort of a change of paradigm, where the typical ML workflow that is composed of a feature engineering stage and a ML prediction stage are replaced,

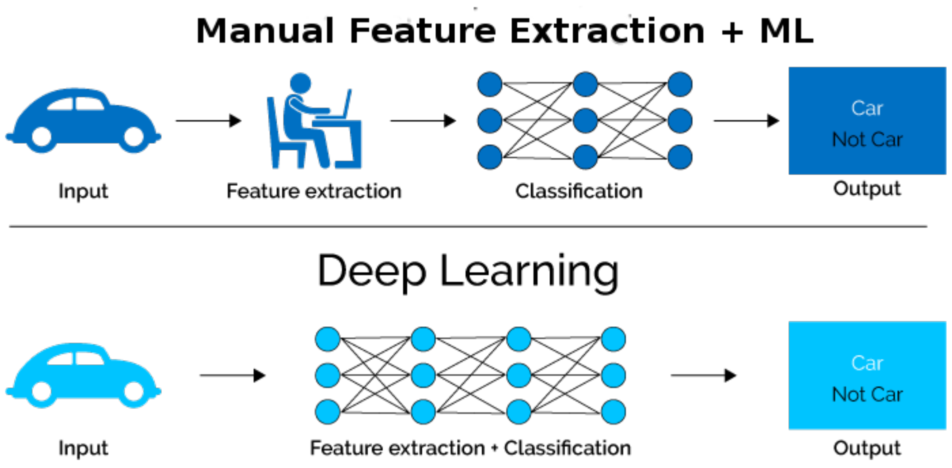


Figure 2.1: Comparison between a classical ML workflow vs Deep Learning approach.

by a fusion system where the whole process is performed by means of artificial intelligence. Fig. 2.1 visually depicts this contrast. We wish to made these remarks because, while in Ch. 5 we do present a GP system specifically aimed at learning new representations, in Ch. 6 we propose a framework based on GP aimed at supervised ML tasks, such as regression or classification, that nevetheless we consider a Deep Learning model, because it process input samples in raw form (even if does not returns as output a new representation per se).

All classical DL methods are based on *artificial neural networks*. In the next section we provide a basic description of these artificial neurons.

2.2 Artificial Neural Networks

ANNs are machine learning models that can be represented by a graph, where nodes represent simple processing units (the artificial neurons) and the edges are weight factors that must be calibrated for the network to correctly perform the desired ML task (Mitchell, 1997). Mathematically, ANNs are linear algebra systems that transform an input through a cascade of simple matrix operations, along with non-linear transformations in-between.

Mathematically, an artificial neuron is the composition of a linear transformation with a non-linear function. Eq. 1 defines the behavior of a single artificial neuron, while Fig. 2.2 shows the typical graphical depiction of an artificial neuron.

$$y = \varphi\left(\sum_{i=0}^{m} w_i x_i + w_b\right) \tag{1}$$

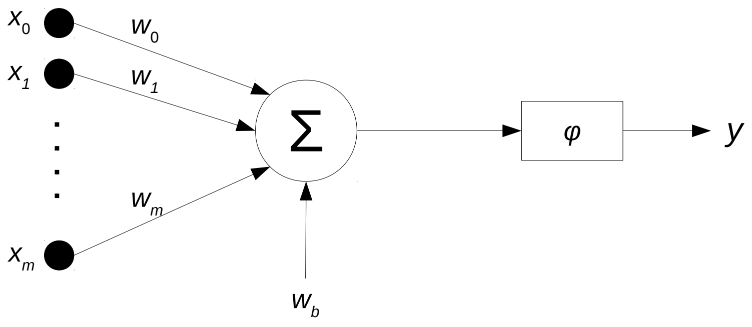


Figure 2.2: An artificial neuron is composed by an input vector, weights that modify its input, a summation of these products and an activation function.

In Eq. 1, y is the neuron output, x is the input vector of size m, w is the weight vector, or parameters that must be tuned, and φ is the non-linear function that transforms summation's output before the final output. φ is a function such as hyperbolic tangent, sigmoid function, unit step function, among others. φ is commonly known as *activation function*.

A single neuron, using a unit step function as activation function, may be already used as a ML model to perform classification. In such case, correct parameters values (weights) may be (efficiently) found by minimizing the error by expressing it as a function of weights and using a gradient optimization method, such as *gradient descent*. This simple model composed of a single neuron is known as the *perceptron* (Rosenblatt, 1957).

However, such an approach would be severely limited because it makes the assumption that the mapping that correctly classifies the input samples is a linear one. So, when dealing with complex ML scenarios, multiple neurons are interconnected in some network topology that allows them to tackle non-linear ML problems. Fig. 2.3 shows an example of one possible topology: the fully connected feed-forward network, also known as the Multi-layer Perceptron (MLP).

When training (finding the weight parameters) multilayer neural networks, an additional algorithm needs to be used in order to express the error (also called cost function) in terms of all the internal weights; this algorithm is known as the *backpropagation* algorithm (Rumelhart *et al.*, 1985; Yann, 1987).

Artificial neurons are the basic building block of all conventional DL models. Even though many DL models exists that are not always referred as neural networks, such as Deep Belief Networks (DBN) (Hinton *et al.*, 2006) or Deep Restricted Boltzmann Machines (RBM) (Smolensky, 1986), all these models do rely on artificial neurons as their backbone. The same applies to ANNs variants specially tailored toward certain domains, such as Convolutional Neural Networks (CNN) (Fukushima, 1988; LeCun *et al.*, 1995) and Recurrent Neural Networks (RNN) (Rumelhart *et al.*, 1985). In the case of CNNs, these ANNs are composed of

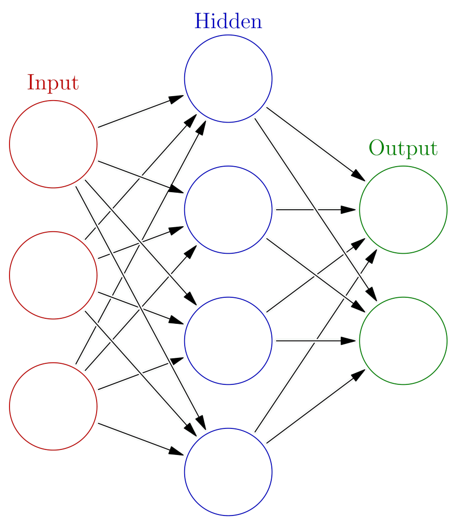


Figure 2.3: Single hidden layer, Multi-Layer Perceptron.

convolutional filters, i.e., artificial neurons that are slid across their input space, while RNNs have neurons with recurrent connections that act as memory units.

On other hand, unconventional DL models that do not use artificial neurons do exist. Those atypical DL models are just beginning to appear in the research literature; we review some of these in Ch. 4. The models that we propose in this research thesis fall under this category of new DL approaches.

At this point, we can evidence some weaknesses and strengths of artificial neurons, and their models. Notice how an artificial neuron is fundamentally a linear transformation coupled with a single non-linear transformation. Even a basic non-linear transformation such as the multiplication of two feature variables cannot be achieved unless at least two sequential layers of neurons are used in network; this is in contrast to the basic building unit of GP, that is an abstract syntax tree, that can represent multiple non-linear operations in cascade. It is one of the reasons we consider important to develop new DL models based on learning units other than artificial neurons.

Artificial neurons and their networks have mathematical properties for training through gradient descent and backpropagation algorithms, which are very efficient algorithms. Attempting to build a DL model without these *goodnesses* is one of the challenges in this research.

One of the most controversial questions in DL is, when an ANN is considered deep? In the next section we review the most distinctive traits that set apart deep neural networks from ANNs.

2.3 Deep Networks

In this section we will discuss defining characteristics of deep learning models, and how they differ or are considered different from standard or older ANNs. We will focus on Deep Neural Networks and Deep Convolutional Neural Networks, though most of what will be exposed here also applies to Recurrent Neural Networks, and to a lesser degree to the rest of the other classical deep learning models referred before (DBNs, RBMs, etc.).

The concept of such thing as ”*deep*” networks probably arose due to the following two factors:

- In the late 1980s and during 1990s researchers from ANNs field began to speculate and gather some empirical evidence that ANN composed of multiple layers could substitute the need for manual feature engineering in ML classification problems, i.e. ANNs composed by several layers of neurons could potentially replace human experts in the problem's domains (Rumelhart *et al.*, 1985; Yann, 1987; LeCun *et al.*, 1995).
- Roughly at the same time, researchers also realized that the backpropagation algorithm -despite its mathematical elegance- struggled to train ANNs composed of more than a single hidden layer (Hochreiter, 1998; Bengio *et al.*, 1994).

These two factors combined lead the search for methods to efficiently train neural networks composed by two or more layers (1 hidden + 1 output) a kind of holy grail status among researchers of the area; while at the same time it suggested that deep networks were a step towards artificial general intelligence.

During the entire decade of the 2000s scientists came up with several advances that eventually led to development and publication of the first deep neural network. AlexNet (Krizhevsky *et al.*, 2012) is a CNN targeted at image recognition tasks. It was published in Neural Information Processing Systems conference in 2012 and it is commonly considered the first deep neural network, or at least, the first *modern* deep neural network. Fig. 2.4 shows the schematics of AlexNet.

AlexNet incorporated some methods that set it apart from earlier attempts at developing deep networks. In the following subsections we will review each of these methods and how they have evolved to date. It should be noted, however, that the reason on why AlexNet was so pivotal is probably due the fact that it surpassed all other ML methods at its time at a difficult image recognition challenge by a significant margin, effectively demonstrating that these so called deep networks could replace the handcrafted features engineered by human experts.

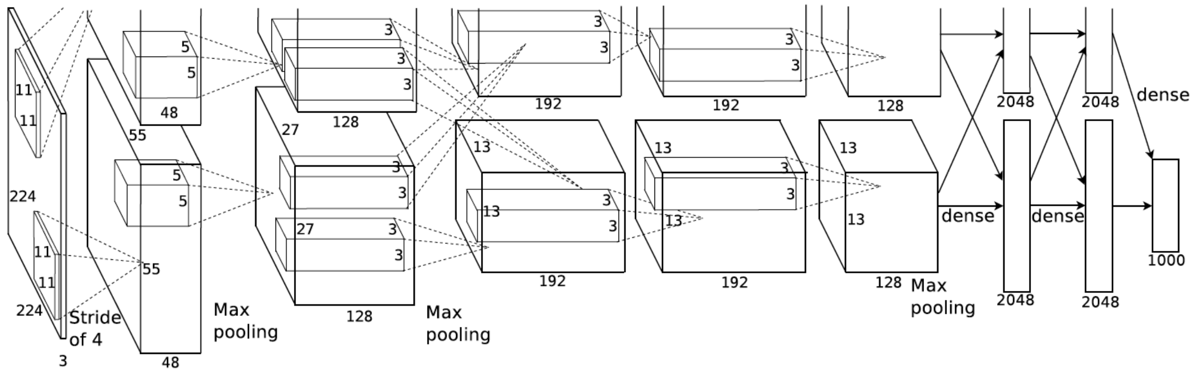


Figure 2.4: Alexnet is a DNN composed of 8 layers in total (Krizhevsky *et al.*, 2012).

2.3.1 Training of Deep and Wide Learning Structures

One of the main characteristics of AlexNet is the activation functions used in the hidden layers of the network. AlexNet uses Rectified Linear Units (ReLU) (Nair & Hinton, 2010), or simply Rectifiers. Rectifiers are the key ingredient that allowed backpropagation algorithm to train more than a single hidden layer efficiently and allowed scientists to train deep networks of up to a couple dozen layers (Simonyan & Zisserman, 2014; Szegedy *et al.*, 2015).

Before the advent of ReLUs, most ANN relied on other activation functions, such as the sigmoid or the $\tanh$. LeNet-5 (LeCun *et al.*, 1998) is an ANN very similar to AlexNet, but one of its main difference is that LeNet-5 uses $\tanh$ as activation functions. ANNs such as LeNet-5 took considerably more time to train and/or required that the initial weights in the network to be close to the correct values, i.e. ANN could not be initialized with completely random weights. Some techniques were developed in order to tune networks' weights to these optimal initial values (Hinton & Salakhutdinov, 2006a), but eventually the usage of ReLUs were fundamental to the birth of modern deep networks.

Nowdays, deep neural networks can consist of several dozens or even hundreds of layers (Huang *et al.*, 2016). These newer deep networks still rely heavily on ReLUs but they also require of other methods in order to train that many layers. Two of the most important techniques in this regard are batch normalization (BN) (Ioffe & Szegedy, 2015) and residual connections (He *et al.*, 2016).

BN allowed to train networks beyond of what ReLUs allowed, giving birth to famous deep networks such as the more advanced versions of the Inception networks (see Sec. 4.1). As the name implies, BN consists in applying a parametrized normalization to the set of training samples in each one of the layers, similar to how it is done to a training or testing dataset at the start of any ML pipeline.

Residual connections on the other hand, consist in each layer sending its output not only to the next immediate layer, but also a few layers ahead. This kind of networks, known as ResNets (Huang *et al.*, 2016), break the traditional paradigm of ANNs' layers being completely sequential.

When rectifier units and residual connections are not used, deep networks may suffer from a phenomenon called exploding/vanishing gradients, which consist in error signal unable to back propagate to, or becomes unstable at, the first layers of the network.

So, while ReLUs, ResNets and BN allow a network to grow deeper, other techniques allow the networks to grow wider, i.e. to have more neurons (convolutional filters) per layer. Another method which AlexNet was pioneer in adopting is *dropout* (Dahl *et al.*, 2013). Dropout consists in setting, with a random probability, each weight in the network equal to zero, i.e. randomly *disconnecting* some inputs for all neurons, in each forward pass of data. This *damage* made on purpose on the networks helps to avoid *overfitting*, which consists of ML methods to memorize training sets and perform poorly on new samples. Large neural networks are particularly prone to overfitting because the number of parameters inside of them may allow them to easily memorize the training dataset, so dropout is a very effective method in deep networks to prevent overfitting.

Newer techniques exists that allow to train even deeper networks, with up to a thousand layers, such as *stochastic depth* (Huang *et al.*, 2016); but these techniques are still more experimental, while the rest of the methods mentioned in this section are nowdays commonly found in DL literature.

2.3.2 Efficient Training with Large and High Dimensional Datasets

Deep learning is, to a large extent, about solving very difficult optimization problems (Kathuria, 2018). The optimization algorithms that make possible to train networks composed of thousands, or even millions, of parameters play a significant role in the field of modern deep learning.

A key trait of modern deep networks is their capability of using *mini-batched* training. In mini-batched, or also called *on-line* training, a ML algorithm is not presented with the entire training dataset each time it has to update the internal parameters of the model being trained; instead, only a small fraction of the training set is used. This approach can accelerate orders of magnitude the training time.

In the case of DNNs, the algorithm *stochastic gradient descent* (SGD) (Bottou, 2010) is the on-line version of a standard gradient descent algorithm. Newer algorithms that belong to the family of gradient following optimizators have been developed recently and specifically for training deep networks, such as

RMSProp (Tieleman & Hinton, 2012) and Adam (Kingma & Ba, 2014). These algorithms also train neural networks using *mini-batches* and can be considered enhanced versions of SGD.

Even though optimizing a DNN is known to be a non-convex problem, the fundamental reasons on why gradient methods converge to very good solutions, or even global minima, remain unknown. Some hypotheses in this regard have been formulated (Haeffele & Vidal, 2017; Frankle & Carbin, 2019; Zhou *et al.*, 2019), but it is still an open line of research.

2.3.3 Accelerated Computing on Graphics Processing Units

It turns out that ANNs are particularly prone to be accelerated orders of magnitude on Graphics Processing Units (GPU), modules originally designed for video games playback (Oh & Jung, 2004). This is thanks to the fact that the fundamental operations used by neural networks (matrix operations) are the same used in 3D rendering.

All current implementations of deep networks are run in GPUs. For example, the authors of AlexNet made publicly available a highly optimized software library for implementing CNNs in GPUs; one remarkable feature of this library, is that allowed to split the implementation of networks across multiple GPUs, thus taking the scalability of parallel execution one step further.

Some researchers (Hernández *et al.*, 2017; Such *et al.*, 2017) have hinted that most of DL success is due to this new available raw computing power, rather than to truly new theories or methods. The reasoning behind this asseveration is true to a certain extent: deep networks are overparametrized (Frankle & Carbin, 2019), and in order to be trained, large training datasets are required, and large datasets require high computational resources if one is to obtain acceptable results in reasonable amounts of time. So, it is true to assert that deep learning may have not been possible without this unprecedented, easy-accessible, processing power; nevertheless, we should not overlook the technical (even if heuristic in nature at times) developments discussed in previous subsections.

2.4 Discussion

We close this chapter with a wrap-up on the DL concepts presented so far, how these relate to the proposed approach, and finally, with our own definition on deep learning.

In the previous sections we discussed some techniques that are commonly used to differentiate ANNs from modern deep networks, which can be summarized as follows:

1. Deep Networks rely on techniques specifically designed to facilitate the training of networks composed of more than a single hidden layer (such as ReLU activations, BD, and residual connections).
2. Deep Networks rely on techniques that allow them to efficiently handle large training datasets (such as SGD, its variants, as well as GPU acceleration).

However, some reseachers have argued that the merits of deep networks such as AlexNet have been overstated, and that DNNs have long existed (Schmidhuber, 2015). According to Schmidhuber (2015), the first networks that can be considered deep, were succesfully trained in the the Soviet Union during the 1970s, by Alexey Ivakhnenko, through a method called *Group method of data handling* (Ivakhnenko, 1968, 1971).

In the framework for this thesis, we share this point of view, and even though (as stated earlier) we do not overlook the great technical advances aimed at enhancing the performance of deep networks presented through Sec. 2.3, we do consider that the only real requirement for considering a method *deep* is that is composed of more than a single layer of non-linear processing.

Now, we can formulate our own definition of deep learning: *deep learning is a set of ML methods that: (a) do not require previous knowledge of the problem's domain being tackled, (b) perform the feature extraction and prediction stages in a single, unified, learning structure/pipeline, and (c) work by performing multiple non-linear transformations in cascade.*

Notice how the above definition does not restrict DL methods to belong exclusively to the area of ANNs. This is important because recent attempts (see Sec. 4.4) at developing DL-alike ML that do not rely on artificial neurons can now be considered DL on their own right, as well as the methods proposed in this thesis research (presented in Ch. 5 and Ch. 6).

Chapter 3. 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). In this chapter we present a thorough conceptual framework on GP.

In the first section of this chapter we describe the basic components of GP. In the second section we introduce a taxonomy for the types of primitives found in the GP works, that in Ch. 4 will allow us to classify different current GP methodologies; this classification system is an original contribution of this research thesis, and will also help us in clearly differentiate our main method, presented in Chapter 6, from the rest of the works found in the area, and why its relevance. The final section of this chapter discusses advanced concepts of GP that will be required to understand, or can be contrasted with, the main proposed method of Chapter 6.

3.1 Basics

In this section we present the basic elements that compose the GP framework. Some of them were briefly introduced in Ch. 1; here are described in greater detail.

3.1.1 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 an *abstract syntax tree* that represents a mathematical function or simple computer programs. In a GP tree, every node represents itself a function. The *primitives set* represent the available functions that nodes can represent, hence functions represented by nodes are known as *primitives*. All node functions must be taken from the primitives set. Internal nodes represent typical functions, in the sense that these are functions which take as input some arguments, apply some transformation over them, and return an output for further processing. Values returned by the children nodes 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*, or *terminals*. 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 and binary or one-class classification; however, they fall short in capabilities for other tasks. Some GP representations make use of forests, a set of trees, to represent a single candidate solution, that is, multiple trees account for a single individual. Forest naturally extend the capabilities of GP to tackle problems as diverse as vector function regression, prototype generation, representation learning, multiclass classification, among others. Nevertheless, it is still important to understand the tree structure as the backbone of individual representation in GP.

3.1.2 Genetic Operations

GP operators are functions that transform the population, and individuals within, in order to find better performing candidate solutions for the problem being tackled. Similarly to standard GAs, the typical operators in GP are *mutation*, *crossover*, and *selection*. 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.

3.1.2.1 Selection

The process of selecting individuals that will pass to the next generation in an EA is sometimes considered also an operator, since these selection mechanisms take as input a number of individuals and return as output a new set of individuals, very much like mutation and crossover operators. However, unlike mutation or crossover that only modify the population implicitly, selection is an operator aimed directly at modifying the population as a unit, by pruning it from poor performing individuals, and shifting it towards promising areas in the search space.

In GP the selection mechanism are the same of those found in GA, such as *roulette-wheel selection* (Baker, 1987), *tournament selection* (Blickle & Thiele, 1996), *truncation selection* (Mühlenbein & Schlierkamp-Voosen, 1993), and *elitist selection* (Thierens & Goldberg, 1994), to name a few. Some of these methods are non-deterministic (roulette-wheel, tournament), while others are deterministic (truncation, elitism).

Selection in GP can happen at two moments during each evolutionary cycle: (1) when selecting the individuals from which offspring will be generated, i.e. individual that will serve as *parents*, and (2) when selecting the individuals, both from offspring and original population, that will survive to the next generation.

In the first selection process (assembling the parents pool), usually mechanisms such as tournament or roulette-wheel selection are used, whereas in the second selection process (assembling the next generation) elitism or no selection mechanism is used at all, depending on the population dynamic used. The available population dynamics will be described in detail in the following subsection.

3.1.3 Population Dynamics

In GP, the dynamics that govern how populations are modified in each evolutionary cycle can be of two main types: *generational* or *steady state*.

A *steady state* population dynamics consists in replacing part of the current population with newly generated offspring. Conventionally, an elitist selection procedure, where the best individuals from both the original population and the offspring generated, are selected in a non-deterministic way to form the next generation. Individuals from the original population and the offspring pool may or may not compete for selection, i.e. even though an elitist selection procedure is used, one may choose to select half of the next generation from the original population and the other half from the offspring pool, or in some other proportion. This can be done this way in order to introduce *diversity* in the population, i.e. that early high performance individuals do not conquer the entire population too quickly.

On the other hand, in *generational* population dynamics, the entire pool of individuals that comprise the current population is replaced by offspring (Eiben *et al.*, 2003). The only evolutionary pressure happens when selecting the pool of highly fitted individuals that undergo reproduction. Elitism can still be used in generational replacement, but only to keep the single best performing individual found so far in each iteration.

3.1.4 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 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

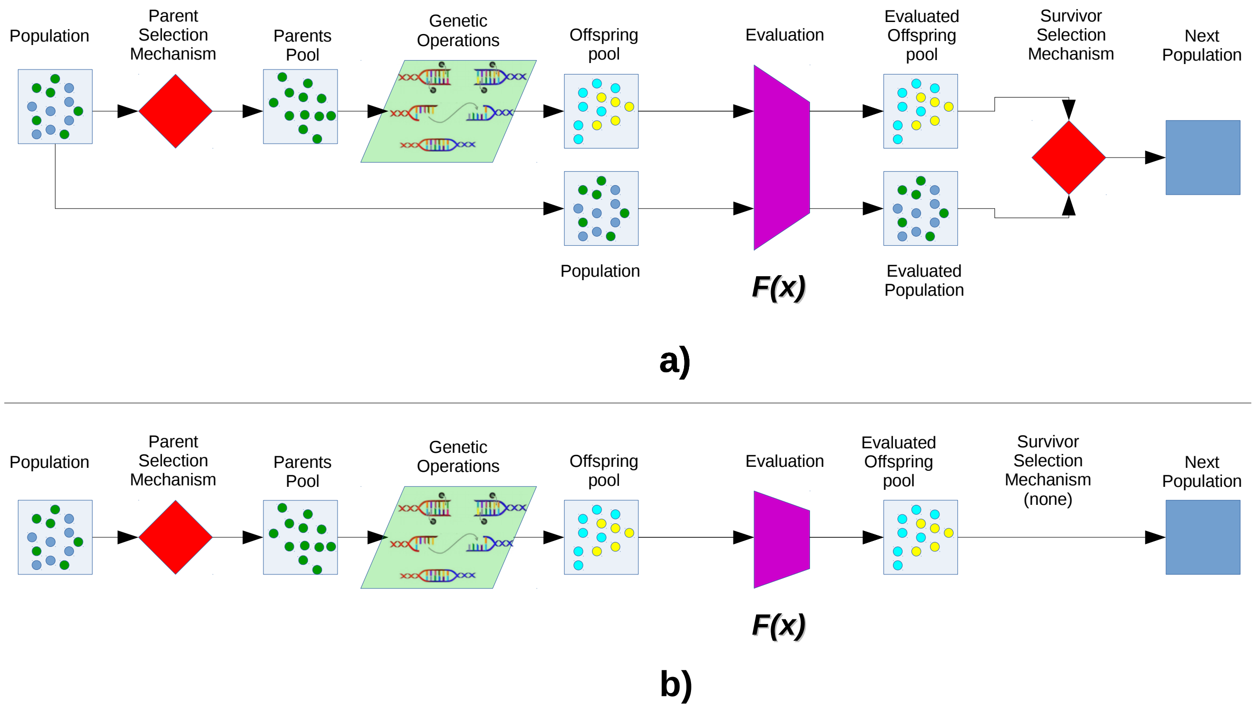


Figure 3.1: Flow diagrams for different evolutionary population dynamics: a) steady state; b) generational replacement. They follow the next steps: 1. Initial random population, 2. Parent selection mechanism (such as binary tournament), 3. Candidate parents pool, 4. Genetic operators are applied, 5. Offspring's pool breeding, 6. Individuals evaluation, 7. Evaluated individuals pools gathering, 8. Survivors selection mechanism (such as elitism or tournament), 9. Next generation assembly.

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).

3.2 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, etc. 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. This taxonomy is an original contribution of this thesis research.

This taxonomy will help us to clearly differentiate our contributions from the rest of the works found in the area, as well as draw some analogies between ANNs/DNNs and GP in the next chapter.

There are two main classes of primitives: input based functions (internal nodes), and zero-argument functions (leaf nodes), also called terminals. Input based functions are further divided into three groups,

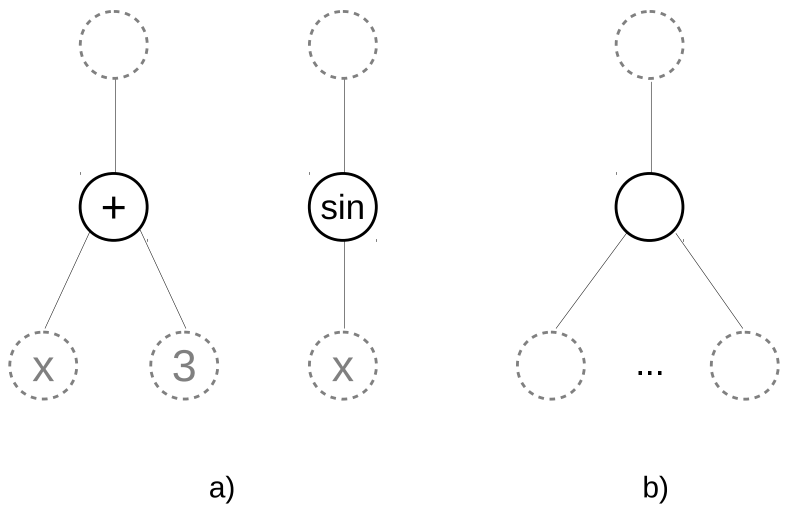


Figure 3.2: a. Two examples of low level functions: arithmetic addition and sine function; b. A generic depiction of low level functions. Low level function may have any fixed number of inputs, but all should be scalars, as well as their output.

depending on the dimension of the inputs and the output of the function, and zero-argument functions into four categories, also depending on their dimension but also on their type (variable or constant).

3.2.1 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. 3.2 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.

3.2.2 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 algebraic operations, such as the *dot product*. These functions serve the purpose of connecting high level functions and complex forms of data, to lower level functions; but at the

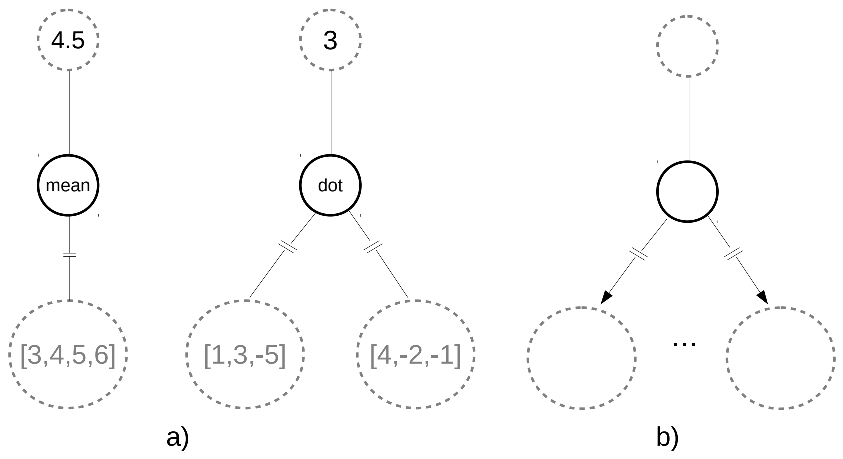


Figure 3.3: a. Two examples of mezzanine level functions: the mean over the elements of a vector, and the dot product of two vectors; b. A generic depiction of mezzanine level functions: mezzanine level function may have any fixed number of inputs, and they may be of any dimension > 0, but their output 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.

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. Fig. 3.3 shows some examples of mezzanine level functions.

3.2.3 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 a convolution of a kernel filter over an image, such as an edge detector, or matrix operations (addition, subtraction, etc.). High level operations, along some type of mezzanine functions, 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. Fig. 3.4 shows some examples of high level functions.

Notice how in a GP tree there can be only a single 'layer' of mezzanine functions, i.e. two mezzanine functions cannot be stacked, while low and high level functions can account for most of the tree depth. Mezzanine functions work mostly as *connectors* or brief feature extraction stages between high dimensional data and low level functions, while at the same time, high-level functions must necessarily connect to low-level ones through mezzanine functions (hence the name *mezzanine*).

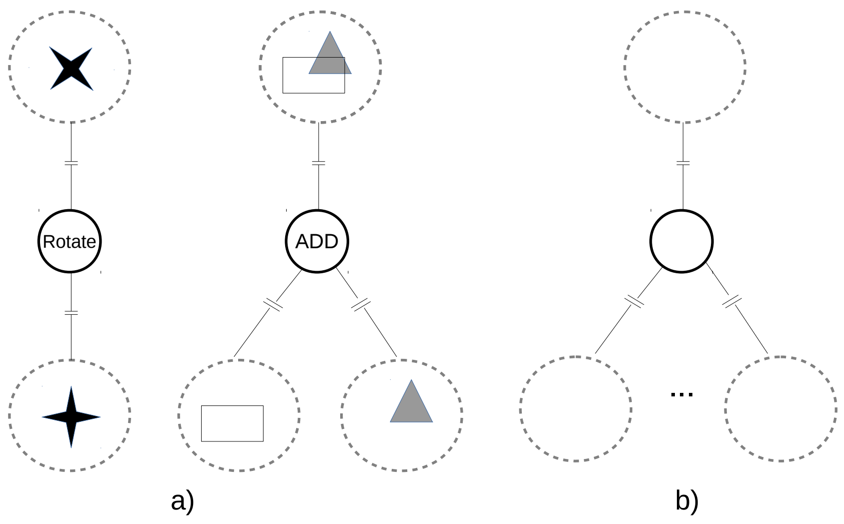


Figure 3.4: a. Two examples of high level functions: the rotation and addition of images; b. A generic depiction of high level functions: high level function may have any fixed number of inputs, and they may be of any dimension > 0, and they yield their output in an vector, array or tensor form of data. In the figure, the dashed arrows that serve as edges to the tree nodes represent the inverse flow of multidimensional data.

3.2.4 Zero-argument Functions

Just as regular functions are classified according to the 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 -or partial- 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_4) 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.

3.3 Advanced Techniques in GP

In the final section of this chapter we cover some GP techniques that are uncommonly used, and that are aimed at enhancing the performance of GP. We review these techniques because the proposed methods in chapters 5 and 6 make use of them, or propose new methods derived from them or that can be contrasted with them.

3.3.1 Diversity Measures

In general terms, *diversity* is understood as how many of the individuals from a population are actually different from each other, i.e. how many unique candidate solutions are in the population. This means that in a GP run, some individuals become repeated in the population as the time progresses. This is not only common, but actually it is the standard behavior, not only for GP, but for all EAs in general.

Over course of a run, the population in a GP (or any EA), will eventually be dominated by a single individual repeated multiple times, taking most part of the population; in other words, *diversity will deplete* or *extinguish*. This happens as a consequence of the very nature of EAs, where only fittest individuals survive each evolutionary cycle.

However, diminishing diversity is an undesired phenomenon, because diversity is the actual key of success for an EA: if all individuals are the same in the population, the effect of crossover operator losses strength, and the evolutionary search may converge to something like multiple hill climbers, pushed forward only by the mutation operator.

In GP in particular, diversity can be measured in two broad ways: at genotypic or phenotypic levels. Genotypic diversity refers to the differences in the individuals' representations, i.e. the how the abstract syntax trees that represent each individual in the population differ from one another. The similarity between two trees can be measured in several ways, such as by the tree edit distance, which measures how many changes have to be done in a tree in order to transform it into the other one.

There are two main problems with measuring diversity at genotypic level: the first one is that it is a computationally expensive way to calculate diversity, and the second problem is that, even if two individuals are different at a structural level, it does not mean they perform the task differently, since one or both of the trees may be composed of non-functional subtrees (known as *introns*), i.e. parts that do not modify the behavior of the tree, such as subtrees that are reduced to the multiplication or addition of the neutro; this would mean that two trees that are considered different at genotypic level in fact perform the same function.

Phenotypic diversity, on the other hand, is measured by observing how many different results are obtained in a population at the task performed, i.e. by comparing their predictions or their fitnesses. Unlike genotypic diversity, phenotypic diversity can be calculated fast, and there is no risk two individuals are considered different, even if they are not.

In the next section we will review one of the main methods used to promote diversity in EA.

3.3.2 Spatially Distributed EAs

One effective technique that allows to sustain diversity for an extended number of generations, is to split the population into multiple subpopulations. There are two main approaches to do so: the island model (Cohoon *et al.*, 1987) and the cellular model (Manderick & Spiessens, 1989; Gorges-Schleuter, 1989). Under this framework, the standard approach with a single population is called panmictic model.

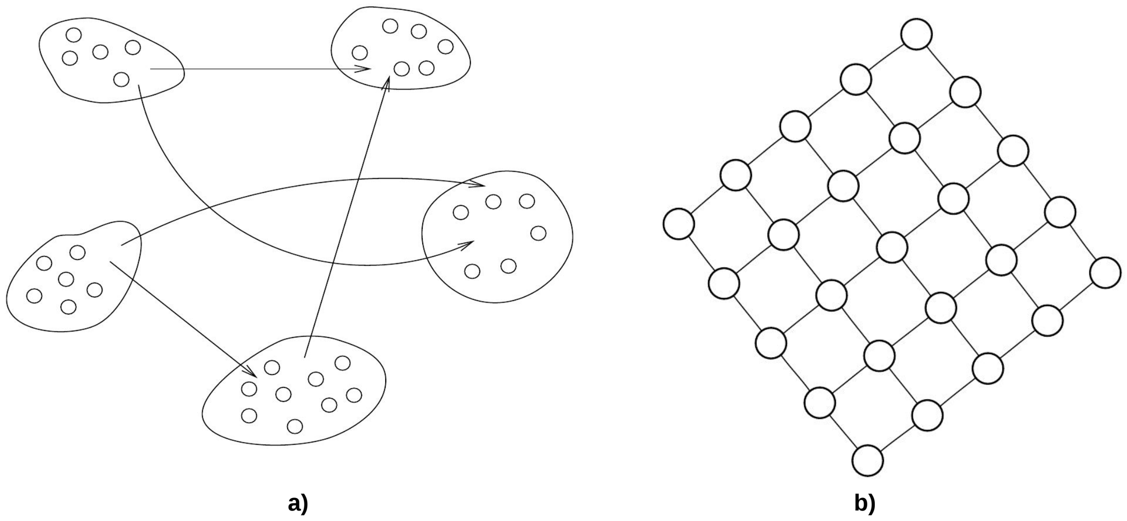


Figure 3.5: Two standard models of spatially distributed EAs: (a) island model; (b) cellular model. (Tomassini, 2006)

In the island model, also called multipopulation model, the population is split into a few smaller populations. Each of these populations evolve in isolation and behave as a panmictic population would do, only exchanging a small percentage of individuals between them every certain number of generations. The links of exchange between these islands can be static or dynamic, if dynamic they may be random, and if static they may adhere to some topology, such as a ring or lattice topology. In Fig. 3.5a shows the concept of a multipopulation scheme composed of 5 islands, with a random dynamic topology.

The cellular models takes the island model to a extreme, where individuals reside in isolated cells, and a spatial configuration is defined for the population, such that each individual can only interact with neighboring cells. In cellular models, the process of replacing the individual in each cell in every generation, consists of assembling a micropopulation composed of the individuals in the surrounding cells, then the processes of parent selection, offspring generation and survivor selection take place in order to find the new individual to reside in the cell. Fig. 3.5b shows a lattice arranged cellular model.

Spatially distributed approaches help to maintain and promote diversity in EAs, because they do not allow top performing individuals to take over the population as fast as in the panmictic model.

3.3.3 Subroutine Finding in GP

Subroutine finding, evolution, synthesis or also called discovery, is the concept of enhancing the GP framework to allow it to discover on its own, re-utilizable pieces of code (expressed in the form of abstract syntax trees), such that the GP individuals become modular (Poli *et al.*, 2008). Just as human programmers rely heavily on subroutines in order to write complex code, the search efficiency of a GP may benefit if such GP could define and re-use pieces of code.

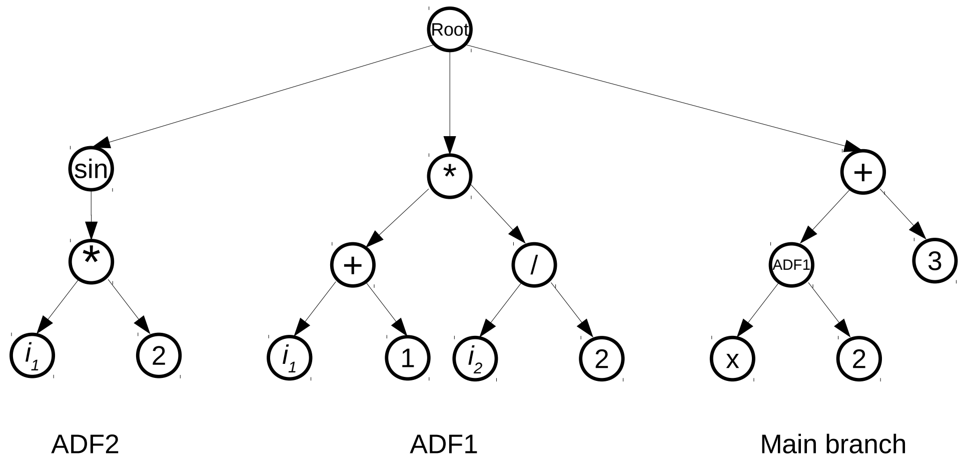


Figure 3.6: Depiction of an GP individual that includes Automatically Defined Functions. In this particular case, the individual includes two ADFs.

Koza's Automatically Defined Functions (ADF) (Koza, 1994) probably remain as the most well-known framework for subroutine synthesize (Poli *et al.*, 2008). The ADF framework consists in including additional trees in the individual representation, so the individual becomes effectively a forest, where the first tree represents the "main branch", i.e. the code or equation that is executed when the individual is used for evaluation or application purposes, and each other tree represents a subroutine that the main branch may call upon. The subroutines may appear in the main branch as nodes, just as if they were primitives; and, depending on how the ADFs are configured, it may also be possible for an ADF to invoke another ADF. Fig. 3.6 shows a GP individual with two subroutines embedded.

Koza proposed an extreme use case of ADFs where ADFs replace all primitives. In this scenario performing crossover can become problematic. Koza proposed a series of complex rules that should be observed in order to use crossover under this scenario. In Ch. 6 we propose a similar modified GP framework that resembles this extreme case, but we propose to use a parametrized representation of GP trees that allows to improve the functions represented by ADF trees through a special mutation operation, instead of relying on crossover and the rules proposed by Koza.

3.3.4 Memetic GP Models

GP has the ability to find symbolic expressions that, depending on the set of primitives available, do not have to constrain to a predefined form (e.g., a polinomial of some degree). However, that does not mean that GP can efficiently search across such large candidate solution space.

For example, let us suppose that a GP is used in a regression problem where the searched function complies to the form of a third degree polinomial (unbeknownst to the designer). A GP may find an expression as the one depicted in Fig. 3.7a, which represents the function $x^3 - 0.3x^2 - 0.4x - 0.6$. This function may yield an acceptable error in the hypothetic regression scenario, nevertheless it is possible that by finding the correct coefficients of the expression, the optimal solution could be found. Notice however that in a standard GP setup, where genetic operations such as subtree crossover or subtree mutation are applied, a lot of computational effort will be wasted by altering the structure of this near-optimal individual in ways that do not push it closer to the optimal solution (i.e., by disrupting the thrid degree polinomial form), before the optimal solution can be found.

Early GP researchers noticed this issue and proposed to enhance the standard heuristic search of GP with some forms of local search (i.e. local optimization methods). At least as early as (Evett & Fernandez, 1998), researchers began to propose methods that focused on altering the numeric constants in leaf nodes of GP trees with the intention of performing a fine tuning of candidate solutions. The idea is to let the GP search act as a global search that sorts through the general *structural* search space, whereas these complementary methods carry a local search. Such methods that combine both types of global and local search, belong to the family of *memetic algorithms* (Moscato *et al.*, 1989).

For example, Evett & Fernandez (1998) proposed to use a mechanim similar to simulated annealing to fine tune the numeric constants in a GP tree, without disrupting the general structure. Experimental evaluation of the proposed method showed promising results when compared against a standard GP.

Notice how once there is a local optimization mechanism available to be combined with the GP search,

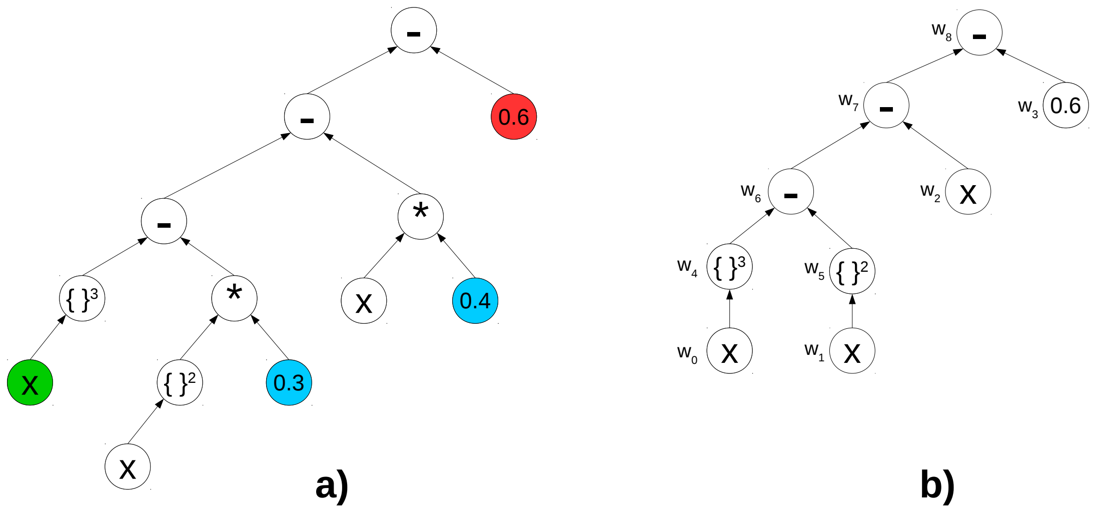


Figure 3.7: Examples of (a) standard GP individual vs (b) memetic enhanced GP individual representations. Notice how in (a) blue nodes represent coeffiecients of the polinomial expression, and the red node is the bias factor; both being subject to be efficiently optimized through numeric optimization methods; unfortunately the coefficient is absent for the green leaf node, in order to optimize the entire expression. In contrast, model depicted in (b) considers all the necessary coefficients for the same class of polinomial of (a), while also being more compact, and potentially more easy to find by a GP search.

the GP individual representation can be modified in order to further take advantage of the local optimizer. Fig. 3.7b depicts an example of such representation. Individual shown in Fig. 3.7b represents the function $w_8(w_7(w_6(w_4(w_0x)^3 - w_5(w_1x)^2) - w_2x) - w_3 0.6)$. Notice how this expression is a generalization of the model of Fig. 3.7a, while also being more compact and pontentially easier to find by the GP search, while the search for the optimal coefficients w_i is left entirely to the local search method.

Memetic GP variants can carry some problems that require designers to take important decisions, most prominently if coefficient values are subject to be inherited by offspring generated through standard GP operations (*Lamarckian* evolution) or if offspring generated will reset coefficient values (*darwinian* evolution). For an in-depth analysis of this and other issues associated to local search-enhanced GPs, as well as experimental evaluations, please refer to Emigdio *et al.* (2014).

Chapter 4. Related Work

In this chapter, we review recent efforts related to the aim of our research thesis: developing a DL framework based on GP or, in more general terms, developing DL frameworks not based on ANN.

First, we perform a quick review and analysis on how DNNs architectures have progressed during the last years. Then, we move into the field of GP by reviewing some efforts related to learning representations with GP, that are related to our research, but were never intended to be the basis of a GP-DL framework. Next, we draw the resemblance that exists between DL architectures and modern GP approaches, and list some works that have leveraged from such similarity. Next, we review the most recent efforts at developing DL frameworks based on learning units other than artificial neurons. We close this chapter with a summary on how the proposed research differs from all works presented through these sections.

4.1 Recent Deep Networks Architectures

In this section, we present a brief review on some of the most popular deep networks from the recent years. Our focus is on how an increase in depth in deep networks has translated, for the most part, in increases in accuracy in ML tasks, at least in what concerns to CNNs and image processing tasks (e.g. classification, denoising, segmentation, etc.), respectively. For a more detailed analysis and discussion on the overall field of DL, refer to Ch. 2.

Fig. 4.1 compares the performance of the winning contestants in the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) from 2010 to 2015. ImageNet (Deng *et al.*, 2009) is a dataset of more than 14 million images, hand labeled and hierarchically organized. A subset of around 1.2 millions images from ImageNet have been used to perform a ML image recognition challenge: images have to be classified among 1000 possible categories, with around 1000 images per class for training purposes. ILSVRC has been pivotal for the birth, development and recognition of the deep learning field; first, because ILSVRC large size has allowed to train deep CNNs that otherwise would have suffered overfitting on smaller datasets, and secondly, because it is a very difficult ML task, where DNNs can shine over many other ML methods.

In 2012, AlexNet, a CNN won the ILSVRC contest by a significant margin, even outclassing methods that relied on handcrafted feature extraction engines designed by experts on the field of image process-

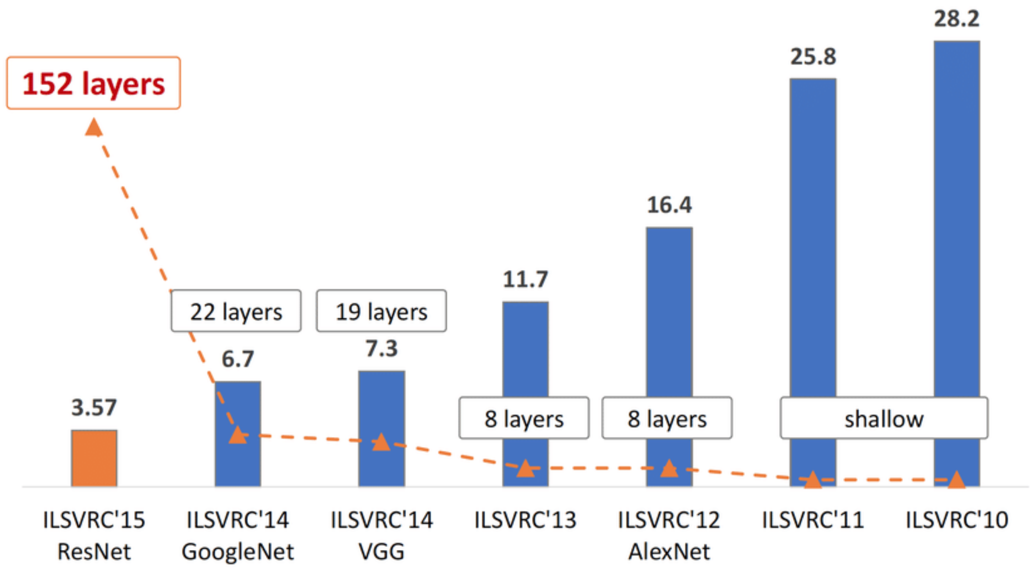


Figure 4.1: Winners of the ImageNet image recognition challenge from 2010 to 2015 (He *et al.*, 2016; Nguyen *et al.*, 2017). Plot depicts classification error, lower is better. Notice plot is time backwards.

ing (Russakovsky *et al.*, 2015), effectively demonstrating that deep networks could replace the need for manually engineered feature transformation stages in ML pipelines.

Notice how Fig. 4.1 also depicts both, a trend in depth increase in CNNs, and a correlation between this increase in depth and the performance obtained by newer CNNs architectures. After the release of AlexNet, newer and deeper CNNs have taken the place as winners of the ILSVRC contests. ILSVRC is fundamentally a natural image classification problem, and winners of each year contest become popular CNNs because they end up serving as templates for other application- or area- specific image processing tasks. Some of these popular networks are VGG (Simonyan & Zisserman, 2014), GoogleNet (Szegedy *et al.*, 2015) and ResNet (He *et al.*, 2016); all these networks being considerable deeper than AlexNet (the number of layers of each one is shown in Fig. 4.1).

Naturally, this correlation translated into a search for increasingly deep ANNs architectures. GoogleNet, also known as Inception network, and ResNet, implement special techniques and methodologies aimed specifically at allowing the successful training of such deep networks (discussed in Sec. 2.3); authors of both architectures acknowledge that the main motivation behind their technical contributions was to develop networks deeper than what was the state-of-the-art at their time. Newer architectures still follow this trend (Huang *et al.*, 2016), though research has also shifted toward understanding why networks need to be so large (Frankle & Carbin, 2019).

At this point is also important to remark how all these networks are aimed at a supervised learning task.

Even though DL is defined as a representation learning method (Bengio *et al.*, 2013; LeCun *et al.*, 2015), notice how all these DL approaches use a supervised learning task as a mean to compare their performance.

The idea of utilizing ANNs for the mere purpose of learning a new representation is probably more related with the past fact that deep networks were difficult to train, and gradually training a deep network by means of unsupervised learning proved to be the way to achieve such feat (Hinton & Salakhutdinov, 2006b; Schmidhuber, 2015). Hinton & Salakhutdinov (2006b) proposed to train deep networks (i.e. ANNs with more than a single hidden layer) by gradually training *autoencoder* structures. In Ch. 5 we follow a similar approach by training a GP-based autoencoder.

However, and as recent progress in the field of DL shows, once researchers managed to train deep networks even when tackling classification or regression tasks, the representation learning process in itself passed to be a by-product, or an automatic hidden/intermediate step, of training these large networks. In Ch. 6 we present a DL architecture based on GP aimed at a regression task, which we could argue it is also an implicit representation learning method.

4.2 Representation Learning with GP

In this section, we present relevant efforts to the task of representation learning with GP. The key characteristic of the works presented in this section is that the only concern was to generate a new representation through GP, while the classification or prediction stage was relegated to another ML algorithm. In the area of feature learning, this is what is known as *filter* or *wrapper* methods to feature selection and extraction (Tran *et al.*, 2016; Gao *et al.*, 2017; Doucette *et al.*, 2012).

The main difference between filter and wrapper methods, is that filter methods perform the feature transformation in an unsupervised learning fashion, i.e. they use some measure or rule to select or transform features without concern if this does enhance the performance of some classification or regression algorithm. In contrast, wrapper methods perform the feature extraction with the aid of some classification algorithm, the score obtained by such algorithm using the generated representation is used as guide to further enhance the feature transformation process; i.e. they work in a supervised fashion and through an iterative cycle, where an initial new representation is generated, then sent to the classification algorithm, and the score is returned to the extraction method as feedback signal, and the process repeats until some stop criterion is met.

Most representation learning approaches performed through GP are probably wrapper methods. Bot (2001) is recognized as one of the first authors to propose the use of GP to generate a representation to enhance the accuracy of a Nearest Neighbour classification algorithm; his approach was limited in the sense that only a single feature was generated each iterative wrapper cycle; Limon et al. further enhanced Bot's approach by enabling GP to generate multidimensional representations each cycle (Limón *et al.*, 2014; Limon *et al.*, 2014; Limón *et al.*, 2015). Zhang & Rockett even experimented with a wide range of classification algorithms fed with GP regenerated representations (Zhang & Rockett, 2009, 2011). The previous examples were limited to low dimensionality problems, but Shao *et al.* (2013) applied a wrapper approach to image classification by linking a GP feature extraction stage with a SVM classifier; later, they applied their approach to video classification too (Liu *et al.*, 2015).

Filter approach to feature extraction with GP is somewhat more complicated, because a non-obvious objective function that guides the search for a new representation, without actual feedback from a classification algorithm, is required. Trujillo & Olague (2006) developed an *interest point detector* that transforms images into a vector representation more prone for classification and detection tasks, in order to do so they used a special objective function aimed at evaluating these kind of representations, without actually using a classification algorithm. Other approaches to learn represententations through GP under the filter approach are: using Fisher criterion (Guo *et al.*, 2008), using clustering measures (Lensen *et al.*, 2017), and very recently, through *manifold learning* (Lensen *et al.*, 2019).

It should be clarified that some of the works just mentioned are not true representation learning methods in the strictest sense, as discussed in Sec. 2.1. Particularly, methods such as (Trujillo & Olague, 2006; Shao *et al.*, 2013; Liu *et al.*, 2015) rely on high or mezzanine level primitives (see Secs. 3.2.2 and 3.2.3) that contain human experts knownledge of the problems' domains. This clarification is important, because it is precisely those works the ones that tackle high dimensionality problems, hence one of the motivations of our research, to develop a deep learning framework (i.e., a representation learning method, not just a feature extraction one) based on GP; in other words, when GP is faced with high dimensionality problems, true representation learning is difficult.

There is a third approach other than filter or wrapper methods, that is *embedded* methods (Doucette *et al.*, 2012; Tran *et al.*, 2016).. In embedded methods, both the feature extractor and the classifier algorithm are trained (evolved, synthetized, etc.) simultaneously. In this regard, the entire deep learning paradigm can be considered merely a embedded method. GP flexible solution representation (syntax trees representing highly non-linear functions) makes GP prone to embody an embedded method all by itself, just like DNNs.

One noteworthy GP method for embedded approach consists in sequentially evolving GP classifiers, such that each new classifier uses the output of the previously evolved one as a new feature that is added to the representation vector (Lin *et al.*, 2007, 2008). This layered structure might be seen as a primitive form of deep learning. The disadvantage of such method is that only a single new feature is generated at each ”layer”. Although some other clever mechanisms to embedded approach have been attempted through GP, such as defining the root node of a GP tree as one of different possible, very simple, classification algorithms (Sherrah *et al.*, 1997), or using co-evolutionary algorithms to build ensembles of GP trees (Doucette *et al.*, 2012), our main concern is with embedded approaches where some of the deeper nodes on a GP individual act as a feature extraction stage, while the upper nodes supposedly act as the classification algorithm. These approaches will be discussed in detail in the following section.

4.3 Deep Learning from a GP perspective

The idea of executing the complete ML workflow (feature extraction + prediction) with a single algorithm, just as DL does, has emerged in a natural way in the GP community in the past. The abstract syntax tree structure of GP has enticed researchers to find ways to allocate some layers of the GP trees for feature extraction while using the rest of the tree to perform the classification or regression per se. There is a vast corpus of academic GP literature where this approach has been attempted, and in this section we will review representative works of such approach.

These works can be considered as *the classical approach to perform deep learning alike processing through GP*, and although they can be considered as a direct precedent to our approach, the main method we propose, in Chapter 6, diverges significantly from all the works mentioned here.

4.3.1 Pseudo Deep Learning with GP

The idea of using bottom level nodes of GP trees as feature extraction stages probably arose in works from the area of financial time series prediction. In the seminal works of Neely *et al.* (1997) and Allen & Karjalainen (1999), a GP methodology is used such that the leaf nodes of GP individuals consist in time series filters that were known to work as financial technical indicators. This approach departs from the traditional GP individual representation where, leaf nodes would typically be individual time ticks (being a time series problem). This methodology was further explored and expanded in the works of Potvin *et al.* (2004); Lohpetch & Corne (2009); Myszkowski & Bicz (2010); Esfahanipour & Mousavi (2011). The modifications proposed consisted in decomposing the specialized leaf nodes into small subtrees of couple levels of depth, such that subtree mutation or crossover operations could make modifications to these feature extraction stages (Potvin *et al.*, 2004; Esfahanipour & Mousavi, 2011), or defining these leaf nodes as parametric functions, along defining new type of mutation operation that modify their parameters (Myszkowski & Bicz, 2010). In this way, a complete ML pipeline can be trained by means of only GP, very similar to the aim of DL.

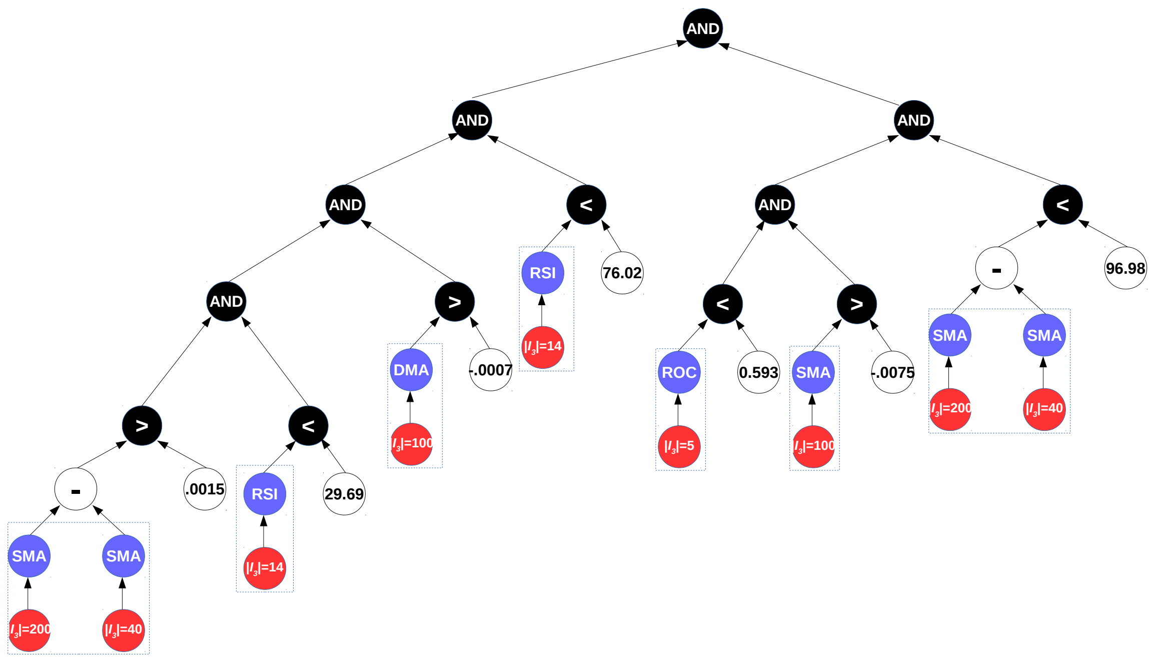


Figure 4.2: This tree was generated by the GP methodology proposed by Myszkowski & Bicz (2010). Blue nodes are time series filters that can operate over different window size inputs (red nodes), i.e. mezzanine functions.

In Fig. 4.2, a GP tree obtained by these type of methodologies is shown. In this tree, the nodes enclosed in boxes compose the feature extraction stage of the tree, while the rest of the nodes can be considered as the classification stage, in a striking similar fashion as how in DL networks convolutions are typically considered feature extraction stages, while fully connected layers are the classifier.

It should be noted, however, that none of these methodologies can be considered deep learning, for the main reason that the feature extraction nodes contribute expert knowledge of the problem's domain (known financial indicators); while a fundamental requirement of any DL approach is to be domain agnostic.

4.3.2 Quasi Deep Learning with GP

The approaches described in the previous subsection eventually resulted in similar attempts at image processing tasks, with the added benefit that these new methods relied on more agnostic feature extraction stages. Al-Sahaf *et al.* (2012a,b) proposed a GP methodology to image classification relying on mezzanine type of nodes. Fig. 4.3 shows the resulting individual generated by their approach; in this type of individual, most of the parent nodes of leaf nodes are mezzanine functions, while the leaf nodes work as the parameters or input data to these kind of mezzanine functions. In this particular case, the mezzanine functions are rather generic statistical measurements, such as the mean or standard deviation over a region, thus this kind of approach can be considered a *quasi* deep learning, except for the fact that, by the very nature of mezzanine functions, the feature extraction stage is limited to be *shallow*, rather than *deep*, as in deep networks.

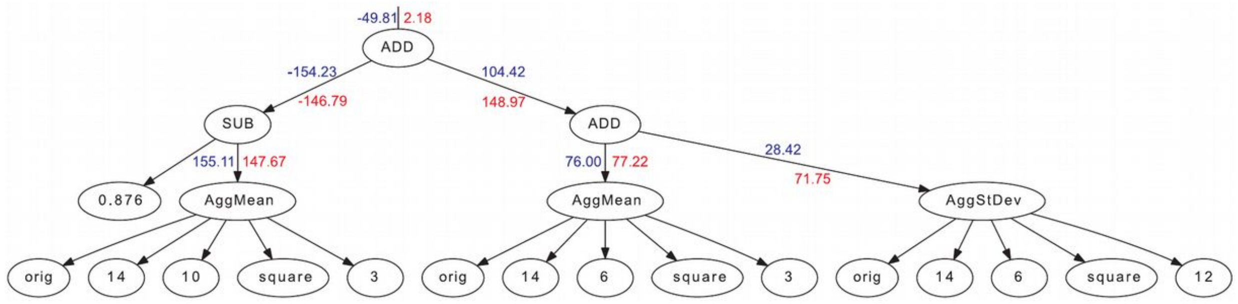


Figure 4.3: This tree was generated by the ”Two-Tier GP” algorithm (Al-Sahaf *et al.*, 2012a).

Previous attempts at increasing the size of the feature extraction stage in these kind of approaches failed (Atkins *et al.*, 2011; Al-Sahaf *et al.*, 2012b), until recently (Evans *et al.*, 2018). Atkins *et al.* (2011) developed a GP method for image classification based on both mezzanine and high level primitives, that allowed to perform a greater amount of high dimensionality processing than if using only mezzanine functions. However, this method was proved to be inferior to that of Al-Sahaf *et al.* (2012b), showing that extending the ’pre-processing’ feature extraction stages of GP trees was not trivial.

4.4 Alternative Deep Learning Architectures

In this section we review relevant works centered around the idea of performing deep learning with frameworks other than artificial neural networks. As discussed in a previous chapter, deep learning can be viewed as a change of paradigm, where the manual, or automatic, feature engineering and extraction stages along the prediction algorithms of a ML pipeline are unified in a single, layered, learning structure, capable of performing multiple non-linear transformations in cascade that both, finds the representations required to correctly make predictions, as well as learning to perform the prediction itself.

Under the above definition, there is no restriction that obliges deep learning to be limited to ANN-alike structures; and the next works reflect the desire of the research community to find novel ways to perform deep learning.

4.4.1 Morphological Neural Networks

Since the inception of the Perceptron by Rosenblatt (1957) in the late 1950s, little has changed regarding the fundamental structure of the artificial neuron other than newer activation functions (Hernández *et al.*, 2017; Schmidhuber, 2015); it is perhaps due to this stability of the basic building block of ANNs and DNNs that the methods to train these networks have been allowed to progress significantly.

Nevetheless, some researchers do have proposed new variants and improvements to the basic artificial neuron model. One of such efforts is known as *morphological neurons* (MN) (Davidson & Hummer, 1993), and their more recent version, *dendrite morphological neurons* (DMN) (Ritter & Urcid, 2003).

Mathematically speaking, MNs and DMNs incorporate *lattice* algebra operators such as $\inf$ and $\sup$ to the function that performs the neuron. MNs and DMNs can form networks, just as regular artificial neurons. In a dendrite morphological neural network (DMNN), the function of neuron j is given by Eq. 2, where $\mathbf{x}$ is the input vector received by neuron j, $\mathbf{w}$ is the vector of synaptic strenghts between neuron j and the ith neuron that connects to neuron j, and a and b are excitation parameters that can take the value of $+1$ or -1, that denote the excitation or inhibition of neuron j and i, respectively. Contrast this with Eq. 1 given for standard artificial neurons.

$$\begin{gathered} \tau_j(\mathbf{x}) = a_j \vee_{i=0}^{n} b_{ij}(x_i + w_{ij}) \\ \text{or} \\ \tau_j(\mathbf{x}) = a_j \wedge_{i=0}^{n} b_{ij}(x_i + w_{ij}) \end{gathered} \tag{2}$$

On a conceptual level, MN and DMN work by projecting hypercubes or even hyper ellipsoids (Arce *et al.*, 2017) in the feature space, that attempt to cluster and separate classes; in comparison, standard artificial neurons are only capable of projecting hyperplanes. DMN are difficult to train, and EC is a useful tool to train these kind of networks (Arce *et al.*, 2018), but other methods exist (Sossa & Guevara, 2014), and even gradient techniques have been now applied to certain extent (Zamora & Sossa, 2017).

In a set of limited experiments (Hernández *et al.*, 2017), it has been empirically proved, that DMNNs can achieve higher, or at least equal, accuracy on classification tasks than that of DNNs of greater total depth (and hence, higher parameter count). This results have opened the door to the possibility of newer DL architectures where some of the artificial neurons are replaced by DMNs. However, it is still not clear yet if these MN can indeed replace standard neurons in what is considered the feature extraction stage of a deep network, or only in the late classification stage; in the first case, it would mean a new generation of DL architectures, while in the second case, it would be more akin to some efforts where the typical MLP at the end of a DL network is replaced by some other classification algorithm, such as a Support Vector Machine.

4.4.2 Deep Forests

Deep Forests (Zhou & Feng, 2017, 2018) are the first explicit attempt at developing a new kind of deep learning architecture. Deep Forests consist in a deep learning framework based on decision trees / random forests, rather than ANNs.

In the context of ML, Decision Trees (DT) is a non-parametric, supervised learning, method that can be used for classification or regression tasks (Pedregosa *et al.*, 2011). DTs work by building a model in the form of a tree where nodes represent decision processes based on the values of the features for some particular instance; DTs' trees are completely unrelated to GP syntax trees. In a DTs trees, prediction is performed from top to bottom, and leaf nodes nodes are the output of the model, where a class or value is returned.

Random Forests (RF) are an extension to the DTs method in the form of an *ensemble*. Ensemble learning is an ML technique that consists in aggregating the prediction of several models, rather than using only one, by some mechanic like averaging or voting. The combined prediction should be stronger than that of any of the single models used, if certain guidelines are followed when generating the individual models. Basically, in a RF, each DT is generated with some degree of variability. This variability can come in the shape of generating DTs with different training samples subsets, and/or different subsets of features.

RF are the conventional approach to use DTs; however Deep Forests take the DTs framework one step further. In Deep Forest, several layers composed of sets of RF are trained sequentially. Architecturally, the concept is very similar to those of deep networks: sequential layers of processing extract features and perform a prediction in an unified pipeline, and even a convolution-alike operator has been employed in Deep Forest in order to be used in image processing tasks; however, there are several key important differences. Unlike artificial neurons, DTs represent non-differentiable, non-linear, non-parametric functions; these traits make RF unable to take advantages of training methods such as gradient descent and backpropagation. This translates in the fact that Deep Forest have to be trained in a completely different fashion to that of deep networks. Fig. 4.4 depicts the general architecture of Deep Forests.

Deep Forests are trained sequentially, layer by layer, unlike most deep networks approaches. In each layer of Deep Forests, there are two sets of RFs: forests of one set are all generated by attempting to perform the classification or regression task desired, while the RFs of the other set are generated by using as objective a clustering function (this is done this way on order to promote feature extraction). Through a special process, features are extracted from the RFs generated in both sets and concatenated with features from all previous layers, and this vector is the new representation generated by this layer that will serve as input for the next layer of RFs.

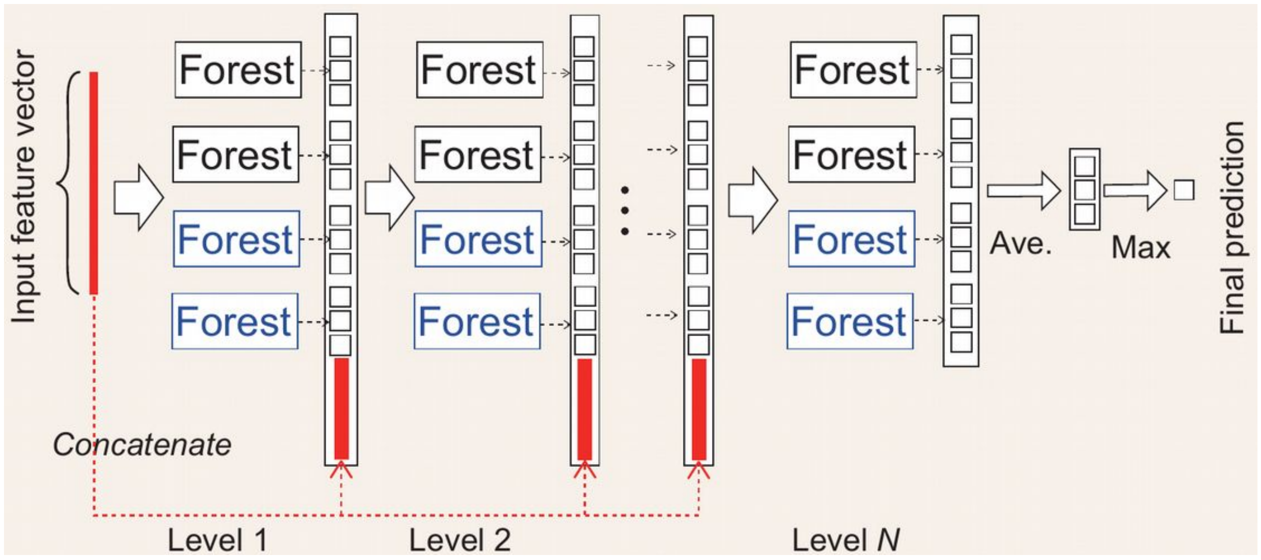


Figure 4.4: Deep Forest architecture. In each layer, there are several RF that generate an intermediate representation. Notice there are two different classes of RF per layer (blacks and blues) (Zhou & Feng, 2018).

Zhou & Feng (2018) tested this deep learning paradigm in a variety of large and low dimensionality ML datasets, from all sort of different ML problems (images, audio, text, etc.) and obtained competitive results when compared with conventional deep networks, effectively giving birth to a new generation of deep learning algorithms not based on ANN. Our thesis work, belongs to this new class of algorithms.

4.4.3 Deep GP

Parallel to the development of our research, another ”Deep GP” paradigm emerged. Evans *et al.* (2018) developed a GP approach to image classification derived from the class of approaches described in Sec. 4.3, but where the feature extraction stage is composed of more than a single layer. Fig. 4.5 shows an example GP tree evolved through this approach.

Instead of relying on image pixel-wise arithmetic operations as in the work of Atkins *et al.* (2011), this new GP approach takes inspiration directly from deep convolutional neural networks, such that the *high level* primitives that compose the generated trees are the convolution and pooling operations. Convolution takes as input two arguments, the first one is the input image, or the output from another convolution or pooling node, and the second one is a 3×3 constant mask. The high level stage of these GP trees (called ”Convolutional layer”) connects to layer composed of *mezzanine* nodes (called ”Aggregation layer”) which finally connects to the top section of the tree, composed of only *low level* nodes, and is considered the classification layer, similar to how a fully connected MLP is considered the predictor component in CNNs.

It is important to remark that one of the main contributions of our research (see Ch. 6) bears little resem-

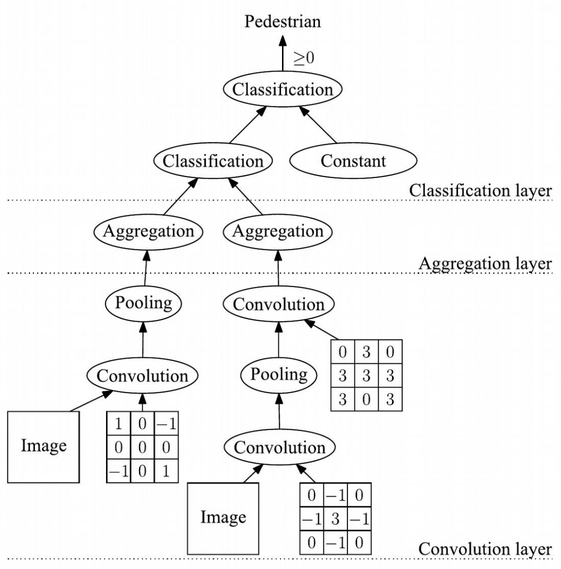


Figure 4.5: A Deep GP architecture as proposed in (Evans *et al.*, 2018). The 'feature extraction' stage has deepen in comparison with earlier works, and now is reminiscent of convolutional deep networks.

blance to the method just described. Our proposed method attempts to find new functions that replace the artificial neurons inside the layers of a deep learning model, whereas the method proposed by Evans *et al.* (2018) works more akin to an architectural search in the feature extraction stage of the DL pipeline, taking for granted the use of convolutions, and only replacing the MLP for a GP tree. In this sense, our method is more related to the Deep Forest method described in Sec. 4.4.2.

4.5 Discussion

In this section, we present a comparison between the proposed approach of this research, and the works described so far in this chapter. The objective is to contrast the novelty of our research against to what has been developed to date.

- The proposed method draws inspiration from the standard, artificial-neuron-based, deep architectures discussed in Ch. 2 and Sec. 4.1; specifically, we take the multi-layer processing architecture of such models, and attempt to export it to the GP framework. However, that is where similarities end, since the model we propose does not rely on the artificial neuron, backpropagation algorithm, or gradient optimization methods, which are the cornerstones of the classical DL field.

- Through Sec. 4.3 we performed a historical review on how GP's scientific community has attempted in the past to develop a DL-alike paradigm shift, such that GP candidate solutions contain both, feature extraction and prediction stages. The main drawback of the first efforts in this regard, is that the feature extraction stages in these GP trees were usually special nodes embedded with some form of human expert knowledge. The work described in Sec. 4.4.3 is one of the most recent developments in this trend, and it is noteworthy because it is the first work that successfully managed to take advantage of more than one feature extraction level in a GP tree. We wish to remark the contrast between their approach and ours: while they include the convolution operation and convolutional masks as available primitives for a GP, we propose to use GP trees as replacements for artificial neurons in a DNN or CNN architecture, i.e. both approaches can be considered sort of the *opposite* to each other.

- In Sec. 4.4.2 we presented a recently proposed, exotic DL model, where rather than layers of artificial neurons, the presented algorithm makes use of layers of RFs; this is perhaps the most related work to the approach proposed in this research thesis, with the difference that we propose the use of GP abstract syntax trees, instead of RFs or DTs.

We can summarize the novelty of the proposed approach as an attempt to import some elements that have proven successful to certain heuristic to tackle high dimensional learning problems (the sequential layered structure in ANNs, a convolution-alike operation), to another framework that also has traditionally struggled in high dimensional scenarios (GP). The expectation, i.e. our hypothesis, is that incorporating these elements into GP, enhance it beyond of what current approaches proposed in the literature can achieve.

Chapter 5. Evolving Autoencoding Structures through GP

Historically, unsupervised learning approaches were employed to train DNNs as a mean to avoid some of the common pitfalls in training DNNs for supervised learning tasks (Schmidhuber, 2015). These methods were employed before the advent of the techniques that defined modern deep networks, as discussed in Sec. 2.3.

In this chapter, we will focus in a particular class of algorithms used for unsupervised learning, known as *autoencoders*. Autoencoders are specifically used to learn new representations and were typically implemented using only ANNs; here we present the first efforts on developing autoencoding structures through GP. The main motiviation behind this approach is to explore some of the methods that were used to train deep networks, and how we can import them to another paradigm such as is GP.

Verbatim sections, images and tables from this chapter were previously published in the following articles: Rodriguez-Coayahuitl *et al.* (2018, 2019b), and have been reproduced here with permission of the editorial.

5.1 Background

Autoencoders (AEs) are models that aim to learn to map inputs to an intermediate representation and also to reconstruct the original input from this intermediate code. Usually, the intermediate representation is more compact than the original one. AEs have been traditionally implemented with artificial neural networks (ANN). Autoencoders are implemented through multilayer ANN with a small, bottleneck, neuron layer at the center of their architecture. Data is compressed (decompressed) as traverses half the network, from the input (central) layer to the central (output) layer. This ANN is fed with training samples, and then trained to mimic its input to its output, in the process forcing the input samples to be converted to a more compact representation at its bottleneck layer.

Neural networks based AEs were proposed as early as the mid eighties (Goodfellow *et al.*, 2016), see, e.g., Yann (1987); Ballard (1987); Gallinari *et al.* (1987); Bourlard & Kamp (1988). Initially, they were methods for dimensionality reduction and feature extraction (Goodfellow *et al.*, 2016), as well as used for unsupervised learning tasks (Schmidhuber, 2015); however, many other applications have been found for

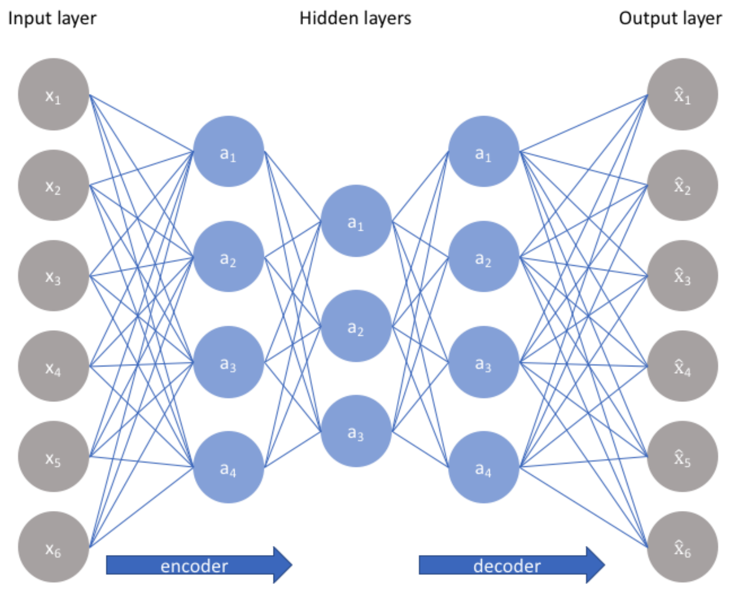


Figure 5.1: Diagram of an autoencoder (Jordan, 2018). AEs are composed of two parts: the encoder and the decoder; amid there is usually a bottleneck layer where a new compact representation is generated.

autoencoders, such as: image denoising (Zhang *et al.*, 2017; Mao *et al.*, 2016) image deblocking (Zhang *et al.*, 2017), classification (Betechuoh *et al.*, 2006), super-resolution (Mao *et al.*, 2016), and image compression (Theis *et al.*, 2017).

Before the advent of the modern field of Deep Learning, Hinton & Salakhutdinov (2006b) proposed that AEs could be used as method to train DNNs, in order to build the feature transformation stages that precede other machine learning tasks. Since training networks composed of more than a single hidden layer had proved difficult at the time, they proposed to take advantage of the autoencoders capability of being trained sequentially, i.e., instead of training an entire ANN through backpropagation/SGD, a deep network could be trained layer-wise, step by step, by connecting first the input layer to the output layer of an AE, then adding the second layer and layer before the output (while keeping the weights of the previously trained pair of layers), and so on. In this way, the fundamental problem of DL could be circumvent, and from the resulting autoencoder, the encoding stage could be used as feature extraction engine for other ML tasks. This primitive idea on how to train deep networks is what served as inspiration for a prototype DL framework based on GP. Fig. 5.1 show a standard, ANN-based, autoencoder

In this chapter, we propose a framework to synthesize AEs through GP as a method for representation learning. In the past, works that proposed generating autoencoding structures through evolutionary computation (EC) focused on discovering and/or optimizing ANN-based AEs architectures with techniques such as NEAT or HyperNEAT, i.e. still conventional AEs. An example of this kind of works can be found in (Fernando *et al.*, 2016). In contrast, AEs presented here are fully generated by standard tree-based GP individuals. One of the main ideas behind this concept is to investigate the possibility of discovering AE algorithms that do not rely on ANN structures. That is, a scheme of deep learning based on evolutionary algorithms.

One important element of the GP based AEs proposed in this chapter is their capability of being evolved in an on-line way (Bottou, 1998). This is achieved by partitioning a training dataset into many small mini-batches, in a reminiscent way of, and directly inspired by, the SGD family of algorithms. In fact, as GP began to be used on increasingly more complex problems with larger training datasets, researchers had to devise clever mechanisms to reduce the number of evaluations required for a GP run, such as using co-evolutionary algorithms that dynamically co-evolved subsets of training samples along the main model solutions (Pagie & Hogeweg, 1997; Dolin *et al.*, 2002; Schmidt & Lipson, 2008; Doucette *et al.*, 2012). Here in contrast we propose to use a rather simple method for evolving GP individuals using only a small fraction of randomly selected training cases each generation, in a similar fashion to both random subsampling (Gathercole & Ross, 1994) and SGD. The experimental results presented in this chapter, that reaffirms GP ability of evolving solutions in an on-line fashion, is one of the key contributions of our research.

In the following sections we propose and describe a methodology for the evolution of autoencoder algorithms through GP, and perform a quantitative comparison with both, results from ANNs models that perform similarly with respect to older classical networks, as well as to more modern architectures. Results show that the proposed GP methodology performs comparabily to deep networks proposed prior to the introduction of AlexNet, i.e. not *modern*, or state of the art, DNNs. Nevertheless, we wish to bring forth the fact that GP's candidate solutions are constructed from a more general set of elements in comparison with the ANNs framework. In general terms, ANNs' ML paradigm consists in searching for the right *numerical* values of a large weights vector (considering all layers in a network); this ML model is only feasible if certain other elements (such as the activation functions) of an ANN architecture are pre-established and known to work. In contrast, in GP, the architecture of the solution or the form of the function (p.e. a polynomial of some degree), are not known or predefined in advance. The ability to achieve the same results that previously required more elements to be predefined, is one of the most important achievements of our research.

5.2 GP Autoencoder

In this section, we describe both the individual representation for GP-based AEs along a partitioning scheme that allows it to be used on high dimensionality problems, as well as an on-line learning approach for GPs efficient evolution when using large training datasets.

In our proposal, a GP based AE is represented by two forests, $\mathbf{E}$ and $\mathbf{D}$, connected through an m dimensional data bus, that receives an n dimensional input vector and outputs a vector with the same dimensionality as the input. The data bus connecting both forests is also a vector. Forest $\mathbf{E}$ ($\mathbf{D}$), from now on the *encoder* (*decoder*), is a list of m (n) standard GP syntax trees.

$E_i \in \mathbf{E}$ is the i-th encoder's tree. Leaf nodes of encoder tree E_i, can be constant values within some range, or can be variables taken directly from the whole autoencoder's input vector, i.e. individual features from input representation. E_i generates as output the i-th element in the data bus vector connecting encoder and decoder.

$D_i \in \mathbf{D}$ is the i-th decoder's tree. D_i leafs, can also be constants or be variables taken from, and only from, the data bus connecting encoder and decoder (i.e., decoder trees cannot *see* any feature variables from the original input representation). A D_i tree generates as output the i-th element in the autoencoder's output vector.

In general, when an autoencoder is used for dimensionality reduction and/or representation learning purposes, it holds that $m < n$, this means that the bus connecting encoder and decoder is an encoded representation of autoencoder's input samples. Fig. 5.2 illustrates a GP based AE individual; arrows from each tree in the figure indicate that its output generates corresponding feature variable in the subsequent vector.

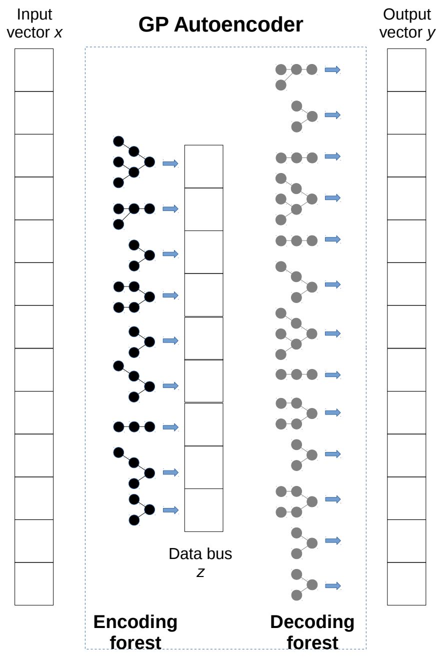


Figure 5.2: GP based Autoencoder's individual representation. Each individual consist of two forests connected through a bus. Variable leaf nodes in encoder trees can only be features from the input vector x, whereas variable leaf nodes in decoder trees can only be features from encoded representation stored in z.

In the following subsection we describe a strategy that allows us to use the proposed GP-AEs in certain types of high dimensionality problems.

5.2.1 Structurally Layered Genetic Programming

In order to use the proposed GP based AE in high dimensional problems (which pretty much represent target problems of representation learning and dimensionality reduction algorithms), we propose to partition the problem into very small subproblems, that GP can deal with, such that each subset is processed by an independent GP, generating their corresponding partial intermediate (compact) representation and output reconstruction. Fig. 5.3 illustrates this kind of partitioning when applied to GP based AEs. Notice that this approach can only be applied when the objective function can be decomposed too[1]. We call this technique, *Structurally Layered Genetic Programming* (SLGP) when applied to AEs which will be described in detail in this section.

Let $\mathbf{x}$ be the vector that represents the autoencoder input; let $\mathbf{y}$ represent the autoencoder output vector; and let $\mathbf{z}$ be the vector that represents the data bus connecting encoder and decoder, from now on called compact representation. Both, $\mathbf{x}$ and $\mathbf{y}$ are composed of n feature variables such that $\mathbf{x} = (x_1, x_2, ...x_n)$, $\mathbf{y} = (y_1, y_2, ...y_n)$, where $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$, while $\mathbf{z}$ is composed of m feature variables such that $\mathbf{z} = (z_1, z_2, ...z_m)$ and $\mathbf{z} \in \mathbb{R}^m$. Input vector $\mathbf{x}$ is partitioned into $c = \frac{n}{\beta}$ indexed partitions C_i, where β is the size of the partition's sets and $\beta > 1$, such that $|C_i| \ll n$ and $|C_i| = |C_j|, \forall i, j$[2]. Both output and compact representation vectors are also partitioned into c indexed subsets, K_i and Q_i, respectively. Subsets C_i, K_i and Q_i are associated in such a way that the compact representation (output) feature $z_j \in Q_i$ ($y_j \in K_i$) is generated by a GP tree which leaf nodes can only be feature variables x_j (z_j) such that $x_j \in C_i$ ($z_j \in Q_i$), or constants within some range. Each set of three associated subsets, C_i, K_i and Q_i, are distributed to independent GPs, and the final solution is assembled by joining the best individuals that emerge from each process. Notice that the above definitions do not contemplate overlappings between subsets C_i; however, there is no real reason why the regions defined by subsets C_i could not share some neighboring elements, (conceivably enhancing the performance of the method) even though in such case, they could not be considered a partition in the mathematical sense.

Notice that the purpose of the SLGP is threefold: (1) it radically reduces the solutions search space by limiting the number of possible variable leaf nodes in every tree, in both encoder and decoder, and distributing different feature variables that might be used as leaf nodes strategically across different forests'

[1]There are tasks such as classification or scalar regression, where the output cannot be decomposed in multiple parts, in such way that each part of the output is solved using a subspace of the entire input, as proposed by our approach

[2]Considering β a factor of n

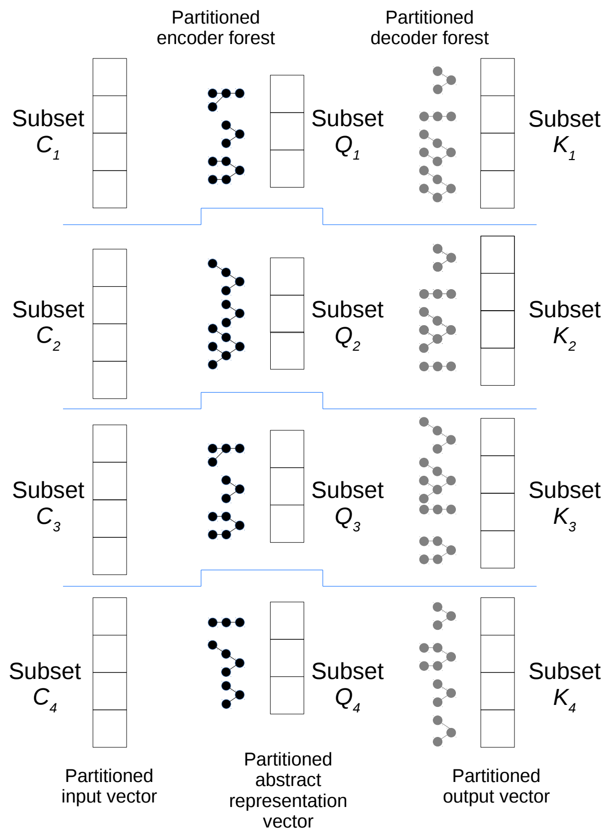


Figure 5.3: Structurally Layered GP based AE that performs a 4-to-3 dimensionality reduction. Input and output vectors, encoder and decoder forests and internal compact representation vector are all partitioned and corresponding partitions are coupled together and processed independently from other partitions sets.

sections; (2) since each set of subsets is processed by a GP, it increases the genetic operators strength. For example, instead of applying a dual single-tree mutation (see Sec. 5.3.1) once per individual for autoencoding a whole image, mutation is applied according to the number of partitions, or in other words, instead of applying a single forest crossover or mutation within a large individual composed of multiple trees, multiple crossovers/mutations happen simultaneously across the equivalent structure that represents the complete solution; (3) and finally, and related to point one, it directs genetic operators effect, such as mutation or crossover, by limiting the range where exchanged genetic material may be imported into an individual. For example, notice that for a GP based AE without any input partitioning scheme, a single-tree crossover might exchange trees between points too distant within the AEs' structure, possibly decoding a region from unrelated encoded features; whereas in a Structurally Layered GP all genetic operators are confined to contiguous regions[3] where it makes more sense to import and export genetic material.

It is important to remark that even though the proposed SLGP has many advantages over a conventional

[3]This holds only if partitions are built by grouping contiguous features, given problems that allow to do so, such as in image, audio, text and time series problems. In unstructured datasets, a manual partionting scheme relying on domain knowledge could be required.

scheme that operates over the entire input space, it also has one important disadvantage: by generating each partial solution from a sample of the input representation, instead of the whole set of feature variables, important information that could enhance the desired outcome, could be lost. This could happen if there are relationships between features that end up in different partitons, relationships that could otherwise be useful if, for example, two input feature are highly correlated, one could be discarded further improving the encoding process.

It is important to remark that Structurally Layered GP can only be performed when the final solution can be assembled by concatenating solutions of small subproblems derived from a whole problem. In the case of an AE used for image encoding and decoding, this can be done if the objective function used to compare the similarity between original and reconstructed samples and to guide the evolutionary search, is a linear function, such as the *mean squared error* (MSE). This is due to the additive property of linear functions, that specifies that a linear function f preserves the addition operation such that $f(x + y) = f(x) + f(y)$; notice how this property allows a problem to be *evaluated* in parts, i.e. that the total error is equivalent either if we evaluate the whole assembled output, or if we average the error obtained by the all the elements of the partition. Notice the emphasis in the aspect that linearity allows the problem to be evaluated, which does not imply that can be also *solved* in parts. This is related to what is expressed in the previous paragraph, in the sense that there is information loss with the SLGP, and that the resulting outcome is only an approximation to the optimal result that could be obtained shall all the feature variables could be used to generate each component of the output. As stated before, there are problems, such as classification or scalar regression, where there is no obvious way to split the output in order to solve it by parts; there has been, however, some proposed approaches that share the same principle of tackling a large scale problem by splitting the input represenation, while requiring a special treatment regarding the assembly of the output; some of those methods are ensemble learning (Geurts *et al.*, 2006) and cooperative co-evolutionary algorithms (Potter & De Jong, 1994).

5.2.2 On-line Learning

Another important aspect of the proposed GP based AE is its capability to be evolved in an incremental way, i.e., instead of using an entire training dataset for every generation, it can be split in many small batches and each batch is used to test individuals in every generation. This allows a more efficient use of a training dataset, as well as provide a form of regularization. In this section, we describe this on-line form of evolution, and in Sec. 5.3.2 we discuss some of the consequences regarding the computational cost for different population dynamics such as *steady state* or *generational replacement*.

The on-line learning approach to evolution we propose is inspired by the stochastic gradient descend (SGD) technique used to train ANN, as a method to accelerate evolution in GP when dealing with very large datasets (hundreds or thousands of training samples). The method consists in splitting the original training dataset into many, much more smaller (in number of samples), *minibatches*. Each generation, individuals are evaluated using only a single minibatch, instead of the entire training dataset. A different minibatch is used every generation. Each sample of the training dataset belongs to only one minibatch, and the union of all minibatches must equal the entire training dataset. Hence, the number of minibatches derived from the training dataset is the number of generations required so that the GP can *see* all training samples available; we call this an *epoch*, similar to SGD/ANN. A GP is not necessarily limited to last only one epoch, as we will see in Sec. 5.4, giving more than one epoch is beneficial for achieving better solutions.

Notice that this method involves a fundamental change to the conventional objective function used in GP. Traditionally, the objective function in GP consists of an average of fitness cases across an entire training set (Poli *et al.*, 2008). For example, if we were to reduce the difference between AEs' input images and its reconstructions, we could use the MSE as a method to compare originals *vs* reconstructions, and average the MSE an individual obtains across all training samples. Reducing such average would be the objective function. The on-line learning proposed method requires however, a *moving target* as an objective function, because, an individual's performance is evaluated using only the current minibatch, and in the next generation, the evaluation of the objective function will change, given that reducing the average over originals and reconstructions differences of a *distinct* minibatch will now be used as objective function.

This method of evolution makes a more efficient use of the training dataset, because it allows more generations to elapse given the same number of training samples as well as taking less time evaluating individuals per generation. However, there is a special computational toll when using this approach to evolution when combined with an *steady state* style of evolutionary population dynamics. This issue will be further discussed in the Sec. 5.3.2.

5.3 GP Configuration

In this section, we discuss two aspects of synthesizing GP based AEs directly related to evolutionary aspects of GP, namely, GP operators and population dynamics. First, we propose three different types of genetic operations for GP AEs' individuals. These operations allow us to evolve acceptable AEs algorithms through GP. Then, in the next subsection we discuss how the on-line evolutionary paradigm proposed in Sec. 5.2.2 affects the computational run times of different EC population dynamics, such as steady state and generational replacement.

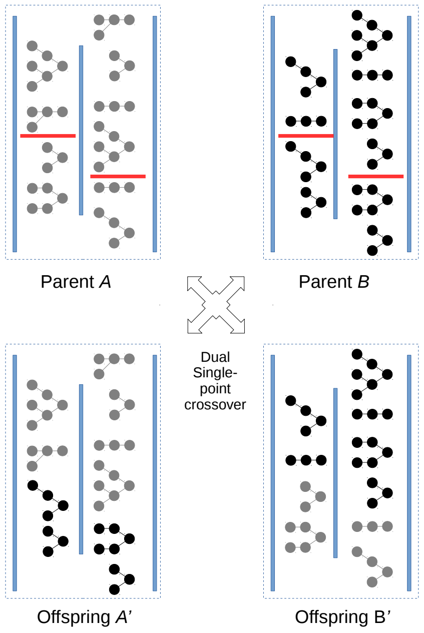


Figure 5.4: Dual Single-point crossover. Red lines in parents' forests show crossover points to create offspring by mixing the forests.

5.3.1 GP Operators

Three different GP operators can be used to evolve GP based AEs: two different types of crossover and one type of mutation. Further down, in Sec. 5.4, performance of both types of crossover are compared experimentally when combined with different mutation probabilities.

5.3.1.1 Dual Single-point Crossover

Single-point crossover is a type of crossover for forest-like GP representations directly inspired in the way crossover is performed on a canonical Genetic Algorithm. For GP individuals comprised of a single forest the operation consists in taking two individuals as parents, A and B, and generates two offspring, A' and B'. Formally, let n be the number of trees in individuals A, A', B, B'. Let i be a random integer between $1 \leq i \leq n$ that will serve as crossover point. A' (B') forest is generated by copying trees A_0 (B_0) up to A_i (B_i) and trees B_{i+1} (A_{i+1}) up to B_n (A_n). In GP individuals composed by two forests, such as in AEs, the operation is carried twice (dual term), but making sure of generating each offspring's forest from corresponding parents' forests[4]. Fig. 5.4 depicts such type of crossover in GP based AEs' individuals.

Single-point or Dual Single-point crossover might be considered as *explorative* genetic operators, because they are disruptive and potentially expands GPs' search to new points in the search space.

[4]Trying to mix portions of encoder and decoder forests to generate an offspring' decoder or encoder would be an error, because two random crossover points would need to be picked in order to guarantee that the resulting forest has a proper number of trees

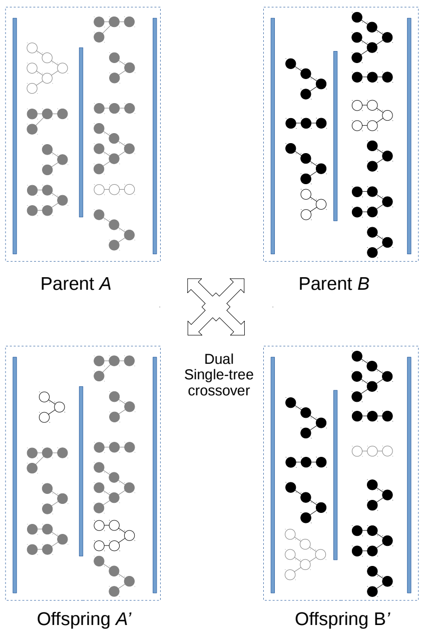


Figure 5.5: Dual Single-tree crossover. Hollow trees are selected to be exchanged between individuals. Encoder (decoder) trees are only exchanged with other encoder (decoder) trees.

5.3.1.2 Dual Single-tree Crossover

In Single-tree crossover, GP forest individuals exchange only one tree to create an offspring. This is in stark contrast to Single-point crossover, where offspring may differ up to 50% from their parents.

The formal definition of Single-tree crossover is as follows. Let n be the number of trees in individuals A, A', B, B'. Let i and j be random integers between $1 \leq i, j \leq n$. Individual A' (B') is generated by making a full copy of A (B) where only tree A'_i (B'_j) are replaced by tree B_j (A_i).

For GP based AEs the process is performed twice. Two random integers are generated, each one to select trees that are copied from parents' encoders and decoders. Offspring encoders can only import trees from parents' encoders, and the same applies for decoders. Fig. 5.5 shows this type of crossover for GP based AEs' individuals.

Even though both Single-tree and Single-point are both crossover operations, they are fundamentally different. While Single-point is an exploration operator, Single-tree is *exploitative* in nature, since it takes two already good solutions and exchange bits of them in order to see if further enhancements can be done to their structures. Notice also how in Single-point crossover, even though structures of individuals are mixed, positional value of each tree is held, because they generate the same features z_i or y_i (even though some of its neighboring trees have changed), whereas in Single-tree crossover, a (supposedly) good GP syntax tree that generates some given feature z_i or y_i, will be tested if it fits in generating a different feature z_j or y_j

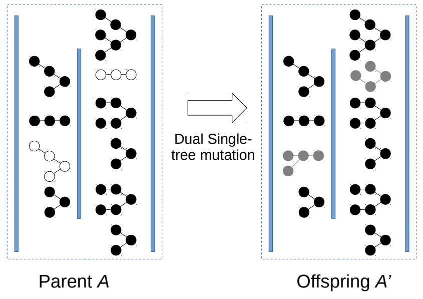


Figure 5.6: Dual Single-tree mutation. The hollow trees are those to be replace by new randomly generated ones.

(and with different structure around it).

5.3.1.3 Dual Single-tree Mutation

Single-tree mutation is similar to Single-tree crossover. From one parent a single offspring is generated by copying the parent's entire forest structure except for a single tree, that gets replaced; but instead of replacing it with one taken from another parent, it gets replaced by a completely new, randomly generated one. For individuals composed by two forests, such as GP based AEs, this process is performed twice, such that one decoder's tree and one encoder's tree is replaced by new randomly generated trees. Fig. 5.6 illustrates this type of genetic operation on a GP based AE's individual.

It is important to stress both the similarities and differences between Single-tree mutation and Single-tree crossover. They can be both considered exploitative genetic operators, because they take good solutions and only perform a small change to them; however, single-tree crossover can be considered even further exploitative because the change applied to generate the offspring comes from another already good solution, whereas in Single-tree mutation, the change is randomly made, without any kind of guarantees regarding the preceding quality of the replacing tree.

5.3.2 Population Dynamics under On-line Learning

In this subsection, we discuss how classical population dynamics models, such as steady state and generational replacement, are affected when applied to the proposed on-line learning approach, in terms of its computational cost. It will be shown that steady state style of evolution requires twice the number of individual evaluations as generational replacement; this extra computational effort is only associated to on-line learning, while under offline learning steady state and generational replacement carry the same computational cost. We also propose a new population dynamics that attempts to close the computational cost gap

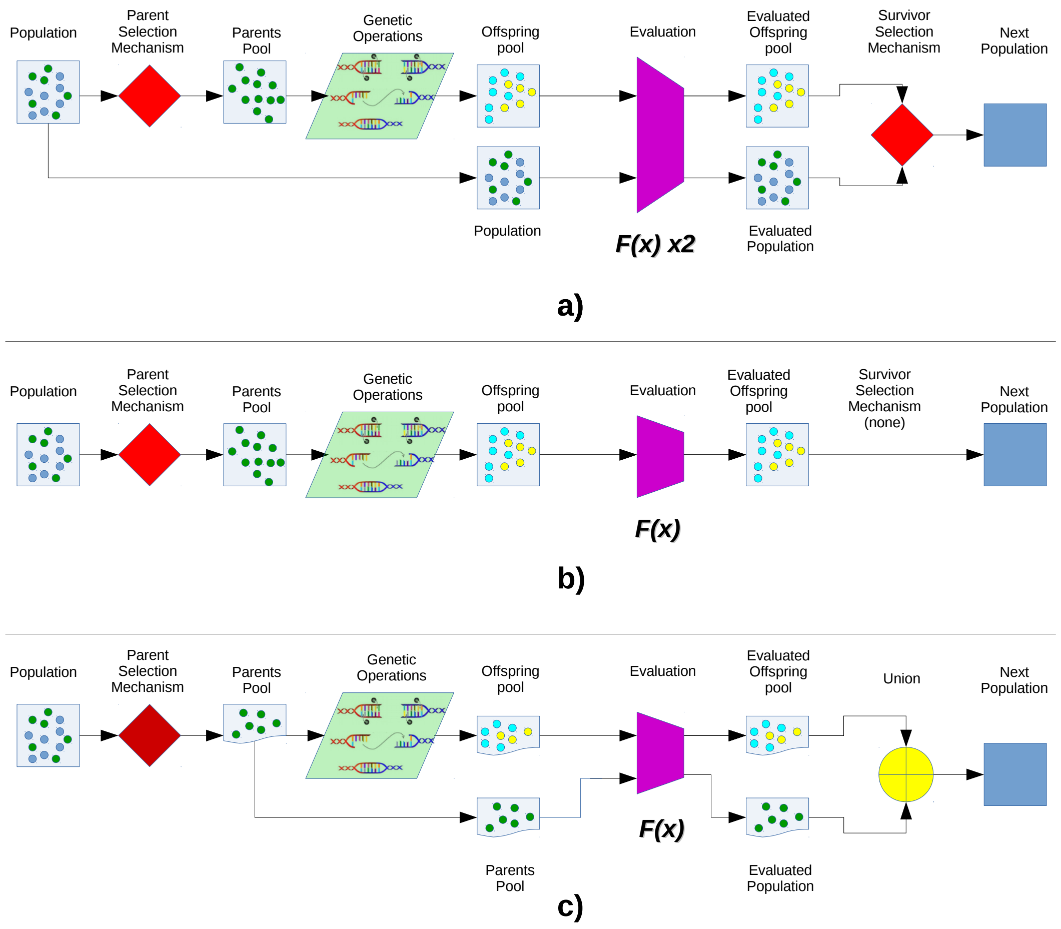


Figure 5.7: Flow diagrams for different evolutionary population dynamics under on-line learning scheme: a) steady state; b) generational replacement; c) efficient steady state. Contrast this flow diagrams, with those presented in Fig. 3.1 for their behavior under an non-incremental learning scheme.

between steady state and generational replacement, we call it *efficient steady state*.

Classical popoulation dynamics were previously introduced in Sec. 3.1.3. Here we will discuss them concerning only on their behavior under on-line learnining schemes.

5.3.2.1 Steady State

A *steady state* population dynamics consists in replacing part of the current population with newly generated offspring. This would require, in theory, an evaluation on both individuals pools (current population and offspring) such that a deterministic or partially stochastic selection process can be performed to select individuals from both pools that will survive into the next generation. Notice however that in a conventional GP without any form of incremental or on-line learning (i.e. all training samples are used every generation) current population evaluation is not necessary at all, because those individuals have already been evaluated: a fitness value was associated with them in the immediately previous generation when they were evaluated in order to be eligible to make it into the next generation or in the initial evaluation when the GP started. Therefore, in such case, only individuals in the offspring pool are evaluated.

In contrast, in the proposed incremental learning method, the samples set changes every generation. Individuals in current generation have to be evaluated again using the current minibatch data even though they already had a fitness value. This means that both the entire current population and the offspring needs to be evaluated before the selection process for the next generation. This double effort is still quite small compared with a non-incremental learning approach (given that the minibatches are small enough). This double evaluation process is not present in other population dynamics, such as generational replacement. Fig. 5.7a illustrates one evolutionary cycle for steady state population dynamics; notice how the initial population needs to be evaluated along newly generated offspring, which means the computational effort of steady state is twice the population size (given that the offspring pool is the same size as the population).

5.3.2.2 Generational Replacement

In *generational replacement* population dynamics, the entire pool of individuals that comprise the current population is replaced by the offspring pool (Eiben *et al.*, 2003). This means that in a generational replacement scheme and under an incremental learning approach, the current population is not compared against the current minibatch, only the offspring, and only because it is evaluated to start the next cycle (parents selection can happen immediately), not because offspring are selected (all offspring pass to the next generation). There is no double effort in generational replacement under on-line learning: the population size is exactly the number of individuals that are evaluated every generation, unlike steady state, where it is twice the number of individuals in the population that are evaluated every cycle.

Even though either generational replacement or steady state can be accelerated orders of magnitude under an incremental learning approach, there still remains the question of which one is actually more efficient (given the same time, which achieves the higher quality solution). So a large part of the experimental evidence presented in this chapter (Sec. 5.4) is geared towards answering such a question. But before that, we will present an alternative version of steady state that aims at achieving the computational cost of generational replacement but with a behavior closer to steady state. Fig. 5.7b shows an abstract depiction of the steps that comprise generational replacement evolution style; notice how the starting population is completely discarded, and consequently there is no extra effort in the evaluation stage.

5.3.2.3 Efficient Steady State

Efficient Steady State (ESS) is a modified version of steady state population dynamics that consists in (1) generating a smaller offspring pool and (2) leveraging on the parent selection process to prematurely select individuals from the original population that will make it into the next generation. If both the offspring pool and the parents pools are setup to be half the size of the entire population, then this variant of steady state has the same computational costs of a generational replacement approach, i.e. the number of individuals evaluated each iteration is the same as the size of the population. There is however one important caveat regarding the use of this approach: individuals from the original population that are selected to pass to the next generation are also based on their performance obtained with **previous** minibatch, even though they are evaluated again with the current minibatch, the selection process has already happened. This is an important difference when compared with the regular version of steady state. Our hypothesis is that this approach will be more efficient than the original steady state dynamics. In Sec. 5.4 we compare both approaches.

ESS is specifically aimed at reducing the computational cost of steady state style dynamics when combined with incremental learning approaches. Notice that ESS does not make sense in an non-incremental learning scenario, because steady state is not twice as expensive in such cases, as explained in Sec. 5.3.2.1. Fig. 5.7c shows the flow diagram of ESS dynamics.

5.4 Empirical Assessment

In this section, we present results of experimental evaluations of the proposed GP based AEs. We performed four main studies. First, we performed a preliminary study where we compare the performance of the SLGP against a standard GP approach where no partitioning of the input space is done, as well as comparing the SLGP with and without on-line learning. The second study is an exhaustive study of EC parameters to control GP, including population dynamics, balancing GP operators probabilities, types of crossovers and population sizes. The third study compares the performance of synthesized GP based AEs with an ANN implementation that performs similar to earlier Deep Neural Networks (DNN); finally, we carried out a small study that examines the diversity of individuals in the populations for the different population dynamics.

5.4.1 Used Datasets

For experimentation we considered three datasets: handwritten digits MNIST (LeCun, 1998), faces dataset Olivetti (Samaria & Harter, 1994) and a special version of Labeled Faces in Wild, LFWCrop (Anderson, 2014); all being gray scale image datasets. We chose MNIST and Olivetti because there has been

Table 5.1: Datasets used for experimentation. All datasets consist in grayscale images; pixel values are normalized to fall in the range [0,1] in all cases.

Dataset	**MNIST**	**LFWcrop**	**Olivetti**
Images Resolution	28x28 (784)	64x64 (4,096)	64x64 (4,096)
No. Training samples	60,000	12,000	360
No. Testing samples	10,000	1,233	40

previous studies with ANN-based autoencoders with such datasets (Hinton & Salakhutdinov, 2006b) that we could compare against. We added LFWCrop to the battery of tests in order to try the proposed methods in a image dataset similar to Olivetti but with more training data available. All datasets were split in training and testing sets. Table 5.1 details the number of training and testing samples of each dataset, as well as image sizes. Input representation for developed AEs is formed by all image pixels, i.e. the number of pixels per image equals the number of features of the initial representation to be encoded.

5.4.2 Preliminar Study

We tested three different configurations for GP autoencoders. The first setup consists of a straightforward GP approach, i.e. all GP trees that generate the encoded representation can see all input features, as well as all GP trees in the decoder can see all encoded features. The compression ratio is setup to $4:3$.

The second setup consists of a structured layer GP autoencoder, as described in Sec 5.2.1. The compression ratio is the same as in the first setup, for a valid comparison, and the n (m) features from representation x (z) are split into $c = \frac{n}{4}$, $(k = \frac{m}{3} = c)$ subsets C_i (K_i), such that $|C_i| = 4$, $(|K_i| = 3)$, $\forall i$. That is, four features from input representation x are assigned to each subset C_i, and from this, and only this, subset of features, is that features in subset K_i can be generated. Mirroring this configuration, the decoding forest is also partitioned in subsets of four trees, such that trees in each subset can only see the subset of three features of some subset K_i. All these subsets, (4) input features-(3) encoding trees-(3) new features-(4) decoding trees, are coupled together, to form a miniautoencoder. The compression ratio is kept low because in these set of experiments we only wish to test GP capability of generating an intermediate representation, i.e. if it is able to evolve at least two layers of processing, where there is function composition, rather than actually test the compression limits/capabilities of the proposed technique, which will be explored in further experiments. We use every four neighboring pixels in the same row to be in subset C_i; Fig. 5.8 illustrates such partitioning scheme. Notice how since there are no overlapping between subsets, borders do not require any special treatment.

The third setting is exactly as the second one, except that we use a *minibatch* based form of training, as described in Sec. 5.2.2.

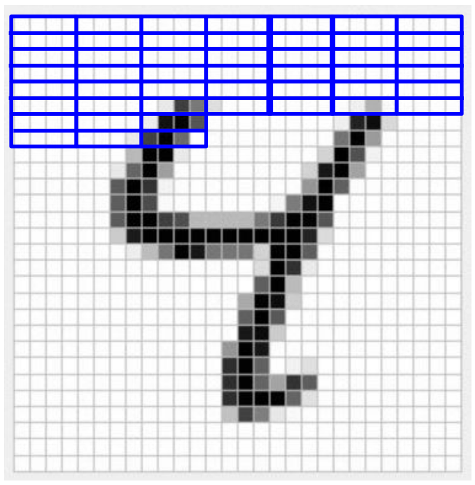


Figure 5.8: Partitioning scheme for a $4:3$ compression ratio setup. Each subsets consisting of 4 pixels is reduced to 3 new variables in the intermediate representation and then decoded back to 4 pixels.

Table 5.2: Evolutionary parameters for the GP runs. Arithmetic operands are 2-ary and trigonometric functions are unary primitives. The division function is protected, meaning that any attempt to divide between zero returns as output 1×10^6, instead of an error.

Parameter	**Value**
Population size	60
Max. Tree depth	4
Set of Primitives	$< +, -, \times, \div, sin, cos >$
Constants range	[0,1]
Crossover Prob.	0.6
Mutation Prob.	0.3
No. Generations	40/40/600

For this set of experiments we used ESS population dynamics, with a 0.6 probability of crossover (single tree) and a 0.3 probability of mutation. Table: 5.2 summarizes the rest of the GP parameters used. Notice that the number of generations given for the on-line version is significantly greater than for the other two approaches; this due the fact that the on-line approach can perform the evaluations much faster.

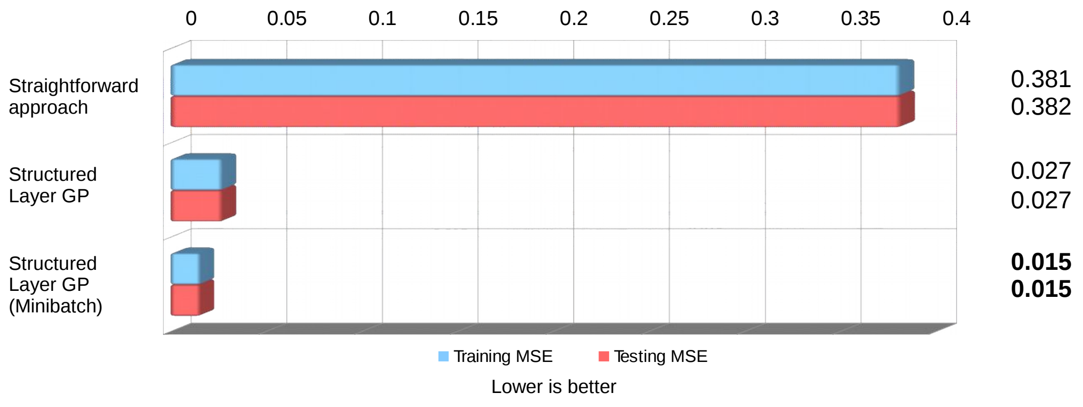
To determine similarity between an original sample and the reconstructed output from the autoencoder, we used the mean square error (MSE), defined in Eq. 3. MSE receives as input an original sample $\mathbf{x}$ and its reconstructed $\mathbf{y}$ vector, and compares them feature by feature, averaging the difference across all features. MSE output can be thought as a *distance* between a sample and its reconstruction:

$$d_{\mathbf{MSE}}(\mathbf{x}, \mathbf{y}) = \frac{1}{n} \sum_{i=1}^{n} (x_i - y_i)^2 \tag{3}$$

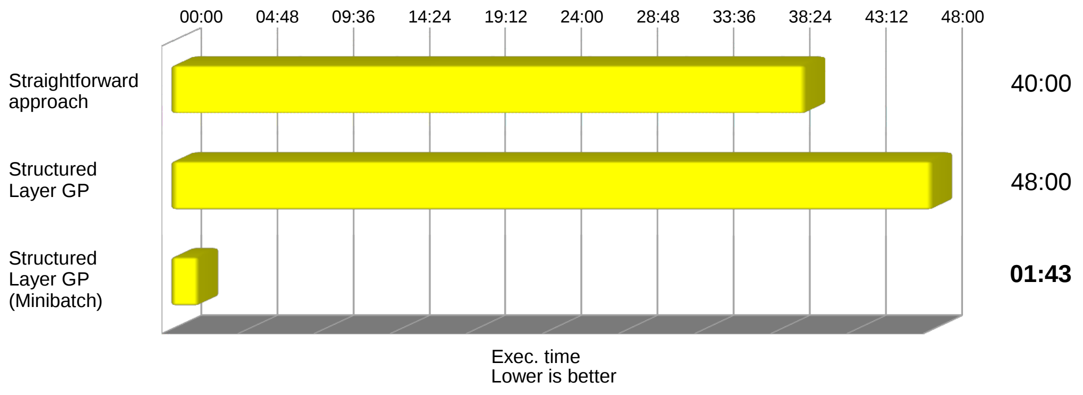
where x_i (y_i) is the i-th variable of original (reconstruction) vector $\mathbf{x}$ ($\mathbf{y}$), and n is the size of representation vectors.

The objective function in the first two setups described is to minimize the average MSE across all pairs sample-reconstruction from some given dataset. At every generation, each individual of the population is tested against the entire training dataset, the resulting MSEs for every instance in the dataset are averaged, and this result is assigned as the fitness for a given individual. On the other hand, the objective function in the third setup is minimizing average MSE for all samples in the current minibatch presented to the population.

Fig. 5.9 shows the results obtained by the straightforward GP, the structured layer GP and the minibatch training version of it; minibatches were composed of 100 samples. We compared the three approaches in MNIST dataset. Models were trained using half of the training set of MNIST (30,000 samples). The straightforward GP can make use of the multiple processing cores by parallelizing the evaluation of sample-reconstructions pairs. On the other hand, both structured layer GP approaches 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)



b)

Figure 5.9: Results obtained by the three different GP setups (a) MSE across all samples in training and testing datasets. (b) Execution time expressed in hh:mm.

Fig. 5.10 compares the reconstruction for the first ten images in the training set, as obtained by the best

Table 5.3: Average MSEs and exec. time for a minibatch approach varying the size of the batches to 30, 60, 100, 300, and 600 samples; and giving 1, 2 and 5 forward passes over the dataset.

	Mini Batch Size														
	30			60			100			300			600		
Epochs	Training	Testing	Time	Training	Testing	Time	Training	Testing	Time	Training	Testing	Time	Training	Testing	Time
1	0.013	0.013	04:44	0.012	0.012	03:36	0.013	0.013	03:15	0.017	0.017	02:49	0.020	0.020	02:39
2	0.010	0.010	09:31	0.011	0.011	07:24	0.011	0.011	06:32	0.014	0.014	05:30	0.017	0.017	05:31
5	0.009	0.009	23:27	**0.008**	**0.008**	18:04	0.009	0.009	16:27	0.011	0.011	14:26	0.014	0.014	13:23

autoencoders generated by the three different experimental setups. For the third setup we also performed a test varying the size of the minibatches and increasing the number of generations in order to allow the GP to see each sample more than just once. Table 5.3 shows results of varying the size of minibatches to 30, 60, 100, 300, and 600 samples; and allowing the algorithm to give one, two and five forward passes over the training data; in this case, the entire training set of MNIST (60,000 samples) was used for training purposes. All experiments were carried on a Intel Core i5 at 4.2 GHz, with 16GB in RAM, running Debian Linux 9. All algorithms were implemented in a in-house GP library developed in Python version 3.6.

From the results presented in this section we can appreciate that the structured layer GP approach is one order of magnitude better than a straightforward GP approach in average MSE . In fact, the straightforward approach does not reach an acceptable solution at all given approximately the same amount of time, making clear the advantage of the proposed approach. Even though the difference, in terms of quality of solutions, is not as decisive between the structured layer GP and its minibatch learning version, the gap in execution time between them is also one order of magnitude. Results also show that the size of the minibatches have to be carefully selected in order to get a balance between quality of solution and execution time. We also confirmed that the minibatch version can benefit from making several passes over the training data.

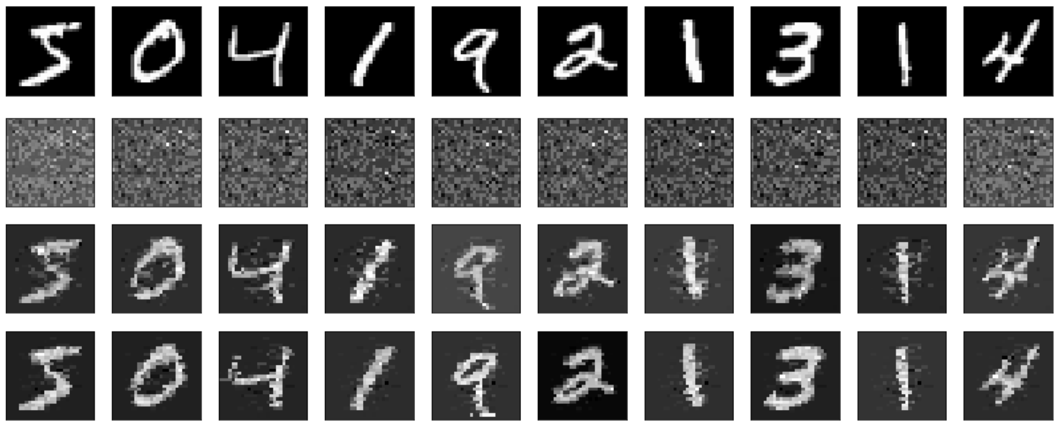


Figure 5.10: Comparison of the reconstruction of the three experimental setups. From top to bottom: original first 10 images from the training set, best straightforward GP reconstruction, best structured layer GP reconstruction, structured layer GP + minibatch training reconstruction.

5.4.3 EC Parameters Study

In this section, we present results of an experimental study carried out in order to gather evidence that sheds light on to the following questions regarding the evolution of GP based AEs structures: (1) is GP a capable method to learn AEs algorithms?; (2) are crossover and mutation a different type of genetic operation, or is crossover equivalent to mutation?; and if so, what is the optimal relation (probability) between both; (3) which population dynamic (generational, steady state, ESS) is more efficient? and (4) which type of crossover (single-point or single-tree) yields better results?.

We organized experiments in the following way: we evolved AEs to perform a 4:3 dimensionality reduction on Olivetti dataset varying crossover / mutation probabilities balance between four options (0.9/0.1, 0.7/0.3, 0.5/0.5, 0.3/0.7), four different population sizes (25, 50, 100, 200), and using generational replacement, steady state and ESS population dynamics; we performed all those experiments using single-point crossover and single-tree mutation, and then repeated the entire set of experiments substituting single-point crossover for single-tree variant. Each experiment (combination of crossover/mutation balance - population size - population dynamic - type of crossover) was performed 5 times in order to obtain an average behavior of the given setup and perform statistical analyses. Olivetti training set is split into six mini-batches of 60 images each. Each experiment ran for 10 epochs (60 generations). The rest of the EC parameters (max tree depth, set of primitives, constants range) are the same as those stated in Table 5.2.

Since in this study we are only interested in observing the effects of different EC parameters combinations on EAs evolution, we did not vary the dimensionality reduction ratio, nor used another dataset than Olivetti for this study; although these limitations could naturally represent a bias in the obtained results, the analysis performed in Sec. 5.4.5 seem to corroborate that the behavior of the parameters herein studied are not related with the dataset used or the reduction ratio, but rather with the preservation of diversity they contribute to the population. In the next experimental set we test GP based AEs performance given different reduction rations on different datasets. Preliminary experiments revealed that the smaller the partitions, the better the performances, this is consistent with the well-documented dimensionality scalability issues of GP (and counter-intuitively with the idea that larger partitions have more context information that could lead to better performance).

The 4,096 pixels in original images from Olivetti dataset are the input representation. These pixels are grouped into 1,024 subsets C_i each consisting of four pixels. These subsets are assembled by picking 4 contiguous pixels from the same row without any overlapping. From each subset C_i, a subset K_i of 3 feature variables from the compact input representation are generated, and from this K_i the corresponding original

4 pixels are reconstructed into subset Q_i. This way the compression ratio of 4:3 is achieved according to the structured layered GP approach described in Sec. 5.2.1. We used average MSE minimization across all images per minibatch as the objective function.

Plots in Figures 5.11 and 5.12 show the average MSE obtained by each setup in the testing dataset after every epoch of training. Fig. 5.11 corresponds to all configurations where Single-point crossover was used, and Fig. 5.12 to those of Single-tree. Each figure contains three plots, corresponding to three different population dynamics tested; and each plot is further divided into four groups that correspond to different population sizes. Table 5.4 shows the MSE obtained in the testing dataset in average for all the different setups after 10 epochs.

For every group of different crossover/mutation probabilities setups that share the other configuration parameters (type of crossover, population size and population dynamics), we performed a statistical test to determine which one of such probabilities balances, if any, provides the best results after 10 epochs. We applied Dunn's test (Dunn, 1961) to determine if there is a significant difference among the probabilities balances in each group. Results show that they are significantly different. We used a level of significance of 0.05 for all tests. Each cell in Table 5.4 groups the different probabilities balances that share the other configuration parameters. For all cells, the configuration with the lowest mean is significantly lower (better) than the two with the highest mean, and not significantly different from that with second-lowest mean.

5.4.3.1 GP as method to discover AEs

The results obtained in the preliminar experiments and those observed in Figures 5.11 and 5.12, show that GP can systematically reduce reconstruction error for most given EC parameters setups, and for two distinct datasets, providing evidence that the proposed method is capable of synthesizing AEs algorithms.

5.4.3.2 Crossover vs. Mutation

According to the results obtained for all setups after 10 epochs (Table 5.4), a generational replacement dynamics yields better results with a balance that favors crossover over mutation, for either kind of crossover (even though more so for Single-tree crossover), whilst the opposite is true for steady state dynamics, where is preferable to perform at least as many mutations as crossover, if not more. ESS is somewhat of a mixed case, depending on the style of crossover utilized, however we can notice that an equal proportion of crossover and mutation is consistently better for this type of population dynamics. Notice how generational replacement does not keep a steady pool of good performing individuals, so it benefits from

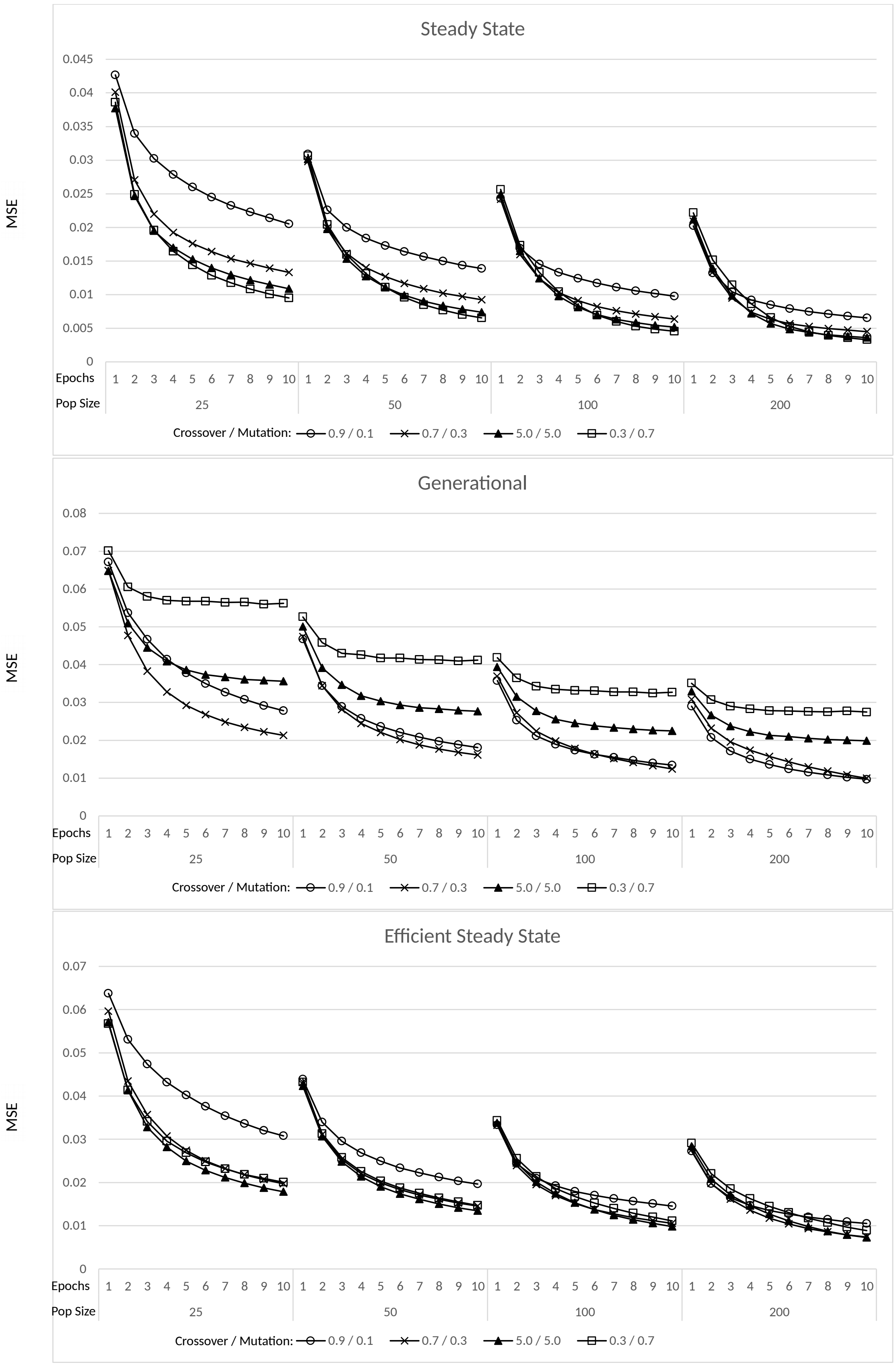


Figure 5.11: Performance of GP based AEs evolved under different setups using Dual Single-point crossover. Results show the average MSE obtained by 5 GP based AEs across 10 epochs when transforming and reconstructing all images from the testing set under the given configuration.

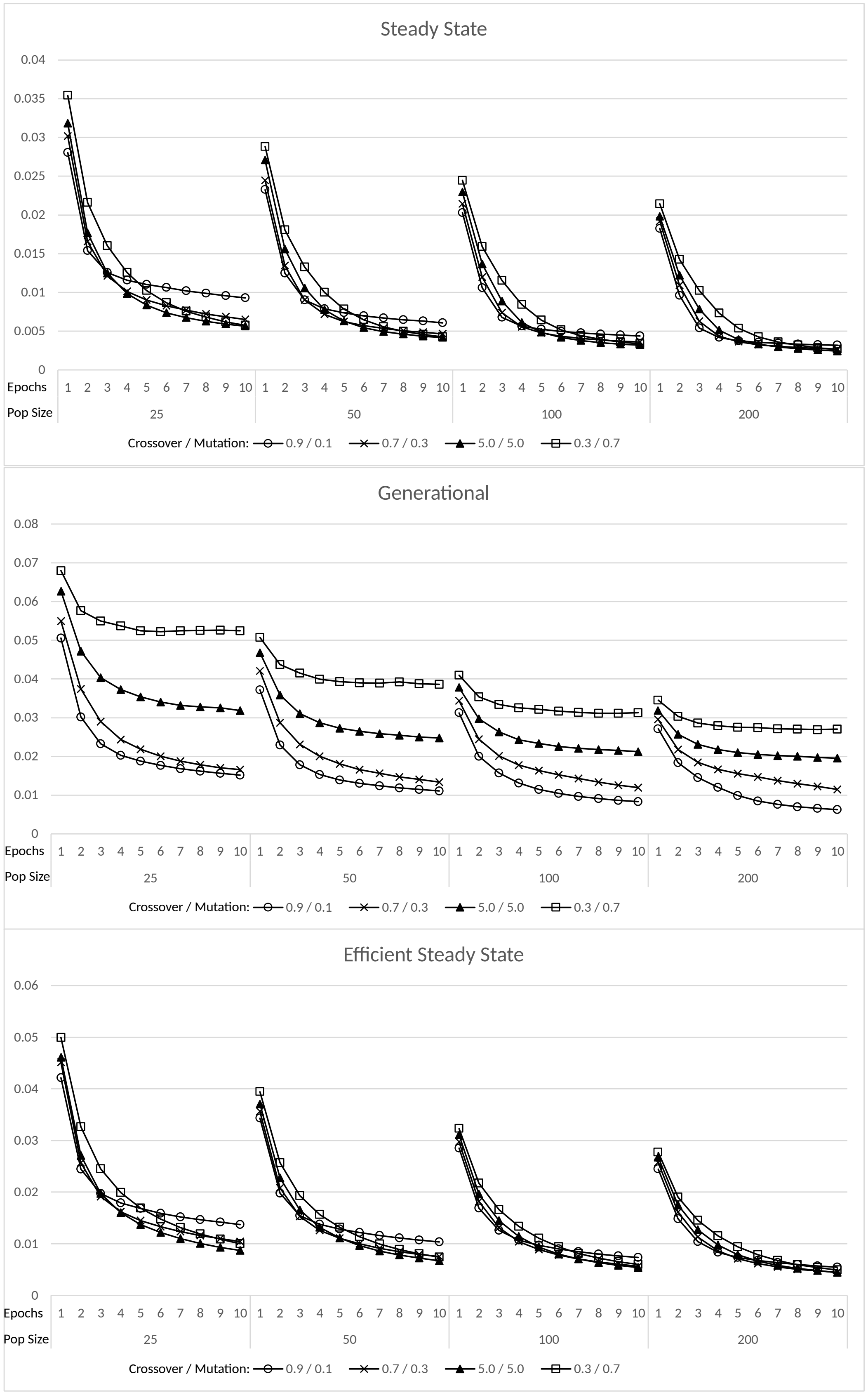


Figure 5.12: Performance of GP based AEs evolved under different setups using Dual Single-tree crossover. Results show averaged MSE obtained by 5 GP based AEs across 10 epochs when transforming and reconstructing all testing set images for the given configuration.

Table 5.4: Results after 10 epochs for every configuration tested. Results in bold are significantly better within their cell. Results with * denote those that are significantly better than the others with equivalent computational cost. Result denoted with ** is the overall best result found. Standard deviations are expressed in hundred thousands.

		Single Point			
Gen.	Pr_c / Pr_m	25	50	100	200
	0.9 / 0.1	**0.0278 ± 46**	**0.0181 ± 30**	**0.0134 ± 30**	**0.0096 ± 7.2**
	0.7 / 0.3	**0.0212 ± 25**	**0.0160 ± 16**	**0.0124 ± 18**	**0.0099 ± 17**
	0.5 / 0.5	0.0356 ± 10	0.0276 ± 45	0.0224 ± 13	0.0198 ± 9.0
	0.3 / 0.7	0.0561 ± 64	0.0411 ± 95	0.0326 ± 28	0.0274 ± 2.0
SS	Pr_c / Pr_m	25	50	100	200
	0.9 / 0.1	0.0205 ± 18	0.0138 ± 23	0.0097 ± 11	0.0065 ± 7.9
	0.7 / 0.3	0.0132 ± 34	0.0092 ± 18	0.0063 ± 11	0.0044 ± 8.0
	0.5 / 0.5	**0.0108 ± 23**	**0.0073 ± 16**	**0.0051 ± 14**	**0.0036 ± 9.5**
	0.3 / 0.7	**0.0094 ± 10***	**0.0065 ± 7.1***	**0.0045 ± 8.4***	**0.0032 ± 9.4**
ESS	Pr_c / Pr_m	25	50	100	200
	0.9 / 0.1	0.0307 ± 43	0.0196 ± 21	0.0145 ± 14	0.0105 ± 13
	0.7 / 0.3	**0.0196 ± 5.6**	**0.0145 ± 9.2**	**0.0105 ± 6.4**	**0.0073 ± 6.3**
	0.5 / 0.5	**0.0178 ± 22**	**0.0134 ± 16**	**0.0098 ± 16**	**0.0072 ± 9.3**
	0.3 / 0.7	0.0200 ± 31	0.0146 ± 11	0.0111 ± 13	0.0088 ± 11
		Single Tree			
Gen.	Pr_c / Pr_m	25	50	100	200
	0.9 / 0.1	**0.0152 ± 28**	**0.0110 ± 32**	**0.0083 ± 16**	**0.0062 ± 11**
	0.7 / 0.3	**0.0166 ± 24**	**0.0134 ± 8.8**	**0.0119 ± 18**	**0.0115 ± 24**
	0.5 / 0.5	0.0318 ± 74	0.0247 ± 25	0.0212 ± 8.7	0.0195 ± 16
	0.3 / 0.7	0.0524 ± 59	0.0386 ± 38	0.0313 ± 14	0.0270 ± 32
SS	Pr_c / Pr_m	25	50	100	200
	0.9 / 0.1	0.0092 ± 27	0.0060 ± 24	0.0043 ± 9.6	0.0031 ± 6.7
	0.7 / 0.3	0.0065 ± 7.5	0.0046 ± 5.9	0.0035 ± 6.3	0.0026 ± 3.1
	0.5 / 0.5	**0.0056 ± 9.2***	**0.0041 ± 6.4***	**0.0031 ± 5.3***	**0.0024 ± 1.6****
	0.3 / 0.7	**0.0057 ± 15**	**0.0042 ± 6.5**	**0.0033 ± 3.4**	**0.0026 ± 5.0**
ESS	Pr_c / Pr_m	25	50	100	200
	0.9 / 0.1	0.0137 ± 27	0.0103 ± 36	0.0073 ± 17	0.0054 ± 4.9
	0.7 / 0.3	0.0104 ± 43	0.0075 ± 16	**0.0056 ± 12**	**0.0044 ± 11**
	0.5 / 0.5	**0.0087 ± 12**	**0.0067 ± 9.5**	**0.0054 ± 7.5**	**0.0044 ± 9.0**
	0.3 / 0.7	**0.0100 ± 24**	**0.0074 ± 13**	0.0059 ± 11	0.0049 ± 4.9

operations that implicitly conserve the best performing individuals such as single tree crossover; in contrast, steady state consists in always ensuring survival of best fitted individuals, that gradually deplete diversity; as such, this dynamics benefits the most from operations that introduce random changes to the individuals in the population, such as mutation.

5.4.3.3 Population dynamics

We also compared each top performer setup after 10 epochs from steady state configurations against the ESS setup configured with twice the population size also after 10 epochs (because they require the same computational effort), i.e. the top setup of steady state with 25 population size against ESS with 50 population size, both with the same type of crossover; the top steady state with 50 population size against the top ESS with 100 population size, and so on, and then again with the other type of crossover. We applied Wilcoxon rank sum test (Wilcoxon, 1945) to compare each pair and used a level of significance of 0.05 for all tests. Results showed that steady state yields statistically better results in all cases. Unfortunately our proposed population dynamics, ESS, did not manage to achieve higher efficiency than steady state, i.e. it is always preferable to use a steady state setup with any given population size than an ESS with twice such population size. Generational replacement dynamics on the other hand lags even further than ESS to steady state.

5.4.3.4 Crossover type

From the results observed in Figures 5.11, 5.12 and Table 5.4 we can realize that there are significant differences between different crossover/mutation balances, meaning that they are indeed different kind of GP operations for this kind of individuals representation (otherwise they would all behave the same). We tested the top performers from each steady state population size study, to determine which of the two crossover styles yields results significantly better, i.e. we tested the top steady state of 25 population size single-point against the top steady state with 25 population size single-tree, and so on. We also used Wilcoxon rank sum test for comparison. Results showed that, in all cases, Single-tree crossover yields significantly lower (better) average MSE values than Single-point crossover.

5.4.4 Comparison with other methods

In this section, we compare the proposed GP based AEs performance against another popular representation learning method. We picked up the best configuration found from the analysis described in the previous section (Steady State - Single-tree - 0.5/0.5 crossover/mutation prob) and compared its performance against an ANN that performs similar to earlier deep networks. We performed the comparison for three different encoding ratios, 2:1, 4:1, and 8:1. For this tests, we used all three datasets described in Section 5.4.1, LFWCrop, Olivetti and MNIST. The number of features generated given the three encoding ratios tested were, for MNIST (Olivetti/LFWCrop), 392 (2,048), 196 (1,024) and 98 (512).

We used fully connected ANNs, composed of three layers for encoding and decoding stages each. The first encoder's two layers are composed of exponential linear units (ELUs) while the last layer (where the encoded representation resides) is made up of linear units. The decoder follows a similar architecture, where the first two layer are made up of ELUs and the last, output, layer is composed of sigmoid units. The architectures of the encoding networks are as follows, for MNIST (Olivetti/LFWCrop), 784-588-490-392 (4098-3072-2560-2048), 784-392-294-196 (4098-2048-1536-1024), and 784-392-196-98 (4098-2048-1024-512). The decoding networks followed an architecture that mirrored that of the corresponding encoding network. We set the minibatch size to 60 samples for all setups, and used Adam Optimization algorithm (Kingma & Ba, 2014) variant of SGD for network training. We trained networks for 100 epochs, at which point they no longer appeared to improve.

Table 5.5: MSE results on the testing set for the compared methods. Values remarked in bold are better.

	Reduction Ratio	**GP**	**ANN**	**CNN**
MNIST	2:1	0.0158	**0.0018**	0.0047
	4:1	0.0241	**0.0016**	0.0079
	8:1	0.0373	**0.0020**	0.0113
Olivetti	2:1	**0.0032**	0.2176	0.0085
	4:1	**0.0058**	0.0089	0.0089
	8:1	0.0177	**0.0063**	0.0108
LFWCrop	2:1	**0.0014**	0.0349	0.0018
	4:1	**0.0024**	0.0068	0.0029
	8:1	0.0050	0.0073	**0.0034**

We also include the comparison with CNNs. The CNNs used are composed of three layers for encoding and decoding stages each. The number of convolutional filters of the encoding networks at each layer are as follows, for MNIST (Olivetti/LFWCrop), 24-24-24 (32-32-32), 24-12-12 (32-16-16), and 24-12-6 (32-16-8), for the 2:1, 4:1, and 8:1 encoding ratios respectively. All convolutional filters are of size 3×3; ReLUs are used as activation functions on all layers; each convolutional layer is also followed by a Max Pooling layer of 2×2. The decoding networks followed an architecture that mirrored that of the corresponding encoding network, where the Max Pooling layers are replaced by Up Sampling layers. The Max Pooling layers in combination with the number of filters at the last enconding layer achieve the desired reduction ratio (approximately in the case of MNIST, and exact in the case of Olivetti and LFWCrop). All CNNs' last layer consist in a single convulutional filter of 3×3 with a sigmoid activation unit. RMSProp was used as optimization method in all cases. These networks' architectures were based on the suggestion presented by Chollet (2016), and cannot be considered state of the art CNNs; the purpose of this comparison is merely to compare the proposed GP-AEs against a CNN and not only against fully connected networks.

For the GP we made some adjustments in order to set it up to perform the desired encoding ratios. We established windows of 4×2, 4×4, and 4×4 pixels, to form sets C_i, from where such patches were reduced to 4, 4, and 2 features respectively, using the SLGP scheme. For all experiments we set the population size to 100 solutions, and let the GP evolve through 10 (20) epochs for MNIST and LFWCrop (Olivetti). We also changed the maximum allowed tree depth to 6 for all experiments, for both encoders and decoders.

Table 5.5 shows the average MSE obtained by each method for the testing sets of each dataset. From these results, we can appreciate that GP is competitive: it yields better results than an ANN when training data is scarce in comparison to the network size, or competitive results when the reduction ratio is not too aggressive.

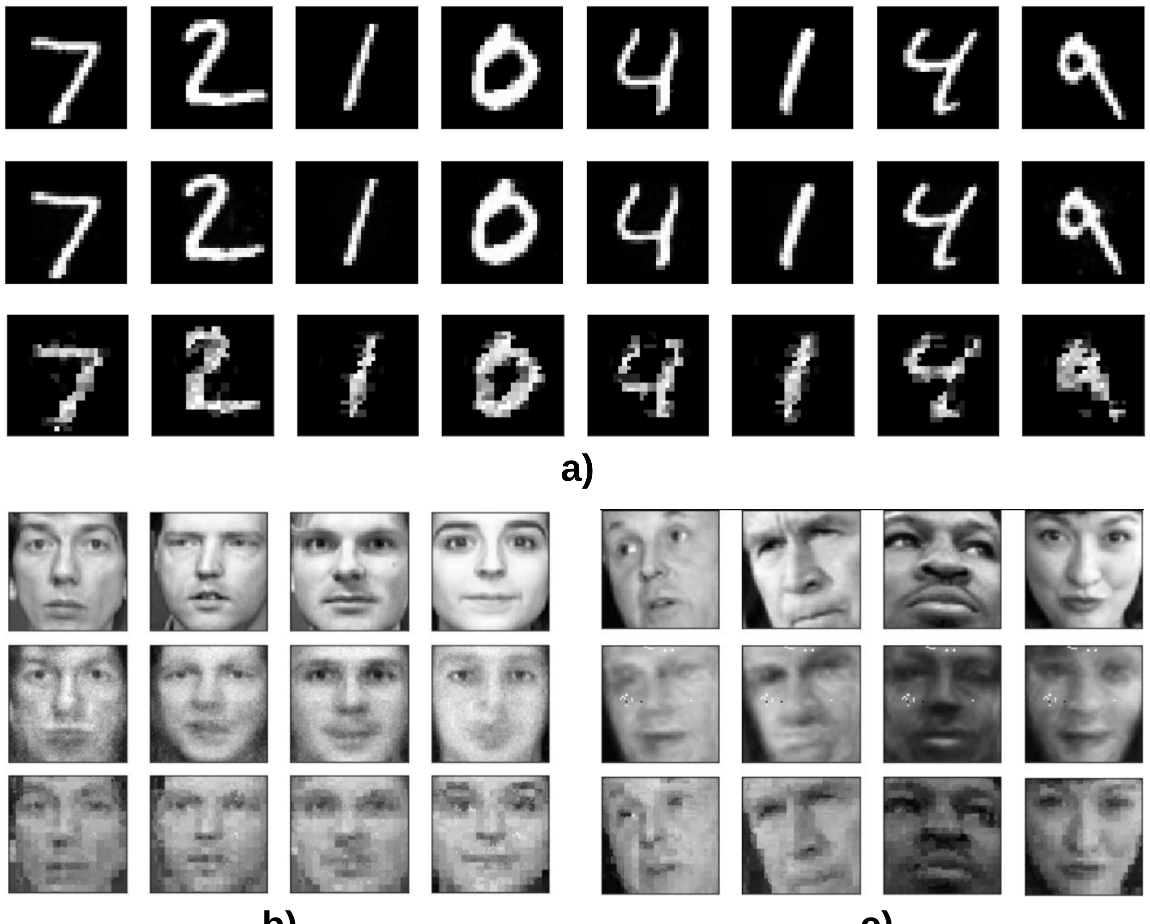


Figure 5.13: Reconstructions generated from encodings of two different methods at a reduction ratio of 4:1. (a) MNIST; (b) Olivetti; (c) LFWCrop. From top to bottom, for all sets: originals, ANN and GP reconstructions.

In Fig. 5.13 we can observe that the qualitative results generated by GP based AEs at a reduction ratio of 4:1 are still acceptable, whereas from the same figure and from Table 5.5 we can conclude that 8:1 is an upper limit for dimensionality reduction of the proposed technique. On the other hand, Olivetti training set is comprised of only a few hundred samples, and a DNN that attempts to reduce such images to only half their initial dimensionality has too many parameters to calibrate while having too few training samples to do so. In contrast, GP does not suffer from this phenomenon, due to its *non-parametric* learning nature. By this we mean that GP does not rely on a number of internal parameters to tune in order to build models, i.e. it is parameter-free (unlike all kinds of ANN). LFWCrop is another case where the lack of massive amounts of training data puts the proposed GP approach on par with the ANN, if not better.

Notice from Table 5.5 how GP performance degrades as the reduction ratio increases, whereas the ANN performance increases. Once the reduction ratio is of certain factor, our proposed scheme cannot make small enough partitions for GP. Further techniques are required to adapt GP to higher dimensionality problems.

5.4.5 Diversity Analysis

Our final study consisted in analyzing the amount of diversity of individuals in GP based AEs populations for different population dynamics set at their corresponding best crossover/mutation balance. The purpose of this study is to analyze the decay in diversity as generations elapse, and to determine if there is a type of GP operator, population dynamics, or combination of both, that allows to better preserve diversity, and see if there is any type of correlation between efficiency delivered by each setup and population diversity. This study also allows us to measure how much computational effort is wasted in evaluating the same individuals over and over again in a GP, a typical problem found in most EC paradigms.

We developed a method to measure diversity in a population of GP based AEs' individuals. The process involves extracting a genotype signature from each individual, so in every generation we store those signatures from all individuals in a list, and then we count how many of the signatures in the list are unique. We define diversity as the ratio of different signatures found over the total size of the population:

$$Diversity = \frac{UniqueSignatures}{PopSize} \tag{4}$$

The signature of each individual is assembled by extracting a genotype signature of every tree in its corresponding forests. A tree signature is formulated from its nodes' list. All encoder trees signatures are concatenated, as well as all decoder trees signatures, then encoder and decoder signatures are concatenated to form an individual's complete signature. Notice, however, that the process for building individuals' signatures also generates encoders' and decoders' independent trees signatures as an initial step. This allow us to also measure diversity of distinct trees found across all encoders and decoders, because it might be the case that two individuals are considered different according to their signature, but they actually are composed of the same trees in their encoders and decoders, but organized differently.

We ran a single diversity analysis for each one of the top performing configurations of population dynamics and crossover style. Figure 5.14 shows the decay of diversity as generations elapse. Results suggest that single-tree crossover is more successful at preserving diversity at the individual level, while steady state does so at tree level, and this is the reason why the combination of both parameters yields the best results[5].

Still, diversity at individual level drops to half of the population (half of the population is duplicated) in just 40 or less generations, and yet diversity at tree levels drops even more dramatically. These results

[5]Consider that diversity at any given number of generations for steady state has to be compared against 2x the number of generations for ESS or generational replacement, because at such points the computational effort is equivalent. Under this caveat, diversity superiority of steady state becomes much more evident

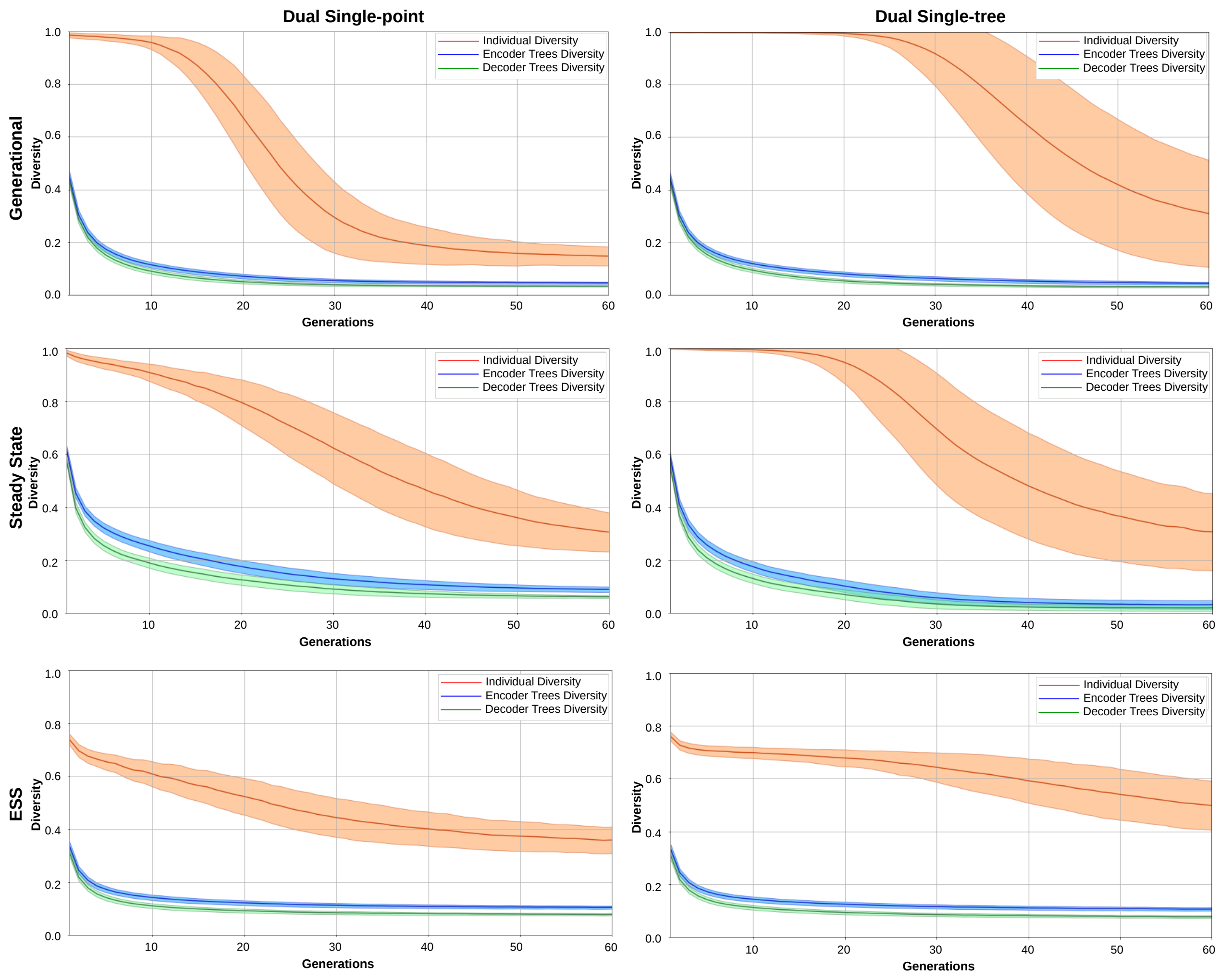


Figure 5.14: Diversity *vs* generations on structurally layered GP under different configurations, to perform 4-to-3 dimensionality reduction in the Olivetti dataset. There is a total of 1024 partitions per setup, plots are individuals, and trees average diversity, while colored areas represent the standard deviation to the mean.

show us that there is a huge amount of computational resources, both memory and processing time, wasted. These results entices us to open new research directions regarding maintaining population diversity, revisit paradigms of evolutionary computation where only the top performer individual survives to the next generation or even propose new paradigms of GP where we evaluate atomic components that conform individuals, and find ways to infer hypothetical individuals performances defined as combinations of the already evaluated atomic components.

5.5 Remarks

In this chapter, we proposed a model for the evolution of autoencoding structures through GP. Instead of relying on evolving ANN-alike algorithms like several previous works, the method presented in this work consisted in evolving native GP structures, i.e. forests comprised of GP syntax trees. Our analysis included a complete comparison on different population replacement methods and genetic operators.

We argued that the on-line learning approach to evolution, discussed in this chapter, represents different computational tolls depending on the population dynamics used. While a generational replacement dynamics requires the evaluation of as many individuals as the population size, a steady state population dynamics requires as twice as many. We also proposed a new population dynamics that attempted to lower the computational cost of steady state dynamics to that of generational replacement, but still preserving its general behavior. Our experimental study showed however, that when given the same number of evaluations (either because the population size is halved for steady state / duplicated for the other methods; or because the steady state is run for half generations, etc.), steady state stands as the most efficient evolutionary dynamics when evolving autoencoding structures. Still, it remains to be seen if this result holds when GP is used to solve other problems, i.e. evolving individuals other than autoencoders, such as classifiers.

Another important result from our research is the outcome from comparing the role of different genetic operators. Our results show that, in the case of evolving autoencoders, both mutation and crossovers are indeed different genetic operations, and that a single-tree crossover yields better results than a single-point type. Results also show that an exact balance between crossover and mutation achieve higher efficiency than other combinations. This results stands among other similar studies (Luke & Spector, 1997, 1998; White & Poulding, 2009) when trying to evolve individuals composed of a single tree, the role of crossover has been controversial. This results vindicates both role and importance of the crossover operator for this type of individuals, and sets GP apart from a mere random search.

Unfortunately results shows that GP based AEs as a method for representation learning still lag behind other available options. Their performance is, at best, comparable to that of state of the art neural networks of fifteen years ago. From the experimental evidence presented here, GP has difficulty with high dimensional problems, and aggressive feature reduction is one such problem. Newer techniques are required to adapt GP to such scenarios.

Nevertheless, it's important to remark that right from the start it was not the purpose of this research to beat the state-of-the-art methods for representation learning or dimensionality reduction, but rather to show GP capabilities to generate autoencoder algorithms that are not based on ANN-alike structures. Therefore, to summarize the contributions of this work, we can highlight:

- A GP-based method of representation learning for high dimensionality datasets, i.e., an approach that allows GP to perform feature extraction on large-scale problems without the need of specialized primitives.

- A GP individual representation for AEs' algorithms.
- We presented experimental results that provide evidence of the performance gains of implementing such an autoencoder with the proposed method compared against a straightforward GP approach.
- An on-line evolution approach for learning in GP, and its detailed study under different EC population dynamics.
- A thorough comparison between different GP operators for evolving GP based AEs.
- Finally, we compared the proposed GP-AE against conventional, ANN-based autoencoder.

Chapter 6. Fractal Genetic Programming

In this chapter, we introduce a new DL framework based on GP. In the proposed model, artificial neurons are replaced by GP abstract syntax trees, and genetic operations along with evolutionary search perform the optimization task typically carried out by stochastic gradient descend algorithms' family in DNNs. The proposed framework is *deep* in the strictest sense: models generated are composed by multiple non-linear processing layers, and perform both feature extraction and prediction stages, in a single, unified, pipeline. Results show that, although not yet competitive with the latest state-of-the-art ANN-based deep models, it does perform comparably to modern GP approaches that require human expert knowledge of the problem domain. However, what we consider the two most important results gathered from these experiments are: (a) evidence that the GP paradigm could circumvent the fundamental DL problem all along, i.e. that GP in itself can generate *complex*, multilayer, sequential data processing, structures; although in a fundamentally different way to that of *backpropagation*; and (b) closer inspection to models generated by the proposed framework indicates new research directions, as well as some reasons on why our approaches so far attempted for a EC-based DL framework might be flawed (these points will be discussed in the final chapter of this thesis document, see. Ch. 7). We call this framework, *Fractal Genetic Programming* (FractalGP).

We also introduce a new form of incremental learning, we call *spatially distributed learning* (SDL). This type of evolutionary training is a requirement for the FractalGP algorithm, because otherwise it is too time consuming.

This chapter is divided in 7 sections. Section 6.1 presents the context under which the FractalGP framework is proposed. In section 6.2, we introduce in detail the FractalGP method, while in Sec. 6.3 we propose a "convolutional" variant of FractalGP. Next, in Section 6.4, SDL method is described. The main motivation that lead to FractalGP development is discussed in Sec. 6.5. Section 6.6 presents an empirical assessment of the proposed approach, and its comparison to modern GP implementations as well as modern deep networks in an image processing task. And finally, Sec. 6.7 discusses some implications of the results found so far.

6.1 Background

DL is, in practial terms, about attempting to train/learn increasingly larger structures. It has been experimentally proved that these larger learning models are capable of solving ML problems where all other methods have failed. One of the possible reasons why those larger models seem to work better than shallow ones might be due to mathematical expressiveness and data transformation capabilities of them over simpler models; i.e., those larger models, such as DNNs, are capable of expressing transformations that are needed in order to map each input to its corresponding output more accurately. Another possible reason behind the success of models such as DNNs, are the number of degrees of freedom offered by larger structures. A parametric deep learning model has a large number of parameters that need to be found, but this could also be translated into a fine tuning capability not present in models with far fewer number of weight parameters.

These are, however, hypothesis on the superiority shown by larger models over smaller ones; other possible strong reasons are proposed and discussed in (Veit *et al.*, 2016; Frankle & Carbin, 2019). One way or another, superior performances offered by larger models should not be understated. It should also be stressed out how ANNs' researchers struggled for a great period of time attempting to came out with methods and mechanisms that allow them to train such deep networks (see Ch. 2)

Here, we present a model that allows the GP framework to evolve considerably larger learning structures than it is normally possible with the canonical solution representation. Not much work has been done regarding evolution of large tree structures in GP. On the contrary, an undesirable phenomenon that commonly occurs in GP is that tree structures grow significantly in size, but without attaining significant gains in their performance; such phenomenon is known as *bloating* (Banzhaf & Langdon, 2002). Since large structures are computationally more expensive, not only in memory footprint, but also -and more importantly- in evaluation time, bloating is an unwanted effect, and a lot of research has been carried out for understanding and mitigating bloating (Langdon & Poli, 1998; Bleuler *et al.*, 2001; Luke & Panait, 2006). Nevertheless we consider that, as the area of DL has proved, some problems require complex solutions that cannot be expressed through shallow computational models.

Some proposed mechanisms that have been explicit or implicit attempts to generate more complex structures through GP are: automatic subroutine discovery (Poli *et al.*, 2008), cooperative co-evolutionary algorithms (Potter & De Jong, 1994), and ensemble learning (Dietterich, 2000). From all these, subroutine discovery (introduced in Sec. 3.3.3) is the most related to the method proposed in this chapter; their similarities and differences will be highlighted at the end of the next section.

6.2 FractalGP

FractalGP is a new extension for the GP framework that we propose in this thesis, and it is a contribution to the overall field of evolutionary ML. In FractalGP, individuals are represented by abstract syntax trees, just

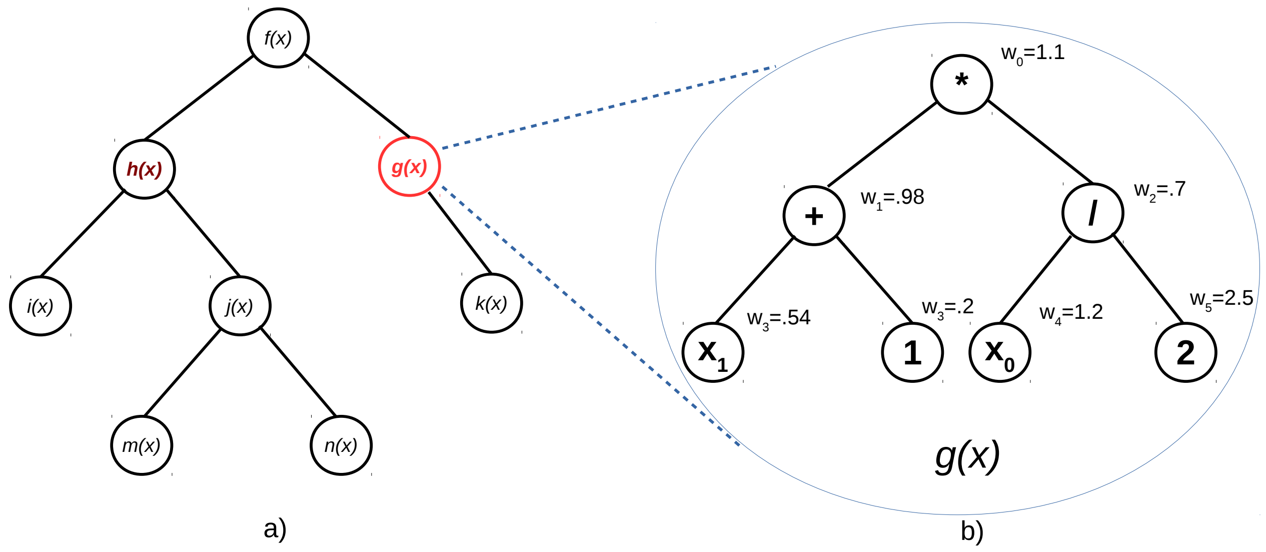


Figure 6.1: Examples of (a) FractalGP individual, and (b) inside-tree in one of its nodes. In a FractalGP tree, each node is a tree in itself, instead of a simple primitive. Each node in the inside-tree has a coefficient that can be tuned in order to enhance its functionality. Notice how Fractal GP greatly increments the real total depth of a GP individual.

as in standard GP, but with one major difference: internal nodes do not take the shape of a function selected from a predefined set of primitives, instead the function they express is defined by an abstract syntax tree that is built using the standard set of primitives one would use in GP. That is, each node in a FractalGP tree is in itself a GP tree. Fig. 6.1 shows a FractalGP tree and the inside of one of its nodes. From now on, we will refer to the trees inside the nodes of FractalGP trees as *molecular primitives* or inside-trees, and to the set of primitives from which they are assembled as, *atomic primitives*.

Trees inside FractalGP trees nodes, i.e. molecular primitives, are just like standard GP trees with one addition: each node in a molecular primitive consists of an atomic primitive as well as a *coefficient* that modifies the node's output; so, in formal terms: let us say that tree t is a molecular primitive; t represents $f_t(x)$ function; each u node, whether internal or leaf, that belongs to t, expresses a f_u function, such that when u is a leaf node, $f_u(x) = w_u \cdot x_i$ or $f_u(x) = w_u \cdot c$, where x_i is a variable taken from x and c is a constant value[1], and when u is an internal node $f_u(z) = w_u \cdot p(z)$, where p is an atomic primitive and z is u children nodes' output. This form of representation where each node in a tree has associated a multiplicative factor is not new and has been used in the past when GP has been exported to the context of *memetic* algorithms (Emigdio *et al.*, 2014); FractalGP is, however, not a memetic algorithm. We will explain this in the following subsection.

For example, the function represented by the inside-tree depicted in Fig. 6.1b is

[1]Note that if t is a leaf node in the FractalGP tree, then x is the input representation vector, otherwise is a vector generated from t children nodes outputs

$$g(x) = 1.1\left(.98(.54x_1 + .2(1)) \times .7\left(\frac{1.2x_0}{2.5(2)}\right)\right) \tag{5}$$

The original purpose of FractalGP was to extend the GP model to allow it to use more complex functions as primitives, instead of just very simple arithmetic or trigonometric operations, but at the same time, without the need to define such functions manually while retaining the ability to *polish* them in some degree. However, notice how the proposed representation carries another benefit: it promotes subroutine finding. This happens because if a FractalGP tree undergoes crossover with itself, integral pieces of code replicate across the tree structure.

6.2.1 FractalGP trees creation

The process of creating an initial population in FractalGP is pretty much the same as it would be in a traditional GP, only requiring additional attention to a few new details. There are two new hyperparameters in FractalGP: *max allowed depth* and *max allowed arity* in molecular primitives.

FractalGP trees are created recursively from the root. Any method such as random depth or full depth may be used. When the root is created, or any other internal node, instead of selecting a primitive for the node as in traditional GP, we randomly generate a syntax tree (the molecular primitive) using atomic primitives and the max allowed arity (as well as random constants within some range) as building blocks. The method to create this tree can also be random or full depth or any other variant of the kind. Once a tree is created, it has to be parsed, to count how many different leaf variables appeared within. This value will determine how many children a node will spring, and it is restricted by the max allowed arity parameter.

When a leaf node of an inner-tree in a FractalGP tree is generated (either because max depth has been reached or because it has been randomly chosen to), the max arity parameter is not used. Instead, molecular primitive's leaves may be any original feature variables of the targeted problem (or constants within some range). Coefficients for all nodes in all inner-trees within a FractalGP tree are initialized to $w_u = 1, \forall u$.

6.2.2 FractalGP Operators

Available genetic operators for FractalGP trees are structural crossover, structural mutation and noise mutation, which will be described below.

Structural crossover or simply crossover, performs GP's conventional crossover operation at FractalGP trees level; i.e. it does not alter in any way insides of molecular primitives. This operations treats FractalGP trees's nodes as black boxes, and just swaps parts of FractalGP trees structures using as a crossover point some node picked at random. Fig. 6.2 illustrates this FractalGP operation.

Structural mutation is a type of mutation that is technically the same as the original mutation defined for standard GP individuals. It works by replacing a subtree from a FractalGP tree with a new, randomly generated, one; the process of generating the new subtree follows the same steps as when generating a complete FractalGP tree (Sec. 6.2.1), but with a reduced max allowed depth, so that the new subtree can be attached to the original tree at the mutation point without breaking the max allowed depth limit. Fig. 6.3 shows an example of this type of operation.

Noise mutation is a new kind of operator defined for FractalGP trees. When a FractalGP tree is selected to undergo noise mutation, a single offspring is derived from the original tree where all coefficients of inner-trees' nodes have been perturbed by adding a small random value generated according to some probability distribution centered around zero, such as Gaussian distribution, $\mathcal{N}(\mu, \sigma^2)$, such that $\mu = 0$. It is important to remark that the proposed FractalGP framework is not a memetic algorithm, because even though inner-trees representation may resemble memetic GP variants, noise mutation does not imply that individuals enter a local or global search algorithm where optimal values in nodes' coefficients are attempted to be found; noise mutation is a single step genetic operation just as crossover or regular mutation. Nevertheless, FractalGP may partially behave as a memetic algorithm during late generations of an evolutionary run, because once population diversity has been exhausted, and one or two individuals repeat throughout the population, applying noise mutation repeatedly becomes a local search. Fig. 6.4 shows a case of how noise mutation modifies the coefficients of an inside-tree.

Notice how noise mutation purpose is to *polish* molecular primitives that were randomly generated when the initial population was created. Then, two original genetic operations can be used as in standard GP, with the difference that primitives composing trees are potentially more complex functions than just atomic primitives.

6.2.3 Relationship to ADFs

FractalGP can be considered a modified version of the extreme ADFs case proposed by Koza (1994) (see chapters 21 through 25). Koza proposed a scenario where all primitives and terminals are replaced by ADFs. The total number of ADFs in this setup is also undetermined. In this extreme case, crossover becomes problematic, and a number of complex rules have to be observed in order to perform crossover (Koza did not established mutation requirements). The FractalGP framework proposes new genetic operations that are

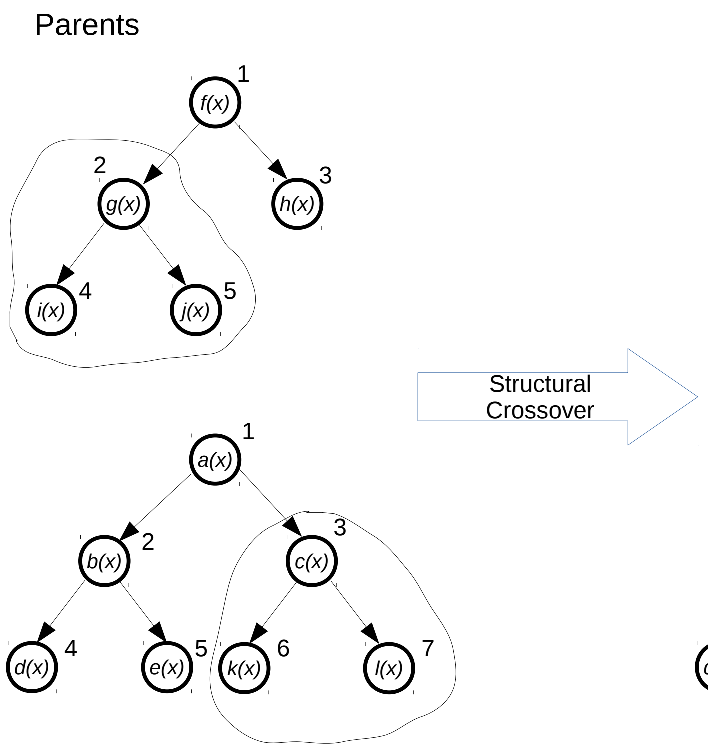


Figure 6.2: Example of Structural Crossover operation defined for FractalGP individuals. Notice that it is the standard GP crossover operation. Molecular primitives inside the nodes of the trees are not altered in anyway.

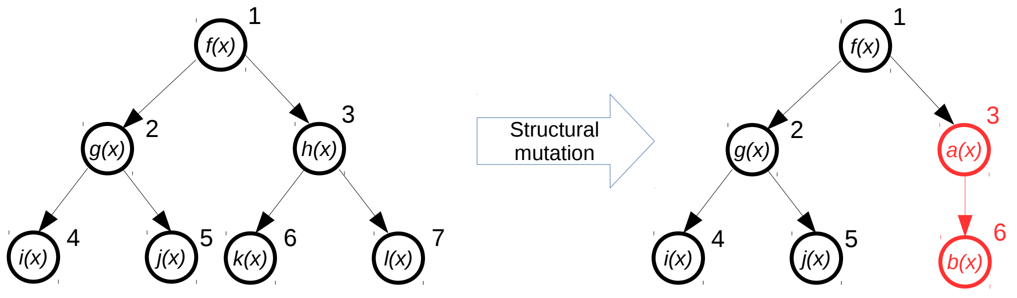


Figure 6.3: Example of Structural Mutation; a subtree in the offspring is replaced by a new, randomly generated, molecular primitives. It is fundamentally the same operation as standard GP subtree mutation.

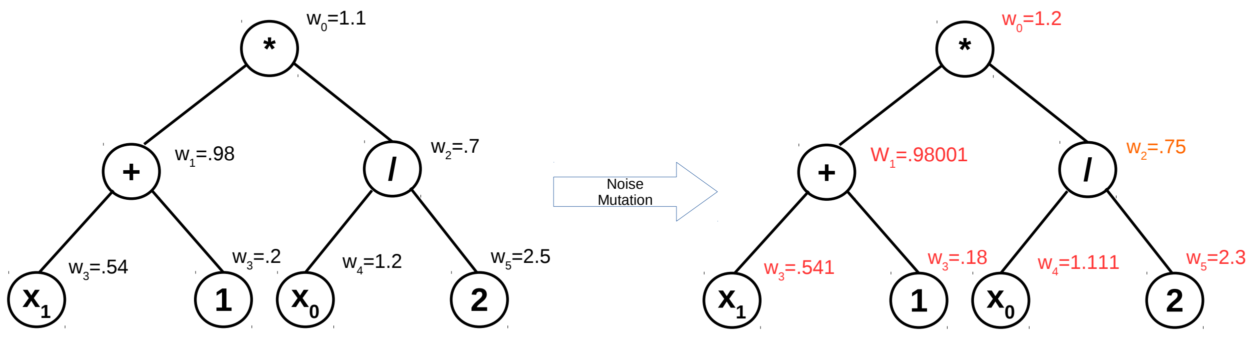


Figure 6.4: Example of the effect of noise mutation inside a molecular primitive. Noise mutation actually alters all molecular primitives in a FractalGP individual. Notice it does not alter the structure of the syntax tree.

more lax in requirements than ADF's crossover. This new operators take advantage of a memetic GP syntax trees variant and recent results from GA-derived optimization techniques for deep networks.

6.3 Convolutional FractalGP

Convolutional FractalGP (CFGP) is a variant of FractalGP to specifically target image processing tasks. In CFGP, nodes in a FractalGP tree receive as input an array, this array is processed by a node (a GP tree or molecular primitive) in a "convolutional" fashion, i.e. a node function is slid through an array, and generates as output another array that results from this convolution-alike operation.

When nodes are leaves, they receive as input an image array to be processed, and as an output a *feature map* array is returned. Feature maps are intermediate transformations of an input representation that are stepping stones for achieving a desired output. Those feature maps can be stacked, forming 3D arrays that serve as input to internal nodes in a CFGP tree. Internal nodes also output a single feature map, for further processing in upper layers of a CFGP tree. In applications where a desired output is a complete image, such as filtering, segmentation or inpainting, an output at root node in a CFGP tree can be used as final result, without having to undergo any further processing; Fig. 6.5 shows this type of architecture. In the case where a desired output is not an image, such in classification tasks, an output at root node in a CFGP tree should be used as input representation for an extra layer of processing, similar to the role that a fully connected MLP layers play at the endpoint of deep convolutional networks[2].

6.3.1 CFGP trees creation

One additional detail has to be observed when creating CFGP trees in comparison to creating regular FractalGP trees. In FractalGP, when a new internal node is created during a tree creation process, an inside-tree has to be parsed to find out how many children such node will spring. This process is straightforward; it simply consists in counting how many different input variables appear in leaf nodes of a molecular primitive (which is limited by the max arity parameter). In CFGP this process is similar but with a caveat.

First, CFGP has an extra configuration hyperparameter: the sliding window size that operate over input arrays. This value defines an input variables range that may appear within molecular primitives but that actually refer to a 3D input array with volume $= 1$, i.e. a molecular primitive only requires a single feature map as input, and thus it only has one children node. If input variables in a molecular primitive overflow this

[2]Although this is an abuse of language, because *in Convolutional Nets there is no such thing as "fully-connected" layers* (Yann LeCun comment on his Facebook page).

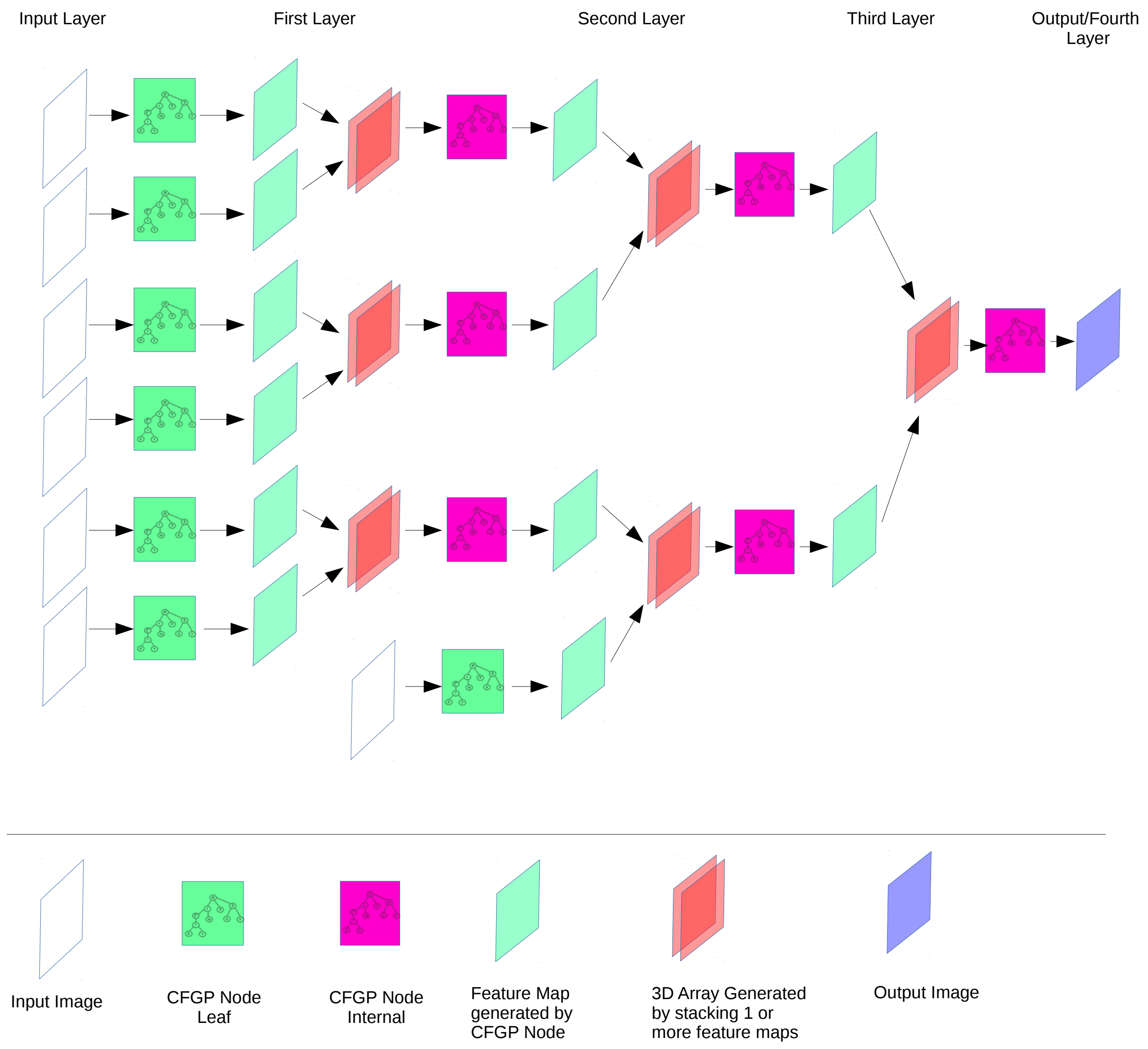


Figure 6.5: Diagram of a CFGP individual. Tree structure is shown horizontally, root node is to the right, leaf nodes are to the left. Each node is a GP tree that operates in a convolutional fashion over its input array; this array may be composed of one or more ”feature maps”, forming a 3D array. In this particular example all internal nodes receive as input 3D arrays composed by 2 feature maps. This individual has 4 processing layers, hence it can be considered an homologous deep learning model.

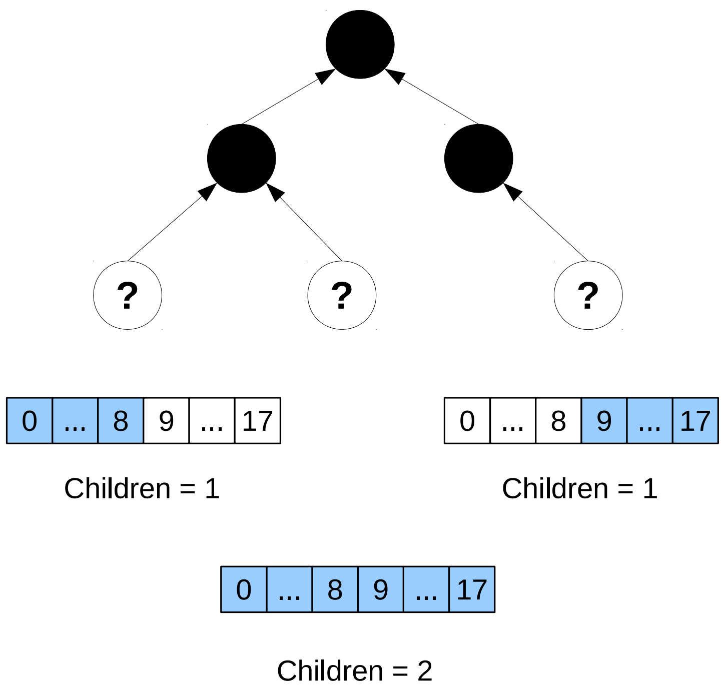


Figure 6.6: Parsing of a CFGP node considering a 3×3 pixels sliding window with maximum arity equals 2. Depending on feature variables at leaf nodes of an inside-tree, a node may have one or two children nodes.

range, then it may or may not, require two feature maps as input, i.e. 3D input array with volume $= 2$; this depends on variables appearing within first range or not.

To clarify this scenario, suppose a CFGP setup with sliding windows of 3×3 pixels, and maximum arity 2. When creating a root's inner-tree, or any other internal node, parsing is required in order to know how many children nodes must be created for that particular node. In this example, there are three possible cases: (1) all inner-tree feature variables fall within $[0, 8]$ range, in such case the node will only spring one children node; (2) all inner-tree feature variables fall within $[9, 17]$ range, then the node will only spring one children node; (3) feature variables fall within entire $[0, 17]$ range, then the node will have two children nodes. Note that feature variables cannot fall beyond $[0, 17]$ range, because maximum arity equals 2; if maximum arity is setup to 3, then range would be extended to $[0, 26]$, and there would be different cases when nodes would spring one, two or three children nodes. Fig. 6.6 illustrates these three possible scenarios.

Those rules do not apply to leaf nodes, simply because they do not spring more nodes, and because their input is fixed to a single 2D array (an input image). Understanding in detail how CFGP trees are created is important because it help us to explain an additional genetic operation specific to CFGP individuals.

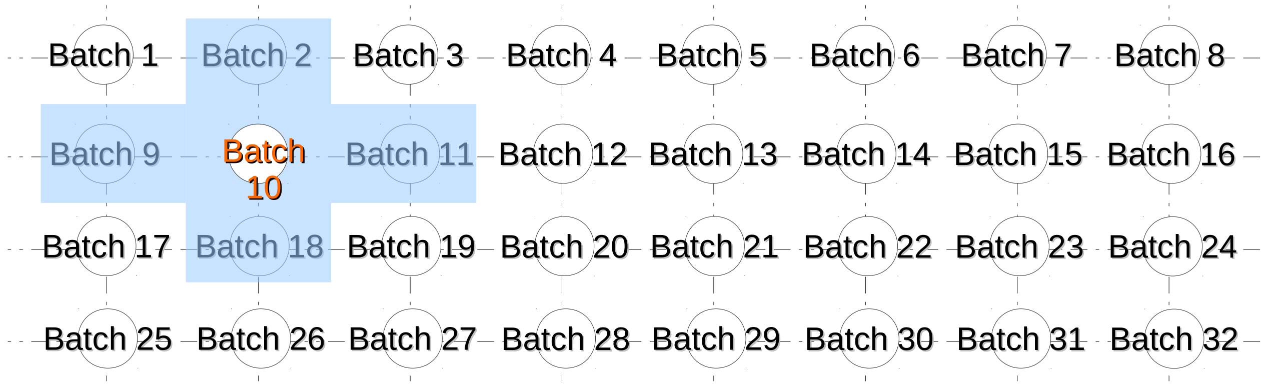


Figure 6.7: SDL example: during cell 10 evaluation, individuals from cells 2, 9, 11 and 18 (cross shape neighbourhood), as well as their offspring, are tested against minibatch 10, and the best performing individual from that micro-population replaces current individual at cell 10.

6.3.2 CFGP Operators

Feature Leaf Mutation is an specific CFGP operation outside FractalGPs regular context. It consists in performing a random shift in every leaf node that is a variable (i.e. not constants) with some probability. This shift should be constrained to the range in which already resides the variable; this is done in this way in order to avoid disconnecting children nodes from their parents; without this prevision it could be the case that a parent node that undergoes feature mutation would end up no longer referencing one of the feature maps it receives as input, which would effectively leave entire subtrees disconnected within individuals.

For example, suppose a CFGP individual configured with sliding windows of 3×3 pixels, and maximum arity equals 2 that generates an offspring through Feature Leaf Mutation. If there is a variable leaf node in one inside-tree set to feature 3 (15), this leaf could take a new value only within $[0,8]$ ($[9,17]$) range in a mutated offspring, otherwise a node would stop using one of its two input features maps, or points to a non existent feature map, if it only has one child.

From our experimental assessment (see Sec. 6.6), CFGP algorithm shows high instability in its performance, with high variance in its results; but even so, in a set of preliminary runs we found that CFGP without feature leaf mutation is severely capped, unable to achieve any acceptable performance in any run. Feature leaf mutation is a requirement for CFGP because it manages to modify and to improve inner-trees in a way that noise mutation (the only other inner-tree altering operation) is unable too.

6.4 Spatially Distributed Learning

SDL is a highly experimental incremental learning scheme that consists in distributing minibatch learning across *space*, instead of time (such as in SGD). SDL is used in combination with a cellular population (see Sec. 3.3.2). Each cell is associated to a different minibatch, and when evaluating such cell, a micro-population assembled with it, its neighboring cells, and the offspring generated with both of them, gets evaluated against the referred minibatch, in order to determine which individual will replace the current cell.

There are three important aspects of SDL to remark: (i) each individual is tested against more than a single minibatch, since it is not only tested against its cell's minibatch, but also against minibatches from neighboring cells, thus good performing and generalizing individuals will propagate across the population; (ii) Under this scheme, each generation roughly corresponds to one epoch, unlike SGD-alike methods where many generations are required to see an entire training dataset, this is because GP as a whole sees all training samples every generation (although the same cannot be said of any individual in the population), thus each generation corresponds to a *pseudo-epoch*; (iii) note that the best individual at the end of an entire GP run will be probably overfit towards some minibatch, and this entices the idea of using the whole population as an ensemble learner; this idea has not been tested.

Fig. 6.7 visually depicts an SDL scheme. Though we have not yet thoroughly tested the SDL method, some of the best solutions found with CFGP were obtained under this learning scheme. Thus, we consider to briefly introduce this scheme even if still is at early development stages. We have used SDL along CFGP only; all other FractalGP instances and other GP variants described in the following section were assessed using standard on-line learning methods.

6.5 Hypothesis

Motivations that led to propose the FractalGP framework are to develop a model that: (1) allows to evolve large GP structures, (2) offers fine-grained tuning degree capability, and to a lesser extent, that (3) promotes subroutine discovery. However, the main idea behind FractalGP is to test the hypothesis that GP can circumvent the credit assignment problem (CAP).

In this section, we state the CAP, its relationship with the deep learning paradigm, and the intuition behind our believe that GP is a viable method to target the CAP.

6.5.1 Credit Assignment Problem

When we develop a *complex* system, how should we distribute the credit obtained from its successes and errors to the multiple parts that compose it? That is, how do we individually evaluate the parts that compose a system when we can only directly quantify the global output generated by the system? Or, in more practical terms, how can we know what parts of a complex system we have to improve and what parts we should left intact, in order to improve its overall performance? This is what is known as the *credit assignment problem*.

The CAP was first formalized by Minsky (1961), however its roots can be traced back to the development of Dynamic Programming (Bellman & Kalaba, 1957). Nowadays, the CAP can be considered a fundamental problem in the entire field of computer science, but perhaps specially for areas of Machine Learning and Artificial Intelligence. The CAP can be found in the context of planning (Bellman & Kalaba, 1957), Reinforcement Learning (Sutton & Barto, 1998), and Evolutionary Computation (Potter & De Jong, 1994).

Schmidhuber (2015) considers that the CAP is the fundamental problem in Deep Learning. This analysis considers the backpropagation algorithm as a Dynamic Programming-derived method. In general terms, we share this point of view, and we believe it is not difficult to understand why the CAP defines the Deep Learning essence.

Considering that even the most basic DL architecture must be composed by two interconnected parts: feature extractor and predictor. How can we evaluate each of them, when we only get a measurable output from predictor's end point? If one may try to make modifications to one of the parts, the other may be rendered useless or viceversa, it does not matter how much we attempt to improve one of the parts if the other one is negatively affected. This problem is even further aggravated when we consider a DL system composed by multiple sequential layers of feature extraction stages.

For ANN-based deep networks, the CAP is resolved through the backpropagation algorithm, a calculus-based method that allows to dispatch and distribute the error signal across all layers within a DL model. Backpropagation can be applied to this particular dynamic programming setting because of the very nature of artificial neurons, that is, learning units that represent differentiable mathematical functions. However, this would not suit a DL model based on non-differentiable learning units, such as the one we are herein proposing, or even a case where a new mechanism to train deep models other than backpropagation is envisioned.

Let us suppose a DL architecture similar to CNNs, with the difference that artificial neurons are replaced with GP abstract syntax trees structures. For simplicity let us assume that we wish to tackle an image processing task that does not necessarily require an MLP at the end of the architecture, such as image denoising, segmentation or inpainting. In its simplest form, this architecture could be composed of a single processing layer with a single filter (GP tree) within that is slid across an input image to generate an output. Fig. 6.8a depicts this first instance. In this case, system's evaluation is trivial, because output's error is directly attributed to the single filter tree in the system.

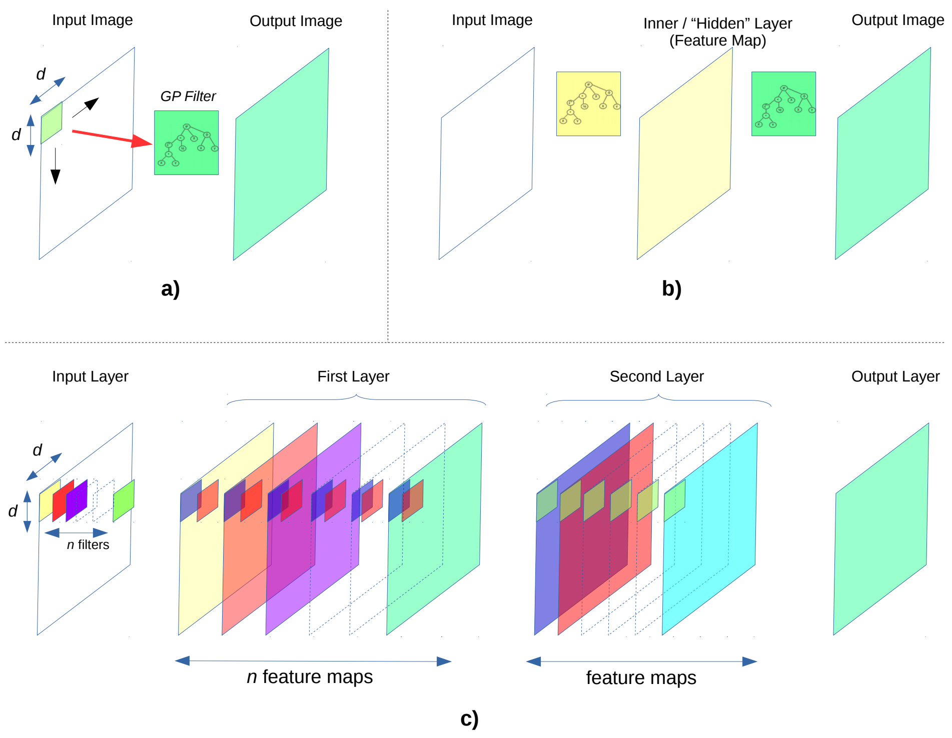


Figure 6.8: Multilayer GP architecture. a) Single layer, single filter; b) Two layer, one filter per layer; c) Three layer, first layer and second with n filters, third layer with only 1, output, filter.

A second variant of this system would be when there are two layers of sequential processing, each one with a single GP tree filter within. Stacking small sliding, i.e. convolution-alike, filters like this has the advantage of being a more efficient way of processing large input spaces than using larger filters or even the whole input (LeCun *et al.*, 1998). But even in this still simple case, evaluating the performance of the system becomes problematic, as there is no obvious way of evaluating the performance of each layer with only the global output as feedback signal. This model is drawn in Fig. 6.8b.

The general case to this hypothetical architecture is when it is composed of several layers, and in each layer there are several GP tree filters, such as in standard CNNs. Fig. 6.8c shows this Multilayer Convolutional GP architecture. In Rodriguez-Coayahuitl *et al.* (2019a), we implemented these types of GP models and called them Convolutional GP. We proposed different mechanisms that attempted to train multilayer variants without success: none of the models composed of two layer or more could outperform the model composed of a single-layer, single filter. The methods proposed in Rodriguez-Coayahuitl *et al.* (2019a) are presented in Appendix A.

6.5.2 Tackling the CAP through GP

In the area of EAs, the CAP has emerged when researchers have attempted to divide complex problems into smaller subproblems, but the evolved subproblems' solutions cannot be evaluated separately (unlike the SLGP Autoencoder proposed in Ch. 5). For those cases, a framework called *Cooperative Co-evolutionary algorithms* (Potter & De Jong, 1994) has been proposed. In cooperative co-evolutionary algorithms, each component is evolved in a separate 'island', and each island imports members (usually only the best) from the rest of the islands so it can complete the processing chain and evaluate each individual. Cooperative Co-evolutionary algorithms were originally proposed for GAs in order to tackle large optimization problems, however this framework has also been used in GP (Krawiec & Bhanu, 2003; Doucette *et al.*, 2012).

However, we hold that GP alone represents a mechanism to tackle the CAP. Notice how GP generated solutions consist in multilayer structures that process data sequentially; if any element in these processing chains is not performing adequately, the entire structure would perform poorly. Take for example the individual evolved for image denoising shown in Fig. 6.9. This GP tree is composed of up to eight processing nodes in sequence in some of its subtrees. Although admittedly each of these processing nodes are much more simpler in nature than a tunable- artificial neuron, the question then arises, could GP be used to tackle the CAP when nodes represent more complex functions? And, could the GP framework benefit, either in terms of efficacy or efficiency, shall nodes represent more complex functions? These are the questions we attempt to provide some clues through the proposed FractalGP framework.

6.6 Experimental evaluation

This section presents an empirical assessment of FractalGP and CFGP proposed models. First, FractalGP is tested in two artificial regression datasets and compared against a standard GP; then both FractalGP and CFGP are tested in a real life image processing task, and their performance is compared against two commonly used GP and DL variants.

6.6.1 Benchmark datasets

For synthetic benchmark datasets, 1000 training samples were generated, and 200 testing samples using the following functions:

$$f(x) = \sqrt{x} \tag{6}$$

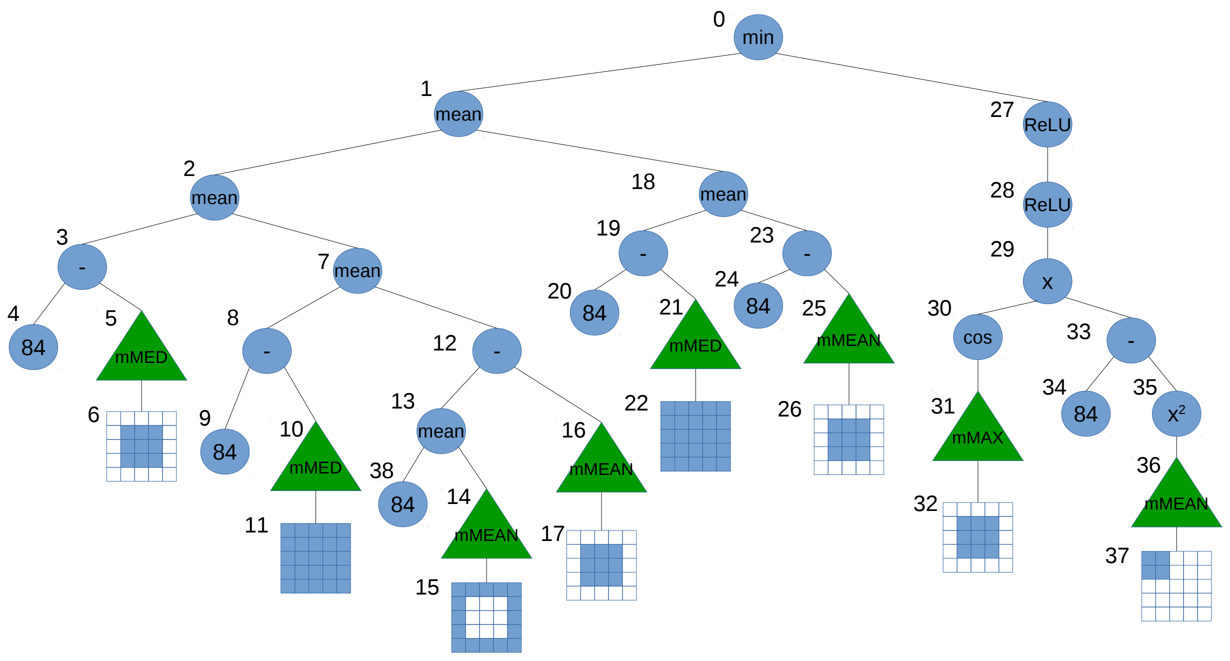


Figure 6.9: Image denoising algorithm synthesized through GP. Mezzanine-type of nodes were used in this evolutionary setup.

and

$$f(x, y) = \frac{\sin(5x(3y + 1)) + 1}{2} \tag{7}$$

Both are non-linear functions. For experiments with both artificial datasets, $\sqrt{x}$, $\sin$ and $\cos$ are removed from the primitives set. Because the experimental purpose is to demonstrate GP's and FractalGP's ability to approximate complex surfaces using as starting point an arbitrary set of simple functions as primitives; consider that in real life scenarios is highly unlikely that the functions we are trying to approximate are already in the primitives set.

On the other hand, for performance testing of both FractalGP and CFGP in a real life scenario, image denoising was selected as the target task. Image denoising consists in extracting a clean image $\mathbf{x}$ from a noisy observation $\mathbf{y}$ such that, for additive noise, $\mathbf{y} = \mathbf{x} + \mathbf{v}$, where $\mathbf{v}$ is a contamination process; a typical example is when $\mathbf{v}$ follows a Gaussian distribution with some given σ, such case is known as Additive White Gaussian Noise (AWGN).

We used the Berkeley Segmentation Dataset (BSDS) (Roth & Black, 2005) for training and testing purposes. We converted 200 images from BSDS to grayscale and extracted 14000 patches of the following sizes: 3×3, 9×9, 13×13, 21×21 pixels. For each patch size, we generated two datasets by contaminating

Figure 6.10: Sample images from the Berkeley Segmentation Dataset

images (before patches extraction) with AWGN with two different noise levels, $\sigma = 25$ and $\sigma = 50$. We used 12000 patches for training and 2000 for testing for each patch size and noise level. Fig. 6.10 show samples images from BSDS.

We chose image denoising because it is a high dimensionality problem, suitable to compare FractalGP and CFGP performance with modern GP variants typically used for image processing and other high dimensionality tasks, as well as current DL networks. These GP variants are described in the following section. Testing different algorithms on different patch sizes allows to observe how they scale to increasingly higher dimensionality problems.

6.6.2 Parameters Setup

For synthetic datasets, we compare traditional GP performance, for now on referred to as *Low-Level GP* (LowGP), configured with two different maximum allowed tree depths, against FractalGP. Table 6.1 resumes most parameters used for all algorithms tested with synthetic datasets.

Table 6.1: Parameters used for experiments carried with synthetic regression datasets.

Parameter	LowGP-8	LowGP-16	FractalGP
PopSize	400		
Generations	Variable / Fixed time (30 min. per run)		
Max Tree Depth	8	16	4
Max Inner-Tree Depth	N/A		4
Max Allowed Arity	N/A		2
Crossover Prob	.50		.25
Mutation Prob	.50		.25
Noise Mutation Prob	N/A		.50
(Atomic) Primitives	$+, -, \times, \div, x^2$, max, min, mean, ReLU		

The set of (atomic) primitives used consists of binary arithmetic operands $+$, $-$, $\times$, $\div$, and binary operations $mean$, max, min, and finally ReLU, which is unary; division is protected meaning that, any division between zero will return zero, which is customary in GP literature (Poli *et al.*, 2008). As already stated in the previous section, typical GP primitives $\sqrt{x}$, $\sin$ and $\cos$ are disabled for this set of experiments.

For image denoising, we tested four algorithms: a Low-Level GP, a *Mid-Level GP* (Mid-GP), a FractalGP and a CFGP. *Low-level* and *mid-level* GPs nomenclature comes from the fact that these implementations are configured to use low-level only, and mezzanine along low level primitives (see Sec. 3.2) respectively. For example, Mid-GP is capable of generating individuals like the one shown in Fig. 6.9, i.e. individuals with mezzanine type of primitives that operate over features arrays, whereas Low-Level GP is only capable of evolving individuals like one shown in Fig. 6.11, i.e. GP trees that can operate only with individual features at their leaf nodes.

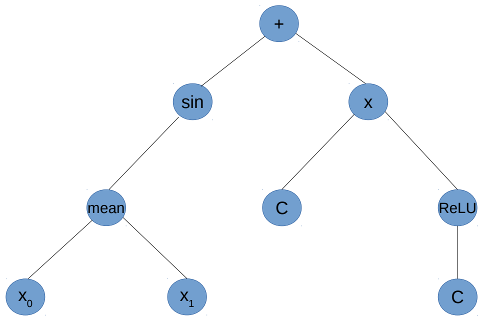


Figure 6.11: ”Low Level” GP individual. Leaf nodes consist only of single feature variables or scalar constants. It can be considered as a ”canonical” GP candidate solution representation.

For Low-Level GP, Mid-Level GP, and FractalGP variants, a square cellular population scheme with 20 $\times$ 20 individuals is used. Table 6.2 resumes all parameters used for these three variants. Crossover and mutation are protected so max allowed tree depth is never exceeded. Max, min and mean listed in low-level functions category are two-arity functions that operate over two single scalar variables or constants, whereas mMean, mMax, etc., are mezzanine functions that operate over a trimmer ’s output (see below). A limited amount of time (24 hrs.) is determined for all setups execution, given the same computational resources (a single Intel Xeon thread at 2.9 GHz), instead of setting a predefined number of generations.

Trimmers serve as leaf nodes for mezzanine nodes. Their purpose is to limit the amount of information, so mezzanine functions do not always operate over the same data (a complete image patch). There are seven types of trimmers defined: Center (takes the center region of 3$\times$3 pixels of a patch), ring (takes first and last columns and rows from a patch), corners (takes one of the four patches’ corners, about $\frac{1}{4}$ of the patch), and full (passes the full patch). Fig. 6.12 illustrates trimmers. Corners and full trimmers adapt to the patch size; the rest of the trimmers remain static for all patch sizes.

Low-GP, Mid-GP and FractalGP are configured to deal with denoising as a simple regression problem: the objective function is to minimize the average mean square error (MSE), from attempting to clean the

Table 6.2: Parameters used for image denoising experiments.

Parameter	**Low-GP**	**Mid-GP**	**FractalGP**	**CFGP**
Pop Size	400			
Generations	Variable / Fixed time (24 hrs. per run)			
Max Tree Depth	8			4
Max Inner-Tree Depth	N/A		5	
Max Allowed Arity	N/A		2	
Crossover Pr.	.5		.25	.25
Mutation Pr.	.5		.25	.25
Noise Mutation Pr.	N/A		.50	.25
Shift Mutation Pr.	N/A			.25
Low-Level	$+, -, \times, \div, sin, cos, \sqrt{}$, max, min, mean, ReLU			
Mezzanine	N/A	mMean, mMax, mMin, mMed	N/A	
Type-1 zero-arg	Individual pixels			
Type-2 zero-arg	Constants within range [-1,1]			
Type-3 zero-arg	N/A	Trimmers	N/A	

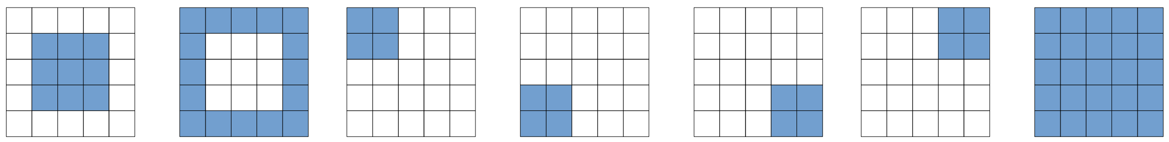


Figure 6.12: Trimmers. From left to right: center, ring, NW, SW, SE, NE, Full.

central pixel from all patches in a training set. In order to clean a complete image, any generated model is slid through the full image in a convolutional fashion to clean it, but for the first round of denoising experiments, we limited ourselves to test generated models in the testing set of 2000 patches, cleaning only the central pixel. The best model that resulted from those experiments is then compared against CFGP, which is trained in a standard convolutional fashion by cleaning complete patches.

CFGP is also trained using different evolutionary dynamics than the rest of the GP variants. CFGP uses SDL (see Sec. 6.4), whereas all other GP approaches use standard on-line learning. Moreover, CFGP also uses a special scheme called centric selection (Simoncini *et al.*, 2009), which consists in enforcing, with a decaying probability, either noise or feature mutation; this allows to *polish* molecular primitives during first generations of an execution, before standard GP operations are applied.

Table 6.3: Results from different GP approaches using synthetic images. Results expressed in MSE. Lower is better. All setups were run for the same amount of time, given the same computational resources.

	LowGP-8			LowGP-16			FractalGP		
	avg	best	worst	avg	best	worst	avg	best	worst
SQRT	8.690 ± 6.434	1.313	24.04	39.149 ± 30.731	8.339	95.31	1.459 ± 0.731	0.341	2.44
SINE	0.079 ± 0.014310	0.061	0.10	0.109 ± 0.013	0.078	0.12	0.079 ± 0.006	0.065	0.087

6.6.3 Results

Table 6.3 shows average, standard deviation and best results from 10 executions per approach of LowGP-8, LowGP-16 and FractalGP using synthetic benchmarks. The purpose of this set of experiments is to test if the FractalGP scheme is capable to capitalize in a more efficient way larger tree individuals than a standard GP representation. Therefore, the idea is to compare FractalGP against a traditional GP configured to an optimal max tree depth (LowGP-8) and against a traditional GP configured to a max tree depth equivalent to the total depth of FractalGP (LowGP-16). Results show that FractalGP is indeed a superior approach. In the case of the square root function, FractalGP manages to reduce the error to a lower level than a standard GP configured to an optimal max tree depth, meanwhile the allocation of very large (*deep*) trees structures cannot be properly exploited by standard GP representation/algorithm. For the second dataset, FractalGP and the optimal standard GP yield the same average result, but variance for FractalGP is lower, making it a more reliable method, while a standard GP approach configured with a conservative max allowed depth still manages to find better results, meaning that given enough trial runs, it is a better approach.

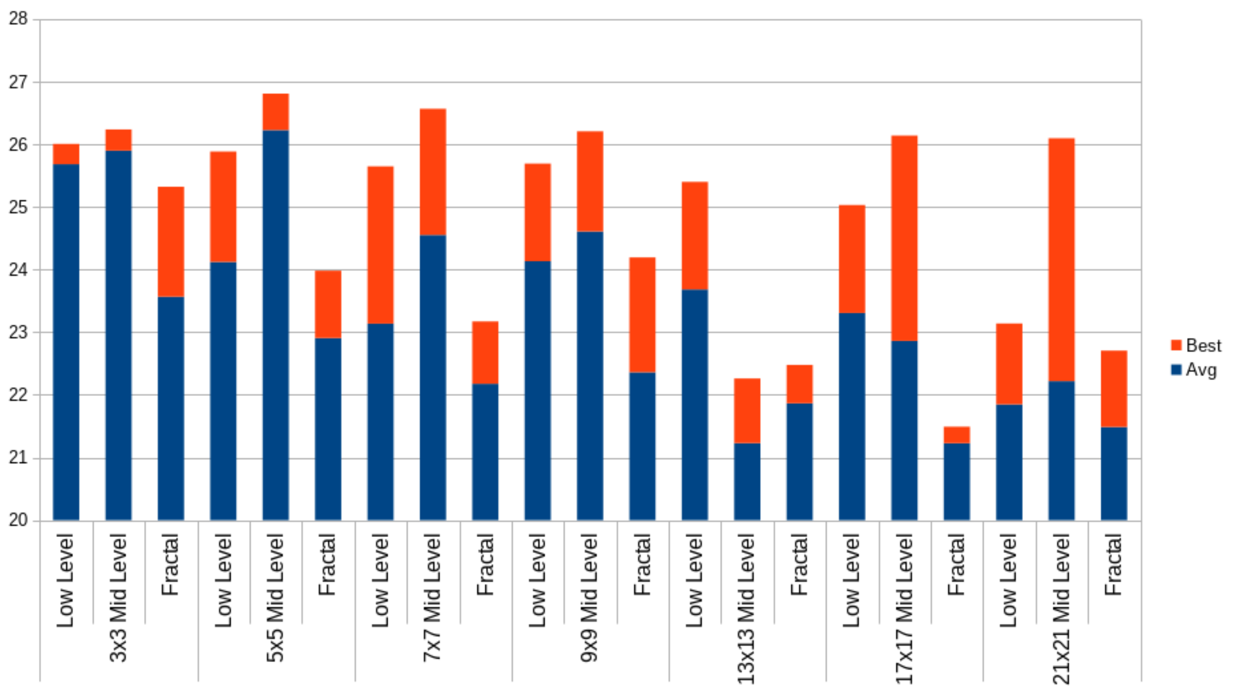


Figure 6.13: Results obtained from LowGP, MidGP and FractalGP approaches, for image denoising with noise level $\sigma = 25$, for different patch sizes. Results expressed in dB; higher is better.

Figures 6.13 and 6.14 show the average and best performances obtained by LowGP, MidGP and FractalGP variants when targeting image denoising, for two noise levels and different patch sizes (perceptive field). Results are expressed in decibels (dB) peak signal to noise ratio (PSNR). Fig. 6.9 is an actual depiction of the best individual found with MidGP configuration with training 13×13 patches, which is also the

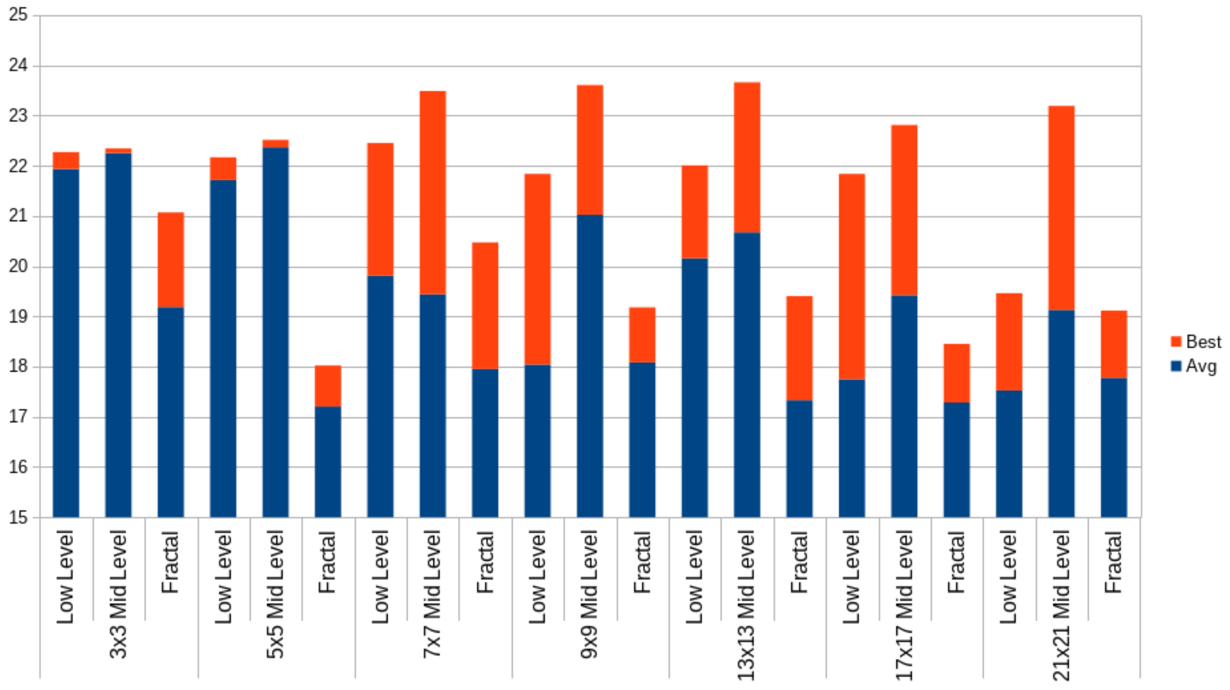


Figure 6.14: Results obtained from LowGP, MidGP and FractalGP approaches, for image denoising with noise level $\sigma = 50$, for different patch sizes. Results expressed in dB; higher is better.

best performer across all the board, for all patch sizes and GP variants. In contrast, the solution shown in Fig. 6.15, which is the best solution found with LowGP variant; this Low level solution was obtained with 7×7 pixels patches.

The purpose of this set of experiments is to test how different algorithmic approaches behave as dimensionality of the problem increases, which would require (presumably) larger candidate solutions (*deeper* trees). Notice how LowGP average and best performances decrease significantly as dimensionality increases, whereas MidGP best performances keep steady even for large patch sizes. MidGP model disadvantage is that system's designer has to define mezzanine functions as well as trimmers; in contrast, a LowGP is a more agnostic machine learning algorithm, which is more in tone with DL concept.

Therefore, even though LowGP is a more desirable approach, these results confirm the status of mezzanine-based models as an standard GP methodology for high dimensionality problems. The proposed FractalGP model is an attempt to close the gap between these two classical GP approaches (note that FractalGP is fundamentally a Low level GP approach, since no mezzanine functions are defined for it). Unfortunately, FractalGP does not show the same competitive performance shown with synthetic datasets when target in image denoising; results suggest that FractalGP requires enhancements to tackle high dimensionality problems.

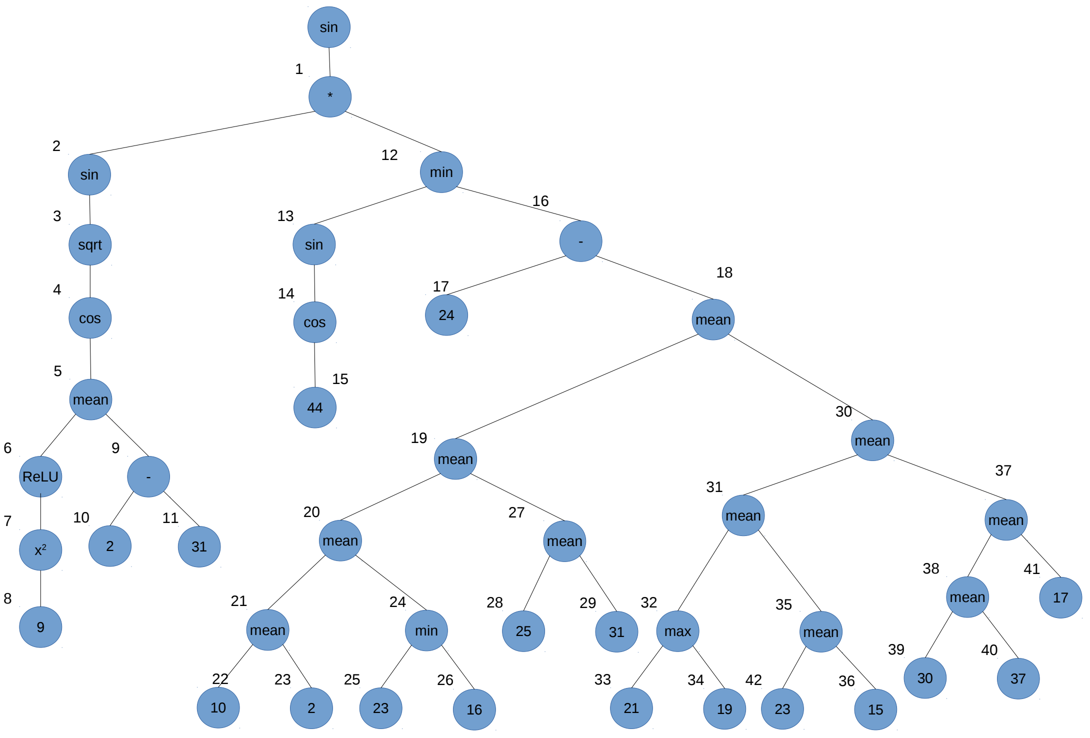


Figure 6.15: GP solution for image denoising using only individual pixels and low-level nodes as available primitives

Table 6.4: Average results obtained by LowGP, MidGP, CFGP and CNN implementations, from 10 independent testing runs, for noise level of 50. Results are expressed in dB; higher is better.

	avg	best	worst
LowGP	17.81 ± 2.31	20.94	15.97
CFGP	19.30 ± 1.12	22.18	18.20
MidGP	20.67 ± 3.32	23.66	16.99
DNN-6	23.46 ± 0.11	23.56	23.27
DNN-32	24.40 ± 0.05	24.43	24.31

Next, we compared MidGP (current top performer) versus CFGP (previous empirical assessment top performer). Table 6.4 shows both approaches for $\sigma = 50$ noise level; MidGP was trained using 13×13 patches, while CFGP model was generated with a maximum tree depth of 4, which is equivalent to have trained it with 11×11 image patches. Results show that, in general, CFGP performance is worst than MidGP that tends to find either very good or very bad solutions with more or less equal probability (hence its high variance), whereas CFGP converges to non-competitive solutions (not too bad, but not acceptable either) most of the time, while in rare occasions converges to good solutions.

We also compare the performance of the CFGP against that of a LowGP and two deep networks. We perform the comparison against a LowGP with an equivalent number of maximum allowed of nodes to that of the CFGP. Both methods can be considered agnostic, i.e. without any human expert knowledge embedded

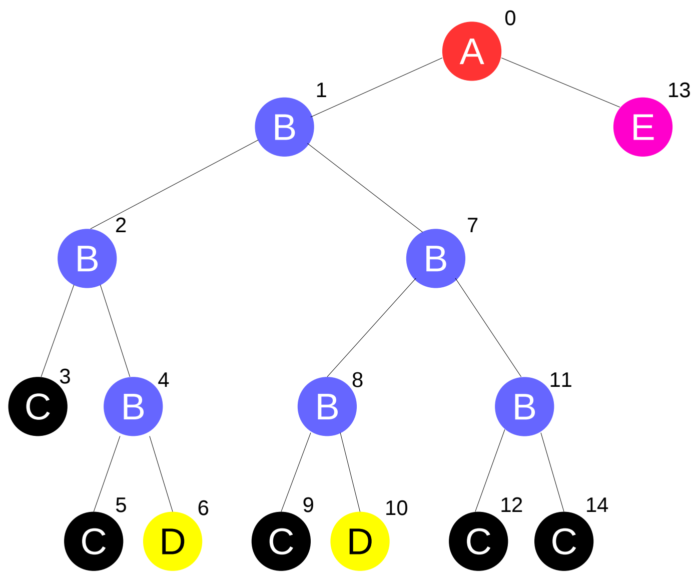


Figure 6.16: Best CFGP solution found. All nodes are "convolutional". This is fundamentally a *deep* model, composed by 5 stacked convolutional processing layers.

(unlike the MidGP), so the purpose is to compare if CFGP can enhance agnostic GP implementations, even though its performance may not be comparable to that of the MidGP. Results show that the CFGP obtains better average performance, as well as better best and worst results than a LowGP.

On the other hand, we also implemented two deep networks for image denoising based on the network proposed by Zhang *et al.* (2017). Zhang *et al.* (2017) proposed a 17-layer CNN for image denoising; each layer is composed of 64 3×3 convolutional filters with ReLU functions, except for the last, output, layer which consists of a single 3×3 convolutional filter without any activation function. We implemented two variant of this network, where we only used a 5 layer architecture (which is equivalent to the number of layers allowed to the CFGP); for one variant (DNN-32) we used 32 convolutional filters per layer, whereas for other variant (DNN-6) we only used six filters per layer. The number of layers of the DNN-6 is configured such that the total number of parameters of the networks is roughly equivalent to that of the CFGP setup tested, in order to perform a parameter-wise comparison. Results shows that in both cases the CNNs outperform all of the GP approaches tested, in all regards: average, best, worst and variance, thus making GP an ineffective method to compete against deep networks at this class of problems.

A closer inspection to the best CFGP individual obtained revealed an interesting pattern, and shed light on some possible reasons behind underwhelming performance of CFGP. These issues will be discussed in the following section. Fig 6.16, depicts the overall architecture of top CFGP performer found, and Figures 6.17 and 6.18 show its molecular primitives.

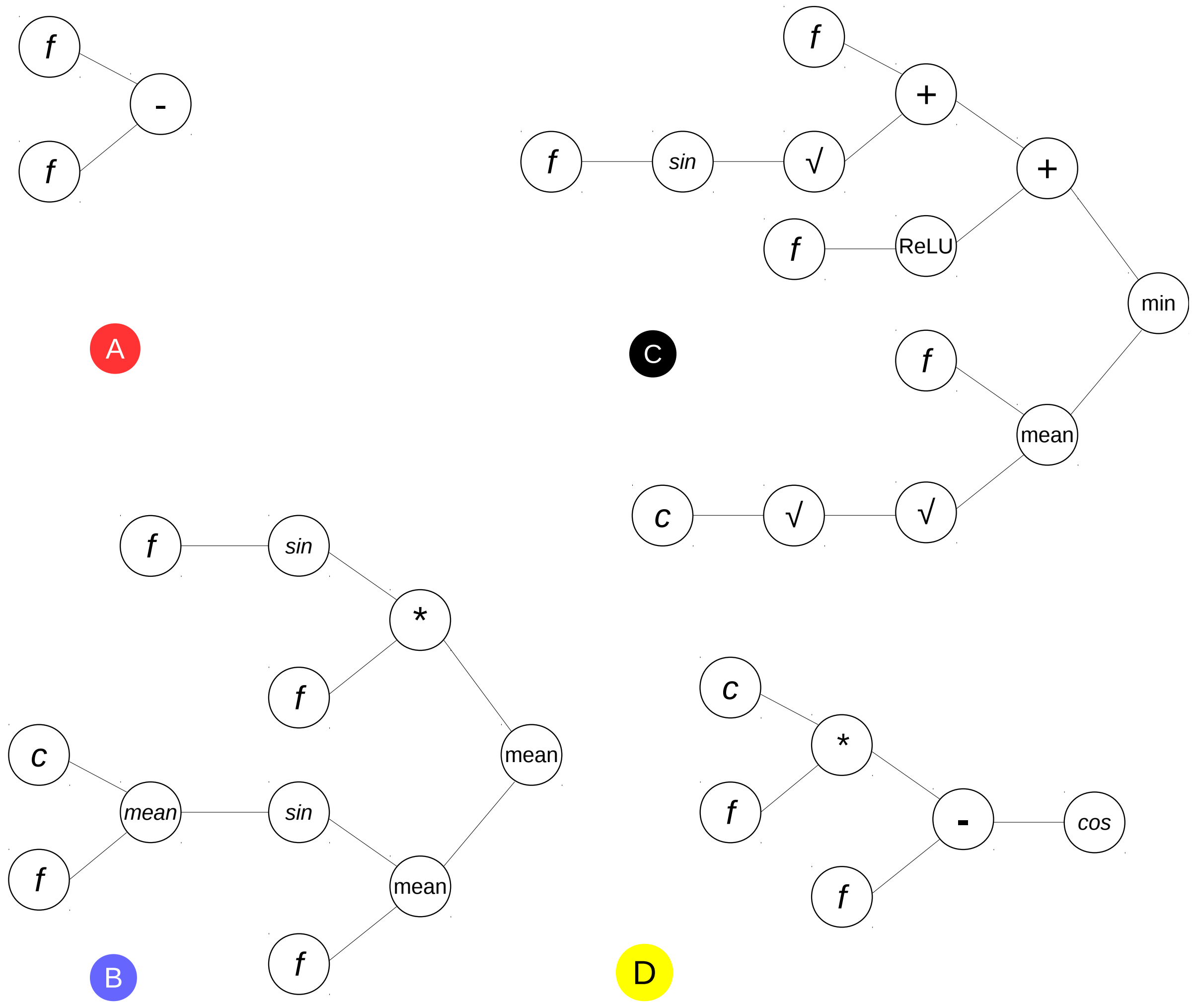


Figure 6.17: Molecular primitives of solution depicted in Fig. 6.16.

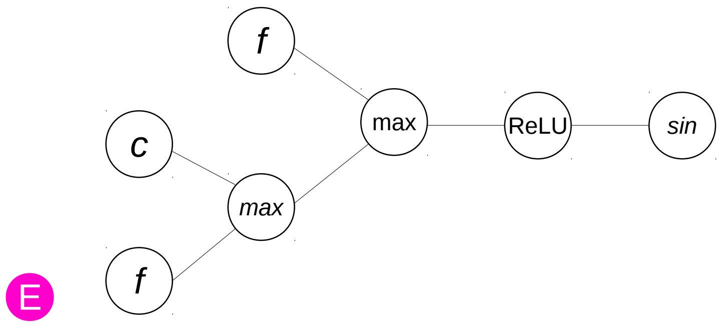


Figure 6.18: Molecular primitives of solution drawn in Fig. 6.16. Nodes denoted with c stand for constants, while f are individual feature variables taken from windows that slide over input feature maps.

6.7 Analysis

In this final section of this chapter, we will discuss some phenomena that can be observed in the best solution found by the CFGP, depicted in detail in Figures 6.16 through 6.18; this analysis will also allow us to pitch some hypotheses regarding the highly unreliable performance of CFGP, and some possible ways to improve it.

In Fig. 6.16 we can notice that the individual is composed, in broad terms, by three classes of nodes: one main type of internal node (B), and two main types of leaf nodes (C and D); there are, however, one internal node (A, the root) and one leaf node (E, a child of the root) that do not belong to the node's main classes. At this point it is important to remember that these molecular primitives, even repeated across the entire main structure, actually have different coefficients assigned in each of their internal nodes, and their leaves may also correspond to different input variables, so they are not fully identical. However, the fact that this solution exhibits a highly modularized nature, suggests that GP is attempting to find sort of artificial neurons, i.e. basic learning units that can replicate across the structure, and tune them later (through noise and feature mutation) in order to improve the overall performance.

From one perspective, this can be considered a significant result because it seems to suggest that GP's artificial evolutionary processes tend towards the very same results than that of natural evolution: finding the nerve cell once, and then build large information processing structures by combining a number of these cells; this results also mimics foundations on which connectionist ML field is built.

From another point of view, these results could also be expected for two main reasons: first, modularity is, perhaps, the ”shortest path” to came up with sophisticated information processing systems, i.e. it is probably simpler to synthesize a ”brain” using very simple units as building blocks, repeated across an entire structure, rather than generating a very heterogeneous system as a whole (this is also probably why nature reached the same type of solution); secondly, and maybe even more importantly, notice that CFGP is actually a method that *imports* some intuition and logic from one heuristic (ANNs are an heuristic in the full sense) into another heuristic of a completely different nature (Genetic Programming). CFGP employs a convolution-alike operation and a layered, sequentially processing, pipeline, both very intrinsic to ANNs nature, so it is only natural to expect that EC/GP fills the gaps by finding the missing pieces of the jigsaw puzzle, i.e., artificial neurons.

This could also explain the reasons behind low CFGP performance: it requires that neuron-type molecular primitives appear in the initial population so it can replicate and propagate them (remember that any genetic operation defined for CFGP can modify molecular primitives at a structural level); and not only that, but it also requires these neuron-alike nodes appear in three different nodes classes: leaf, internal and root nodes. This last issue is important to clarify further: notice how internal nodes can never become leaf nodes, so nodes prone to work as basic learning units are required to appear both as internal and as leaf nodes; also, internal nodes cannot become the root (there is no genetic operation that ”prunes” a GP tree from atop), thus the root node is either also a neuron-alike node or at least a ”transparent” node that allows to pass information from children nodes without too many modifications (notice how this is the case for node A in Fig. 6.17). Therefore, it can be said that CFGP requires to hit the jackpot three times; hence its poor performance.

There are at least two possible ways to improve CFGP: (a) define new genetic operations that modify molecular primitives at a structural level (and not only their internal coefficients or leaf variables), in this way CFGP does not has to win the lottery three times, i.e. instead of requiring the necessary neuron-alike primitives to appear in the initial population, it can gradually work towards building (evolving) them; and (b) restrict CFGP to make use of only a few available molecular primitives (more akin to the original ADF paradigm proposed by Koza), in this way CFGP has to sort for neuron-alike primitives over a lesser number of possible options. Both proposed approaches are not exclusive in between, on the contrary, they complement each other and would probably be required to implement both of them to truly improve the performance of the proposed CFGP.

Despite CFGP lower performance when compared with more conventional MidGP approach, we consider both CFGP and FractalGP successful in their main purpose, that was to prove if GP could evolve complex, layered, systems where building blocks are composite elements rather than mere basic arithmetic operations or trigonometric functions.

Therefore, we conclude that proposed FractalGP, and its convolutional variant, CFGP, are promising new techniques that deserve further study and improvements; some possible methods to enhance CFGP have also been proposed in this section.

Chapter 7. Conclusions

7.1 Recapitulation

In this thesis, we researched the idea of developing a new DL framework based on learning units other than artificial neurons; more specifically, we thoroughly explored the possibility of performing the same type of tasks at which DL methods excel, through GP.

In Chapter 2, we presented a review on the main concepts and ideas behind the field of Deep Learning. We presented a brief historical review on how the field of Deep Learning came to be, and discussed some techniques that define and distinguish *modern* deep networks implementations from previous attempts at developing these large learning models.

In Chapter 3, we presented a thorough description on the some of the most relevant concepts from the GP framework. GP is an evolutionary algorithm that can be used for ML tasks, as such GP has been used for classification, regression, feature extraction as well as for representation learning. GP candidate solutions consist in abstract syntax trees that represent mathematical functions or simple computer programs.

In Chapter 4, we performed a literature review on works that are related to the approach proposed in this thesis research. We discussed how functions represented by GP trees can be highly non-linear in nature, and their intrinsic layered structure has enticed researchers to develop DL-alike, or pseudo-DL, approaches for ML tasks based on GP, i.e., evolving algorithms that perform both, feature extraction and prediction, represented by an unified processing pipeline of a GP tree. We also presented newer and exotic models of DL, where artificial neurons are replaced by other learning models, such as decision trees, very similar in spirit to the framework we hypothetize.

In this research, we proposed two main schemes that attempted to fusion some classic DL aspects or strategies, such as training deep learning structures or applying sequential convolutional layers for greater high dimensionality processing efficiency, but from the general GP learning model perspective. We proposed both, supervised and unsupervised learning, approaches for DL-alike processing, using the GP paradigm.

For the unsupervised model, in Chapter 5, we developed a GP framework for the evolution of *autoencoder* algorithms. The proposed framework allowed us to confirm the ability of GP for on-line learning, greatly reducing processing time for large training datasets. The local processing concept of neighboring features was also implemented, which laid foundations for convolutional-alike schemes based on GP.

In the case of supervised learning, in Chapter 6 we presented a GP framework extension, we call FractalGP, and a further variant aimed at image processing tasks, CFGP. FractalGP purpose is to allow, in a simple way, evolving considerably larger (deeper) tree structures through GP. Meanwhile, models evolved through CFGP resemble convolutional deep networks, where all artificial neurons have been replaced by a GP variant of abstract syntax trees.

Nevertheless, none of the proposed models could reach modern, ANN-based, DL models' performance. This opens an entire new line of questioning regarding the reasons behind this lackluster performance. Our research carried so far, and presented in this document, focused mostly on one possible reason: high dimensionality problems have always been challenging for GP. Both in chapters 5 and 6 empirical evidence on this issue was presented, as well as in Ch. 4 scientific literature recollected related to works that also struggled with this particular issue.

Also noteworthy, is the taxonomy presented in Sec. 3.2, where we classified different types of nodes used in modern GP implementations: from our perspective many of these special nodes emerged as a workaround to adapt GP to high dimensionality scenarios.

There could be other possible reasons (beside the curse of dimensionality) that hold GP from attaining the same level of performance of deep networks. Some of these issues will be discussed in further sections of this chapter.

7.2 Contributions

In this section, we recapitulate the main contributions of our research, already stated in the first chapter of this thesis, in greater detail.

1. We have developed two significant extensions to the GP framework: FractalGP and CFGP. FractalGP allows GP to evolve larger structures to those possible with GP's canonical representation. We presented empirical evidence that shows that, in low dimensionality problems, FractalGP succeeds in exploiting with greater efficiency larger GP individuals. Meanwhile, CFGP is a Fractal GP variant aimed at high dimensionality problems, such as image processing tasks. Notice also how CFGP generates fundamentally deep architectures where all neurons have been replaced with abstract syntax

trees, and where a GP algorithm manages to optimize them favourably up to a certain degree. Both methods can be considered deep learning models in their full right.

2. We generated autoencoder algorithms through GP, which also marked the first time autoencoders are learned with an ML tool other than any belonging to the family of ANN-related methods. We consider this an important result, because it shows that GP is as flexible as an ML approach based on ANNs, thus further providing evidence that a DL-paradigm based on GP, that is competitive with classical DL-models, might be possible in the near future.

3. We developed a GP primitives classification system (taxonomy), that allowed us to categorize different efforts from GP's research community regarding the development of complex ML systems. Reviewed through the lens of this proposed taxonomy, we presented an argument about GP's community long attempt to came up with the ”Deep Learning” paradigm shift (performing feature extraction + prediction in a unified pipeline), in an almost parallel race to that from the ANN community. We consider this analysis of equal relevance to that of our technical contributions, because it opens discussion on why GP researchers failed (or at least stopped) where ANN researchers succeed, and what are the true technical or theoretical limitations of the GP framework that does not allow to tackle the same kind of problems ANNs excel at.

7.2.1 Additional Developments

1. The development of an opensource GP library is one of the main contributions that resulted from this thesis research. The new library is aimed specifically at ML tasks, and is designed to evolve GP individuals that contemplate different classes of nodes described in our proposed taxonomy, as well as FractalGP and CFGP variants. The library is available for download at: `https://sourceforge.net/projects/turbogp/`.

7.3 Discussion

The final question we wish to answer is whether or not our hypothesis hold. Our research began with the assumption that if we biased GP towards models that mimic deep networks layered architectures, then GP could also learn intermediate representations necessary to tackle complex, high dimensionality, ML problems.

Considering the results gathered during this research we reach the conclusion that there is little evidence that support our hypothesis. If we focus on CFGP results (which directly imitates deep networks architectures), and compare it with more classical GP approaches performance, it is clear that there is no apparent benefit in attempting to do so. Classical GP models (such as MidGP) outmatch CFGP. In contrast, standard FractalGP is a more promising approach when compared against other typical GP models, and FractalGP imports few ideas from the field of DNNs, i.e. is more akin to the GP paradigm. However, FractalGP is still not ready for high dimensionality problems.

From inspection of the best individuals generated by CFGP, we found out that the evolutionary process converge towards finding few building blocks that can replicate in order to build complex information processing systems, instead of coming up with different functions in each processing node. As already discussed in the previous chapter, this well could be the byproduct of importing so many elements of one heuristic, so that the evolutionary process naturally tends towards finding the missing elements of the original heuristic. Therefore, if what we are after is efficiency, then we should probably restrict CFGP to evolve a few processing units that can replicate across the entire structure. In such way, the evolutionary search would be considerable more focused, and better results could be achieved in less amounts of time. In fact, this is how nature actually worked: the nerve cell evolved once, and then from it, complex nervous systems were built.

However, the question still remains: why GP cannot achieve a performance comparable to that of deep learning? is it a problem of high dimensionality or is it something else? To answer this question we will have to compare GP's performance against DNNs in low dimensionality problems. We suspect that there might be other factors besides high dimensionality that detriment GP's performance. One factor we believe could affect GP from achieving better results is the optimization process accuracy.

In Ch. 2 we briefly discussed how DL is really about solving very difficult optimization problems, and how gradient descent family of algorithms have come a long way even since the era of AlexNet. Notice how, in contrast, GP genetic operations are far more coarse in their manipulation of candidate solutions than a gradient based technique that operates over thousands or even millions of tunable parameters. This is one of the reasons we proposed FractalGP with internal weight factors, but even the proposed genetic operator that modifies these weights is primitive in comparison with modern gradient descent variants; in this regard more research is necessary.

7.4 Future lines of research

Finally, in this section we discuss some possible routes our research can take from here.

- One of the most immediate issues that should be answered is, how GP compares to ANNs in low dimensionality problems? A battery of tests, in both artificial and real life datasets, should be carried on with both methods, but preferably with artificial datasets where emphasis is made on generating complex surfaces that are challenging to approximate, and where the number of training samples can be controlled. Results obtained with our GP-AE suggested that GP outclasses ANN when training data is scarce, and recent research suggest that deep networks are overparametrized (Frankle & Carbin, 2019), further signaling GP as a preferable choice for problems with scarce training data over deep networks; but high dimensionality has to be taken out of the equation for proper investigation into the issue.

- FractalGP showed promising results in low dimensionality problems, but its performance declined dramatically when faced with higher number of feature variables. We propose that genetic operations similar to *feature leaf mutation*, defined only for CFGP, could adapt FractalGP to problems like the image denoising task.

- We consider CFGP an algorithm that showed some interesting properties. Notice how it is an algorithm capable of generating deep models without having to manually define some hyperparameters (no. of layers/filters per layer), as well as finding skipping connections, that resemble ResNets to a certain extent. Its current drawback is that, since it cannot modify the overall structure of molecular primitives, it depends on a chance to find suitable neuro-alike primitives to achieve acceptable performance. Therefore, it is important to develop new genetic operations for CFGP that can perform changes inside molecular primitives structures.

- Another important priority is to further develop the noise mutation operation. So far it is a very simple operation. We propose to enhance its behavior by incorporating the logic of optimization heuristics such as simulated annealing, i.e. deviation of distribution from which random additive factors are extracted decays over time. Another option is to incorporate elements from more current optimization methods such as the cross-entropy method (Rubinstein & Kroese, 2013). We consider this issue of the upmost importance, considering how, as already discussed in this thesis, DL methods have progressed to a large extent thanks to the development of enhanced gradient optimization methods.

- Finally, and considering a longer term research, we draw the possibility of developing a capped CFGP version where a single molecular primitive is searched for, which is then repeated across the entire deep structure and finally its weights tuned. The purpose of such model is to find new possible forms of artificial neurons with GP's help. Notice how modern deep models birth happened thanks to the artificial neuron model refinement with the activation function substitution, i.e. ReLU incorporation; also, as reviewed in Ch. 4, newer variants of the artificial neuron model are also being proposed and researched. GP could be an useful tool for this kind of research.

7.5 Publications

The following journal publication is a direct product of this thesis research:

- Rodriguez-Coayahuitl, L., Morales-Reyes, A., & Escalante, H. J. (2019). Evolving autoencoding structures through genetic programming. *Genetic Programming and Evolvable Machines*, 1-28. **JCR Q2**

The following conference publications are also a direct product of this thesis research:

- Rodriguez-Coayahuitl, L., Morales-Reyes, A., & Escalante, H. J. (2018, April). Structurally layered representation learning: towards deep learning through genetic programming. In *European Conference on Genetic Programming* (pp. 271-288). Springer, Cham.
- Rodriguez-Coayahuitl, L., Morales-Reyes, A., & Escalante, H. J. (2019, June). Convolutional Genetic Programming. In *Mexican Conference on Pattern Recognition* (pp. 47-57). Springer, Cham.
- Rodriguez-Coayahuitl, L., Morales-Reyes, A., & Escalante, H. J. (2019, November). Comparison between levels of abstraction in Genetic Programming. In *Autumn Meeting on Power, Electronics and Computing*. IEEE Xplore. Received *Best Paper Award* in the Computing track.

Appendix A. Convolutional Genetic Programming

In this section we explore the possibility to implement the fundamental architecture of Convolutional Neural Networks through GP. CNNs are a type of connectionist machine learning (ML) algorithms particularly adept at image processing tasks (LeCun *et al.*, 1989, 1995), thanks to a clever architectural design that allows them to scale well to high dimensionality problems.

In recent years, convolutional variants of DNNs have achieved record performance in typical ML tasks such as classification, regression or prediction. DNNs have achieved this performance thanks to an ever increasing number of stacked convolutional layers (Krizhevsky *et al.*, 2012; Szegedy *et al.*, 2015; He *et al.*, 2016).

Our motivation to import the architectural design of CNNs into GP is twofold: first, we wish to explore the idea of replacing neurons in CNNs with GP syntax trees, as we believe they have the same, or even higher, computational power than that of CNNs' neurons; and secondly due to the fact that GP does not scale well to high dimensionality problems (Gathercole & Ross, 1997), and we suspect it might benefit from the same architectural design than that of CNNs.

We use image denoising as target problem in order to test our proposed approach. The purpose of image denoising is to recover a clean image from contaminated original. The contamination model may be of different kinds. In this work we attempt to clean images from additive gaussian noise.

Verbatim sections, images and tables from this appendix were previously presented in Rodriguez-Coayahuitl *et al.* (2019a).

A.1 Image Denoising

The problem of image denoising is defined as follows: extract a clean image $\mathbf{x}$ from a noisy observation $\mathbf{y}$ such that $\mathbf{y} = \mathbf{x} + \mathbf{v}$, where $\mathbf{v}$ is a contamination process; a typical example is when $\mathbf{v}$ follows a Gaussian distribution with some given σ, which case is known as Additive Gaussian Noise (AWGN).

A.2 Related Work

GP has been succesfully used in the past to synthetize image filers. Examples of these type of works can be found in Yan *et al.* (2014); Khmag *et al.* (2017). However, these works rely on a modified version of the canonical GP individual such that primitive functions may include already specialized image filters or at least well known image processing functions, i.e. *high level* primitives (see Sec. 3.2). This property is undesirable if we wish to build ML systems that rely as little as possible on domain human expert's knowledge, i.e. highly automated learning systems. A more agnostic approach has been proposed in Hernández-Beltrán *et al.* (2016), where terminals of the syntax trees consist in simple statistics taken over regions of pixels.

It is relevant to contrast such specialized GP approaches with recent developments in the area of DL. There is really nothing specialized regarding image processing in the architecture of DNNs other than the use of convolution to efficiently process images. DnCNN (Zhang *et al.*, 2017) is a recent DNN designed to tackle image denoising; its flexibility is such that, by just switching the dataset with which is trained, the same network can learn to remove vastly different types of noises such as Gaussian noises with different or unknown levels of deviation, deblocking artifacts, and can even perform super resolution. DnCNN is competitive with fully and partially handcrafted image filters designed by human experts, thus positioning DNNs as very powerful learning systems.

In more general terms, high dimensionality issues have been long acknowledged in the GP community (Gathercole & Ross, 1997). Standard approaches to tackle such issues generally involve grouping input features in one way or another, process each cluster separately, and then attempt to assemble a joint global solution (Tran *et al.*, 2017; Rodriguez-Coayahuitl *et al.*, 2018). In contrast, in this work we draw inspiration from CNNs and propose a single sliding GP window that swipes an input image for processing, instead of many multiple independent GP processes.

A.3 Proposed Method

Our approach to evolve image denoising filters through GP is to leverage from the CNNs' architecture, where we replace neurons with GP syntax trees. Initially we propose to evolve a single sintax tree that acts as image filter by sliding over the noisy input image and cleaning pixel by pixel. Thereafter, we propose to stack multiple layers of these GP filters. We explain the theoretical advanges of stacking filters in this manner further below in this section.

A.3.1 Single Layer Convolutional GP Filter

We propose to use a standard GP individual representation, i.e. a syntax tree, to act as an image filter. This filter operates over a small window region of $d \times d$ pixels (where d is an odd number), receiving as input the pixels within such region, and returning as output a single value that is the level of noise of the central pixel in the operating window; in order to filter a whole image, the window is slided over an the entire image, generating a residual image the same size of input image that we desire to clean of noise. Fig. A.1a shows a depiction of the proposed sliding GP filter. This residual image represents the (estimated) level of noise of each pixel that composes the input image. In order to retrieve an approximation of the clean image, we simply substract the residual image from the noisy input image.

The leaf nodes of the GP individual should be the individual pixels in the region being processed, or constanst values within some range. The primitives can be any function that can operate at this individual pixel level. This is done in this way to avoid the use of any image filtering expert's knownledge.

A.3.2 Multi-layer Convolutional GP

Additionally we also propose to stack multiple of these sliding GP filters, both in parallel and in series, since DNNs are actually designed this way. That is, instead of using a single GP syntax tree that filters the image, we can slide multiple, different, GP syntax trees that generate as output different *feature maps*, which are intermediate transformations of the input that may be useful for generating the desired output. All these feature maps form a volume of codified information that is further processed by another GP sliding tree that generates the final output, i.e. the residual image. Fig. A.1b shows a GP filter architeture composed of two stacked filter in series, while Fig. A.1c depicts an architecture with multiple GP filters both in series and in parallel.

Stacking these convolutional filters in series carries the advantage of increasing the *field of view*. This means that if we use two sliding filter with windows of 3×3 in series, when we reconstruct the central pixel at the output of the second filter, we are actually using information of a 5×5 window size around it (this is as along as the first filter did manage to codify information at feature map it outputs). Notice however how when using this approach in such example scenario, we are using only 18 features as inputs (9 for the first filter and 9 for the second) whereas if we attempt to directly process filters of 5×5 window size, then such filter would need to process 25 input features. This is one of the reasons on why CNNs scale well to high dimensionality problems and image processing tasks.

On the other hand, stacking filters in parallel per layer allows to generate more than one feature map at

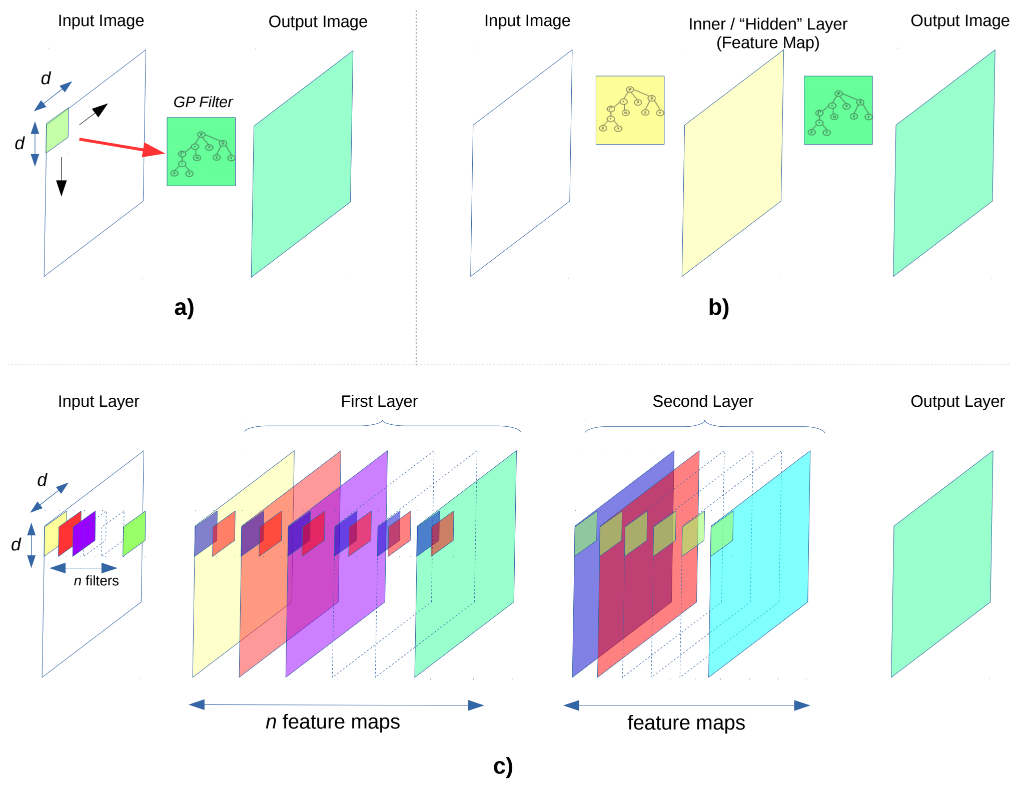


Figure A.1: Multilayer GP architecture. a) Single layer, single filter; b) Two layer, one filter per layer; c) Three layer, first layer and second with n filters, third layer with only 1, output, filter.

each layer. Each feature map might codify different information useful for the next layer of processing.

The canonical form of GP contemplates individuals that are composed of a single syntax tree. In our proposed method, in the case of multiple stacked filters, we would need to evolve more than a single GP tree. Although there do exists GP individual representations based on forests (multiple trees), in this type of representations the trees are loosely dependent on each other, whereas in the multilayer architecture we are proposing here, the filter trees series rely completely on the output generated by the previous trees in the structure.

A.3.3 Evolving Multiple Layers of Convolutional GP filters

In order to train this complex architecture, we propose three different approaches: (**straightforward**) define the GP individual as the entire set of trees across all layers, evolve individuals by applying genetic operations layer-wise; (**sequential**) evolve the multi-layer structure sequentially, i.e. evolve the first layer for fixed number of generations; once this first evolution is finished, the second layer of filters are evolved, which take as input a cleaner version of the noisy image generated by the first layer, and so on; (**ensamble**) the third approach is based on the idea that the multiple feature maps at the penultimate layer might actually

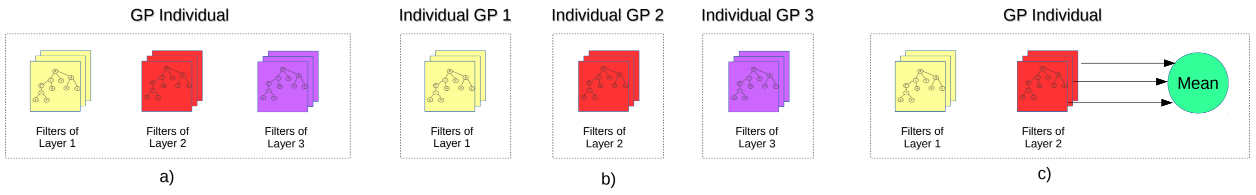


Figure A.2: Different possible GP individual representations for multilayer GP filters.

act as ensamble learner, with the last layer only performing the mean function, so in this architecuture we enforce this behavior by taking as output the mean over the feature maps of the last layer. Fig. A.2 illustrates these three variants.

A.4 Experimental Results and Analysis

A.4.1 Training and Testing Datasets

We generated the training data following the works of Zhang *et al.* (2017); Schmidt & Roth (2014); Chen & Pock (2017). From the Berkeley Segmentadion Dataset (Roth & Black, 2005) we extracted 19,200 unique 40×40 image patches for training purposes. For testing, we use the same classic image processing set used in Zhang *et al.* (2017); Yan *et al.* (2014); Khmag *et al.* (2017), composed of well-known pictures such as ”Lena” and ”Boats”. A total of 12 (seven 256×256 and five 512×512) pictures were used for testing.

We contaminated both the training patches and the testing images by adding them noise masks generated with a Gaussian distribution of $\sigma = 25$. All training and testing was performed on grayscale images.

A.4.2 Evolutionary Algorithm Setup

For all experiments we used a multi-population, island based, model. We used a population of 500 individuals splitted across 5 islands each with 100 individuals. We used an heterogeneous and asynchronous model where each island had different crossover/mutation probabilities, and every 10 generations send their top 10 performing individuals to another, randomly selected, island (migration). The crossover/mutation probabilities were set as follow for each island: $[0.9/0.1, 0.7/0.3, 0.5/0.5, 0.3/0.7, 0.1/0.9]$. The set of primitives used consist on binary arithmetic operators, $[+, -, \times, \div]$, binary functions *max*, *min*, *mean*, and unary functions x^2, x^3, and Rectifier Linear Units (ReLUs).

We utilized an on-line form of learning defined in Rodriguez-Coayahuitl *et al.* (2018). We partitioned the entire training dataset into mini-batches of 60 samples, and use one mini-batch per evolutionary cycle for evaluating both individuals and offspring generated. We used a steady state population replacement policy.

Table A.1: Average performance of all Convolutional GP architectures tested. Values expressed in decibels. Higher is better.

Noisy Image	Single GP, 3x3	Single GP, 5x5	Strfwd-GP (2 Layers)	Sequential GP (2 Layers)	Ensamble (2L + Mean)	DnCnn
20.32	25.96	25.07	25.22	25.93	23.60	30.43

A.4.2.1 Fitness function

We used the minimization of the mean square error (MSE) between the predicted noise level and the actual noise level to drive the evolution of all systems proposed.

A.4.3 Results

We tested two Single Layer Convolutional GP, one consisting in a sliding window of 3×3 pixels, and another with a window of 5×5 pixels.

We tested three different Multi-layer Convolutional GP, each under one of the three different proposed methods for evolving multi-layer GPs. All Multi-layer architectures consisted in only 2 layers (2 layers + mean, in the case of the ensamble method). Both the straightforward and the sequential architectures were composed of 3 filters in the first layer, and 1 filter in the second layer (3 filters in both layers for the ensamble method). All filters were 3×3 windows.

Table A.1 shows the results obtained by the different tested approaches. We include in Table A.1 the values of the unfiltered noisy images (to understand how much the proposed approaches actually denoise the images), as well as the performance of the DnCNN network (Zhang *et al.*, 2017), to fully appreciate how far GP is from modern DNNs. These results were obtained on the same testing dataset for all approaches (including the DnCNN), and using the same training dataset (also applies for DnCNN). All the GP approaches were given the same computational time[1]. Therefore these results are based on a comparison as fair as possible.

Fig. A.3 shows the performance of a 2-Layer, Sequentially evolved variant GP, on ten training patches. We found no visually appreciable difference between this output and the one from a single layer GP.

A.4.4 Additional Results

We also performed experiment using 10 filters per layer for the Multi-layer GP architectures. Although we found them to be consistently inferior in performance to the 3 filters per layer reported above, we found

[1]DnCNN runs in less time than GP, due to being accelerated in GPU and implemented in highly optimized DL software libraries.

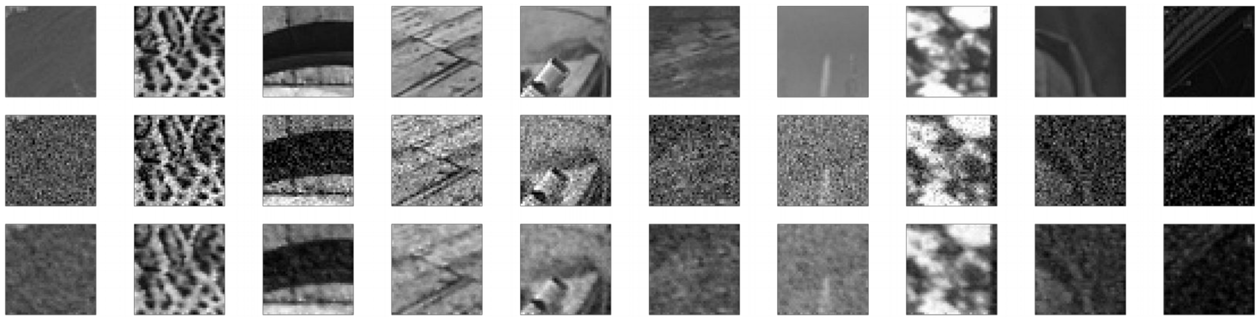


Figure A.3: Visual results of the output generated by a 2-Layer Convolutional, Sequentially evolved, GP. From top to bottom: original images, noisy samples, filtered images.

that these GP variants generated interesting paterns in the hidden layer. Fig. A.4 shows the feature maps generated by the ten filters for ten different training patches. Some of the feature maps appear to be signaling borders or other points of interest.

A.4.5 Discussion

Results shows that GP can succesfully synthetize image denoising filters, even though none of the proposed methods allows GP to benefit from a multi-layer convolutional architecture, thus positioning a single layer GP filter as the reference method-to-beat in future works based on GP.

Results also confirm that GP struggles with high dimensionality problems. In this case, a single layer 5×5 window GP filter does not performs any better, if not worse, than a 3×3 window one, even though the first one has more than twice context information that theoretically should allow it to perform a better filtering.

A.5 Conclusions

In this section we have introduced a method to evolve image denoising filters with GP, through an architecture inspired by CNNs. Our results have confirmed that:

- GP is a viable method to synthetize image denoising filters, even when processing images at individual pixel level.
- GP struggles with high dimensionality problems, since it cannot make use of input samples with as low as 25 features.
- GP cannot directly benefit from a stacked convolutional architecture. More research is necessary in this direction.

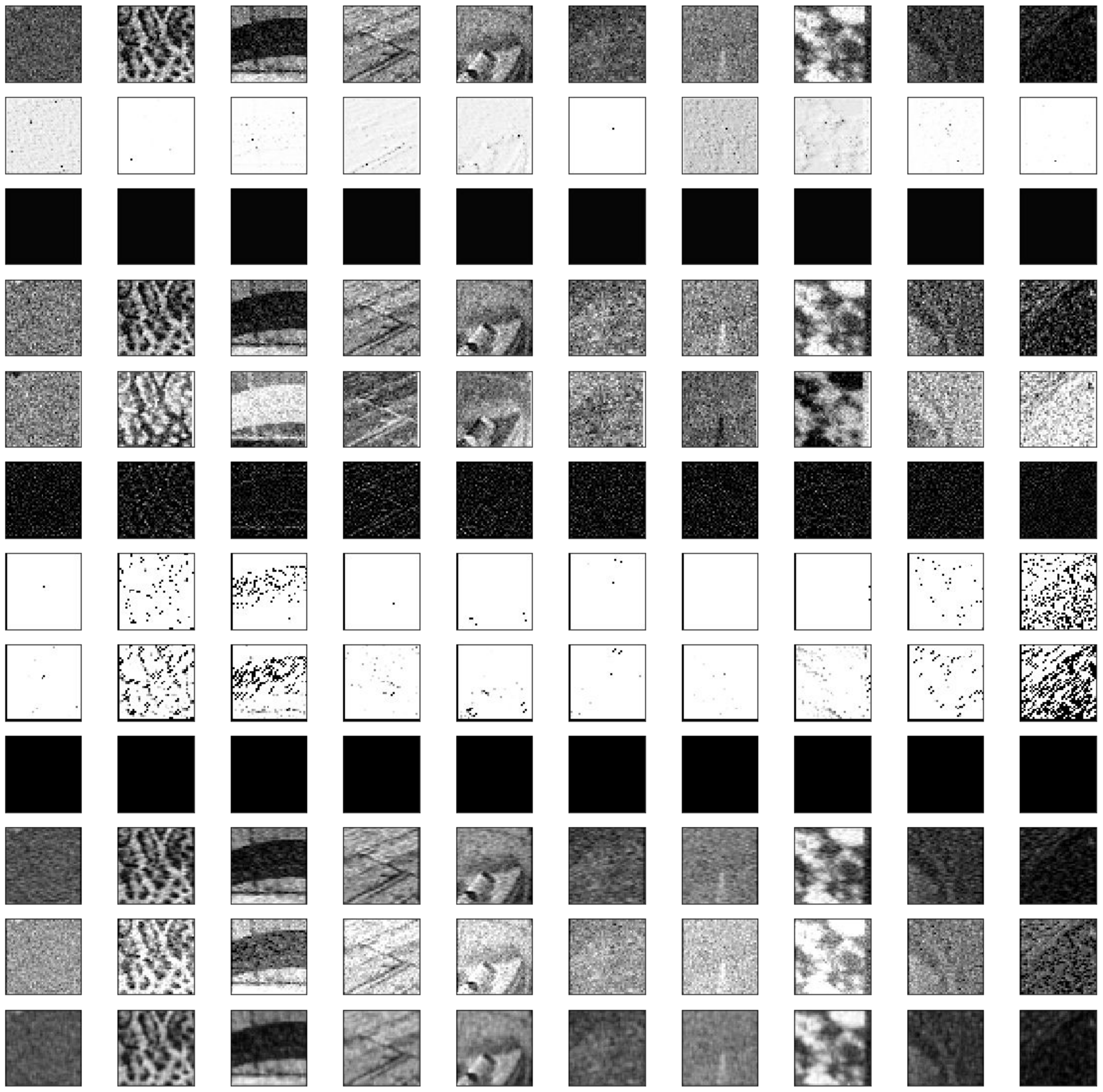


Figure A.4: Feature maps generated by a 2-layer GP in the hidden layer given 10 different input patches. From top to bottom: first row, noisy patches rows 2 to 10, feature maps; last row, filtered final output.

We have also draw a clear, quantitavive, performance gap between GP and DL based methods, by using the same exact training and testing datasets, and making head-to-head direct comparison with modern DNN architectures.

We believe this work should serve a reference for future works that attempt to attack problems with GP in which DL excels at.

Bibliography

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., *et al.* (2016). Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*.

Agrawal, R., Imieliński, T., & Swami, A. (1993). Mining association rules between sets of items in large databases. En: *Acm sigmod record*. ACM, Vol. 22, pp. 207–216.

Ahmed, S., Zhang, M., Peng, L., & Xue, B. (2014). Multiple feature construction for effective biomarker identification and classification using genetic programming. En: *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*. ACM, pp. 249–256.

Al-Sahaf, H., Song, A., Neshatian, K., & Zhang, M. (2012a). Extracting image features for classification by two-tier genetic programming. En: *2012 IEEE Congress on Evolutionary Computation*. IEEE, pp. 1–8.

Al-Sahaf, H., Song, A., Neshatian, K., & Zhang, M. (2012b). Two-tier genetic programming: Towards raw pixel-based image classification. *Expert Systems with Applications*, **39**(16): 12291–12301.

Al-Sahaf, H., Zhang, M., & Johnston, M. (2014). Genetic programming evolved filters from a small number of instances for multiclass texture classification. En: *Proceedings of the 29th International Conference on Image and Vision Computing New Zealand*. ACM, pp. 84–89.

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

Alpaydin, E. (2014). *Introduction to machine learning*. MIT press.

Anderson, C. (2014). Lfwcrop face dataset. *http://conradsanderson.id. au/lfwcrop/*.

Andre, D., Bennett III, F. H., & Koza, J. R. (1996). Discovery by genetic programming of a cellular automata rule that is better than any known rule for the majority classification problem. En: *Proceedings of the 1st annual conference on genetic programming*. MIT Press, pp. 3–11.

Angeline, P. J. & Pollack, J. B. (1992). The evolutionary induction of subroutines. En: *Proceedings of the fourteenth annual conference of the cognitive science society*. Bloomington, Indiana, pp. 236–241.

Arce, F., Zamora, E., & Sossa, H. (2017). Dendrite ellipsoidal neuron. En: *2017 international joint conference on neural networks (IJCNN)*. IEEE, pp. 795–802.

Arce, F., Zamora, E., Sossa, H., & Barrón, R. (2018). Differential evolution training algorithm for dendrite morphological neural networks. *Applied Soft Computing*, **68**: 303–313.

Atkins, D., Neshatian, K., & Zhang, M. (2011). A domain independent genetic programming approach to automatic feature extraction for image classification. En: *2011 IEEE Congress of Evolutionary Computation (CEC)*. IEEE, pp. 238–245.

Axelrod, B. (2007). Genetic programming. `https://en.wikipedia.org/wiki/File:Genetic_Program_Tree.png`. Accessed 05/05/17.

Ayodele, T. O. (2010). Types of machine learning algorithms. En: *New advances in machine learning*. InTech.

Baker, J. E. (1987). Reducing bias and inefficiency in the selection algorithm. En: *Proceedings of the second international conference on genetic algorithms*. Vol. 206, pp. 14–21.

Ballard, D. H. (1987). Modular learning in neural networks. En: *AAAI*. pp. 279–284.

Banzhaf, W. & Langdon, W. B. (2002). Some considerations on the reason for bloat. *Genetic Programming and Evolvable Machines*, **3**(1): 81–91.

Bellman, R. (1961). *Adaptive control process: a guided tour*.

Bellman, R. & Kalaba, R. (1957). On the role of dynamic programming in statistical communication theory. *IRE Transactions on Information Theory*, **3**(3): 197–203.

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

Bengio, Y., Simard, P., Frasconi, P., *et al.* (1994). Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, **5**(2): 157–166.

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

Betechuoh, B. L., Marwala, T., & Tettey, T. (2006). Autoencoder networks for hiv classification. *Current Science*, pp. 1467–1473.

Beyer, H.-G. & Schwefel, H.-P. (2002). Evolution strategies–a comprehensive introduction. *Natural computing*, **1**(1): 3–52.

Bhowan, U., Johnston, M., Zhang, M., & Yao, X. (2012). Evolving diverse ensembles using genetic programming for classification with unbalanced data. *IEEE Transactions on Evolutionary Computation*, **17**(3): 368–386.

Bishop, C. M. (2006). *Pattern recognition and machine learning*. springer.

Bleuler, S., Brack, M., Thiele, L., & Zitzler, E. (2001). Multiobjective genetic programming: Reducing bloat using spea2. En: *Proceedings of the 2001 Congress on Evolutionary Computation (IEEE Cat. No. 01TH8546)*. IEEE, Vol. 1, pp. 536–543.

Blickle, T. & Thiele, L. (1996). A comparison of selection schemes used in evolutionary algorithms. *Evolutionary Computation*, **4**(4): 361–394.

Bot, M. C. (2001). Feature extraction for the k-nearest neighbour classifier with genetic programming. En: *European Conference on Genetic Programming*. Springer, pp. 256–267.

Bottou, L. (1998). Online learning and stochastic approximations. *On-line learning in neural networks*, **17**(9): 142.

Bottou, L. (2010). Large-scale machine learning with stochastic gradient descent. En: *Proceedings of COMPSTAT'2010*. Springer, pp. 177–186.

Bourlard, H. & Kamp, Y. (1988). Auto-association by multilayer perceptrons and singular value decomposition. *Biological cybernetics*, **59**(4-5): 291–294.

Cano, A., Ventura, S., & Cios, K. 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.

Chen, Y. & Pock, T. (2017). Trainable nonlinear reaction diffusion: A flexible framework for fast and effective image restoration. *IEEE transactions on pattern analysis and machine intelligence*, **39**(6): 1256–1272.

Chollet, F. (2016). Building autoencoders in keras.

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

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

Ciregan, D., Meier, U., & Schmidhuber, J. (2012). Multi-column deep neural networks for image classification. En: *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*. IEEE, pp. 3642–3649.

Cohoon, J. P., Hegde, S. U., Martin, W. N., & Richards, D. (1987). Punctuated equilibria: a parallel genetic algorithm. En: *Genetic algorithms and their applications: proceedings of the second International Conference on Genetic Algorithms: July 28-31, 1987 at the Massachusetts Institute of Technology, Cambridge, MA*. Hillsdale, NJ: L. Erlhaum Associates, 1987.

Collobert, R., Weston, J., Bottou, L., Karlen, M., Kavukcuoglu, K., & Kuksa, P. (2011). Natural language processing (almost) from scratch. *Journal of Machine Learning Research*, **12**(Aug): 2493–2537.

Comisky, W., Yu, J., & Koza, J. (2000). Automatic synthesis of a wire antenna using genetic programming. En: *Late Breaking Papers at the 2000 Genetic and Evolutionary Computation Conference, Las Vegas, Nevada*. Citeseer, pp. 179–186.

Dahl, G. E., Sainath, T. N., & Hinton, G. E. (2013). Improving deep neural networks for lvcsr using rectified linear units and dropout. En: *2013 IEEE international conference on acoustics, speech and signal processing*. IEEE, pp. 8609–8613.

Damianou, A. & Lawrence, N. (2013). Deep gaussian processes. En: *Artificial Intelligence and Statistics*. pp. 207–215.

David, O. E. & Greental, I. (2014). Genetic algorithms for evolving deep neural networks. En: *Proceedings of the Companion Publication of the 2014 Annual Conference on Genetic and Evolutionary Computation*. ACM, pp. 1451–1452.

Davidson, J. L. & Hummer, F. (1993). Morphology neural networks: An introduction with applications. *Circuits, Systems and Signal Processing*, **12**(2): 177–210.

Davies, D. L. & Bouldin, D. W. (1979). A cluster separation measure. *IEEE transactions on pattern analysis and machine intelligence*, (2): 224–227.

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., & Fei-Fei, L. (2009). ImageNet: A Large-Scale Hierarchical Image Database. En: *CVPR09*.

Dietterich, T. G. (2000). Ensemble methods in machine learning. En: *International workshop on multiple classifier systems*. Springer, pp. 1–15.

Dolin, B., Bennett III, F. H., & Rieffel, E. G. (2002). Co-evolving an effective fitness sample: experiments in symbolic regression and distributed robot control. En: *Proceedings of the 2002 ACM symposium on Applied computing*. ACM, pp. 553–559.

Doucette, J. A., Mcintyre, A. R., Lichodzijewski, P., & Heywood, M. I. (2012). Symbiotic coevolutionary genetic programming: a benchmarking study under large attribute spaces. *Genetic Programming and Evolvable Machines*, **13**(1): 71–101.

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

Dunn, O. J. (1961). Multiple comparisons among means. *Journal of the American Statistical Association*, **56**(293): 52–64.

Eiben, A. E., Smith, J. E., *et al.* (2003). *Introduction to evolutionary computing*, Vol. 53. Springer.

Emigdio, Z., Trujillo, L., Schütze, O., Legrand, P., *et al.* (2014). Evaluating the effects of local search in genetic programming. En: *EVOLVE-A Bridge between Probability, Set Oriented Numerics, and Evolutionary Computation V*. Springer, pp. 213–228.

Escalante, H. J., García-Limón, M. A., Morales-Reyes, A., Graff, M., Montes-y Gómez, M., Morales, E. F., & Martínez-Carranza, J. (2015). Term-weighting learning via genetic programming for text classification. *Knowledge-Based Systems*, **83**: 176–189.

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

Evans, B., Al-Sahaf, H., Xue, B., & Zhang, M. (2018). Evolutionary deep learning: A genetic programming approach to image classification. En: *2018 IEEE Congress on Evolutionary Computation (CEC)*. IEEE, pp. 1–6.

Evett, M. & Fernandez, T. (1998). Numeric mutation improves the discovery of numeric constants in genetic programming. *Genetic Programming*, pp. 66–71.

Fernando, C., Banarse, D., Reynolds, M., Besse, F., Pfau, D., Jaderberg, M., Lanctot, M., & Wierstra, D. (2016). Convolution by evolution: Differentiable pattern producing networks. En: *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference*. ACM, pp. 109–116.

Folino, G., Pizzuti, C., & Spezzano, G. (2004). Boosting technique for combining cellular gp classifiers. En: *European Conference on Genetic Programming*. Springer, pp. 47–56.

Frankle, J. & Carbin, M. (2019). The lottery ticket hypothesis: Finding sparse, trainable neural networks. En: *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net.

Fukushima, K. (1988). Neocognitron: A hierarchical neural network capable of visual pattern recognition. *Neural networks*, **1**(2): 119–130.

Gallinari, P., LeCun, Y., Thiria, S., & Fogelman-Soulie, F. (1987). Memoires associatives distribuees. *Proceedings of COGNITIVA*, **87**: 93.

Gao, L., Song, J., Liu, X., Shao, J., Liu, J., & Shao, J. (2017). Learning in high-dimensional multimedia data: the state of the art. *Multimedia Systems*, **23**(3): 303–313.

Gathercole, C. & Ross, P. (1994). Dynamic training subset selection for supervised learning in genetic programming. En: *International Conference on Parallel Problem Solving from Nature*. Springer, pp. 312–321.

Gathercole, C. & Ross, P. (1997). Tackling the boolean even n parity problem with genetic programming and limited-error fitness. *Genetic programming*, **97**: 119–127.

Geurts, P., Ernst, D., & Wehenkel, L. (2006). Extremely randomized trees. *Machine learning*, **63**(1): 3–42.

Gogna, A. & Majumdar, A. (2016). Semi supervised autoencoder. En: *International Conference on Neural Information Processing*. Springer, pp. 82–89.

Goldberg, D. E. & Deb, K. (1991). A comparative analysis of selection schemes used in genetic algorithms. *Foundations of genetic algorithms*, **1**: 69–93.

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning*. MIT press.

Gorges-Schleuter, M. (1989). Asparagos an asynchronous parallel genetic optimization strategy. En: *Proceedings of the third international conference on Genetic algorithms*. pp. 422–427.

Guo, H., Zhang, Q., & Nandi, A. K. (2008). Feature extraction and dimensionality reduction by genetic programming based on the fisher criterion. *Expert Systems*, **25**(5): 444–459.

Haeffele, B. D. & Vidal, R. (2017). Global optimality in neural network training. En: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. pp. 7331–7339.

He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. En: *Proceedings of the IEEE conference on computer vision and pattern recognition*. pp. 770–778.

Hernández, G., Zamora, E., & Sossa, H. (2017). Comparing deep and dendrite neural networks: a case study. En: *Mexican Conference on Pattern Recognition*. Springer, pp. 32–41.

Hernández-Beltrán, J. E., Díaz-Ramírez, V. H., Trujillo, L., & Legrand, P. (2016). Restoration of degraded images using genetic programming. En: *Optics and Photonics for Information Processing X*. International Society for Optics and Photonics, Vol. 9970, p. 99700K.

Hinton, G., Deng, L., Yu, D., Dahl, G. E., Mohamed, A.-r., Jaitly, N., Senior, A., Vanhoucke, V., Nguyen, P., Sainath, T. 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, G. E. & Salakhutdinov, R. R. (2006a). Reducing the dimensionality of data with neural networks. *Science*, **313**(5786): 504–507.

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

Hinton, G. E., Osindero, S., & Teh, Y.-W. (2006). A fast learning algorithm for deep belief nets. *Neural computation*, **18**(7): 1527–1554.

Hochreiter, S. (1998). The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, **6**(02): 107–116.

Holland, J. H. (1992). *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. MIT press.

Hornik, K., Stinchcombe, M., & White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural networks*, **2**(5): 359–366.

Huang, G., Sun, Y., Liu, Z., Sedra, D., & Weinberger, K. Q. (2016). Deep networks with stochastic depth. En: *European Conference on Computer Vision*. Springer, pp. 646–661.

Huang, G. B., Ramesh, M., Berg, T., & Learned-Miller, E. (2007). Labeled faces in the wild: A database for studying face recognition in unconstrained environments. Reporte técnico 07-49, University of Massachusetts, Amherst.

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

Iba, H. (1999). Bagging, boosting, and bloating in genetic programming. En: *Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation-Volume 2*. Morgan Kaufmann Publishers Inc., pp. 1053–1060.

Ioffe, S. & Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*.

Ivakhnenko, A. G. (1968). The group method of data of handling; a rival of the method of stochastic approximation. *Soviet Automatic Control*, **13**: 43–55.

Ivakhnenko, A. G. (1971). Polynomial theory of complex systems. *IEEE transactions on Systems, Man, and Cybernetics*, (4): 364–378.

Jordan, J. (2018). Introduction to autoencoders. Accessed 30/09/19.

Kathuria, A. (2018). Intro to optimization in deep learning: Gradient descent. Accessed 29/08/19.

Khmag, A., Ramli, A. R., Al-haddad, S., Yusoff, S., & Kamarudin, N. (2017). Denoising of natural images through robust wavelet thresholding and genetic programming. *The Visual Computer*, **33**(9): 1141–1154.

Kingma, D. P. & Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

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

Koza, J. R. (1994). Genetic programming ii: Automatic discovery of reusable subprograms. *Cambridge, MA, USA*.

Koza, J. R., Bennett, F. H., Andre, D., Keane, M. A., & Dunlap, F. (1997). Automated synthesis of analog electrical circuits by means of genetic programming. *IEEE Transactions on evolutionary computation*, **1**(2): 109–128.

Krawiec, K. & Bhanu, B. (2003). Coevolution and linear genetic programming for visual learning. En: *Genetic and Evolutionary Computation Conference*. Springer, pp. 332–343.

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

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. En: *Advances in neural information processing systems*. pp. 1097–1105.

Langdon, W. B. & Poli, R. (1998). Fitness causes bloat. En: *Soft Computing in Engineering Design and Manufacturing*. Springer, pp. 13–22.

LeCun, Y. (1998). The mnist database of handwritten digits. *http://yann. lecun. com/exdb/mnist/*.

LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., & Jackel, L. D. (1989). Backpropagation applied to handwritten zip code recognition. *Neural computation*, **1**(4): 541–551.

LeCun, Y., Bengio, Y., *et al.* (1995). Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks*, **3361**(10): 1995.

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

LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *Nature*, **521**(7553): 436–444.

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

Lee, D. D. & Seung, H. S. (2001). Algorithms for non-negative matrix factorization. En: *Advances in neural information processing systems*. pp. 556–562.

Lensen, A., Xue, B., & Zhang, M. (2017). Improving k-means clustering with genetic programming for feature construction. En: *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. ACM, pp. 237–238.

Lensen, A., Xue, B., & Zhang, M. (2019). Can genetic programming do manifold learning too? En: *European Conference on Genetic Programming*. Springer, pp. 114–130.

Limon, M., Escalante, H. J., Morales, E., & Morales-Reyes, A. (2014). Simultaneous generation of prototypes and features through genetic programming. En: *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*. ACM, pp. 517–524.

Limón, M. G., Escalante, H. J., & Morales, E. F. (2014). Towards simultaneous prototype and feature generation. En: *Power, Electronics and Computing (ROPEC), 2014 IEEE International Autumn Meeting on*. IEEE, pp. 1–6.

Limón, M. G., Escalante, H. J., Morales, E., & Pineda, L. V. (2015). Class-specific feature generation for 1nn through genetic programming. En: *Power, Electronics and Computing (ROPEC), 2015 IEEE International Autumn Meeting on*. IEEE, pp. 1–6.

Lin, J.-Y., Ke, H.-R., Chien, B.-C., & Yang, W.-P. (2007). Designing a classifier by a layered multi-population genetic programming approach. *Pattern Recognition*, **40**(8): 2211–2225.

Lin, J.-Y., Ke, H.-R., Chien, B.-C., & Yang, W.-P. (2008). Classifier design with feature selection and feature extraction using layered genetic programming. *Expert Systems with Applications*, **34**(2): 1384–1393.

Littman, M. & Isbell, C. (2015). Machine learning - supervised learning. `https://www.youtube.com/watch?v=Ki2iHgKxRBo`. Accessed 11/02/16.

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

Liu, L., Shao, L., Li, X., & Lu, K. (2015). Learning spatio-temporal representations for action recognition: A genetic programming approach. *IEEE transactions on cybernetics*, **46**(1): 158–170.

Lohpetch, D. & Corne, D. (2009). Discovering effective technical trading rules with genetic programming: Towards robustly outperforming buy-and-hold. En: *Nature & Biologically Inspired Computing, 2009. NaBIC 2009. World Congress on*. IEEE, pp. 439–444.

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

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

Loveard, T. & Ciesielski, V. (2001). Representing classification problems in genetic programming. En: *Evolutionary Computation, 2001. Proceedings of the 2001 Congress on*. IEEE, Vol. 2, pp. 1070–1077.

Luke, S. & Panait, L. (2006). A comparison of bloat control methods for genetic programming. *Evolutionary Computation*, **14**(3): 309–344.

Luke, S. & Spector, L. (1997). A comparison of crossover and mutation in genetic programming. *Genetic Programming*, **97**: 240–248.

Luke, S. & Spector, L. (1998). A revised comparison of crossover and mutation in genetic programming. *Genetic Programming*, **98**(208-213): 55.

Manderick, B. & Spiessens, P. (1989). Fine-grained parallel genetic algorithms. En: *Proceedings of the third international conference on Genetic algorithms*. pp. 428–433.

Mao, X., Shen, C., & Yang, Y.-B. (2016). Image restoration using very deep convolutional encoder-decoder networks with symmetric skip connections. En: *Advances in neural information processing systems*. pp. 2802–2810.

Martinez, Y., Trujillo, L., Naredo, E., & Legrand, P. (2014). A comparison of fitness-case sampling methods for symbolic regression with genetic programming. En: *EVOLVE-A Bridge between Probability, Set Oriented Numerics, and Evolutionary Computation V*. Springer, pp. 201–212.

Miikkulainen, R., Liang, J., Meyerson, E., Rawal, A., Fink, D., Francon, O., Raju, B., Shahrzad, H., Navruzyan, A., Duffy, N., *et al.* (2019). Evolving deep neural networks. En: *Artificial Intelligence in the Age of Neural Networks and Brain Computing*. Elsevier, pp. 293–312.

Mika, S., Ratsch, G., Weston, J., Scholkopf, B., & Mullers, K.-R. (1999). Fisher discriminant analysis with kernels. En: *Neural Networks for Signal Processing IX, 1999. Proceedings of the 1999 IEEE Signal Processing Society Workshop.*. IEEE, pp. 41–48.

Miller, G. F., Todd, P. M., & Hegde, S. U. (1989). Designing neural networks using genetic algorithms. En: *ICGA*. Vol. 89, pp. 379–384.

Minsky, M. (1961). Steps toward artificial intelligence. *Proceedings of the IRE*, **49**(1): 8–30.

Mitchell, T. (1997). Machine learning. wcb.

Montana, D. J. & Davis, L. (1989). Training feedforward neural networks using genetic algorithms. En: *IJCAI*. Vol. 89, pp. 762–767.

Morse, G. & Stanley, K. O. (2016). Simple evolutionary optimization can rival stochastic gradient descent in neural networks. En: *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference*. ACM, pp. 477–484.

Moscato, P. *et al.* (1989). On evolution, search, optimization, genetic algorithms and martial arts: Towards memetic algorithms. *Caltech concurrent computation program, C3P Report*, **826**: 1989.

Moscinski, R. & Zakrzewska, D. (2015). Building an efficient evolutionary algorithm for forex market predictions. En: *International Conference on Intelligent Data Engineering and Automated Learning*. Springer, pp. 352–360.

Mühlenbein, H. & Schlierkamp-Voosen, D. (1993). Predictive models for the breeder genetic algorithm i. continuous parameter optimization. *Evolutionary computation*, **1**(1): 25–49.

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

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

Neely, C., Weller, P., & Dittmar, R. (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, A., Yosinski, J., & Clune, J. (2015). Deep neural networks are easily fooled: High confidence predictions for unrecognizable images. En: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. pp. 427–436.

Nguyen, K., Fookes, C., Ross, A., & Sridharan, S. (2017). Iris recognition with off-the-shelf cnn features: A deep learning perspective. *IEEE Access*, **6**: 18848–18855.

Oh, K.-S. & Jung, K. (2004). Gpu implementation of neural networks. *Pattern Recognition*, **37**(6): 1311–1314.

Olague, G. & Trujillo, L. (2011). Evolutionary-computer-assisted design of image operators that detect interest points using genetic programming. *Image and Vision Computing*, **29**(7): 484–498.

Olague, G., Clemente, E., Dozal, L., & Hernández, D. E. (2014). Evolving an artificial visual cortex for object recognition with brain programming. En: *EVOLVE-A Bridge between Probability, Set Oriented Numerics, and Evolutionary Computation III*. Springer, pp. 97–119.

Otero, F. E. & Johnson, C. G. (2013). Automated problem decomposition for the boolean domain with genetic programming. En: *European Conference on Genetic Programming*. Springer, pp. 169–180.

Pagie, L. & Hogeweg, P. (1997). Evolutionary consequences of coevolving targets. *Evolutionary computation*, **5**(4): 401–418.

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

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., & Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, **12**: 2825–2830.

Plaut, D. C. (2016). Unsupervised learning.

Poli, R., Langdon, W., & McPhee, N. (2008). A field guide to genetic programming (with contributions by jr koza)(2008). *Published via http://lulu. com.*

Potter, M. A. & De Jong, K. A. (1994). A cooperative coevolutionary approach to function optimization. En: *International Conference on Parallel Problem Solving from Nature*. Springer, pp. 249–257.

Potvin, J.-Y., Soriano, P., & Vallée, M. (2004). Generating trading rules on the stock markets with genetic programming. *Computers & Operations Research*, **31**(7): 1033–1047.

Real, E., Moore, S., Selle, A., Saxena, S., Suematsu, Y. L., Tan, J., Le, Q. V., & Kurakin, A. (2017). Large-scale evolution of image classifiers. En: *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org, pp. 2902–2911.

Ritter, G. X. & Urcid, G. (2003). Lattice algebra approach to single-neuron computation. *IEEE Transactions on Neural Networks*, **14**(2): 282–295.

Rodriguez-Coayahuitl, L., Morales-Reyes, A., & Escalante, H. J. (2018). Structurally layered representation learning: towards deep learning through genetic programming. En: *European Conference on Genetic Programming*. Springer, pp. 271–288.

Rodriguez-Coayahuitl, L., Morales-Reyes, A., & Escalante, H. J. (2019a). Convolutional genetic programming. En: *Mexican Conference on Pattern Recognition*. Springer, pp. 47–57.

Rodriguez-Coayahuitl, L., Morales-Reyes, A., & Escalante, H. J. (2019b). Evolving autoencoding structures through genetic programming. *Genetic Programming and Evolvable Machines*, pp. 1–28.

Rosenblatt, F. (1957). *The perceptron, a perceiving and recognizing automaton Project Para*. Cornell Aeronautical Laboratory.

Roth, S. & Black, M. J. (2005). Fields of experts: A framework for learning image priors. En: *null*. IEEE, pp. 860–867.

Rubinstein, R. Y. & Kroese, D. P. (2013). *The cross-entropy method: a unified approach to combinatorial optimization, Monte-Carlo simulation and machine learning*. Springer Science & Business Media.

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1985). Learning internal representations by error propagation. Reporte técnico, California Univ San Diego La Jolla Inst for Cognitive Science.

Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A. C., & Fei-Fei, L. (2015). ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, **115**(3): 211–252.

Sakanashi, H., Higuchi, T., Iba, H., & Kakazu, Y. (1996). Evolution of binary decision diagrams for digital circuit design using genetic programming. En: *International Conference on Evolvable Systems*. Springer, pp. 470–481.

Salakhutdinov, R. & Hinton, G. (2009). Deep boltzmann machines. En: *Artificial Intelligence and Statistics*. pp. 448–455.

Salimans, T., Ho, J., Chen, X., Sidor, S., & Sutskever, I. (2017). Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint arXiv:1703.03864*.

Samaria, F. S. & Harter, A. C. (1994). Parameterisation of a stochastic model for human face identification. En: *Applications of Computer Vision, 1994., Proceedings of the Second IEEE Workshop on*. IEEE, pp. 138–142.

Sanderson, C. (2014). Lfwcrop face dataset.

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

Schmidhuber, J. (2015). Deep learning in neural networks: An overview. *Neural networks*, **61**: 85–117.

Schmidt, M. D. & Lipson, H. (2008). Coevolution of fitness predictors. *IEEE Transactions on Evolutionary Computation*, **12**(6): 736–749.

Schmidt, U. & Roth, S. (2014). Shrinkage fields for effective image restoration. En: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. pp. 2774–2781.

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

Sermanet, P., Chintala, S., & LeCun, Y. (2012). Convolutional neural networks applied to house numbers digit classification. En: *Pattern Recognition (ICPR), 2012 21st International Conference on*. IEEE, pp. 3288–3291.

Shao, L., Liu, L., & Li, X. (2013). Feature learning for image classification via multiobjective genetic programming. *IEEE Transactions on Neural Networks and Learning Systems*, **25**(7): 1359–1371.

Sherrah, J. R., Bogner, R. E., & Bouzerdoum, A. (1997). The evolutionary pre-processor: Automatic feature extraction for supervised classification using genetic programming. *Genetic Programming*, pp. 304–312.

Simoncini, D., Verel, S., Collard, P., & Clergue, M. (2009). Centric selection: a way to tune the exploration/exploitation trade-off. En: *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*. ACM, pp. 891–898.

Simonyan, K. & Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.

Smolensky, P. (1986). Information processing in dynamical systems: Foundations of harmony theory. Reporte técnico, Colorado Univ at Boulder Dept of Computer Science.

Sossa, H. & Guevara, E. (2014). Efficient training for dendrite morphological neural networks. *Neurocomputing*, **131**: 132–142.

Sotelo, A., Guijarro, E., Trujillo, L., Coria, L. N., & Martínez, Y. (2013). Identification of epilepsy stages from ecog using genetic programming classifiers. *Computers in biology and medicine*, **43**(11): 1713–1723.

Srivastava, N., Hinton, G. E., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *Journal of machine learning research*, **15**(1): 1929–1958.

Stanley, K. O. & Miikkulainen, R. (2002). Evolving neural networks through augmenting topologies. *Evolutionary computation*, **10**(2): 99–127.

Stanley, K. O., D'Ambrosio, D. B., & Gauci, J. (2009). A hypercube-based encoding for evolving large-scale neural networks. *Artificial life*, **15**(2): 185–212.

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

Such, F. P., Madhavan, V., Conti, E., Lehman, J., Stanley, K. O., & Clune, J. (2017). Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning. *arXiv preprint arXiv:1712.06567*.

Suganuma, M., Shirakawa, S., & Nagao, T. (2017). A genetic programming approach to designing convolutional neural network architectures. En: *Proceedings of the Genetic and Evolutionary Computation Conference*. ACM, pp. 497–504.

Sutskever, I., Vinyals, O., & Le, Q. V. (2014). Sequence to sequence learning with neural networks. En: *Advances in neural information processing systems*. pp. 3104–3112.

Sutton, R. S. & Barto, A. G. (1998). *Reinforcement learning: An introduction*, Vol. 1. MIT press Cambridge.

Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., Erhan, D., Goodfellow, I., & Fergus, R. (2013). Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*.

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., & Rabinovich, A. (2015). Going deeper with convolutions. En: *Proceedings of the IEEE conference on computer vision and pattern recognition*. pp. 1–9.

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

Teredesai, A., Park, J., Govindaraju, V., *et al.* (2001). Active handwritten character recognition using genetic programming. *Lecture notes in computer science*, pp. 371–379.

Theis, L., Shi, W., Cunningham, A., & Huszár, F. (2017). Lossy image compression with compressive autoencoders. *arXiv preprint arXiv:1703.00395*.

Thierens, D. & Goldberg, D. (1994). Elitist recombination: An integrated selection recombination ga. En: *Proceedings of the First IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence*. IEEE, pp. 508–512.

Tieleman, T. & Hinton, G. (2012). Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, **4**(2): 26–31.

Tomassini, M. (2006). *Spatially structured evolutionary algorithms: Artificial evolution in space and time*. Springer.

Tran, B., Xue, B., & Zhang, M. (2016). Genetic programming for feature construction and selection in classification on high-dimensional data. *Memetic Computing*, **8**(1): 3–15.

Tran, B., Xue, B., & Zhang, M. (2017). Using feature clustering for gp-based feature construction on high-dimensional data. En: *European Conference on Genetic Programming*. Springer, pp. 210–226.

Trujillo, L. & Olague, G. (2006). Synthesis of interest point detectors through genetic programming. En: *Proceedings of the 8th annual conference on Genetic and evolutionary computation*. ACM, pp. 887–894.

Trujillo, L. & Olague, G. (2008). Automated design of image operators that detect interest points. *Evolutionary Computation*, **16**(4): 483–507.

Veit, A., Wilber, M. J., & Belongie, S. (2016). Residual networks behave like ensembles of relatively shallow networks. En: *Advances in neural information processing systems*. pp. 550–558.

Vladislavleva, E. Y. (2008). *Model-based problem solving through symbolic regression via pareto genetic programming*. Tesis de doctorado, CentER, Tilburg University.

White, D. R. & Poulding, S. (2009). A rigorous evaluation of crossover and mutation in genetic programming. En: *European Conference on Genetic Programming*. Springer, pp. 220–231.

Wilcoxon, F. (1945). Individual comparisons by ranking methods. *Biometrics bulletin*, **1**(6): 80–83.

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

Wold, S., Esbensen, K., & Geladi, P. (1987). Principal component analysis. *Chemometrics and intelligent laboratory systems*, **2**(1-3): 37–52.

Yan, R., Shao, L., Liu, L., & Liu, Y. (2014). Natural image denoising using evolved local adaptive filters. *Signal Processing*, **103**: 36–44.

Yann, L. (1987). *Modeles connexionnistes de lapprentissage*. Tesis de doctorado, PhD thesis, These de Doctorat, Universite Paris 6.

Yates, A., Cafarella, M., Banko, M., Etzioni, O., Broadhead, M., & Soderland, S. (2007). Textrunner: open information extraction on the web. En: *Proceedings of Human Language Technologies: The Annual Conference of the North American Chapter of the Association for Computational Linguistics: Demonstrations*. Association for Computational Linguistics, pp. 25–26.

Zamora, E. & Sossa, H. (2017). Dendrite morphological neurons trained by stochastic gradient descent. *Neurocomputing*, **260**: 420–431.

Zhang, K., Zuo, W., Chen, Y., Meng, D., & Zhang, L. (2017). Beyond a gaussian denoiser: Residual learning of deep cnn for image denoising. *IEEE Transactions on Image Processing*, **26**(7): 3142–3155.

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

Zhang, Y. & Rockett, P. I. (2009). A generic multi-dimensional feature extraction method using multiobjective genetic programming. *Evolutionary Computation*, **17**(1): 89–115.

Zhang, Y. & Rockett, P. I. (2011). A generic optimising feature extraction method using multiobjective genetic programming. *Applied Soft Computing*, **11**(1): 1087–1097.

Zhou, Y., Yang, J., Zhang, H., Liang, Y., & Tarokh, V. (2019). Sgd converges to global minimum in deep learning via star-convex path. *arXiv preprint arXiv:1901.00451*.

Zhou, Z.-H. & Feng, J. (2017). Deep forest: towards an alternative to deep neural networks. En: *Proceedings of the 26th International Joint Conference on Artificial Intelligence*. AAAI Press, pp. 3553–3559.

Zhou, Z.-H. & Feng, J. (2018). Deep forest. *National Science Review*, **6**(1): 74–86.