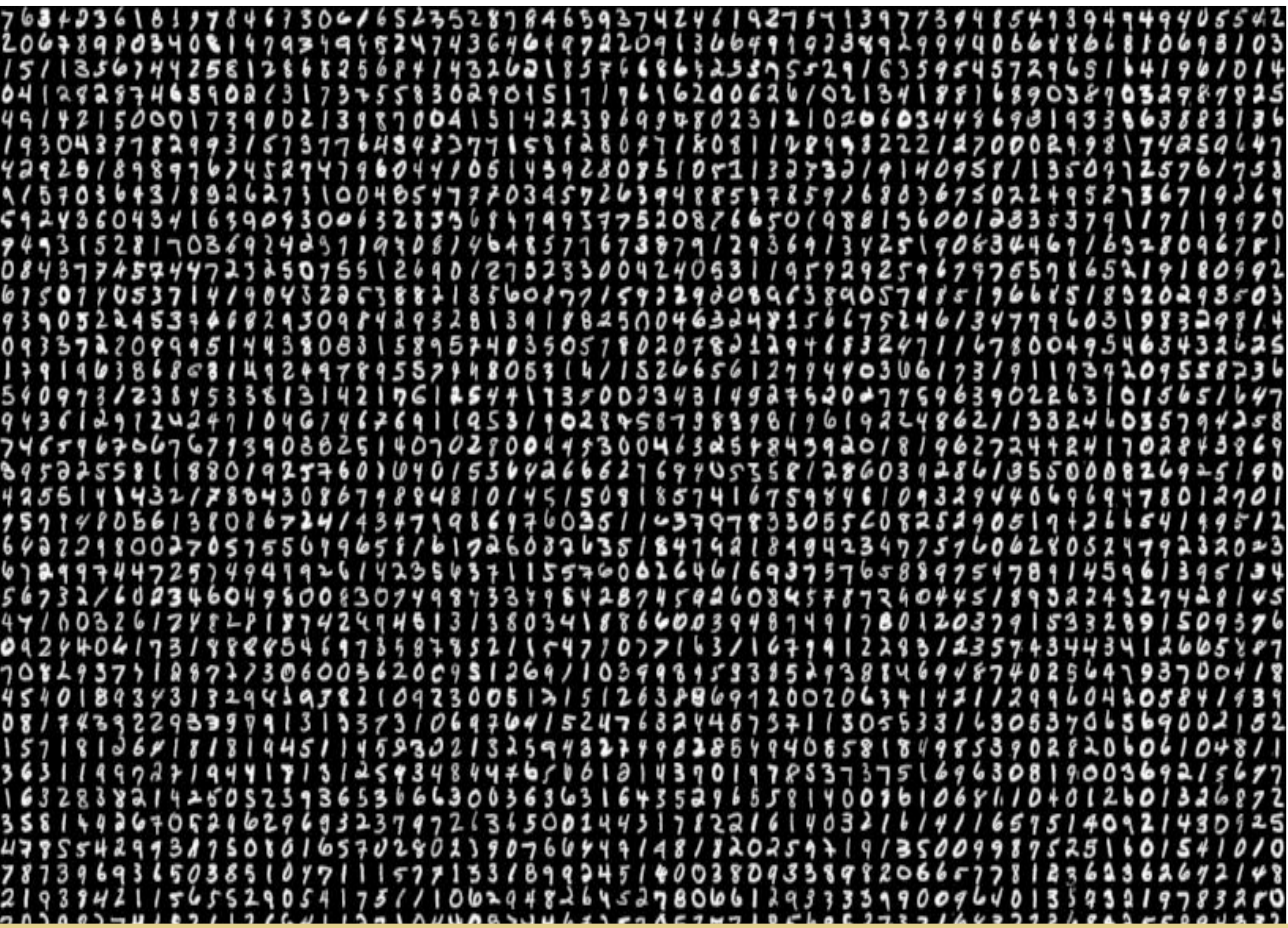


David Martins Duarte Sereno

# Automatic Evolution of Deep AutoEncoders

Dissertation/Internship Report
Master in Informatics Engineering
advised by Penousal Machado & Bernardete Ribeiro
and presented to the Departament Informatics Engineering
of the Faculty of Sciences and Technology of the University of Coimbra

January 2018

## Abstract

In this work we have created a versatile evolutionary algorithm that can evolve an auto encoder neural network structure in an attempt to maximize the performance of different classifiers by using the resulting compressed version of the instances. During this process the algorithm searches for structures that compress as much as possible the representation to facilitate the classifiers training while maintain the necessary information in the datasets.

This approach is set around the evolution of the number and size of the layers of a deep autoencoder, which is then trained using back propagation in a semi supervised fashion. The tests executed spanned multiple classifiers, and show promising results in which we observed an overall improvement in the classification on most the cases and, as expected, significant decrease in the training times.

On the context of this thesis, a methodical approach was taken to analyze the impact that an autoencoder has, and how it behaves when its structure is evolved by means of Evolutionary Computation. As a stepping stone for the final work, preliminary experiments were performed, where multiple auto encoders were implemented and tested to confirm their correct behaviour and performance. To complement this a an evolutionary algorithm was tested in order to assess the usefulness and potential of evolving the structures, without imposing any restrictions on their shape.

## Keywords

Deep Learning, AutoEncoder, Evolutionary Algorithms

## Resumo

No decorrer desta dissertação foi criado um algoritmo evolucionário altamente versátil capaz de evoluir com sucesso a estrutura de um rede neural de auto encoder, que procura maximizar a performance de diferentes classificadores. Durante este processo, o algoritmo procura maximizar a compressão de forma a facilitar a tarefa de treino dos classificadores sem que exista perda de performance dos mesmos.

Esta abordagem consiste na evolução do número de camadas e número de neurónios presentes em cada uma, sendo a estrutura treinada de forma semi supervisionada através de retropropagação. Foram executados testes sobre um leque variado de classificadores, onde observámos uma melhoria na sua performance bem como uma significativa redução nos tempos de treino.

No contexto desta tese , consta tambem uma análise metódica sobre o funcionamento e performance de autoencoders profundos e quais são as vantagens práticas de evoluir a sua estrutura. Como primeiro passo, no decorrer do trabalho, foram testados múltiplos autoencoders e abordagens evolucionárias de forma a confirmar o seu comportamento e performance.

## Palavras-Chave

Deep Learning, AutoEncoder, Computação Evolucionária

# Contents

# Acronyms

**AE** AutoEncoder.

**ANN** Artificial Neural Network.

**BCE** Binary Cross Entropy.

**CAE** Convolutional Autoencoder.

**CNN** Generative Adversial Network.

**DAE** Denoising Autoencoder.

**DBN** Deep Belief Network.

**EA** Evolutionary Algorithm.

**EC** Evolutionary Computation.

**GAN** Generative Adversial Network.

**LSTM** Long Short Term Memory.

**NE** Neuro Evolution.

**VAE** Variational Autoencoder.

# List of Figures

# List of Tables

# List of Algorithms

# Chapter 1

# Introduction

Nature has succeeded in bringing forth humanity as a whole through evolution, and these manage to mold their surrounding to their will using the immense potential of their intellectual. The question arises, if each succeeded on these seemly impossible tasks, can't we learn from them and apply them to solve the problems that we encounter? In computer science, fueled by these question, arose the Evolutionary Computation and Neural Networks fields which arbor nearly endless potential to solve close to any problem when tailored to fit a certain solution, but are ultimately limited by the amount of data available for the given problem and computational power needed to be considered effective solutions.

Recent years have motivated a significant increase in the computational power which has led deep learning to flourish, consequently leading to an increased and widespread enthusiasm in the Pattern Recognition and Machine Learning areas.

Deep architectures are inspired in the neurosciences and aim at decomposing a hard problem into simpler tasks (recursive decomposition), by stacking several layers of neurons, each one responsible for learning a different set of features. However, the problem of finding the appropriate topology for an Artificial Neural Network (ANN) often follows a trial-and-error approach, which is a difficult and time consuming task. In order to overcome this challenge, researchers have focused their attention on the development of algorithms to automate the discovery of adequate topologies (and/or weights) of ANNs relying on methods such as Evolutionary Computation(EC).

The main goal of this thesis is the study, understanding and development of an evolutionary framework for the evolution of the topology of Deep autoencoders(AE). In brief words, autoencoders are unsupervised learning models that aim at rebuilding the original data, i.e., whereas typical feed-forward neural networks try to predict the correct classes (y) from the input data (x), auto encoders try to predict x from x. Therefore, one of the main advantages of autoencoders is their ability to learn compressed representations of the original data, in a process that resembles similarities with common feature selection techniques, such as the Principal Component Analysis (PCA) algorithm.

There are many variables in attempting to test this concept such as:

- How does an autoencoder behave with different types and sizes of data?

- What benefits does a deep architecture bring to the basic function of an auto encoder?

- What elements in the topology of an autoencoder are relevant to its performance?
- What kind of evolutionary approach can we take on the training and/or design of an autoencoder?
- How to measure the network compression and what is considered to be an acceptable result of the autoencoder compression?

## 1.1 Scope

On the current thesis we will focus on the development and implementation of an Evolutionary Algorithm(AE) that is both versatile and effective in the compression of data sets for use in other tasks such as classification, reducing the time required while improving the performance of these methods. The final result should be able to evolve a population of AutoEncoder Structures and provide the user with a fully trained AutoEncoder, that should provide compression and feature selection for the task selected beforehand.

## 1.2 Goals

As stated before, the main goal of this study is to analyze the potential applications of Deep autoencoders in a practical setting, while using an evolutionary approach to design the autoencoders. We hope to achieve an evolutionary framework that can produce high performing autoencoders, that can be applied to multiple problems with a minimum need for user tweaking.

We intend to analyze the impact of choosing a structure for the Auto Encoder in its performance and ability to improve the classification of standard classifiers.

## 1.3 Document Structure

This thesis is organized into three different Chapter, excluding the current one.

Chapter 2 presents itself as the analysis of the state of art and goes through a brief overview of EC and NNs, followed by an analysis into autoencoders and NeuroEvolution(NE), which are the key topics of this thesis.

Chapter 3 contains the detailed description of all the work done in this project, which includes what tests were planned and how they were executed, followed by an analysis of their results.

Chapter 4 closes off the thesis with the roundup of the work made and how it translates into results that met our goals followed by the observation of what work might be made in the future.

# Chapter 2

# State of Art

This section aims at describing and presenting the work that has been done in the fields within the scope of this thesis. As such we will start by analyzing Evolutionary Computation, what it is and how it has been used in the past followed by an introduction and analysis of what are Artificial Neural Networks (ANNs) and what this field has produced in practical settings. In the follow up we will explain what are AutoEncoders and how this concept will be used in context of this thesis. To close this section we will briefly survey the area that combines both the previous approaches (ANNs & EC), which is known as NeuroEvolution, what different kind of algorithms and approaches does this area of study contain and what results were obtained.

## 2.1 Evolutionary Computation

Finding solutions to real world problems has been for centuries the aim of scientists and engineers alike. This problems can range from simple choices to how the universe is expanding. Although some of these problems are very simple, others take years to discover or are currently impossible to solve.

To fill this void, researchers changed their aim towards what has happened in nature, how does nature solve this problems, how does it adapt to its ever changing environment and thrives with what would seem custom tailored species that inhabit even the harshest environments. The immense variety of different species and their adaptation to the changes in the environment through generations, can be explained by the overall accepted evolutionary theory proposed by Darwin [9]. In this theory Darwin states that the evolution observed in species occurs naturally due to the selection pressure the environment exerts on the individuals, such as predatory relationships and resource scarcity, which result in the demise of the poorly adapted individuals. This results on a higher chance of better individuals to survive and reproduce (survival of the fittest) transferring their distinct and vital characteristics to the next generations, thus improving the overall quality and adaptation of a species as a whole.

This is interesting to computer science, since we can observe how evolution has been able to solve and create interesting responses to problems across the world, from creating cooperative behaviours in ants to evolving birds capability to extract nectar from plants, by imposing morphological changes onto the species. It becomes clear that evolution approaches is the ideal algorithm, one which can solve entirely different complex problems in multiple ways, given only the context.

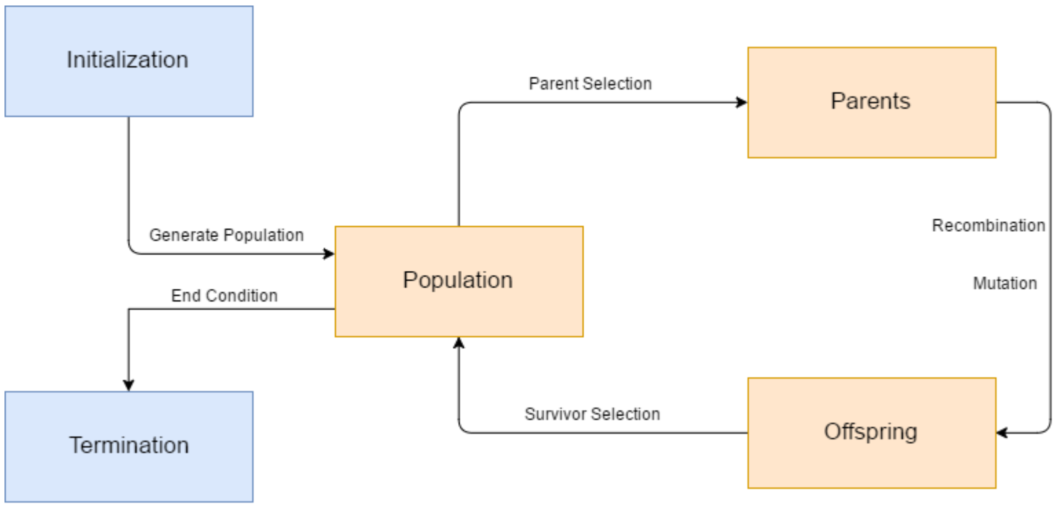


Figure 2.1: Evolutionary Algorithm flowchart.

This concept was brought to life in the realms of computation as what is known as **Evolutionary Computation** (EC) and its corresponding Evolutionary Algorithms (EAs). These can be seen as stochastic search procedures that aim to solve problems[11]. One of it's greatest advantages lies in their capability of finding a good solution to a problem in a limited amount of time or in search spaces that are unfeasible to be tackled by exact algorithmic approaches, such as NP-hard, and have been used to create solutions for many different real life problems that range from System Optimization to Pattern Recognition [11].

Although there are many variations as to what is, and how an EA works, the basic structure that is used in most of the approaches is still rooted in the traditional evolutionary process. As such, EAs work on a **population** of individuals, where each represents a possible solution to the problem that is being solved. The individuals are then evaluated on how well they perform as a solution, usually by a numeric value attributed by a fitness function (objective function) that seeks to module the problem to be solved. Knowing the individual's representation (genotype) and how it represents a solution (phenotype) and its corresponding quality (fitness) we can simulate natural selection by means of reproduction, where we select the individuals that will become parents. Usually, two individuals are selected as parents, based on their fitness.Genetic operators are applied to the parents. We apply a crossover function to produce new solutions (offspring), from the combination of the parents genotype. As in nature we make it possible to occur mutation to the offspring which enables new genetic material to be generated which might be advantageous for the species. Once we have created the offspring we then select the next generation; this is usually obtained by comparing the existing solutions fitness (survival selection). This process is then repeated to the next generations of solutions until a stop condition is fulfilled, which can range from the selected number of generations have gone by, a time limit as been exceeded or the quality of the solutions has not improved after a (fixed) period of time. We can represent this procedure as a flowchart, as seen below in figure 2.1.

**Algorithm 1:** Evolutionary Algorithm pseudo-code

```
Result: Best_Solution
population = GENERATE_POPULATION(size);
for indiv in population do
    indiv.fitness = EVALUATE(indiv);
end
while stopCondition() == False do
    parents = PARENT_SELECTION(population);
    offspring = RECOMBINATION(parents);
    offspring = MUTATION(offspring);
    for indiv in offspring do
        indiv.fitness = EVALUATE(indiv);
    end
    population = SURVIVOR_SELECTION(population, offspring);
end
return best_indiv in population;
```

By analyzing figure 2.1 and algorithm 1, one can simply conclude that EAs fall into the category of generate-and-test algorithms. The fitness function represents an heuristic function of a solution's quality and the search process is led by the selection, recombination and mutation operators. EA possess some features that strengthen their position within the category of generate-and-test, such as:

- Evolutionary Algorithms operate at population level. A collection of candidate solutions are processed simultaneously;
- They mostly use recombination to create new candidate solutions from existing ones;
- They perform actions according to probabilistic information;
- They are easily parallelizable.

As the field matured we saw many different approaches appear with different interpretations and implementations of the basic concepts of evolution. In many cases the approaches differ on how each individual presents itself, some take the form of string over a finite alphabet, such as Genetic Algorithms (GA)[24] where others use real-valued vectors (ES)[5] , finite state machines in classical evolutionary programming (EP)[13] or even binary trees in classic Genetic Programming (GP)[28]. Other approaches differ from the classical structure above by implementing variations on the algorithm, some were based on other popular methods such as Particle Swarm Optimization (PSO)[48] or on other biological systems such as the human immunity systems [10], other simply apply changes such as sexual reproduction[43], competing species or evaluate individuals based on how unique they are in novelty exploration approaches [37].

A given representation or approach might be more adequate than others if it matches the problem better, i.e., if the encoding of candidate solutions represents an easier or more natural form of solution to the problem. It is important but not vital to understand the problem we wish to solve in order to select and fine tune the approach we intend to use for the best results.

For a better understanding, we can further divide the structure of an EA into the several elements, which are detailed in the following sections.

### Representation

In order to represent real world problems it is almost always necessary to represent the problems in such a way that enables the computation of a problem, and it is clear that there are multiple valid ways to represent the same problem. In order for a computer to execute and manipulate a solution to produce new solutions there is a need to select the representation of our individuals (genotypes), that will be modified and evaluated, which is then mapped to a solution to the problem at hand (phenotype). The evolutionary search takes place in the genotype space. A solution is obtained by decoding a genotype, and as such it is desirable that all possible feasible solutions can be represented.

### Fitness Function

The fitness function is one of the most important parts of an EA, since in its core is the representation of the problem to be solved, and defines what improvement is for a solution, serving as a way to distinguish between the many possible solutions. As such, it is the basis for selecting the most fit individuals and thus inducing progress in the algorithm.

Technically, it is a function or procedure that assigns a quality measure to genotypes. Typically, this function is composed from the inverse representation (to create the corresponding phenotype) followed by a quality measure in the phenotype space.

### Population

The population in any point of the execution, is the set of possible solutions (their representation) that encode the species that is being evolved by the algorithm. The initial population is often composed of multiple randomly generated individuals which can be biased depending on prior information obtained on the problem.

### Parent Selection and Survivor Selection

In order to simulate evolutionary pressure done by the environment it is necessary to select which individuals manage to reproduce and which individuals, both parents and offspring manage to survive.

Parent selection is the method used to select what solutions will mate in order to create new solutions, this is usually done by taking into account the individuals fitness. This selection together with the survivor selection mechanism is responsible for generating selective pressure over the population, ideally creating improvements in future generations.

This methods are typically probabilistic, in such a way that high quality individuals have higher chances of being selected to mate than those with lower quality, which in most cases still have a low chance of being selected since they might possess important genetic information relevant to a good solution, and are important to avoid greedy local search, or the loss of variety in the population.

Some of the most used methods for parent selection are: roulette-wheel selection, stochastic universal sampling, tournament selection and truncation selection. [11]

In survivor selection, similarly to the parent selection method, its purpose is to select the most fit individuals that will pass onto the next generation of the population. This is done to apply selective pressure to the population in order to improve its overall fitness over the generations. The most common approach to this selection is based only on the fitness value, where the worse individuals are removed from the population, reducing the population to the initial population size(before reproduction).

### Variation Operators

As we have observed previously in order for a population to evolve, new individuals must be created from the existing population through what is commonly known as Variation Operators (Crossover and Mutation). Base on genetics crossover, the crossover operators in EC usually are composed by a process which divides each parents genotype into small parts and assigning each of the offspring created with different ones from each parent. This ensures that each offspring will be different from their parents and one another while carrying the genetic material of their parents which translates into similar yet different solutions which will then compete in the population for survival.

In the second method, Mutation, an individual will suffer a small, or large depending on the function, modification to their genotypes, usually by modifying one or more of its basic representation elements, usually known as genes.

In order to achieve good results in search procedure it is essential to achieve variation. This is done by the variations imposed by both the mutation and crossover functions together, since they can make small variations to the solution, thus carrying out exploitation, or large variations which result in exploration. Exploration which linked with the principle of global search, and exploitation, which linked to local search, must strive for a balance in order to achieve a successful search, as they must be able to avoid the local optimal solutions while searching for increasingly better solutions.

## 2.2 Neural Networks

Neural Networks found in the human brain are composed of millions of interconnected neurons, where each neuron receives signals through synapses located on its dentrites or membrane. When the signals received are strong enough (surpass a certain threshold), the neuron is activated and emits a signal through the axon, which might be connected to other neurons synapses, and may cause them to also activate. The complex combination of exterior and internal input and network structure is what makes the behaviour we can observe on a daily basis possible. An image representing the neurons can be found in Figure 2.2.

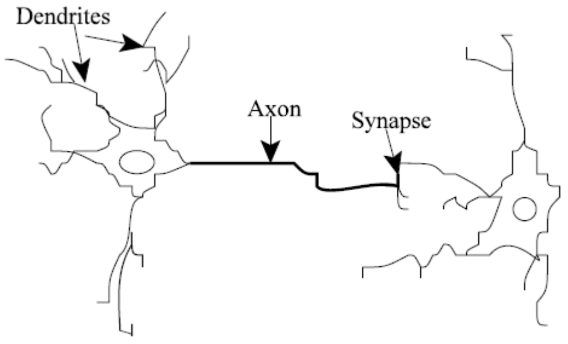


Figure 2.2: Scheme of a natural neuron, and synapse.

Given the incredible potential of the human intellectual, which is mostly due to the brain structure and capability to learn almost anything given the right information and time, many have sought to make machines which present the same problem skills of a human, i.e, universal approximators/problem solvers. This serves as the basis to the entire field of Artificial Intelligence, but on a more direct interpretation of this concept, ANNs were created in order to simulate a human brain.

First created in the 1950s [14], ANNs are mathematical/computation models that mimic the activity of the human brain at a conceptual level. In its simplest form, an ANN is a group of many interconnected mathematical modules (Artificial Neurons) or nodes, which usually consist of applying a mathematical function (Activation Function) to the sum of the weighted inputs. In many cases a bias may also be applied, which is then output to the neurons it is connected to, producing a sequence of real-valued activation's. These structures are usually organized in layers, starting with the input layer and ending in the output layer. All the layers between the input and output re known as hidden-layers. The structure of an artificial neuron and of an ANN are depicted in Figures 2.3 and 2.4 respectively.

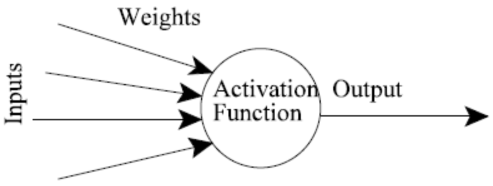


Figure 2.3: Artificial neuron representation.

When observed mathematically, an Artificial Neuron is composed by a set of weights $w$

which refer to its inputs $x$, with an overall bias $b$ and an activation function $\sigma$. When all pieces are set together we get the activation value $z$ for the neuron $j$ given by:

$$z_j = \sigma(w \cdot x + b) \tag{2.1}$$

This leads to the basic neural network model, which can be described as a series of functional transformations [6]:

$$a_k = \sum_{i=1}^{M} w_{kj} z_j + w_{j0} \tag{2.2}$$

Although the structure is quite fascinating, a model by itself is just that, a model with no purpose. When creating an ANN, usually we want it to model a function that can attempt to solve a given problem, but in order to do this we need to design the architecture of the network and, above all, to train the network to fit the problem. In supervised problems, the task of training is the adaptation of the network weights $w_{kj}$ so that the ANN can map a set of input patterns to the desired corresponding outputs or behaviour.

This is usually done by utilizing techniques such as gradient descent on a loss function such as the mean squared error, using problem dependent previously acquired training samples. For such purpose the use of training algorithms like BackPropagation (BP) [53] have become widespread for the scientific community even though other methods like EAs have also been used for this purpose.

The study of neural networks has evolved through the decades and many different concepts and types of networks have been used to produce excellent real world performances, with approaches that range from Convolutional Neural Networks (CNNs) specialized in image and sound analysis [29], to recurrent architectures [22] that enable networks to keep and use collections of sequential data, correlating them to produce great results in problems such as prediction, which also include Long short-term memory (LSTM) [23] networks that have risen in popularity in the last years.

In principle ANNs are universal aproximators [38]. Due to their flexibilty, ANNs can can be fairly easily adapted in structure to deal with specific problems/domains,and as long as data is available to learn from, it is likely that it is possible to build effective models. As example, ANNs have been applied to problems which range from computer vision [55] to security intrusion [54] presenting on many the state of art when it comes to result and performance.

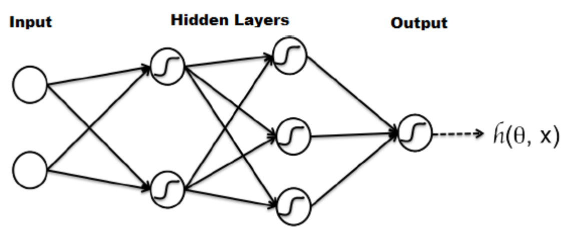


Figure 2.4: Feed-forward Artificial Neural Network structure.

### 2.2.1 Deep Learning

The field of ANNs has a long history, dating back to the 1950's. Perhaps the earliest example of ANNs is the Perceptron algorithm developed by Rosenblatt in 1957 [14]. In the late 1970's, researchers discovered that the Perceptron cannot approximate many nonlinear decision functions, such as the XOR function. In 1980's, researchers found a solution to that problem by stacking multiple layers of linear classifiers (hence the name multilayer perceptron) to approximate nonlinear decision boundaries. Neural networks again took off for a while but due to many reasons, e.g., the lack of computational power and labeled data, neural networks were left out of mainstream research in late 1990's and early 2000's. Since the late 2000's, neural networks have recovered and become more successful thanks to the availability of inexpensive, parallel hardware (graphics processors, computer clusters) and a massive amount of labeled data. There are also new algorithms that make use of unlabeled data and achieve impressive improvements in various settings, but it can be argued that the core is almost the same with old architectures of the 1990's.

In a naive definition, ANNs are said to be shallow when they possess a single or few hidden layers with a large number of neurons, and as its counterpart deep networks are composed by multiple layers (Figure 2.5).

When the problem at hand exhibits nonlinear properties, deep networks are computationally more attractive than shallow networks due to being computationally more efficient. It has been observed empirically that in order to get to the same level of performances of a deep network, one has to use a shallow network with many more connections (e.g., 10x number of connections in speech recognition [2]). It is thus much more expensive to compute the decision function for these shallow networks than the deep network equivalences because for every connection we need to perform a floating-point operation (multiplication or addition).

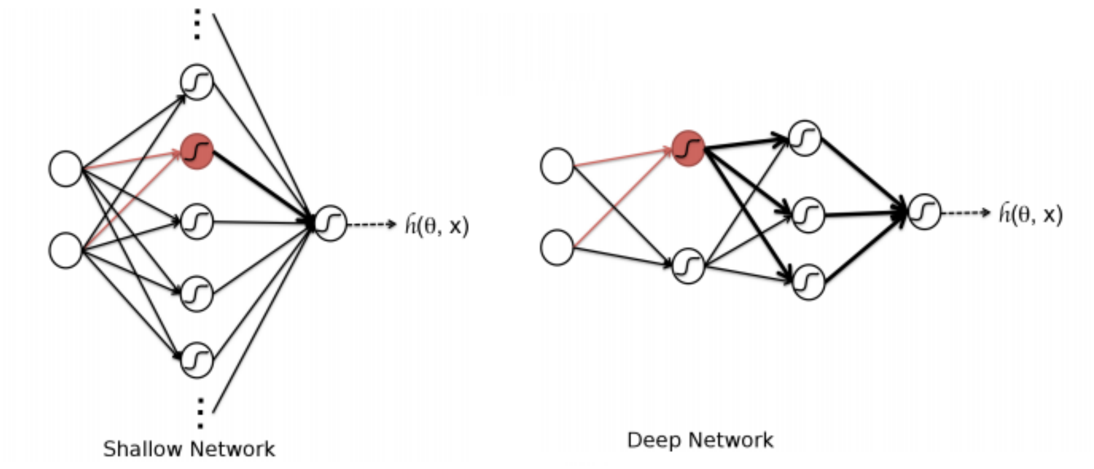


Figure 2.5: Architecture of shallow vs deep networks.

Key results are obtained when the networks are deep and are used on massive amounts of data: in speech recognition [18], computer vision [8], and language modeling [42]. And thus the field is also associated with the name Deep Learning. There are many reasons for such success. Perhaps the most important reason is that neural networks have a lot of parameters, and can approximate highly nonlinear functions.

## 2.3 Autoencoders

In the context of this thesis we will attempt to evolve a specific type of ANN: AutoEncoders (AEs).

An AutoEncoder is an unsupervised neural network which is structured and trained to map the inputs ($x$) to themselves ($x'$), i.e the number of neurons at the input layer and output layer is equal, and thus the optimization goal for the output layer is set to $x = x'$. In this process we convert, implicitly, the input into an intermediary state $h$ by passing the data through the hidden layers in the network which can be seen a learning an higher level representation of the input. When decomposed further we can identify that the AE is made out of two distinct parts: the encoder ($f$) which maps $x$ to $h$, $f(x) = h$, and the decoder ($g$) which maps $h$ to $x'$, $g(h) = x'$, which lets us describe an the AE functionally as $g(f(x)) = x$, as we can observe in figure (Figure 2.6).

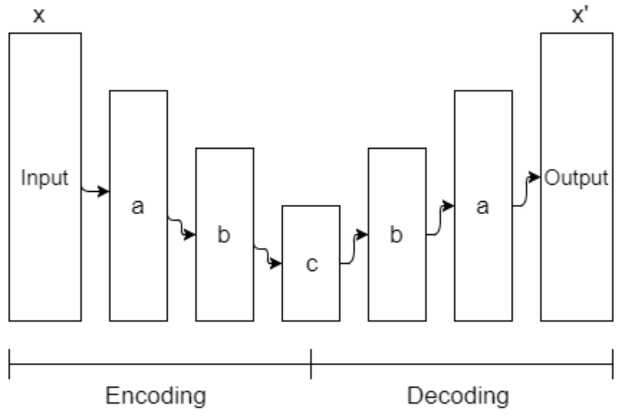


Figure 2.6: Example of a symmetric Deep AutoEncoder

Traditionally AEs possess an hidden layer which has a lower number of neurons when compared to the input which creates a bottleneck shape in the AE, also called an undercomplete AutoEncoder. This restriction is done with the intention of forcing $h$ to represent a compressed representation of $x$ by learning the main properties of the information and their relationships while making it impossible for the network to learn the identity function. We can then throw away the decoder and use the $h$ representation as the input for tasks such as classification problems. This can be seen as a dimensionality reduction method, similar to methods such as Principal Component Analysis (PCA). In fact a simple AE with a linear decoder and mean squared error as the loss function will indeed replicate the results of PCA, since it will map the linear relationship between the principal components of the data. However, AEs can, and usually are composed of neurons with nonlinear activation, which in theory are capable of learning more powerful nonlinear relations present in the data, improving their capacity to find more robust and reliable dimensional reductions.

In recent advances, AEs have generalized the idea of encoder and decoder beyond deterministic functions to stochastic mappings, $Pe(h|x)Pd(x|h)$ [4]. This becomes clearer when we observe sparse Autoencoders. In this approaches, such as the Variational autoencoders (VAEs)[31], the training criterion includes a sparsity penalty $\Omega(h)$ on the code layer h in addition to the reconstruction error, i.e., an AE that has been trained to be sparse must respond to unique statistical features of the dataset rather then simply performing the copying task, which can result in the learning useful features as a byproduct. These approaches enable the use of overcomplete AE encodings, i.e., where the layer h has an higher number of neurons than the inputs, since it will not learn the identity function due to the restrictions imposed. Such encoding have showed improvement of classifications performance in certain applications [20] . When using VAEs, many consider $h$ as being latent vectors, which can easily be manipulated or generated to modify the characteristics of the resulting decoding process, as such AEs have also been studied and applied with success in generative models such as Generative Adversial Networks (GAN) [32].

Up to now we have mentioned the applications of autoencoders as a powerful dimensionality reduction method, compression and use in generative models, but some approaches prefer to take their capacity to rebuild data using the entire AE. One of the most promising uses of this concept comes from the capacity of an AutoEncoder to learn to remove noise from the input when trained with corrupted inputs. These models, also called denoising autoencoders (DAE) [60], are trained using corrupted data from x with some form of noise, having achieved remarkably good results in denoising applications [63]. Additionally DAEs also manage to create good representations [61] in the dimension $h$, since the network is forced to learn useful new features when there is information missing and as such will never learn the identity function.

Another prominent use of autoencoders is the ability to stack multiple simple AEs into a single more complex structure, to support an existing structure, such as weight initialization in Deep ANNs (Figure 2.7). This translates into the creation of Deep Networks (DBN) such as Deep Belief Networks [19][25] as an alternative to the use of Restricted Boltzman Machines [35]. Given an input, into a DBN, it will be passed through the deep structure of the network, resulting in high level outputs. In a typical implementation, the outputs may then be used for supervised classification if required, serving as a compact higher level representation of the data [19].

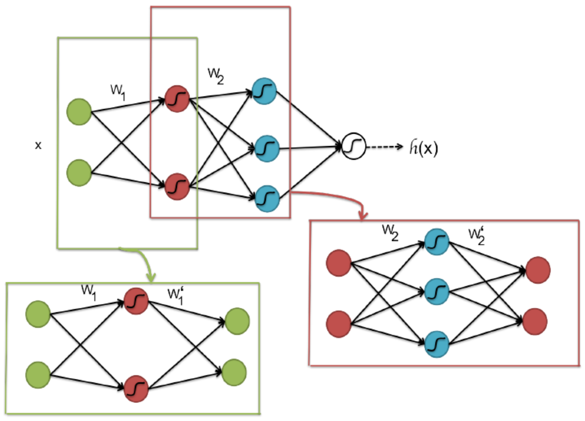


Figure 2.7: Stacking autoencoders to initialize weights [34]

AEs are also capable of using convolutional layers in their structure to create Convolutional Auto Encoders (CAE) [51]. These specialized AE can also be stacked [40] or combined with sparse representations [3] to produce great results. As a matter of fact many of the most interesting results presented by AEs can be observed when we are dealing with deep architectures [33][46], which more often than not, combine the use of DAE [39], CAE [21] and VAE [49] with problem specific configurations.

The purpose of this thesis revolves around exploring the potential of deep networks serving as autoencoders, and what benefits we can extract from using larger encoders spawned from multiple layers and what applications it can have on real world problems. Our motivations are focused on exploring the novelty of using EAs to evolve the structure of a more unorthodox AE, composed by a deep network, which as far a our knowledge goes has never been explored in state of art in such a manner.

In this context we will consider all deep networks that map input $x$ to $x_i$ as explained above, with at least one layer that has lower dimensions than both the input and the corresponding output such as the networks represented in Figure 2.8. In theory the output of the layer with the lowest dimensionality can be seen as a compressed representation of the input.

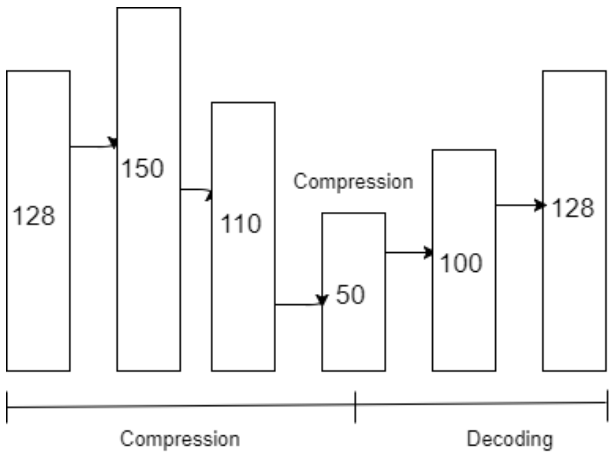


Figure 2.8: Example of a non symmetric Deep AutoEncoder.

## 2.4 Neuroevolution

Evolutionary Computation and Artificial Neural Networks, both spawned from the need to solve problems in the real world by observing the solutions already implemented by nature itself, be it the survival of the fittest or the incredible complexity of human neural systems. As one might expect, there are many situations where both can be combined in order to account for the two main issues that occur when looking at ANNs: how do to design their structure and evolve (train) its weights to account for the desired behaviour. The application of this concept is known as NeuroEvolution (NE).

EAs possess multiple properties that fit both of the problems that need to be solved whilst designing and training a neural network, such as their capacity to search globally, handle infinitely large, non-differential and multimodal search spaces.

With that said, it is easy to distinguish NE approaches based on the aspects that they optimize: The weights [50][15][45] , network topology [17] [52] [56] or both the topology and weights simultaneously [62] [57][59].

One other issue to take into account is the intended depth of the networks. NE that aim to evolve deeper structures tend to have a greater focus on high level feature evolution, such as hyper parameters or resorting to layer-based encodings [58], in contrast to the most common approaches that target the evolution of small networks for very specific tasks. This is a natural shift, due to the significant increase of the search space size when we are evolving the connections [27][12] or nodes [59][44] or weights in deeper and larger networks, which renders some of them impractical for discovering high performing networks.

On the same reasoning, it is unfeasible to directly evolve the weights of the networks, which might reach millions of parameters to optimize. As such when the training of the networks is optimized using EC usually only the hyper-parameters are tuned and the networks trained using gradient-descent algorithms [41] [58].

The idea of optimizing hyper-parameters for deep networks is further extended in Coevolution DeepNEAT (CoDeepNEAT) [41], where the structure of the network is searched combining the ideas behind Symbiotic, Adaptive Neuro-Evolution (SANE) [44] and NeuroEvolution of Augmenting Topologies (NEAT) [57]. Two populations are evolved in simultaneous: one of modules and another one of blueprints, which specify the modules that should be used. Learning and data augmentation parameters are also optimised.

On the topic of evolving autoencoders, as far as our research went, only once did we stumble upon the use of EC to evolve the structure and weights of a single layered Autoencoder. In this study [30], the authors explore the possibility of evolving basic autoencoders with a single hidden and fully connected layer. By initializing multiple autoencoders and training them using backpropagation. Using each node and their weigths as gene allows them, to combine multiple networks by exchanging their nodes, or modifying them by adding nodes from other encoders, which are once again trained using BP. The shortcoming of this approach come from being tailored to work on single layered fully connected AEs which won't be our focus in this work. As such we expect our approach to be a novel method for evolving deep autoencoders.

# Chapter 3

# Automatic Evolution of Deep Autoencoders

In the previous section we severed and described all the parts that serve as the foundation for this work, namely Evolutionary Computation, Neural Networks, NeuroEvolution and autoencoders. While any of these topics presents a moderate amount of complexity, and some time was required to fully understand each of them, to the best of our knowledge this is the first approach focused on the automatic evolution of Deep Topologies for autoencoders by means of an EA. Only one case was found that makes use of EC implicitly to evolve autoencoders [30] where the algorithm evolved the structure and weights of a basic AutoEncoder with a single fully connected hidden layer, but due to its specificity it cannot scale to deep structures. Additionally our approach does not intend to restrict autoencoders to a symmetric structure which is the most common in the literature; this should provide us with interesting insight into the potential of this approach.

## 3.1 AutoEncoder Implementation

To get us started, we implemented simple autoencoders using the KERAS framework [7] using Tensor Flow [1] as our primary back-end. At this phase we assumed simple symmetric topology ANNs in a traditional "V" shape, composed by multiple layers of fully connected neurons, where the left side of the network (the structure responsible for the encoding) is symmetric to the right part (responsible for the decoding) and at the middle is the narrowest part of the network that defines the amount of compression to be executed(Figure 2.6). This is implemented by converting a list containing the first part of the layers in a list similar to [a b c], and then creating the appropriate tensor model and stacking the appropriate layers in the correct order, making sure to add the input and output layers to complete the structure. In an initial approach all the intermediary networks possessed the activation function ReLU and were trained using GPU assisted BP. The loss function to be minimized was the binary cross entropy [16](BCE) between the original and reconstructed data. As per hardware, most of the tests explained further were executed in servers with GTX 1080 Ti GPUs.

Our initial concern was to find a way to compare the performance of multiple autoencoders by using a generic approach. For this goal we randomly handpicked similar autoencoders with a symmetric structure composed of 8 fully connected layers. This correspond of a structure composed by 3 components such as [a b c]. The AEs are trained using the training samples belonging to the MNIST dataset[36]. A simple value comparison does not present interesting results by itself, since we do not know how the variation in the loss value translates into the actual images. For this purpose, we attempted to recreate intervals of error values, and visually explore the images that were being reconstructed to have an idea of which values corresponded to images that were correct and easily recognized by a human (Figure 3.1).

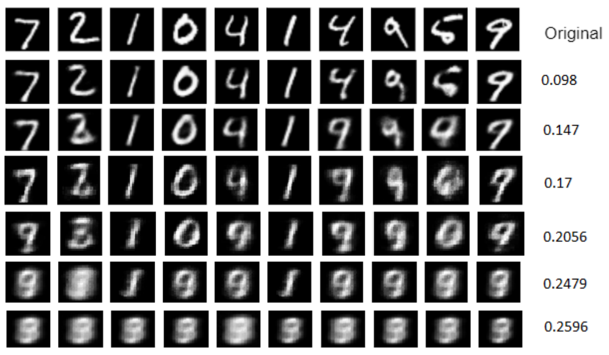


Figure 3.1: Visual inspections of error intervals

We can thus see that our autoencoders that manage to reach error values above **0.18** were incapable of producing the result we intended as the images presents mostly a blur that on occasion appears to represent the average result of the dataset, however when the error values are bellow 0.14 the results were clearly recognizable with the expected noise from the imperfect reconstruction expected of this approach. On a second observation, it is very difficult for a human eye to recognize a significant difference between values that are bellow 0.1 . Taking these results into account we made the informed assumption that a deep AE trained using back propagation is able to produce the results that we expected, and reproduce with fidelity the input images when the error values are bellow a threshold. We could then expect these error values to be a reasonably good approach to measuring the quality of the structures, or as the fitness in an EA.

The results obtained so far indicate that our approach is valid but some questions and curiosity inspired new tests. Since we are dealing with deep networks it is impossible to ignore the cost and importance that the training of these structures impose on the overall result of the network, regardless of how simple or complex the structure is. Additionally the training process has multiple parameters that need to be set, which many times can make or break the results. On this few initial tests our neurons were set to the ReLU activation function with Adam as the training optimization algorithm and the training epochs were changed accordingly to attempt to generate error values in each interval, keeping the smaller layer as small as possible to attempt to recreate a good compression.

Taking into account the scope of this thesis, a full study of each parameter and their impact on the performance would be too time expensive. However as we have mentioned before, our EA would most likely need to train and obtain the results of over a thousand networks and since we know that our networks need to avoid having their performance bellow an expected threshold to result in good results, we need to balance the time spent during training and the relative performance gained. With this in mind further tests were run in an attempt to understand which optimizer and how many epochs would be enough to obtain satisfactory results. The Table 3.1 presents results from six optimizers available in the framework, trained during 2 different amounts of epochs on a single network structure [626,271,23]. The values presented correspond to 20 runs per configuration.

Observing the average performance shown by each optimizer we opted to use the adam

| Optimizer | Epochs | MEAN | STD |
|---|---|---|---|
| adadelta | 7 | 0.1694 | 0.004053 |
| adadelta | 15 | 0.1363 | 0.002206 |
| adam | 7 | 0.0933 | 0.002220 |
| adam | 15 | 0.0858 | 0.002135 |
| nadam | 7 | 0.1052 | 0.001830 |
| nadam | 15 | 0.0948 | 0.001520 |
| adamax | 7 | 0.0970 | 0.002127 |
| adamax | 15 | 0.0865 | 0.001847 |
| adagrad | 7 | 0.1185 | 0.002551 |
| adagrad | 15 | 0.1060 | 0.002907 |
| rmsprop | 7 | 0.1107 | 0.002147 |
| rmsprop | 15 | 0.0990 | 0.001125 |

Table 3.1: Initial comparison between optimizers and epochs

optimizer [26] with the reference values (learning rate = 0.001, beta_ = 0.9, beta_2 = 0.999, epsilon = $1e^8$, decay = 0), trained during 7 epochs as it is seen that more epochs do not lead to an increase in performance. Using 7 epochs seems to be the most appealing choice since when using adam the networks consistently surpass acceptable values with this small number of training epochs, and when time execution is important we accept the loss in the final performance that it might bring as we assume that this difference in error will not affect the EA performance.

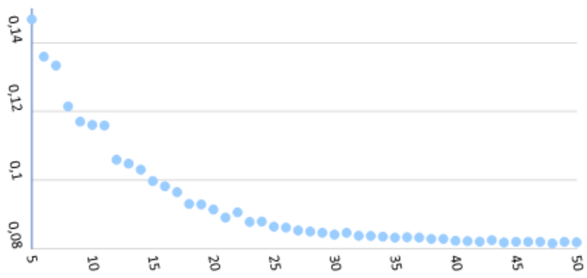
On another note, we speculate that an AE does not need to be symmetric to compress and rebuild the data, so we created multiple non symmetric autoencoders, which we trained to obtain the error values and images, that resulted in reconstructions that were on par with the symmetric autoencoders we have tried so far. This might be due to the nature of the MNIST dataset but this approach appears to be valid under these circumstances.

Since our main purpose for evolution at this point, was to attempt to create a good compression or effective pre-processing of the information it was also necessary to study the effect of the dimensionality size of $h$ in this context. With this in mind we executed a simple test that consists in using a fixed structure and modifying the size of the compression layer h. The networks structure was [610, 1000, 100, 400, h, 740, 500, 710]. The results shown on Figures 3.11 are an average of 20 executions per configuration.

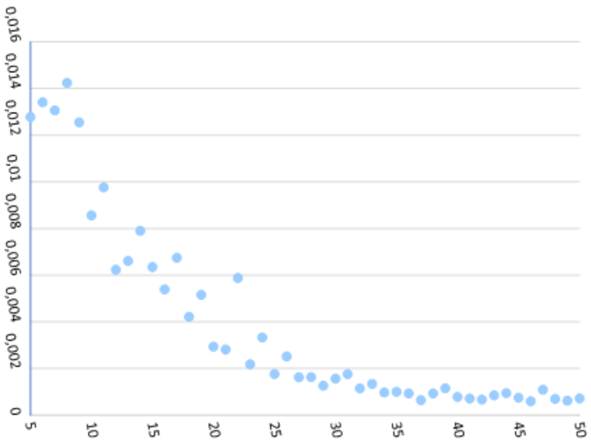
While observing the results, and although we cannot assume that this effect is universal among all networks and problems, we can observe that as we expected the performance does decrease significantly at lower dimension sizes (more or less from 5 to 14) and increases as the compressing layer becomes larger (Figure 3.2a).This may be due to the nature of the data-set having 10 different classes, but it would seem that the optimal compression to be found by our EA will likely be above these values. In Figure 3.2b we can also verify that the standard deviation from the test samples increases as the number of neurons in the compression layer decreases, and the AE struggles in the training process.

## 3.2 EA Implementation

Given the results we obtained during the first batch of tests, we were confident that we were able to create by hand networks capable of fulfilling our expectations, but we still had to decide how we were going to evolve the structures of our autoencoders.



(a) Loss value



(b) Standard Deviation of Error

Figure 3.2: Impact of the compression value (x) on the error on the test set (y)

Due to how our networks were structured in previous tests, the direct translation into an evolutionary approach would orbit around evolving the size of the layers and number of layers in an AE, where we would seek to evaluate their fitness based on the loss function after train. Keeping in mind that our objective is to evolve networks to have the best features present in the layer $h$ we must take into account the size of the compression being applied. What came forth was a textbook evolutionary algorithm that would serve as a basic prototype for the work to come.

In this first approach we opted to use an indirect representation that translates the symmetric V structure, as used in the initial tests to represent and build our autoencoders. Since we are dealing with a minimalist indirect representation in order to test our concept, the representation selected was a simple $n$ sized vector of integer values, where $n$ is the number of layers in the decoding side of the AE and each value represents the number of neurons in that layer since we are assuming a symmetric topology. As an example, the genotype [a , b , c] would map to the AE [INPUT a b c b a OUTPUT] where a,b and c represent the size of the corresponding layer, with the restriction that $a > b > c$ to generate networks with the typical funnel structures of AEs.

### 3.2.1 Fitness Function

Recall that the purpose of this EA is to evolve the structure of autoencoders in order to obtain the topologies that result in a greater compression, while maintaining a low error in the representation. Consequently, our **fitness function** can be obtained by assigning a quality to each individual based on their performance after a short period of training and the amount of compression, which is given by the following equation:

$$\alpha E + \beta C, \tag{3.1}$$

where $E$ is the error given by the structure on the test set, after being trained on the framework explored in the initial tests, and $C$ is the size of the narrowest layer. $E$ and $C$ are multiplied by the weight factors $\alpha$ and $\beta$, respectively.

With the problem at hand we can additionally define a threshold $T$ representing a value in which our compression is good enough, and any improvements above this error can be ignored in the evolution process without the risk of compromising the end results.

This improvement could potentially lead to the improvement of the structure and avoid potentially local optima or bloat. The final fitness function is given by equation 3.2.

$$\alpha(min(E, T)) + \beta C \tag{3.2}$$

### 3.2.2 Mutation Operators

As the mutation operator, a simple integer gaussian mutation was used, where an individual has a probability of $m$ to have one of its layers size modified by adding a value taken from a gaussian distribution with mean = 0 and deviation = 5. Using this method it is likely that the mutated individual will be similar to the original, making use of exploitation, but it is also possible for the same mutation to have a greater impact which results in a greater exploration of the fitness landscape.

### 3.2.3 Crossover Operator

The crossover methods implemented in this prototype were a simple one point cross and a random matrix crossover (see Figure 3.3). In the first one a cut position is selected in the representation, which divides each parent into two distinct halves. As a result each offspring is created by joining opposing halves from both parents. On the random matrix crossover, a binary vector is randomly generated in order to create the mapping of what genes are passed to the offspring.

These operators in particular were chosen due to their capability of producing good offspring combinations due to the limited size of the representation.

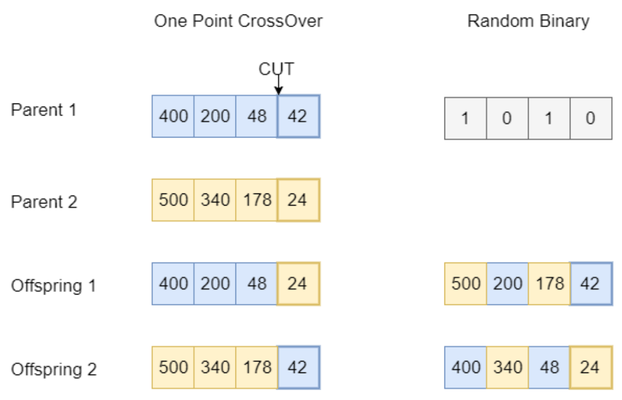


Figure 3.3: Example of the application of the used crossover operators.

### 3.2.4 Parent Selection

The parent selection implemented in this prototype was a simple tournament selection. In this method we randomly select a sample of $tSize$ individuals from the population, selecting the one with the highest fitness to be used as a parent. This process is then repeated until the required number of parents have been selected.

### 3.2.5 Survivor Selection

The selection of individuals that form the next generation is an elitism-based method. In this approach we guarantee that a percentage ($e$) of the best individuals chosen as parents are retained to the next generation while the remaining individuals are chosen from the most fit offspring created.

### 3.2.6 Initialization

In order to initialize a population in this approach, four different network creation methods were developed:

**Random decreasing generation** – We select sequentially n (the number of encoding/decoding layers) while making sure that each selected number is lower than the previous one.

**Evenly space creation** – We divide the space into n parts, generating a random number for the first layer then adding the input/n to this value to generate the remaining layers.

**Region generation** – We divide the space into n regions, as in the previous method, and proceed to generate a random number for each region.

**Random generation** – The simplest method, n random numbers are selected and ordered in order to generate an individual.

To generate a population one of the above methods is selected and used multiple times to generate the initial population, no heuristics are used at the moment.

### 3.2.7 Stop Criteria

Our algorithm will run until a predefined number of generations have passed or the fitness of the best individual has stagnated, i.e, not improved for a given number of generations. This stagnation value was set as 7 generations as default.

## 3.3 Evolutionary Experiments

In order to test this approach we perform 10 evolutionary runs, with the parameters of Table 3.2. The obtained results are reported in Table 3.3 and Figure 3.5.

| crossP | elitism | mutP | PopSize | tSize | trainE | threshold |
|---|---|---|---|---|---|---|
| 0.5 | 0.02 | 0.7 | 100 | 2 | 7 | 0.09 |

Table 3.2: Evolutionary Approach Parameters

In this experiment the number of generations needed to imply stagnation was set to 7, with $\alpha = 1000$ and $\beta = 0.5$, as the fitness function parameters. The initial population was generated by using the random decreasing generation method mentioned in the previous section.

| Run Number | Fitness Max | Individual | Generations | Compression | Error |
|---|---|---|---|---|---|
| 1 | 105,4336793 | [626, 271, 23] | 26 | 23 | 0,093933679 |
| 2 | 105,2465687 | [636, 321, 29] | 17 | 29 | 0,090746569 |
| 3 | 104,9859118 | [634, 321, 29] | 30 | 29 | 0,090485912 |
| 4 | 105,624386 | [668, 252, 27] | 23 | 27 | 0,092124386 |
| 5 | 105 | [697, 515, 30] | 32 | 30 | 0,09 |
| 6 | 105 | [663, 287, 30] | 30 | 30 | 0,09 |
| 7 | 105,3969231 | [633, 443, 29] | 22 | 29 | 0,090896923 |
| 8 | 105,3466392 | [703, 318, 29] | 16 | 29 | 0,090846639 |
| 9 | 104,2987375 | [736, 231, 27] | 40 | 27 | 0,090798737 |
| 10 | 105,8262697 | [666, 300, 30] | 14 | 30 | 0,09082627 |
| Mean | 105,2159115 | | 25 | 28,3 | 0,091065912 |
| Std | 0,403106653 | | 7,771743691 | 2,051828453 | 0,001107277 |

Table 3.3: Initial Experiment Results

We can observe in the results (Figure 3.5 and Table 3.3) that the EA is indeed evolving the overall population and producing individuals that possess a good compression value while reaching performance values very close to the set threshold. By observing the individuals created we can see a pattern where the resulting networks layers are more or less evenly spaced, however this cannot be fully verified in this test since we did not plan to analyze the evolution of each layer.

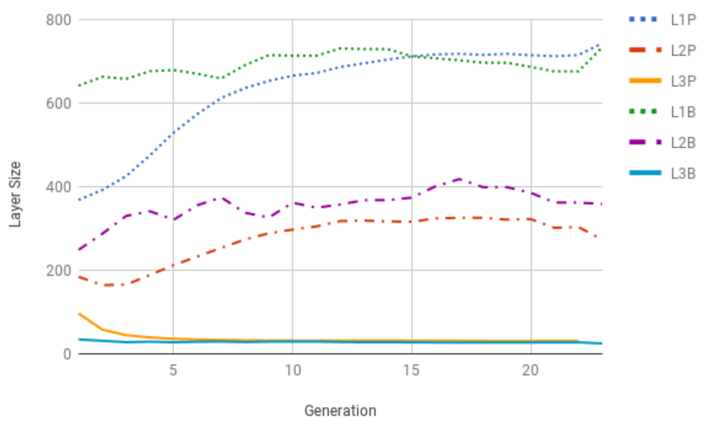


Figure 3.4: Evolution of compression layer size. Results are average of 10 independent runs.

As such, to cover this last point, and to attempt to observe the behaviour of the algorithm with slightly different parameters another test was run with the same basic assumptions of the first, as can be observed in Tables 3.4, 3.5, and in Figure 3.5.

| crossP | elitism | mutP | PopSize | tournSize | trainE | optimizer |
|---|---|---|---|---|---|---|
| 0.8 | 0.01 | 0.5 | 100 | 2 | 7 | adam |

Table 3.4: Evolutionary approach parameters.

As we were expecting, the change in both the mutation and crossover probabilities did not impact the overall quality of the results, as these were similar on both tests. In Figure 3.5

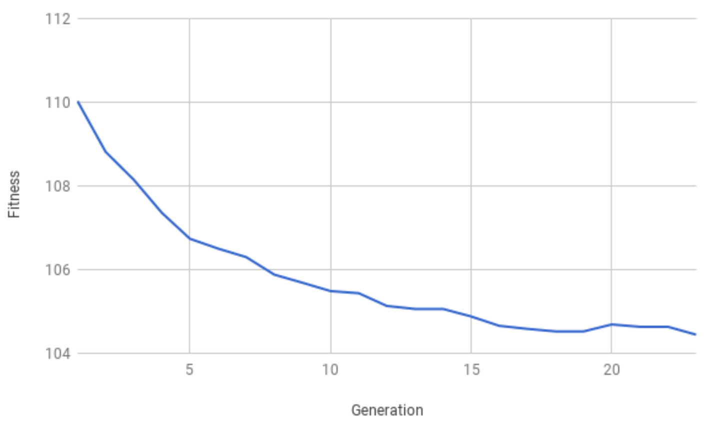


Figure 3.5: Best solutions fitness evolution.

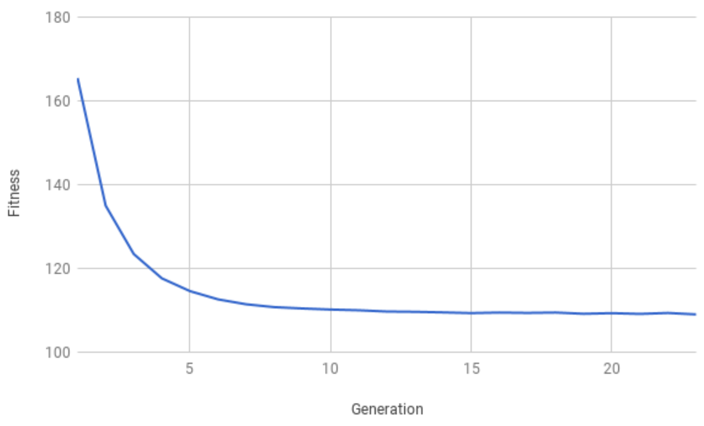


Figure 3.6: Populations fitness evolution.

| Run number | FitnessMax | Individual | Generations | Compress | Error |
|---|---|---|---|---|---|
| 1 | 104,3522324 | [667, 285, 28] | 22 | 28 | 0,090352232 |
| 2 | 105,5781388 | [741, 409, 31] | 15 | 31 | 0,090078139 |
| 3 | 104,4446218 | [732, 358, 25] | 23 | 25 | 0,091944622 |
| 4 | 104,8861725 | [731, 514, 28] | 17 | 28 | 0,090886172 |
| 5 | 104,8552107 | [718, 455, 26] | 20 | 26 | 0,091855211 |
| 6 | 105,1016529 | [626, 442, 30] | 22 | 30 | 0,090101653 |
| 7 | 105,0912969 | [734, 296, 27] | 16 | 27 | 0,091591297 |
| 8 | 105,2155912 | [712, 152, 27] | 15 | 27 | 0,091715591 |
| 9 | 103,8599937 | [736, 452, 27] | 19 | 27 | 0,090359994 |
| 10 | 106,0392149 | [607, 492, 31] | 11 | 31 | 0,090539215 |
| Mean | 104,9424126 | | 18 | 28 | 0,090942413 |
| Std | 0,592265126 | | 3,660601044 | 1,949358869 | 0,000718581 |

Table 3.5: Second experiment results.

we can visually observe the evolution of each layer in the AE, which allows us to observe that the population is indeed evolving and converging towards the best solution while evolving all three layers, in what appears to be an evenly spaced structure as we observed in the previous test. That being said, we can verify that our EA has managed to evolve successfully the structure of these simple, fixed size autoencoders to an interesting result. This is nevertheless quite restricting as far as possible structures go, as we would like to be able to make our EA responsible for deciding the number of layers of the evolved solutions.

### 3.3.1 EA Changes

Taking the algorithm one step further we enabled the structures to be able to change in size by adding two additional mutation functions, which can add a random size layer or remove one of the existing layers. This does not, remove the symmetry restriction from the EA.

Although these changes were very subtle, we still feel like it would be interesting to observe the behaviour of the algorithm, without any parameter changes other than slightly adjusting the mutation probabilities which resulted in the results seen bellow in the Figures 3.7,3.9 and 3.8 that maps the average among all evolutionary runs.

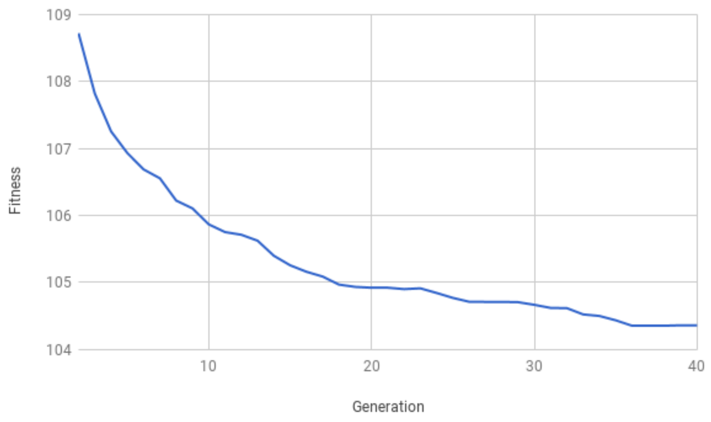


Figure 3.7: Evolution of fitness of the best solutions across generations. Results are average of 10 independent runs.

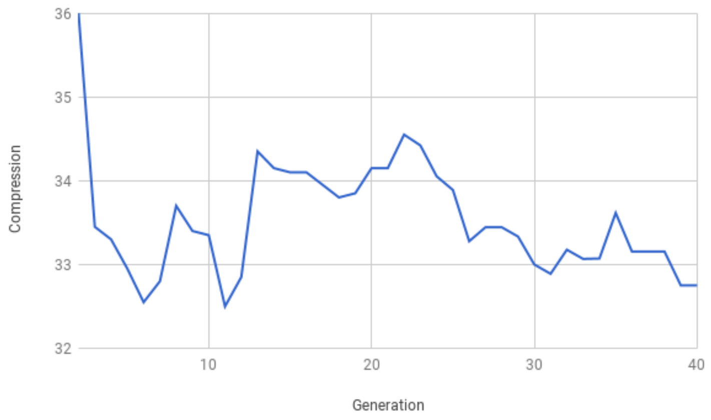


Figure 3.8: Compression of the best individuals per generation.

As one might have expected, the algorithm tends to evolve structures that are smaller than the ones imposed by our previous algorithm. This might be so because we are dealing with symmetric autoencoders, and incrementing a layer to the structure implies adding

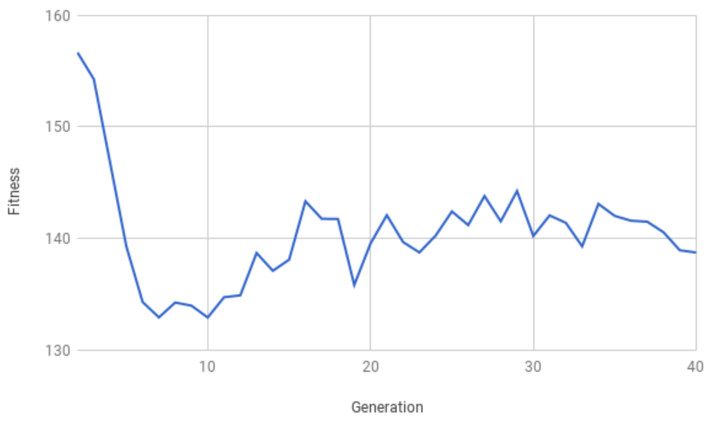


Figure 3.9: Fitness of the population per generation.

two layers to the final AE. Since we have limited time to train the networks, smaller networks without fine tuning will have a slight advantage against their deep counterparts. When we are only trying to evolve based on the loss error we are very likely to be left with a basic network which is not our initial objective.

On a further study on the matter we once again modified the algorithm to evolve any kind of structure by removing the symmetry constraint out of the picture and making the necessary adaptations:

- The crossover operator no longer sorts the networks and we now mainly use a modified version of the binary crossover which takes into account that the size of two layers might not match. Based on this second occasion, the new networks are formed by appending the next available gene present in the binary selection array.Consequently, it is possible that the parents and the offspring differ in the number of layers as observable in Figure 3.10.

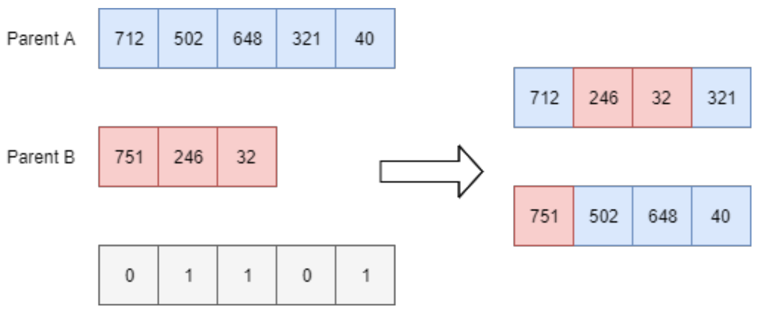
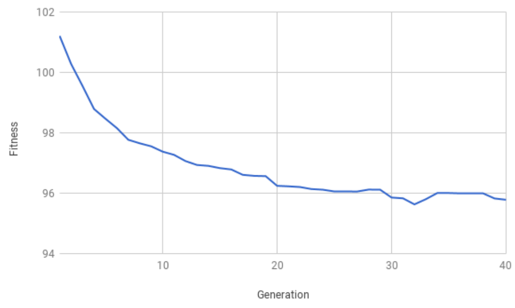


Figure 3.10: Updated binary crossover.

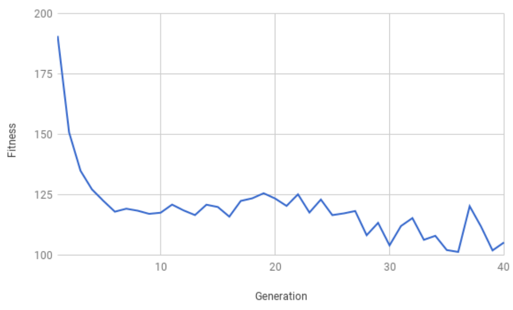
- The network initialization now generates structures with a random size, within a given range. In practice the layers were commonly restricted to a size between 2 and 6 hidden layers.

- A swap mutation was implemented, which simply swaps the values in 2 genes along with a reset mutation that restarts the value of one gene.

With this changes, a few more tests were executed to attempt to visualize the EA behaviour, where we maintained the previous configurations, but lowered the error threshold to 0.08 to observe what results were obtained when the EA attempts to maximize the loss function over the compression.

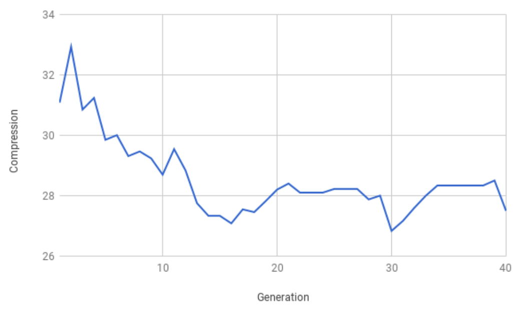


(a) Average best individual fitness per generation.

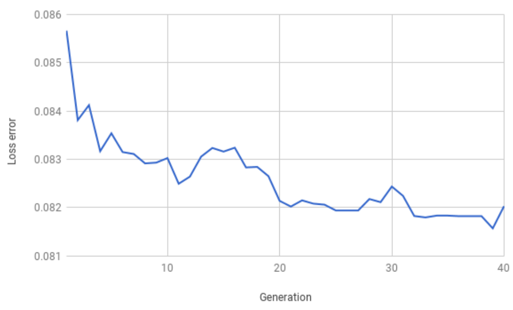


(b) Average population fitness per generation.

Figure 3.11: Fitness evolution averaged over 10 independent runs.



(a) Average best individual compression per Generation



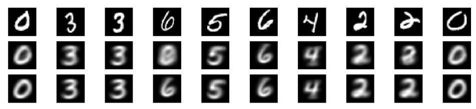
(b) Average best individual error per generation

Figure 3.12: Evolution of fitness parameters averaged over 10 independent runs.

The results of these experiments show us that our algorithm can pick networks capable of achieving error values below 0.09 (Figure 3.12b) with some ease when not constraint by the threshold while still maintaining compression values similar to what we had seen before in our previous EA setup. Observing the size of the compression and error side by side (Figure 3.12) we can see that the EA in its current form manages to evolve both the compression and the error simultaneously across the generations. On this tests the resulting networks tend to have a genotype of size 4, corresponding to an AE with 6 layers on total. It is also important to note that on its current state the algorithm tends to produce networks with uneven encoder and decoder parts, namely the decoder size on most cases far exceeded the encoder, which we might have to pay attention when fine tuning it further.

### 3.3.2 Final Approach

At this point we had created an EA capable of evolving the structure of asymmetric autoencoders as intended, but we have had yet to link our approach and results with our goal, to which we have to assess the quality of autoencoders as a means to pre-process the information without losing the minimum needed for classification. For this we would need to make use of more than the reconstruction error of our current approach. As a result we


(a) Class Average Target

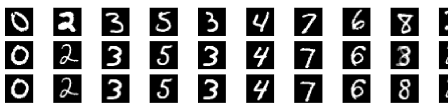

(b) Select numbers Target

Figure 3.13: Results of the reconstruction with the target digits updated. On the top row we have the original digits, on the middle the reconstructions and on the bottom the targets

would need to possess concrete proof that our compression translates into classification performance and then on a further test validate these results.

As such, we modified our EA by changing the fitness function to include the classification performance of a classifier on the compressed data, rather than the compression value. To accomplish this, every time an individual is evaluated we construct the AE, which is trained using the training set in the same fashion as previous tests to minimize the error of reconstruction, and then proceed to remove the decoding part of the AE and compress the entire entire dataset which is then used to train a simple MLP in order to evaluate its accuracy on the test set. This MLP is a basic classifier with 1 hidden layer of size 100 and ReLU activated neurons.

The fitness thus becomes:

$$\alpha(1 - A_c) + \beta C, \tag{3.3}$$

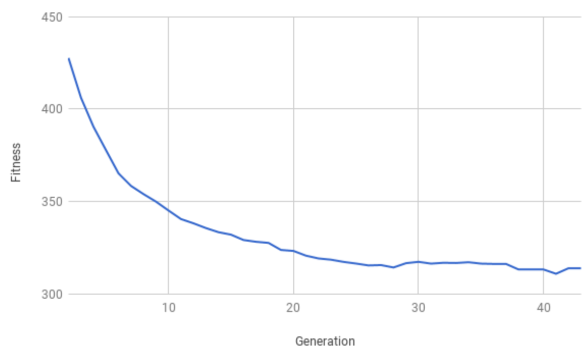
where $A_c$ is the accuracy provided by the classifier. On the previous tests our structures seemed to create larger decoders, which feels like it might be counterproductive when we attempt to create good representations for the information at $h$ to improve classification instead of minimizing the reconstruction error. Although this might not occur since we no longer take this into a account, we still provided the fitness function with an additional parameter $Dp$ which stands for the decoder proportion of the autoencoder which we wish to minimize. This value is given by the (total number of layer) / (number of decoder layers) which brings us to the final fitness function as follows:

$$\alpha Ac + \beta h + \gamma Dp \tag{3.4}$$

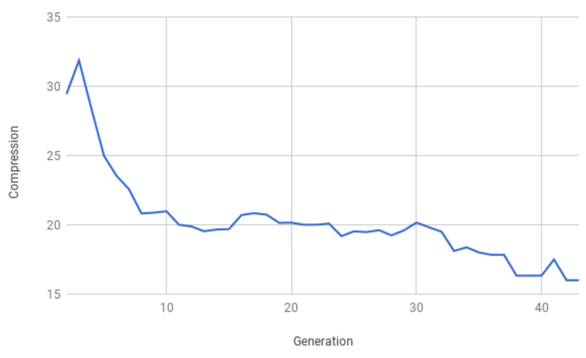
Even though these changes to the EA would take us closer to our objective, it would be interesting to make use of the labels available to maximize the changes of our autoencoders collecting essential information of each digit identity. On our original approach we attempt to transform $x$ into $h$ while attempting to recreate $x$ in the output. In order to make use of the class information, we will train the AutoEncoder to map $x_c$ to $g_c$ where $g_c$ is a number image that represents the class c. With this modification we hope that our compressed representation will inevitably present the information that represents each class identity thus improving the changes of increasing the accuracy of the classifiers.

Similar to what we have done previously, simple manual tests were executed to prove if this concept is promising (Figure 3.13). These initial results point out that the autoencoders can reliably compress and construct the pretended images that represent the class, when used on the MNIST data while also being able to use a small dimension size for $h$. This remained true when testing with different targets, including hand picked numbers (Figure 3.13b) to represent each class or simply using the class average (Figure 3.13a).

Using the updated EA, capable of evolving the structure of asymmetric AEs, with the updated fitness function, new experiments were planned and conducted with the parame-



(a) Fitness Values of the best individuals.



(b) $h$ size of the best individuals.

Figure 3.14: Results from the best solutions found per generation averaged over 30 runs.

ters detailed in Table 3.6. 30 runs were executed, while dividing the MNIST dataset three disjoint sets: train, test, and validation, of sizes 60000, 5000, and 5000, respectively. The train set will be used to train both our autoencoders and, after compressed our classifiers in the evolutionary process, while the validation set is used for posterior validation experiments.

| **Parameter** | **Value** |
|---|---|
| Crossover Probability | 0.8 |
| Crossover Function | Binary Crossover |
| Increment Mutation Probability | 0.15 |
| Decrement Mutation Probability | 0.15 |
| Swap Mutation Probability | 0.05 |
| Reset Mutation Probability | 0.05 |
| Gaussian Mutation Probability | 0.6 |
| Elitism Percentage | 0.02 |
| Population size | 100 |
| Tournament size | 2 |
| Min Number of layers | 2 |
| Max Number of layers | 6 |
| Alfa | 20000 |
| Beta | 2 |
| Gamma | 10 |

Table 3.6: Evolutionary Approach Parameters

As far as the performance of the EA goes (Figure 3.14), we observe that the algorithm is able to evolve the initial population, finding better solutions as generations pass until the runs are interrupted by stagnation, and both the parameters being studied are being successfully minimized. Even with a very small incentive, the networks created by the algorithm tend to have a much larger encoder than decoder. We also observed that the resulting structures are contain, on average, 4 hidden layers.

The average compression obtained using this method was around 16 and the error values were consistently above 0.98 accuracy, compared to the 0.95 obtained using the entire dataset on this particular neural network. This improvement,when compared to the results of the NN with the full dataset, represents a significant shift as the returned a value of p=9.42976e-60. This test was the result of comparing 30 different AE results and 30 runs with the original dataset.

To attempt to validate our results, and to assess whether our algorithm is capable of speeding up different small solutions while improving their overall performance, 7 different classifiers were chosen:

- Neural Network (**NN**) with one 100 ReLU activated neurons on the hidden layer. (This solution was used in the EA)
- Adaboost (**AB**); estimators = 50, Learning rate 1
- Decision Tree (**DT**); max_depth = 5
- Random Forest (**RF**); max_depth = 5, n_estimators=10, max_features = 1
- Naive Bayes (**NB**)
- **SVM**; linear kernel and C = 0.025
- KNeighbors Classifier (**KN**); number of neighbours = 3

For all the algorithms above, the sklearn [47] package implementation was used with most parameters set to their default values. These were chosen due to their ease of use and simplicity which goes along with our intent to improve the speed and performance of simple and easy to use solutions.

For this test we used all of the 30 autoencoders obtained during the previous runs of the EA to compress the dataset which was then used to train each of the classifiers. The classification target for these networks was the validation set which was set apart during the EA execution.

| Classifiers | NN | AB | DT | RF | NB | SVM | KN |
|---|---|---|---|---|---|---|---|
| original | 0.9565 | 0.7299 | **0.6747** | 0.6089 | 0.5558 | 0.9445 | 0.9705 |
| compressed | **0.983** | **0.7587** | 0.6177 | **0.98** | **0.9827** | **0.9843** | **0.9833** |
| pvalue | **9.42 e-60** | **0.0323** | 0.5218 | **9.59 e-59** | **3.04e-59** | **6.21 e-60** | **1.78 e-56** |

Table 3.7: Classifier performance comparison with and without using compression. These results are averaged over the result of 30 AEs. The pvalue are the result of a independent Mann Whitney.

| Classifiers | NN | AB | DT | RF | NB | SVM | KN |
|---|---|---|---|---|---|---|---|
| original | 25.7 | 46 | 3.7 | **0.164** | 0.7 | 366 | 561 |
| compressed | **12.2** | **0.7** | **0.617** | 0.55 | **0.02** | **1.2** | **1.4** |

Table 3.8: Execution time in seconds, to train the classifier and to test on the validation set. These results are averaged over the result of 30 AEs.

These results show a clear pattern, where the majority of the algorithms tested have significant improvements on the time they take to train and evaluate the tests, which was to be expected due to the significantly reduced dimensionality of the dataset. We also observed that on nearly all the tested classifiers, the resulting accuracy with the compressed dataset managed to improve their performance, on some cases, such as the Naives Bayes and Random Forest we saw a significant leap in the classifiers final accuracy.

With the current results of these tests, we are able to observe that autoencoders selected by our EA can be used with success to aid the classification of problems in a accurate and efficient manner. The EA was able to successfully evolve the structure of the networks, maximizing the classification gain by using the autoencoders while demanding that the data is compressed as much as possible which minimizes the time needed to train and test the classifiers to obtain good results.

This opens a large window of options as to how this can be used in real applications. The reduced training times without the expense of the classification results clearly aids time sensitive operations, which leads us to believe it can be used to optimize the information being processed in real time from visual sensors or other similar tasks. As a matter of fact, reducing the complexity of the problem also enables lower specs systems to be able to reliably solve the problems in usable time, leaving training of the autoencoders is left to a more robust system.

Chapter 4

# Conclusions and Future Work

During the duration of this thesis, we iteratively built an EA that is capable of evolving deep structures of autoencoders. This approach, as far as our knowledge goes has never been done before, and even though the scopes of the tests for our final approach is rather small, it presents interesting results that may spark the interest of the community or the execution of more tests in the matter.

This takes us back to our final solution, where we implement a semi-supervised training of the autoencoders in order to maximize its compression while retaining the information that makes each class recognizable. We then completed this approach with an EA capable of evolving the number of layers and their respective size taking into account the improvements on the classification error of a Neural Network classifier. Once the resulting autoencoders were tested, using different basic classifiers we saw clear improvements in the classification performance on most of them and a significant improvement on their training times.

We have thus reached our initial purpose of developing an evolutionary framework that is capable of reliably returning deep autoencoders that are able to consistently improve other solutions performance.

It would be interesting to test on future work, how these kind of approach behaves when dealing with other types of data, or harder bench-marking datasets. On a different note, different aspects of the network could be taken into account such as the possibility to evolve the weights of the autoencoders, or implementing the use of different autoencoders such as the use of sparsity constraints, adding noise in the training process or enabling convolutional layers into the evolutionary process. This could lead into a more versatile approach that enables for more diverse solutions to be created.

And as any scientific work, one would hope that it is possible to translate our research into solutions for real life problems down the line. We have only scratched the surface of what this powerful structure and evolutionary approach can bring to the table, and with time I think this structure will be able to prove that purely unsupervised or semi-supervised autoencoders will present themselves as the solution for many problems to come.

# References

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

[2] Jimmy Ba and Rich Caruana. Do deep nets really need to be deep? In *Advances in neural information processing systems*, pages 2654–2662, 2014.

[3] Moez Baccouche, Franck Mamalet, Christian Wolf, Christophe Garcia, and Atilla Baskurt. Spatio-temporal convolutional sparse auto-encoder for sequence classification. In *BMVC*, pages 1–12, 2012.

[4] Yoshua Bengio, Pascal Lamblin, Dan Popovici, and Hugo Larochelle. Greedy layer-wise training of deep networks. In *Advances in neural information processing systems*, pages 153–160, 2007.

[5] Hans-Georg Beyer and Hans-Paul Schwefel. Evolution strategies–a comprehensive introduction. *Natural computing*, 1(1):3–52, 2002.

[6] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[7] François Chollet et al. Keras, 2015.

[8] Dan C Ciresan, Ueli Meier, Jonathan Masci, Luca Maria Gambardella, and Jürgen Schmidhuber. Flexible, high performance convolutional neural networks for image classification. In *IJCAI Proceedings-International Joint Conference on Artificial Intelligence*, volume 22, page 1237. Barcelona, Spain, 2011.

[9] Charles Darwin. *On the origin of species by means of natural selection: or the preservation of favoured races in the struggle for life. By Charles Darwin,...* John Murray, Albemarle Street, 1880.

[10] Leandro Nunes De Castro and Jonathan Timmis. *Artificial immune systems: a new computational intelligence approach*. Springer Science & Business Media, 2002.

[11] Agoston E Eiben, James E Smith, et al. *Introduction to evolutionary computing, Second Edition*, volume 53. Springer, 20015.

[12] Chrisantha Fernando, Dylan Banarse, Charles Blundell, Yori Zwols, David Ha, Andrei A Rusu, Alexander Pritzel, and Daan Wierstra. Pathnet: Evolution channels gradient descent in super neural networks. *arXiv preprint arXiv:1701.08734*, 2017.

[13] David B Fogel. Evolutionary programming: An introduction and some current directions. *Statistics and Computing*, 4(2):113–129, 1994.

[14] R Frank. The perceptron a perceiving and recognizing automaton. *tech. rep., Technical Report 85-460-1*, 1957.

[15] Faustino Gomez, Jürgen Schmidhuber, and Risto Miikkulainen. Accelerated neural evolution through cooperatively coevolved synapses. *Journal of Machine Learning Research*, 9(May):937–965, 2008.

[16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[17] Steven A Harp, Tariq Samad, and Aloke Guha. Designing application-specific neural networks using the genetic algorithm. In *Advances in neural information processing systems*, pages 447–454, 1990.

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

[19] Geoffrey E Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554, 2006.

[20] Geoffrey E Hinton and Ruslan R Salakhutdinov. Reducing the dimensionality of data with neural networks. *science*, 313(5786):504–507, 2006.

[21] Geoffrey E Hinton and Ruslan R Salakhutdinov. Reducing the dimensionality of data with neural networks. *science*, 313(5786):504–507, 2006.

[22] Sepp Hochreiter, Yoshua Bengio, Paolo Frasconi, Jürgen Schmidhuber, et al. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies, 2001.

[23] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[24] John H Holland. Genetic algorithms. *Scientific american*, 267(1):66–73, 1992.

[25] Yuming Hua, Junhai Guo, and Hua Zhao. Deep belief networks and deep learning. In *Intelligent Computing and Internet of Things (ICIT), 2014 International Conference on*, pages 1–4. IEEE, 2015.

[26] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[27] Hiroaki Kitano. Designing neural networks using genetic algorithms with graph generation system. *Complex systems*, 4(4):461–476, 1990.

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

[29] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[30] Sean Lander and Yi Shang. Evoae–a new evolutionary method for training autoencoders for deep learning networks. In *Computer Software and Applications Conference (COMPSAC), 2015 IEEE 39th Annual*, volume 2, pages 790–795. IEEE, 2015.

[31] Anders Boesen Lindbo Larsen, Søren Kaae Sønderby, Hugo Larochelle, and Ole Winther. Autoencoding beyond pixels using a learned similarity metric. *arXiv preprint arXiv:1512.09300*, 2015.

[32] Anders Boesen Lindbo Larsen, Søren Kaae Sønderby, Hugo Larochelle, and Ole Winther. Autoencoding beyond pixels using a learned similarity metric. *arXiv preprint arXiv:1512.09300*, 2015.

[33] Quoc V Le. Building high-level features using large scale unsupervised learning. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 8595–8598. IEEE, 2013.

[34] Quoc V Le et al. A tutorial on deep learning part 2: Autoencoders, convolutional neural networks and recurrent neural networks. *Google Brain*, 2015.

[35] Nicolas Le Roux and Yoshua Bengio. Representational power of restricted boltzmann machines and deep belief networks. *Neural computation*, 20(6):1631–1649, 2008.

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

[37] Joel Lehman and Kenneth O Stanley. Revising the evolutionary computation abstraction: minimal criteria novelty search. In *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, pages 103–110. ACM, 2010.

[38] Moshe Leshno, Vladimir Ya Lin, Allan Pinkus, and Shimon Schocken. Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. *Neural networks*, 6(6):861–867, 1993.

[39] Xugang Lu, Yu Tsao, Shigeki Matsuda, and Chiori Hori. Speech enhancement based on deep denoising autoencoder. In *Interspeech*, pages 436–440, 2013.

[40] Jonathan Masci, Ueli Meier, Dan Cireşan, and Jürgen Schmidhuber. Stacked convolutional auto-encoders for hierarchical feature extraction. *Artificial Neural Networks and Machine Learning–ICANN 2011*, pages 52–59, 2011.

[41] Risto Miikkulainen, Jason Liang, Elliot Meyerson, Aditya Rawal, Dan Fink, Olivier Francon, Bala Raju, Arshak Navruzyan, Nigel Duffy, and Babak Hodjat. Evolving deep neural networks. *arXiv preprint arXiv:1703.00548*, 2017.

[42] Tomáš Mikolov. Statistical language models based on neural networks. *Presentation at Google, Mountain View, 2nd April*, 2012.

[43] Geoffrey F Miller. Exploiting mate choice in evolutionary computation: Sexual selection as a process of search, optimization, and diversification. In *AISB Workshop on Evolutionary Computing*, pages 65–79. Springer, 1994.

[44] David E Moriarty and Risto Miikkulainen. Forming neural networks through efficient and adaptive coevolution. *Evolutionary computation*, 5(4):373–399, 1997.

[45] Gregory Morse and Kenneth O Stanley. Simple evolutionary optimization can rival stochastic gradient descent in neural networks. In *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference*, pages 477–484. ACM, 2016.

[46] Jiquan Ngiam, Aditya Khosla, Mingyu Kim, Juhan Nam, Honglak Lee, and Andrew Y Ng. Multimodal deep learning. In *Proceedings of the 28th international conference on machine learning (ICML-11)*, pages 689–696, 2011.

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

[48] Riccardo Poli, James Kennedy, and Tim Blackwell. Particle swarm optimization. *Swarm Intelligence*, 1(1):33–57, Jun 2007.

[49] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*, 2015.

[50] Amr Radi and Riccardo Poli. Discovering efficient learning rules for feedforward neural networks using genetic programming. In *Recent advances in intelligent paradigms and applications*, pages 133–159. Springer, 2003.

[51] Salah Rifai, Grégoire Mesnil, Pascal Vincent, Xavier Muller, Yoshua Bengio, Yann Dauphin, and Xavier Glorot. Higher order contractive auto-encoder. *Machine Learning and Knowledge Discovery in Databases*, pages 645–660, 2011.

[52] Miguel Rocha, Paulo Cortez, and José Neves. Evolution of neural networks for classification and regression. *Neurocomputing*, 70(16):2809–2816, 2007.

[53] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[54] Jake Ryan, Meng-Jang Lin, and Risto Miikkulainen. Intrusion detection with neural networks. In *Advances in neural information processing systems*, pages 943–949, 1998.

[55] Robert J Schalkoff. *Digital image processing and computer vision*, volume 286. Wiley New York, 1989.

[56] Khabat Soltanian, Fardin Akhlaghian Tab, Fardin Ahmadi Zar, and Ioannis Tsoulos. Artificial neural networks generation using grammatical evolution. In *Electrical Engineering (ICEE), 2013 21st Iranian Conference on*, pages 1–5. IEEE, 2013.

[57] Kenneth O Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127, 2002.

[58] Masanori Suganuma, Shinichi Shirakawa, and Tomoharu Nagao. A genetic programming approach to designing convolutional neural network architectures. *arXiv preprint arXiv:1704.00764*, 2017.

[59] Andrew James Turner and Julian Francis Miller. Cartesian genetic programming encoded artificial neural networks: a comparison using three benchmarks. In *Proceedings of the 15th annual conference on Genetic and evolutionary computation*, pages 1005–1012. ACM, 2013.

[60] Pascal Vincent, Hugo Larochelle, Yoshua Bengio, and Pierre-Antoine Manzagol. Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th international conference on Machine learning*, pages 1096–1103. ACM, 2008.

[61] Pascal Vincent, Hugo Larochelle, Yoshua Bengio, and Pierre-Antoine Manzagol. Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th international conference on Machine learning*, pages 1096–1103. ACM, 2008.

[62] Darrell Whitley, Timothy Starkweather, and Christopher Bogart. Genetic algorithms and neural networks: Optimizing connections and connectivity. *Parallel computing*, 14(3):347–361, 1990.

[63] Junyuan Xie, Linli Xu, and Enhong Chen. Image denoising and inpainting with deep neural networks. In *Advances in Neural Information Processing Systems*, pages 341–349, 2012.