EvoAAA: An evolutionary methodology for automated neural autoencoder architecture search

Francisco Charte*   Antonio J. Rivera   Francisco Martínez   María J. del Jesus

January 18, 2023

Abstract

Machine learning models work better when curated features are provided to them. Feature engineering methods have been usually used as a preprocessing step to obtain or build a proper feature set. In late years, autoencoders (a specific type of symmetrical neural network) have been widely used to perform representation learning, proving their competitiveness against classical feature engineering algorithms. The main obstacle in the use of autoencoders is finding a good architecture, a process that most experts confront manually. An automated autoencoder architecture search procedure, based on evolutionary methods, is proposed in this paper. The methodology is tested against nine heterogeneous data sets. The obtained results show the ability of this approach to find better architectures, able to concentrate most of the useful information in a minimized coding, in a reduced time.

1 Introduction

Intelligent appliances based on machine learning systems [1] can be found in many everyday tasks. They are in charge of filtering spam email [2], detecting fraudulent transactions [3], recommending new products to buyers [4] and many other apparently simple jobs. To do that, ML methods need to extract knowledge from raw data. The usefulness of that knowledge mostly depends on the quality of data features. The goal is to produce descriptive and/or predictive models, depending on the task at hand.

Choosing a curated set of attributes, or building a new one from the original features, tends to produce better results [5] than using raw variables. Hence the interest in feature engineering techniques in late years, including well-known preprocessing procedures [6] such as feature selection [7, 8] and feature extraction [9]. Representation learning (REPL) [10, 11] is a term tightly linked to perform feature engineering relying on deep learning techniques [12, 13].

Autoencoders [14, 15, 16] are a modern general-purpose DL-based family of tools for facing REPL. An AE is an unsupervised symmetric neural network [17] aimed to build a coding that maximizes the reconstruction of data patterns. The obtained coding can be applied to many different tasks [18], including visualization, anomaly detection, hashing and noise removing. However, finding the proper AE architecture for each data set and function is not a trivial process. Usually, it is a challenge that experts have to deal with.

In this study EvoAAA, an automated methodology for designing AE architectures maximizing their reconstruction power when used with a specific data set, is proposed. The job is confronted as a hard optimization problem [19], unfeasible to solve by means of exhaustive search. Our hypothesis is that evolutionary methods [20] would be able to find near-optimal AE architectures in a reasonable time.

To verify the competitiveness of EvoAAA a thorough experimentation, including nine data sets and five search methods, three of them based on evolutionary optimization, is conducted. The architectures found through this approach are way better than those retrieved by exhaustive search.

1.1 Problem formulation

That AEs are effective tools for REPL is a known fact [14, 15, 16], having proved their superiority against classical feature engineering methods such as PCA, LDA, ISOMAP and LLE [21]. They are also a tool closely related to nonstandard learning [22] problems. An AE is a symmetrical ANN [17], so it shares many of the characteristics of any ANN.

Training an ANN implies adjusting the weights that connect their neurons, using the backpropagation method [23] and any variation of the gradient descent algorithm such as SGD [24]. The ANN architecture, i.e. number of layers, amount of units per layer, activation functions, etc., is set in advance, prior to the training process.

An inadequate ANN architecture could produce bad output results, regardless of the weights learned through the training process. If the ANN is too simple, adjusting too few parameters, it will be not able to learn. On the contrary, too complex

---

*Corresponding author: Francisco Charte, Computer Science Department, A3-241 Universidad de Jaén, Campus Las Lagunillas, 23071 Jaén, Spain.

Table 1: Summary of terminology and acronyms

Term/Acronym	Description
AE	Autoencoder
ANN	Artificial Neural Network
Chromosome	Codification which represents the set of parameters of an individual in the population
CNN	Convolutional Neural Network
Cross-over	Operator to produce a new chromosome from two or more individuals (parents) by mixing their genes
DE	Differential Evolution
DL	Deep Learning
Elitism	Technique which preserves the best individuals among generation in an EM
EM	Evolutionary Method
ES	Evolution Strategy
EvoAAA	Evolutionary Methods for Automated Autoencoder Architecture search
Fitness	Quality measure linked to each individual
GA	Genetic Algorithm
Gene	Each value in a chromosome
Generation	Each one of the iterations in an evolutionary method
Individual	A potential solution in the search space denoted by a chromosome and the corresponding fitness value
LDA	Linear Discriminant Analysis
LLE	Locally Linear Embedding
ML	Machine Learning
MLP	Multi-Layer Perceptron
MSE	Mean Squared Error
Mutation	Operator to produce a new chromosome altering genes in an existing individual
PCA	Principal Component Analysis
Population	Group of individuals in a generation
REPL	Representation Learning
SGD	Stochastic Gradient Descent

ANNs suffer from overfitting [25] due to existing enough parameters to memorize the whole training data. These same problems also affect AEs. Achieving a balanced architecture, the point where the ANN extracts enough information from seen data to generalize well while processing future never seen patterns, is still an unsolved problem. As a result, dozens of papers which contribute the design of ANNs to tackle specific problems [26, 27] are published every year.

The interest is in choosing a proper AE architecture to process an specific data set, so that the AE is able to learn an optimum representation of the data. Until now the design of AEs has been in charge of human experts. It is not an easy task to automate, since it is a hard combinatorial problem [19]. In fact, finding a good architecture can be seen as an optimization problem. This is a field where EMs [20] have shown their efficacy in the past.

Our proposal is a formulation to code any AE architecture so that it can be evolved by means of evolutionary approaches. The goal is to reduce the reconstruction error as much as possible. This way, the AE encoding will concentrate the maximum general-purpose information, rather than a coding aimed to improve class separability or any other specific goal. Regarding evolutionary techniques, they will be used as the tool to optimize a set of parameters. The classical terminology in this field summarized in Table 1 along with the acronyms appearing in the text, will be used.

1.2 Literature review

Feature engineering is a manual or automated task aimed to obtain a set of features better than the original one. Feature selection [6] consists in choosing a subset of attributes while maintaining most useful information in the data. It can be manually performed by an expert in the field, but mostly is faced with automated methods based on feature correlation [7] and mutual information [8]. By contrast, feature extraction methods transform the original data features to produce a new, usually reduced, set of attributes. Popular algorithms to do this are PCA [28] and LDA [29], whose mathematical foundations are relatively easy to understand.

More advanced studies work with the hypothesis that the distribution of variables in the original data lies along a lower-dimensional space, usually known as *manifold*. A manifold space works with the parameters that produce the data points in the original high-dimensional space. Finding this embedded space is the task of manifold learning [30] algorithms. Unlike PCA or LDA, manifold methods apply non-linear transformations, so they fall into the non-linear dimensionality reduction [31] category.

Autoencoders, as detailed in [17], are ANNs having a symmetric architecture, as shown in Figure 1. The input and output layers have as many units as features there are in the data. Inner layers usually have fewer units, so that a more compact representation of the information hold in the data is produced. The goal is to reconstruct the input patterns into the output as faithfully as possible.

Although AEs are mainly used to perform feature fusion [17], searching the manifold in which the parameters to rebuild the data are found, they have many other practical applications [18]. A properly configured AE can project data of any
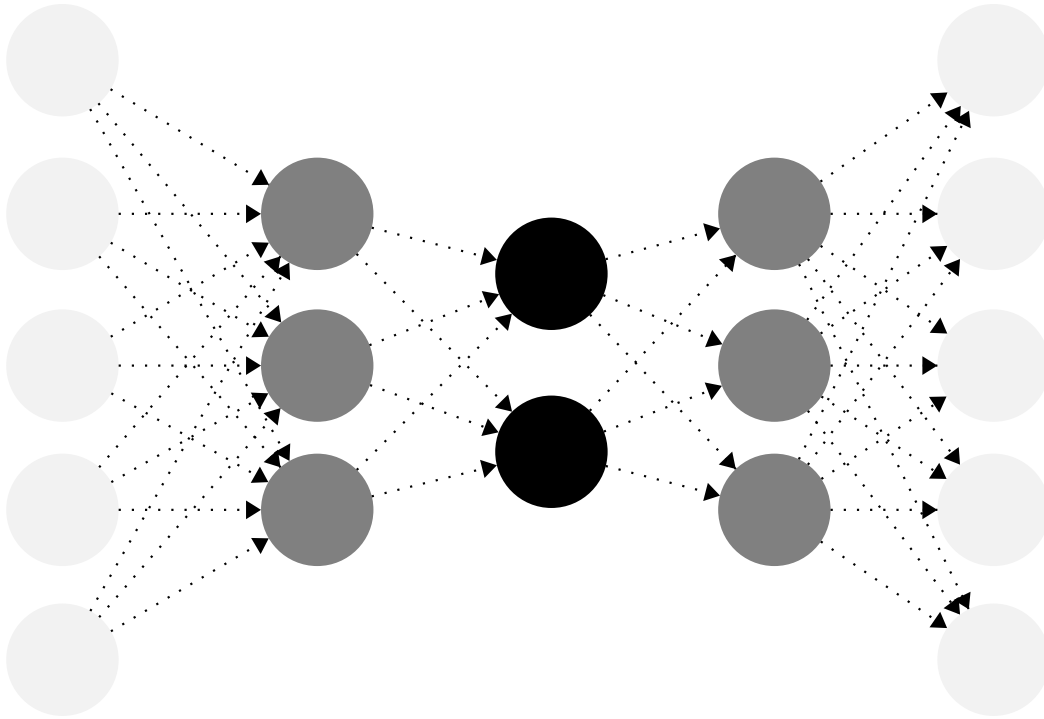
Figure 1: Classic architecture for an AE. Black nodes denote a 2-variable encoding layer. Dark gray nodes are intermediate hidden layers. Light gray ones are the input (left) and output layers.
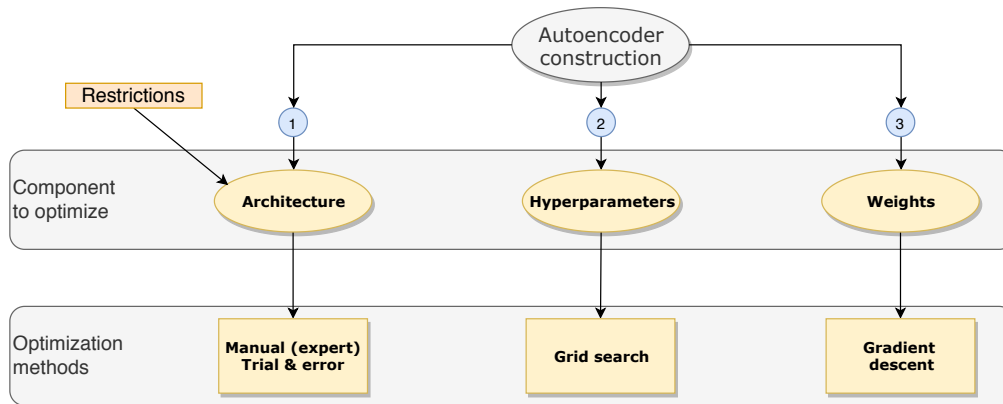

Figure 2: Components to be optimized during AE construction and the methods used for it. The numbers inside circles indicate the usual optimization order: firstly an architecture is set, then a set of hyperparameters is chosen, lastly the weights are adjusted.

dimensionality into 2 or 3 dimensions so that patterns can be graphically visualized [32]. AEs can be used to detect anomalies [33, 34], training them to faithfully reconstruct normal patterns. When anomalous data enters the AE, it produces a high output error denoting that these patterns do not follow the known distribution. Another interesting application of AEs, specifically of denoising AEs, is data noise removing. This kind of AE has been used to successfully denoise images [35] and speech [36]. Usually the loss function of the AE has to be adapted to the specific task to be faced, so that the obtained encoding promotes separability, topology preservation or any other desired characteristic. When only maximum performance is pursued while reconstructing the input patterns, the AE will produce a general purpose coding in its inner layer.

AEs can be configured with a variable amount of inner layers, each of them having different lengths. The proper architecture will mostly depend on the complexity of the patterns to be reconstructed and the restrictions imposed by the encoding layer. These restrictions prevent the AE from simply copying the input onto the output [17], for instance by reducing the number of neurons in the coding layer, forcing the output of most neurons to be zero (sparse representation), etc.

Finding the best parameters to tune a machine learning model is an uphill battle. Performing a grid search through an internal validation process is an usual approach. However, it is useful only for limited sets of parameters taking known ranges of values. Disparate search algorithms, metaheuristics [37] and optimization approaches [38], aimed to perform combinatorial

optimization, could be applied. These go from relatively simple search methods [39, 40, 41, 42] such as A* or IDA to more advanced and complex approaches such as memetic algorithms [43], including classic means as simulated annealing [44] or tabu search [45]. Evolutionary methods [46, 47] have been also used to face optimization problems [48, 49, 50, 51] for long time. Many of these algorithms are based on the behavior of certain populations, such as ant colonies [52] and particle swarms [53]. In the following, we focus specifically in approaches based on natural evolution [54].

In evolutionary algorithms [55] the search space is defined by the *chromosome*. It is made up of several *genes*, each of them representing a specific trait or dimension. The values taken by the genes in a chromosome are limited, and each combination is a potential solution (a point in the search space). These are also known as *individuals*. Usually a set of these are randomly generated, producing an initial *population*. Each individual is assigned a fitness value that represents the goodness of the solution. From this point, a number of iterations are run consisting in the same steps. Firstly, certain individuals of the population are selected for reproduction based on their fitness. Then, a set of operators are applied to selected individuals in order to create new ones. Common operators are crossing, that mixes genes of two individuals to produce a new chromosome, and mutation, which randomly changes the value of one or more genes. Lastly, the whole population is evaluated by computing the new fitness values and the population is updated, usually replacing old individuals with new ones, although the best overall solutions can be kept (elitism) whether they are new or not. Three popular evolutionary methods are genetic algorithms [56], differential evolution [57] and evolution strategy [58]. The first follows the methodology just described of selection, crossing, mutation and evaluation. Although GAs can be seen as an old technique, they are still in widely use [38, 50, 59] due to their simplicity and good performance. The second method produces new individuals from differences between existing ones, while the third one focuses on evolving only a few individuals using the mutation operator as only tool.

Evolutionary methods are also suitable for combinatorial optimization, and they have been used for instance for support vector machines [60] and more recently for deep learning networks [61]. Even though EMs have been already used to optimize ANNs, many of the proposals have been focused on learning the weights linked to each connection. This is known as conventional neuroevolution [62, 63, 64, 65] and its main foundation is to use an EM instead of the traditional gradient descent algorithm to optimize weights. Therefore, a fixed network architecture is the base for all the population. The only difference among individuals is the set of weight matrices connecting each layer, values optimized by the EM. Neuroevolutive algorithms able to also evolve the network topology appeared later [66], being aimed most of them to optimize MLPs. There are different approaches to construct the ANN architecture, being one of the most popular subnetworks composition [67]. A recent survey in that matter can be found in [68].

More recently, the fusion of different techniques to fully automate the process of choosing a proper model structure and hyperparameters, adjusting the weights, etc., has given birth to a new field known as AutoML [69, 70, 71]. AutoML tools can be based on EMs, but also in Bayesian techniques and other optimization algorithms. Existing AutoML tools are mostly aimed to aid in the design of MLPs and CNNs [72], maybe the two most popular kind of ANNs. Essentially, these tools have a limited set of *cells* or *blocks* that they can combine to define the ANN topology. Depending on the task at glance a specific strategy is followed to choose these blocks. For instance, the AutoKeras tool [73] defines tasks that allow the automated design of ANNs for image, text and structured data classification. The ANN architecture is made up of predefined blocks such as ImageBlock, TextBlock or StructuredDataBlock, with some adjustable parameters. As consequence, these AutoML tools have a coarse granularity and only can be used to build some specific types of ANNs.

By contrast, in the present proposal EMs are used to evolve the architecture of AEs, a kind of ANN with some specific aspects such as its symmetric layout or its unsupervised learning approach. The AE layout is generated with a fine granularity, as described below, instead of using predefined blocks. The weights are learned through the usual back-propagation algorithm. The EvoAAA procedure proposed here could be a piece in an AutoML chain.

The rest of this paper is structured as follows: in Section 2 the different aspects to optimize while building an AE are outlined and the proposed architecture search methodology is presented. Section 3 describes the experimental framework, provides the analysis of performance results and discuss them. Lastly, conclusions are drawn in Section 4.

2 Autoencoder architecture search with EvoAAA

This section presents the proposal to face the problem formulated in subsection 1.1, outlining the aspects that have to be taken into account while building an AE and detailing the methodology to follow.

2.1 Components to be optimized during AE construction

The process to build a new AE, adjusted to produce a good representation from input patterns, implies several steps. Each one is linked to the optimization of a certain component. A summary of the procedure, components and methods is shown in Figure 2. The components are tuned in the following order:

1. **Architecture:** The structure of the AE has to be set, deciding the amount of layers it will be made of, the number of units per layer, which activation functions will be used in each unit, etc. For years, this has been manually done by experts. The conventional trial and error procedure is also a frequent approach, readjusting the AE architecture after
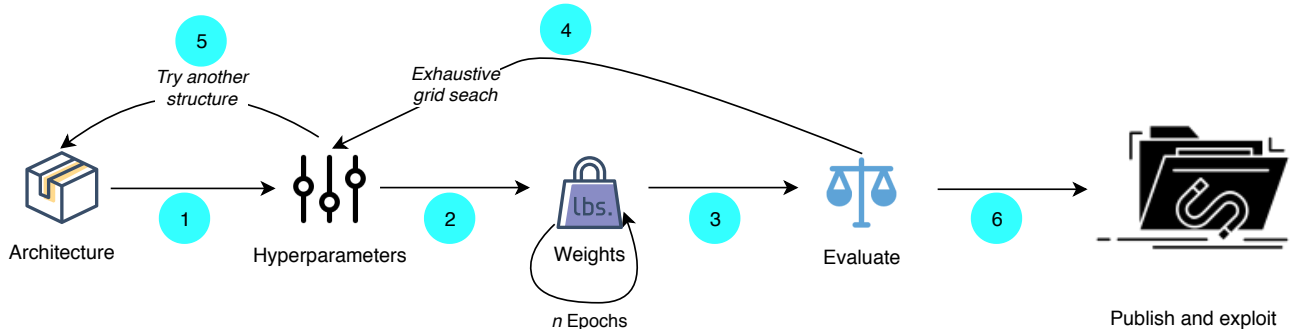
Figure 3: Before an AE can be exploited each architecture has to be tested with multiple hyperparameters combinations, and the weights of everyone of these configurations have to be adjusted.

the three building steps have been completed and the AE performance evaluated. During this step specific restrictions, as the symmetrical structure of the network, have to be taken into account.

2. **Hyper-parameters:** Having decided on the design of the AE, the following step would be choosing the proper values for several hyperparameters. These are not part of the network structure nor are they part of the weights configuration. They are in charge of controlling the tuning of weights during the training process. The most common hyperparameters are the learning rate, the batch size and the number of epochs the network is trained. Although the values could be manually picked, usually a grid search algorithm is used. Once more, trial and error using part of the training patterns as validation, allows to find the best configuration.

3. **Weights:** Once the architecture of the network and its hyperparameters have been set, the last action would be adjusting the weights that connect every unit in each layer with all the units in the following one. Unlike what happens for finding a proper network structure and good hyperparameters, there is a solid mathematical background related to how these weights should be tuned. Derivatives allow to know the contribution of each connection to the global committed error, so that small adjustments can be made in the correct direction. This is the foundation of the well-known gradient descent method.

The three previous steps are iterative in a nested way, as shown in Figure 3. Usually several different architectures will be tested. For each architecture, the first step (noted as 1 inside a circle) would be getting different sets of hyperparameters to be tried. Adjusting the weights of each configuration, a procedure that is iterative by itself, is be the following step. The third stage, after an architecture has been fixed, a set of hyperparameters is chosen and the weights are adjusted, is evaluating the model. The output of this action would lead us to step 4 or step 6, depending on whether a certain criterion is satisfied or not. In the first case additional sets of hyperparameters, generally obtained by grid search, would be used. If all potential combinations have been tried, a step backward (noted as 5) would be altering the AE architecture. Step 6 marks the end of the process, having an AE ready to be exploited in the system.

2.2 The EvoAAA proposal

As previously stated, there are well-known procedures for steps 2 and 3 (see Figure 2) of the optimization process: grid search for finding the hyperparameters (step 2) and gradient descent for weight adjusting (step 3). On the contrary, finding a proper AE architecture is still an open problem.

The experience acquired while working with a certain data set could not be applicable when data change occurs. The amount of combinations is potentially infinite, so automating this process by means of exhaustive search is unfeasible. Most experts follow some heuristics to choose the number of layers and units, depending on the quantity of variables they have to deal with, while aspects as activation functions are statically assigned.

Seeking a way of automating the AEs design process, a suitable architecture could be found through a simple search if the number of combinations is restricted beforehand. It would be similar to the grid search procedure followed for hyperparameters. This approach would allow to choose the best structure among a bag of predesigned ones. However, it does not offer any guarantee over the performance of these predesigned models. Other straightforward search heuristics could be adopted, such as adding/deleting layers/units from a base structure as long as the performance of the AE improves.

The EvoAAA approach proposes to solve the task through EMs, since they have amply demonstrated their ability to solve disparate optimization problems. Specifically, it aims to use population based algorithms to evolve a set of AE architectures over time. To do so, a way to code all possible AE configurations is introduced. It will be the chromosome representation that the EMs will work with.

1	2	3-6	7-13	14	15
Type	Layers	Units per layer	Activation function per layer	Activation out layer	Loss
[1,6]	[0,3]	[1,f]	[1,8]	[1,4]	[1,5]

Figure 4: Chromosome genes, name and interval of values they can get.

Table 2: Purpose of each gene and description of their values.

Name	Purpose	Values
Type	Sets the type of AE to be used	1) Basic, 2) Denoising, 3) Contractive, 4) Robust, 5) Sparse, 6) Variational
Layers	Number of additional layers in coder/decoder	0) Only a coding layer, 1-3) Additional layers in both coder and decoder
Units	Sets the number of units per layer, with f being the amount of features in the dataset	The first integer (gen 3) configures the number of units in the outer layer, while the last one (gen 6) sets the coding length.
Activation	Activation function to use in each layer, both for the coder and decoder	1) linear, 2) sigmoid, 3) tanh, 4) relu, 5) selu, 6) elu, 7) softplus, 8) softsign
Out act.	Activation function for the output layer	1) linear, 2) relu, 3) elu, 4) softplus
Loss	Loss function to evaluate during fitting	1) Mean squared error, 2) Mean absolute error, 3) Mean absolute percentage error, 4) Binary crossentropy, 5) Cosine proximity

2.2.1 Chromosome representation

Evolutionary algorithms usually work with binary or real-valued genes. A set of genes builds a chromosome or individual of the population. In EvoAAA each chromosome will code the complete architecture of an AE. However, an integer gene representation is used rather than binary or real-valued genes.

The chromosome will be made up of 15 genes, as shown in Figure 4. The number of each gene is shown above, their names inside and just below the range of values that can be assigned to them. The purpose of each gene, as well as the meaning of its values, are portrayed in Table 2. The main characteristics of this AE encoding are the following:

- Different types of representations can be learned with AEs depending on the imposed restrictions. Some of those restrictions are linked to the type of AE [17], designed to induce sparsity, learn from noisy samples, etc. The first gen in the chromosome allows to choose among six different AE types (see details in Table 2).

- The AEs will have a coding layer and up to six additional hidden layers that have to be taken in pairs: 2, 4 or 6. The value of the second gen, between 0 and 3, indicates the number of pairs of hidden layers. Therefore, the simplest AE would have only 3 layers, the input one, the coding one and output one, while the most complex would be made up of 9 layers in total.

- Genes 3 to 6 state the number of units to have in the hidden layers. The last value is associated to the innermost layer, so it sets the coding length. The other three values are linked to each layer pair, from outer to inner. The number of features in the data, noted as f, will limit the amount of units in any layer. An additional restriction is that inner layers cannot be larger than outer ones.

- With the exception of the input layer, which is limited to transferring values to the next one, all other layers in the AE use an activation function. Although it is common for all units of an AE to have the same activation function, there is nothing that restricts the use of different functions. The proposed encoding uses 7 genes (7 to 13) to choose the activation functions to be applied in each inner layer, allowing eight different options (see Table 2). The output layer is treated independently with gene 14, since only activation functions producing positive output are allowed.

- The last gene in the chromosome can take values from 1 to 5, stating the loss function to be internally evaluated while training the AE. The meaning of these values is provided in Table 2.

Through the restrictions imposed in genes 3 to 6, the resulting AEs would be always undercomplete [17]. This means that the learned representation will be more compact than the original one, with a vector containing less values. In addition, the loss function evaluated while adjusting the weights will exclusively focus on reconstruction error. AEs can be trained to improve class separability, reduce the data complexity and other goals. In this case the main objective is to have a reconstruction of data patterns as good as possible. This will force the inner layer to concentrate as much general purpose useful information as possible, so that the AE can be later used in any kind of task.

2.2.2 Autoencoder complexity penalization

When designing an AE for learning new representations the main interest will be in the raw performance. In an AE the performance is usually measured as the error produced while reconstructing data patterns from the learned encoding. Nonetheless, the time needed to obtain the encoding is also a factor to consider. This is the motivation to include a penalization factor in EvoAAA: α. It will be used in the fitness function of the evolutionary method in order to assess the goodness of each AE configuration.

The fitness function, that will decide the quality of the solutions, will be computed as shown in (1), where *trainloss* is the reconstruction loss produced by the AE with training data, *Layers* is the number of additional hidden layers (gen 2), *Units coding* is the size of the encoding layer (gen 6), and α is a coefficient setting the level of penalization applied according to the complexity of the AE.

$$fitness = trainloss + \alpha(Layers \times Units\ coding) \tag{1}$$

Therefore, AEs having a similar reconstruction performance but simpler architecture (see subsection 3.5.8 for additional details) will be preferred over the more complex ones. The bias to obtain AEs with less layers and a shorter encoding is modulated with the α value. This is the only parameter needed by EvoAAA.

2.2.3 Search space

The reason for using a complex evolutionary algorithm to find a proper AE architecture is that the search space is huge. So, a strategy is needed to pick a good solution without having to explore much of this space. But, how large is the search space assuming the chromosome previously described?

Excluding genes 3-6, whose values would vary depending on the number of features in the dataset, there are more than a billion combinations (see Eq. 2).

$$Type \times Layers \times Act^7 \times Act\ out \times Loss =$$

$$6 \times 4 \times 8^7 \times 4 \times 5 = 1\ 006\ 632\ 960 \tag{2}$$

Working with very small datasets, those having a few dozens of attributes only, the amount of combinations will grow to several billions. We would face trillions of solutions or even more for high-dimensional datasets. Evaluating all those solutions to find the best one is currently unfeasible. As a result, looking for an optimal AE architecture would not be always possible by brute force. However, we could find good enough solutions through optimization mechanisms based on evolutionary strategies.

3 Algorithmic framework of EvoAAA

The EvoAAA methodology is a general evolutionary proposal to search good AE architectures. It can be instantiated using any population based evolutionary algorithm. In this study three specific instances of EvoAAA are tested, one with a genetic algorithm as underlying search method, another one relying on differential evolution and the third one using evolution strategy. The three of them will be compared against each other and also versus two baseline search methods, a simple exhaustive search and a random search. Nine different data sets will be used in the conducted experimentation.

3.1 Evolutionary search algorithms

We propose instantiating EvoAAA three times using three different evolutionary algorithms. All of them will use the former chromosome representation. These three approaches are:

- **Genetic algorithm (GA).** A classical genetic algorithm [56], in which a population of individuals evolves through a crossover operator, to give rise to new ones, and to which a mutation operator is applied with a certain probability.

- **Evolution strategy (ES).** An aggressive solution-seeking algorithm [58], working with a few individuals who give rise to new ones exclusively through mutation.

- **Differential evolution (DE).** A population-based optimization algorithm [57] in which new individuals (*agents*) are produced from the differences between two random agents with respect to another taken as reference.

Table 3 summarizes the main parameters used to run these methods. For GA and ES, each gene in the chromosome is mutated with a probability of 1/15, a value based on the chromosome length itself. Elitism is used in the GA to preserve the tenth percent of individuals having better fitness. The ES is intrinsically elitist, choosing only the best candidates among the union of new mutated individuals and the old population. For GA, in each iteration the best 5 individuals are chosen and preserved. Then, two random parents are picked up from the remaining individuals by using roulette wheel probability.

Table 3: Main parameters of the evolutionary algorithms.

Parameter	GA	ES	DE
Population size	50	4	150
Iterations	100	500	30
Prob. mutation	1/15	1/15	NA
Prob. cross-over	1.0	NA	0.5
Elitism (individuals)	5	NA	NA
Termination cost	0	0	0

Table 4: Datasets used in the experimental study

Dataset	Type	Variables	Instances	Source
cifar10	Integer	1 024	60 000	[75]
delicious	Binary	983	16 105	[76]
fashion	Integer	784	70 000	[77]
glass	Real	9	214	[78]
ionosphere	Real	34	351	[79]
mnist	Integer	784	70 000	[80]
semeion	Binary	256	1 593	[81]
sonar	Real	60	208	[82]
spect	Binary	22	267	[83]

A multi-point crossover operator is applied, being the crossing points established according to the diagram in Figure 4. This allows the two individuals acting as parents to interchange several of their genes to produce childhood. This way, new individuals are produced until the population size is met, replacing all the old individuals. Lastly, the mutation operator is used with some individuals according to the probability previously stated. As can be seen, this is an aggressive scheme that maximizes the exploration in such a huge search space. For DE, the DE/local-to-best/1 optimization approach is followed. The population size is obtained from the representation length (15 genes × 10). Cross-over probability, as well as other specific parameters, has been set following the recommendations in [57, 74]. The number of iterations for each method has been adjusted so that a similar amount of evaluations is made.

3.2 Data sets

In order to compare the performance of the three instances of EvoAAA along with the exhaustive and random search approaches, nine data sets are used. Their main traits are detailed in Table 4. The last column provides the origin reference for each one of them. The criteria followed to choose them have been:

- **Attribute type:** An AE is a specialized ANN that, through a series of computations, reconstructs the input values. These have to be numeric, and three cases are considered: real values, integer values, and binary values. The goal is to evaluate how the AEs performance changes depending on the type of attributes they have to reconstruct.

- **Number of attributes:** Half of the data sets have several hundreds of attributes, even more than a thousand in the case of cifar10. The other five are not that large, with only a handful of variables. This way the behavior of found AEs while working with a few versus a lot of variables will be analyzed.

Since an AE is an unsupervised representation learning method, class attributes have been removed for all the data sets. The number of variables indicated in Table 4 is the effective amount of them being processed.

3.3 Restrictions and evaluation

Five AE architectures will be obtained for each data set, EvoAAA-Gen: the instance based on a genetic algorithm, EvoAAA-Evo: the instance using evolutionary strategy as underlying search method, EvoAAA-Dif: the one based on differential evolution, and the two produced by the exhaustive (Exh) as well as random search (Ran). The same restrictions and evaluation procedure are applicable to all of them:

- **Performance evaluation:** The common mean squared error metric is used to assess the performance of the AEs. This is an unbounded measure, which depends on the original range of values. The lower the MSE the better performance the AE has. This value is the *trainloss* factor in Eq. 1.

- **Termination cost:** As shown in Table 3, the three evolutionary algorithms have been configured with 0 as termination cost. This means that the search will stop if a configuration returning $MSE = 0$ is found, but not before unless the maximum runtime or maximum number of iterations are reached.

- **Maximum runtime:** The five search approaches will be limited to running for 24 hours. After that the best solution found until then is returned as preferred AE structure. In practice, this limitation will impede the exhaustive search from going through all the possible AE configurations, limiting as well the amount of combinations tested by the random search.

Aside from the MSE, the amount of architectures tested by each approach is also recorded during execution, as well as each individual solution. The number of combinations is limited by both the maximum runtime and the population size and iterations of the evolutionary methods.

3.4 Experiments and results

The 45 experiments[1] (9 data sets times 5 search approaches) have been executed in the same hardware, a PC with an NVidia RTX-2080 GPU, and using the following software configuration: Arch Linux [84], CUDA 10.1 [85], cudnn 7.6 [86], Tensorflow 1.14 [87], Keras 2.24 [88] and ruta 1.1 [89]. The RMSProp [90] optimizer of Tensorflow has been used. Since it relies on an adaptive learning rate, optimizing this hyper-parameter is not necessary. The default batch size value in Keras has been chosen, 32. Lastly, the training of the AEs is made in 20 epochs. Although a higher number of epochs, such as 100 or even more, is usual while trying to perform a final optimization of a representation, in this case the main interest is in comparing the search procedures rather than in obtaining fully optimized AEs. Reducing the number of epochs allows to try more architectures in a certain time interval. Regarding how the data instances are used, 20% of them are reserved from the beginning to assess AE performance, computing the MSE. The remainder 80% patterns are given to Keras to train each AE architecture.

A summary of results is provided in Table 5. For each data set the columns indicate its name, the search method, the amount of individuals (AE configurations) tested, the number of hidden layers and encoding length of the best individual, its complexity (assuming that $\alpha = 0.0001$) and the MSE obtained with test data. For these last two columns the lower the value the better the AE would be, having less complexity and superior reconstruction performance. The best configuration (lower MSE) for each data set has been highlighted in bold. These values include the complexity penalization, real best MSEs are provided in Table 6. Average error values, computed from all the evaluated solutions, and best values are presented in it.

The number of tested individuals is limited by the population size and iterations , as well as the maximum runtime, for the three EvoAAA instances. In addition all the invalid configurations produced during the exploration are discarded before they enter the training and testing phase. The exhaustive and random strategies only generate valid AE architectures, and the only limitation is the running time.

3.5 Discussion

The results shown in Table 5 can be analyzed from several perspectives, raw performance: the lower MSE the better; size of explored space: although the more inspected individuals could be considered the better, the ability to find good solutions exploring less space has to be also taken into account, and solution complexity: the fewer layers and shorter encoding length the better. In addition, other aspects such as running times, convergence, etc., can be studied.

3.5.1 Analysis of performance

It is easy to see that the highest error values always correspond to the exhaustive search approach. MSE is unbounded and depends on the original attribute values. So, it does not allow to make comparisons among different data, but only between the five methods for each data set. In general, the error committed by **Exh** is one or two orders of magnitude above EvoAAA-Dif, EvoAAA-Gen and EvoAAA-Evo.

Comparing the three EvoAAA instances, EvoAAA-Gen is the best performer in 5 out of 9 datasets, with EvoAAA-Dif in a close second position gaining 3 out of 9 cases although the differences against EvoAAA-Evo are almost negligible in some cases. The biggest difference can be found with the fashion dataset, where the EvoAAA-Dif approach achieves one-quarter of the error shown by EvoAAA-Gen while using a simpler architecture. This analysis is made taking into account the complexity of AEs, i.e. the MSE is increased by the penalization factor.

In addition to best values, it would also be interesting to analyze the average behavior of the search strategies. For doing so, Table 6 shows for each data set (columns) and method (rows) the average and minimum (best) MSE[2] achieved in each

---

[1]The R code to reproduce these experiments, as well as the datasets used in them, are available to download from `https://github.com/fcharte/EvoAAA`. To execute this code you will need a current R version, install several R packages (detailed in the provided scripts) and the frameworks enumerated in this section. Each run will provide two result files containing all evaluated solutions and the AE structure for the best ones.

[2]These are raw MSE values rather than MSE penalized by complexity (the penalization factor was explained in subsection 2.2.2 and it depends on the α parameter, set to 0.0001 in the described experiments). Therefore, best values in Table 6 are lower than those in Table 5.

Table 5: Summary of results. For each combination dataset-optimization strategy the amount of evaluated individuals, number of layers and encoding length of the AE, its complexity and MSE (including the penalization factor) are provided.

Dataset	Method	Individuals	Layers	Coding length	Complexity	Error (MSE) ↓
cifar10	EvoAAA-Dif	2 537	1	38	0.004	0.0137
	EvoAAA-Evo	4 001	1	52	0.005	0.0133
	Exh	1 739	1	1	0.000	0.0395
	EvoAAA-Gen	1 115	1	39	0.004	**0.0131**
	Ran	2 101	3	5	0.002	0.0260
delicious	EvoAAA-Dif	4 650	2	19	0.004	0.0124
	EvoAAA-Evo	4 001	1	39	0.004	0.0119
	Exh	5 837	1	1	0.000	0.0165
	EvoAAA-Gen	1 401	1	32	0.003	**0.0102**
	Ran	5 901	2	10	0.002	0.0134
fashion	EvoAAA-Dif	2 604	1	256	0.026	**444.8169**
	EvoAAA-Evo	755	3	61	0.012	565.4571
	Exh	1 494	1	1	0.000	4 180.2370
	EvoAAA-Gen	439	5	156	0.047	1 881.0680
	Ran	1 501	2	35	0.007	782.3572
glass	EvoAAA-Dif	4 650	2	3	0.001	24.8785
	EvoAAA-Evo	4 001	3	3	0.001	5.4093
	Exh	35 368	1	1	0.000	29.2589
	EvoAAA-Gen	4 505	1	5	0.000	**0.4473**
	Ran	35 301	3	7	0.002	1.2705
ionosphere	EvoAAA-Dif	4 650	2	23	0.005	**0.0740**
	EvoAAA-Evo	4 001	3	15	0.003	0.0931
	Exh	29 905	1	1	0.000	0.2099
	EvoAAA-Gen	2 302	3	15	0.003	0.0917
	Ran	29 901	3	23	0.007	0.1049
mnist	EvoAAA-Dif	2 754	1	162	0.016	**192.5133**
	EvoAAA-Evo	732	3	332	0.066	431.6960
	Exh	1 491	1	1	0.000	4 104.1400
	EvoAAA-Gen	403	1	156	0.016	254.7397
	Ran	1 401	2	35	0.007	611.5036
semeion	EvoAAA-Dif	4 650	1	136	0.014	0.0459
	EvoAAA-Evo	4 001	1	134	0.013	**0.0355**
	Exh	19 861	1	1	0.000	0.1940
	EvoAAA-Gen	4 505	1	110	0.011	0.0376
	Ran	19 901	2	132	0.026	0.0604
sonar	EvoAAA-Dif	4 650	1	29	0.003	0.0142
	EvoAAA-Evo	4 001	3	17	0.003	0.0162
	Exh	34 557	1	1	0.000	0.0462
	EvoAAA-Gen	4 505	3	7	0.001	**0.0139**
	Ran	35301	4	8	0.003	0.0145
spect	EvoAAA-Dif	4 650	2	13	0.003	0.0959
	EvoAAA-Evo	4 001	3	14	0.003	0.0834
	Exh	30 740	1	1	0.000	0.1829
	EvoAAA-Gen	4 505	5	15	0.004	**0.0703**
	Ran	30 701	3	14	0.004	0.1145

Table 6: Average and best performance (raw MSE) by data set and search strategy

	cifar10	delicious	fashion	glass	ionosphere	mnist	semeion	sonar	spect
EvoAAA-Dif (Avg.)	0.0256	0.0186	8979.9337	604.3130	0.2719	5073.9444	0.2348	0.0404	0.2575
EvoAAA-Evo (Avg.)	**0.0085**	0.0086	**891.1958**	562.3818	**0.1152**	**616.6640**	**0.0322**	0.0223	0.1145
Exh (Avg.)	0.0498	0.0195	18734.2379	611.4017	0.3687	9646.3325	0.2836	0.1331	0.3213
EvoAAA-Gen (Avg.)	0.0097	**0.0075**	2852.7007	**345.4758**	0.1227	3323.4105	0.0405	**0.0157**	**0.1006**
Rnd (Avg.)	0.0461	0.0206	8557.0936	606.7591	0.3303	4997.0538	0.2886	0.0761	0.2910
EvoAAA-Dif (Best)	**0.0020**	0.0131	**444.7913**	563.1547	**0.0694**	**192.4971**	0.0323	**0.0106**	0.0924
EvoAAA-Evo (Best)	0.0080	0.0080	565.4449	316.0380	0.0963	431.6296	**0.0221**	0.0128	0.0806
Exh (Best)	0.0394	0.0164	4180.2370	576.7005	0.2098	4104.1400	0.1939	0.0461	0.1828
EvoAAA-Gen (Best)	0.0090	0.0055	1881.0210	**0.4468**	0.0872	254.7241	0.0266	0.0115	**0.0658**
Rnd (Best)	0.0047	**0.0021**	782.3502	564.4037	0.0980	564.2114	0.0339	0.0114	0.1098

Table 7: Performance ranking of the tested methods

Dataset	EvoAAA-Dif	EvoAAA-Evo	Exh	EvoAAA-Gen	Ran
cifar10	1	3	5	4	2
delicious	4	3	5	2	1
fashion	1	2	5	4	3
glass	3	2	5	1	4
ionosphere	1	3	5	2	4
mnist	1	3	5	2	4
semeion	3	1	5	2	4
sonar	1	4	5	3	2
spect	3	2	5	1	4
Average	2.00	2.56	5.00	2.33	3.11

case. It can be observed that the best MSE obtained by the exhaustive and random search is far from the average MSE of EvoAAA-Dif, EvoAAA-Gen and EvoAAA-Evo. This leads to the conclusion that even a few iterations with EvoAAA would achieve a better result than 24h of exhaustive or random search. Internal differences between best and average values for each search method tend to be minimal in most cases (cifar10, delicious, ionosphere, semeion and sonar). In general, these differences seem lower in the case of EvoAAA-Evo than with EvoAAA-Gen or EvoAAA-Dif. For instance, average and best values for fashion, glass and mnist are closer in the former case than in the latter. This suggests that EvoAAA-Evo would be preferable if only a few iterations are affordable.

If we strictly focus on the best values achieved by each approach, these returned at the end of all iterations without complexity penalization, EvoAAA-Dif stands out over the other algorithms. It has 5 best values, against 2 for EvoAAA-Gen, 1 for EvoAAA-Evo, and 1 for the random search. To assess the overall performance of each method a ranking is provided in Table 7, with the last row showing the average rank. As can be seen, EvoAAA-Dif is the best performer, closely followed by EvoAAA-Gen and EvoAAA-Evo. The rank difference between random search and exhaustive search is considerable. The bias in the exhaustive search, which tries consecutive configurations until the runtime is out, implies less opportunities to explore the solution space than the random search. By applying a Friedman statistical test over the best MSE values (bottom half of Table 6) a $p\text{-}value = 0.0004248178$ was obtained. This means that there are statistically significant differences among the evaluated optimization strategies.

3.5.2 Analysis of explored space

As can be seen in Table 5, in general the exhaustive and random search approaches explore a much larger space of solutions than the evolutionary methods. In some cases, such as glass, ionosphere, semeion, sonar and spect, this approach examines up to 15 times more configurations than EvoAAA. This is due to that these search algorithms devote all their time in analyzing candidate solutions while the evolutionary methods have other tasks to do, such as individuals selection, crossing, mutation, etc. However, neither random nor exhaustive search never achieve the best performance when AE complexity is taken into account. On the contrary, the MSE for these methods is always higher. Exhaustive search gets stuck in simpler architectures, as stated by its complexity values, despite its usually longer run times (analyzed later) that always go to 24 hours. This behavior is due to the way the exhaustive search has been implemented, trying all possible solutions allowed by the representation in Figure 4 from right to left as is usually done when an interval of values is going to be traversed from beginning to end. Random search has the ability to explore all the solution space, but it is a non-guided approach. In a way the strategies followed by the exhaustive and random search are antagonistic. The former chooses to exploit the local space, trying all the possible solutions of a very reduced area. The second one jumps all over the space of solutions, without taking advantage of past configurations to exploit the locality of the most promising solutions.

The amount of solutions explored by the EvoAAA-Dif, EvoAAA-Gen and EvoAAA-Evo methods is quite similar while working with small datasets, such as glass, semeion, sonar or spect. By contrast, the EvoAAA-Evo and EvoAAA-Dif approaches are able to inspect more candidate structures when larger datasets are used. This could be due to the simpler procedure of EvoAAA-Evo to produce its offspring with respect to EvoAAA-Gen, since crossing is not necessary and the population size is smaller, and to the lower number of iterations conducted by EvoAAA-Dif with respect to EvoAAA-Gen.

3.5.3 Analysis of solution complexity

Another fact to take into account while comparing the different search procedures is the complexity of found architectures. Theoretically, simpler architectures achieving a similar performance would be preferable to more complex ones.

As can be stated by looking at the sixth column in Table 5, the lowest complexity is always that of the **Exh** approach. As said before, the exhaustive search is stuck in small architectures with thousands of combinations for activation functions and loss functions (see Figure 4). However, these are not the best solutions as the MSE values demonstrate.

As would be expected, the random search also produces disparate AE configurations. Sometimes are simpler than those produced by the EvoAAA methods and sometimes are more complex.

Despite some exception, such as fashion and spect, EvoAAA-Gen usually produces simpler AEs with fewer layers and a more compact encoding than EvoAAA-Dif and EvoAAA-Evo. Therefore, at first sight the EvoAAA-Gen search seems to be the best choice, as it provides simpler AEs with better reconstruction capability. However, this comes with a cost as explained below.

3.5.4 Analysis of running times

The running times recorded during the experiments for each configuration are provided in Table 8. As in Table 5, for each data set there are five rows corresponding to the five search approaches. Analyzing this information the following conclusions can be drawn:

Table 8: Running time summary

| Dataset | Running times ↓ | | | | |
	EvoAAA-Dif	EvoAAA-Evo	Exh	EvoAAA-Gen	Rnd
cifar10	24h	24h	24h	24h	24h
delicious	18h 4m.	17h 54m.	24h	6h 51m.	24h
fashion	24h	24h	24h	24h	24h
glass	1h 41m.	1h 20m.	24h	2h 20m.	24h
ionosphere	1h 33m.	0h 44m.	24h	1h 59m.	24h
mnist	24h	24h	24h	24h	24h
semeion	2h 28m.	1h 32m.	24h	5h 0m.	24h
sonar	1h 24m.	1h 10m.	24h	4h 13m.	24h
spect	1h 31m.	0h 59m.	24h	5h 57m.	24h

- The exhaustive and random strategies always reach the limit of 24 hours without being able to examine enough configurations to find a good solution, although the random approach is close to the evolutionary methods in some cases.

- For the larger data sets (cifar10, fashion and mnist) none of the five approaches finish the search process, running out of time.

- In general, EvoAAA-Dif and EvoAAA-Evo only need a fraction of the time used by EvoAAA-Gen to find AE architectures slightly more complex but with a similar performance.

From this analysis a simple guideline can be drawn, choose the EvoAAA-Dif EvoAAA instance if performance is the main and only goal, but consider the EvoAAA-Evo instance if running time is important while sacrificing a bit of reconstruction power. For large data sets the latter approach can save many hours in obtaining a good-enough AE architecture.

3.5.5 Solutions explored over time

The following aspect to be analyzed is how each approach explores the solution space. For doing so, the plots in Figure 5 to Figure 12 show the MSE (Y axis) of the solutions examined through time (X axis)[3]. Since not all methods take the same running time, there are differences among the X axis for a given data set.

---

[3]The analysis described here has been made with the overall results, although only some illustrative cases are graphically shown. Plots corresponding to all combinations of dataset × search method are available in the repository.
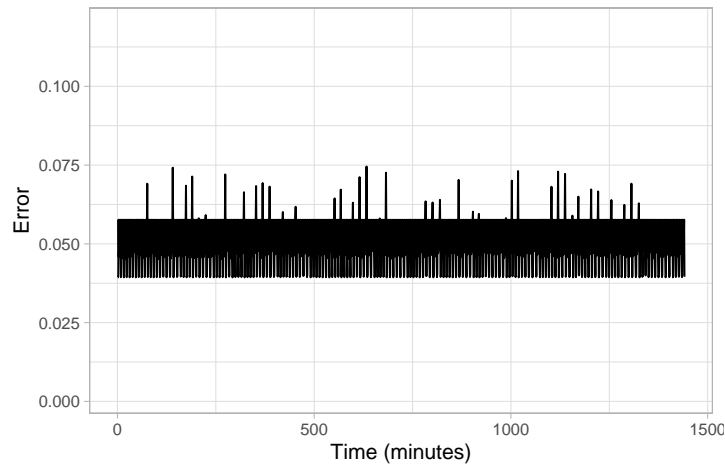
Figure 5: Solutions explored through time

As can be seen, the exhaustive search (Figure 5) keeps a high error rate from start to end. For fashion and mnist, whose attributes are quite similar, good and bad solutions are mixed over time (see Figure 6). For the remainder data sets the search does not seem able to improve much as new solutions are evaluated.
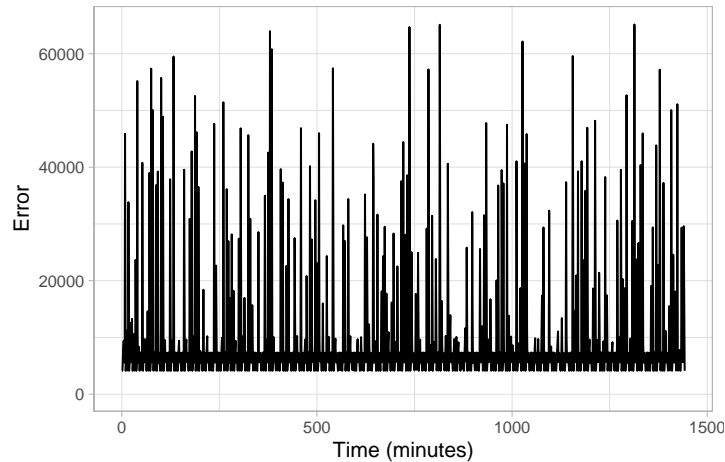

Figure 6: Solutions explored through time

As can be observed in Figure 7, the random search approach has a similar behavior to the exhaustive search, although it has larger fitness variability (the lines are less condensed) among the explored individuals. It is due to its ability to jump over all the solution space, instead of going through every possible combination of parameters.

The behavior of the EvoAAA-Gen and EvoAAA-Evo candidates is similar, as shown in Figure 8 and Figure 9. Both start with lower error rates than the exhaustive and random strategies, and then improve as the running time goes by. This advance in the quality of solutions is not highlighted in the plots, as the Y axis is kept fixed to ease the comparison among the five approaches.

The behavior shown by the EvoAAA-Evo and EvoAAA-Gen methods while working with the glass data set is anomalous as can be observed in Figure 10 and Figure 11. As stated in Table 4, this is a data set with only 9 attributes and a handful of instances. It is probably the hardest case for an AE, since there is no much room to reduce the data representation nor enough patterns to learn it.

By contrast with EvoAAA-Gen and EvoAAA-Evo, the EvoAAA-Dif algorithm seems to have a wider exploration range in most cases, probably due to its larger population. Figure 12 shows the quality of individuals evaluated through time by this search approach.
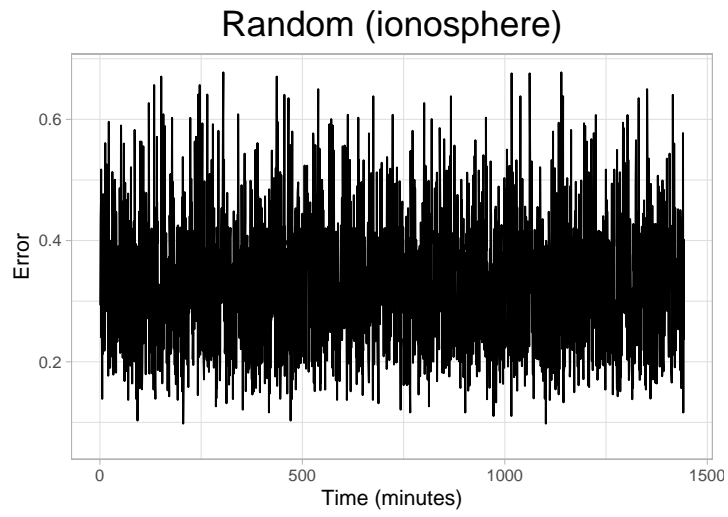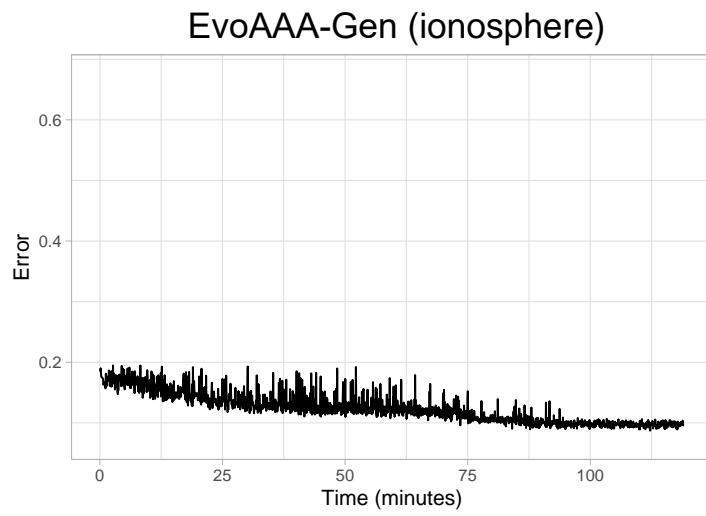
Figure 7: Solutions explored through time

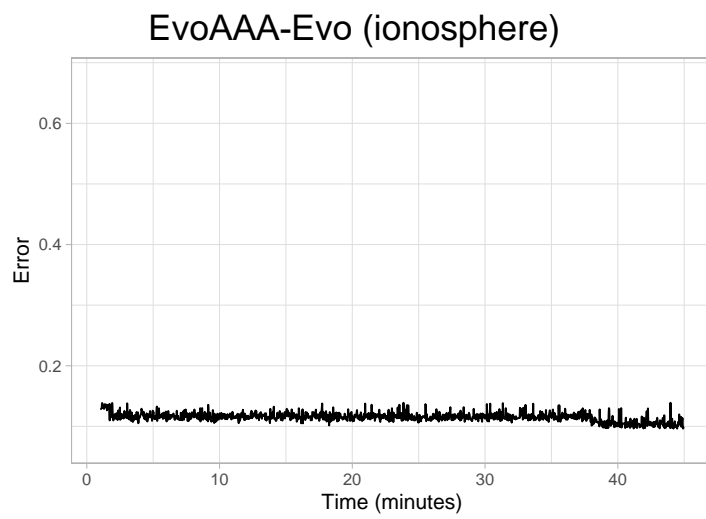Figure 8: Solutions explored through time
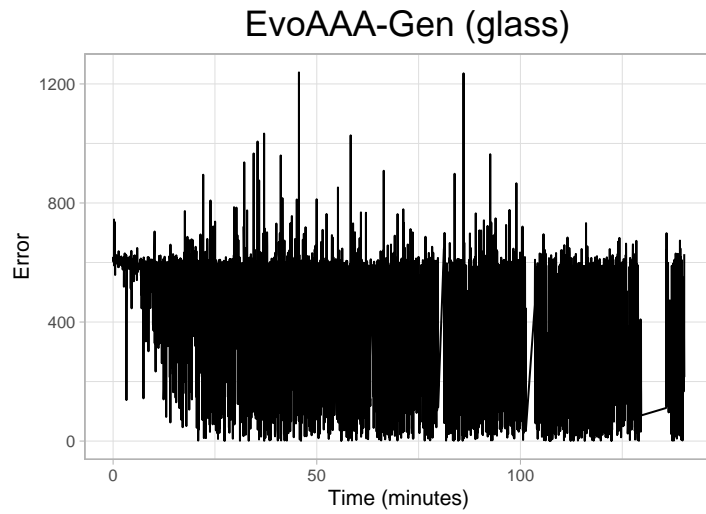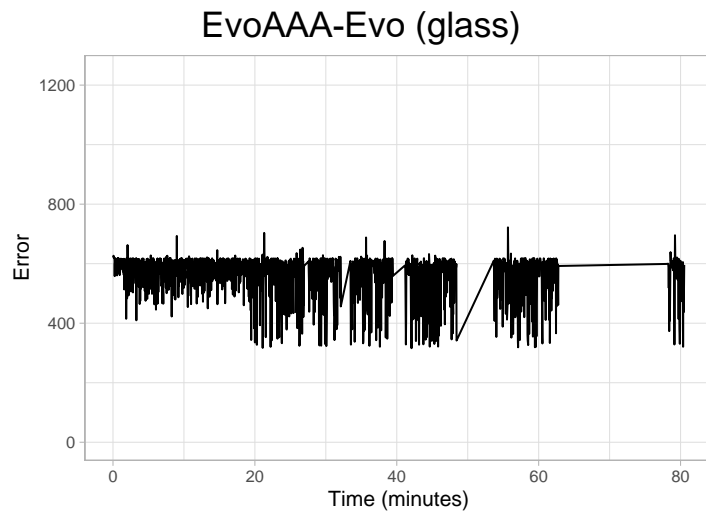

Figure 9: Solutions explored through time

EvoAAA-Gen (glass)

Figure 10: Solutions explored through time

EvoAAA-Evo (glass)

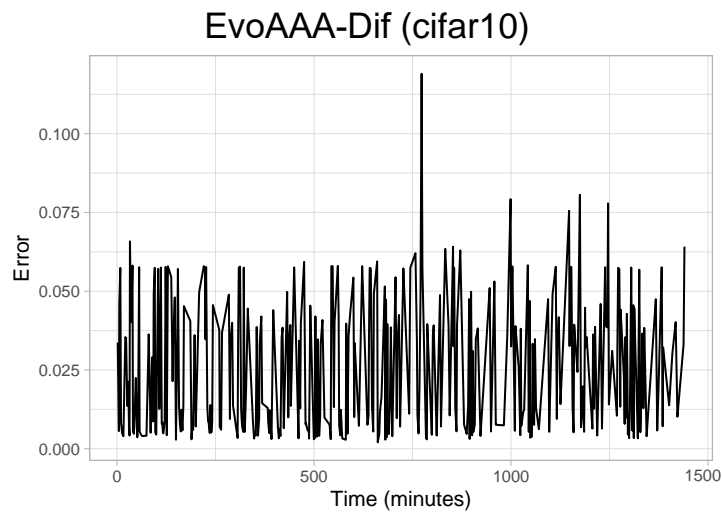Figure 11: Solutions explored through time

EvoAAA-Dif (cifar10)

Figure 12: Solutions explored through time

3.5.6 Achieved improvement vs explored solutions

In Figures 13 to 15 the X axis has been changed from time to number of explored solutions, while the Y axis shows the error of the best solution found until now. The Y axis scale is kept fixed for each dataset, but the X axis scale changes since each

approach examines a different amount of candidates. The goal is to analyze the improvement achieved by each optimization strategy as the they explore more solution space. All plots are available in the repository.

As might be expected, all five methods find better solutions as the number of possible architectures examined grows. However, both the exhaustive method and random search show some rungs as the search progresses, getting sometimes stuck for a long time in the same error level. An example of this behavior can be seen in Figure 13.

Figure 13: Improvement achieved as solutions are explored

The improvements achieved by EvoAAA-Gen and EvoAAA-Evo seem more progressive until they reach their minimum level (see Figure 14). In addition, this lower error rate is achieved after exploring a lower number of potential solutions.
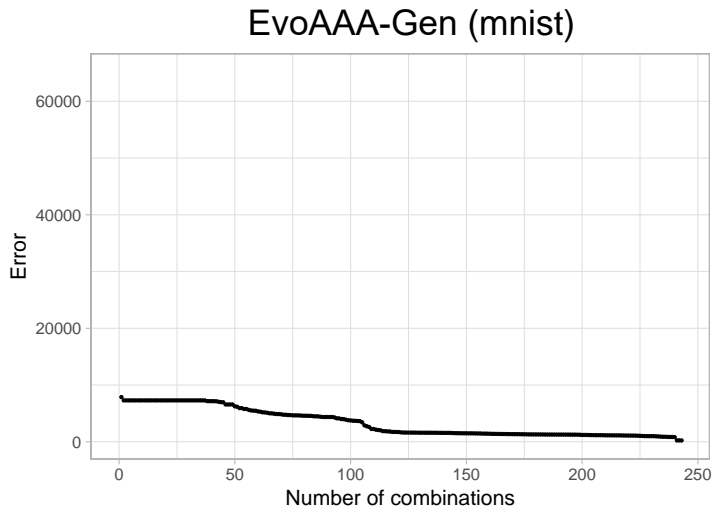

Figure 14: Improvement achieved as solutions are explored

The behavior of the EvoAAA-Dif algorithm is somehow a mix of the previous ones, with a continuous improvement of results and starting with a lower error but having certain similarities with the random strategy.

3.5.7 Search convergence

The following step is to analyze the methods' speed of convergence. In this case there are nine plots available in the repository, one per data set. One of them, shown in Figure 16, is taken as reference for the following analysis. This way the same X and Y scales are shared by the five optimization strategies. The X axis corresponds to running time. Only a portion of the time spent is represented, otherwise the lines that depict the EvoAAA instances would occupy only a small portion of the area, to the left. The lines that continue to the right mean that better values were found later, whereas those that do not reach the X limit indicate the best value achieved is in the plot (i.e. random search with the mnist dataset). From these plots the following conclusions can be extracted:
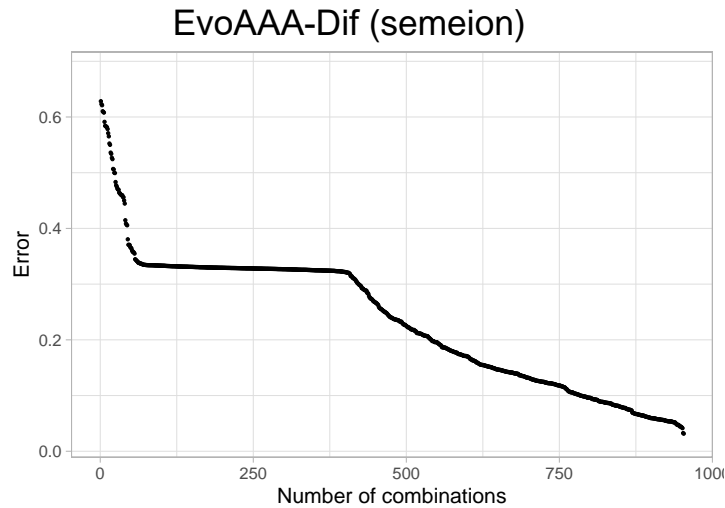
Figure 15: Improvement achieved as solutions are explored
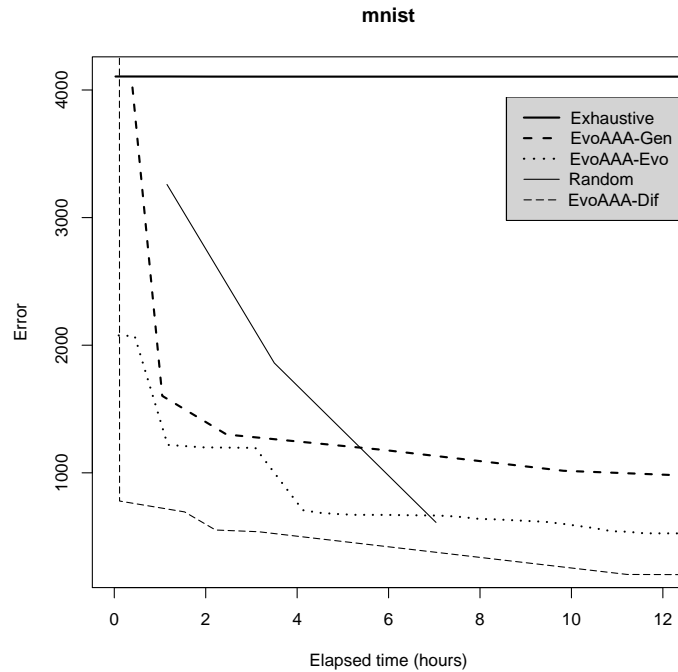

Figure 16: Speed of convergence for each EvoAAA approach against exhaustive and random search

- In general, the exhaustive approach (thick solid line) is stuck in a high error rate most of the time. The only exception is the glass data set, with this search method almost chasing the three EvoAAA strategies.

- The random search (thin solid line) behaves aimlessly as it would be expected. Sometimes it shows a slow progress over time similar to that of the exhaustive strategy (e.g. cifar10 and delicious) while other times it finds a good solution very quickly and then no better solutions are found (e.g. fashion and mnist). It is a strategy that could provide a good result in short time but without guarantees.

- In general, the three EvoAAA instances converge quickly to a lower error value than the two baseline strategies, then stabilize and keep improving at a slower rate.

- In most occasions the ES approach (dotted line) reaches its optimum (lowest error value) before the GA (thick dashed line) does, then finishes the execution. On the contrary, the GA method keeps improving for longer. Due to this behavior, it is able to beat ES in many cases.

- The DE strategy (thin dashed line) usually starts with worse solutions than GA and ES, but in most cases it converges

faster, particularly with the most complex datasets such as cifar10, fashion and mnist. It is able to get the best result in a fraction of the time with respect to the other alternatives in some cases.

On the basis of this analysis, it would be possible to adjust the parameters of the DE and ES algorithms, increasing the population or their number of iterations, in order to run longer and keep improving as the GA does. They would presumably achieve results comparable to those of the GA method for some small datasets (e.g. sonar, glass and spect), or even better with DE.

3.5.8 Influence of the penalization factor

To finish this analysis, how the penalization factor linked to the AEs complexity influence the obtained results is scrutinized. For doing so, the DE strategy has been used over the sonar dataset with α varying from 1 to 0 following a logarithmic scale.

The goal of the α penalization factor is to prefer simpler AE architectures for similar reconstruction performances. Intuitively, lower α values would produce AEs with a higher reconstruction power but also with more layers and units, and the opposite for higher α values (i.e. simpler architectures having lower reconstruction accuracy).
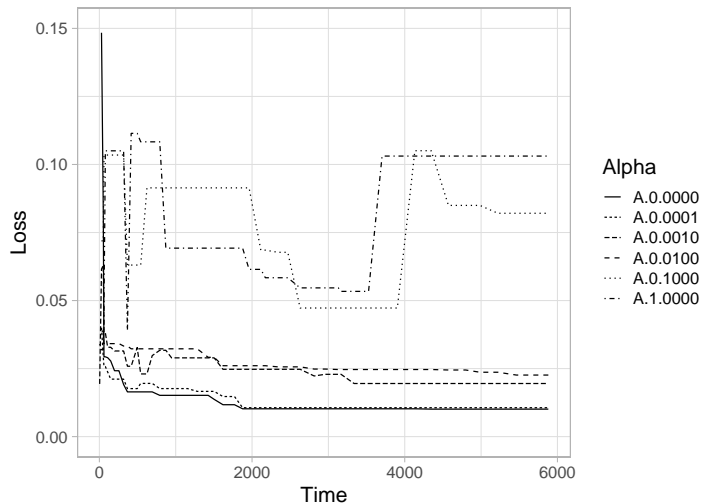

Figure 17: Loss change through time for different α values using EvoAAA-Dif with the sonar dataset

To start with this analysis, Figure 17 shows how the MSE changes through time, as the architectures are explored by the DE algorithm, for the considered α values. As can be observed, the two highest penalization values severely affect the search procedure as the abrupt loss changes reflect. As the DE method tries to improve the performance testing more complex DEs the penalization grows. These rungs are reduced as the α value lowers, until there is no impact with $\alpha = 0$. So, the first outcome would be that it is preferable to have small penalization factors, specifically values that are a fraction of the MSE.

In order to better appreciate the extent to which performance degrades as α increases, the final results obtained with DE from the sonar dataset for each penalization have been represented in Figure 18. Black bars are linked to the left Y axis scale and denote MSE (higher bars are worse), while the line indicates complexity level (right Y axis). Observe that for $\alpha = 0$ and $\alpha = 0.0001$ there is almost no change in performance, but the difference in complexity is remarkable. As penalization factor increases, to the left, the complexity scores reduces but the reconstruction error does the opposite.

Table 9: Loss and AE configuration obtained with each α value using EvoAAA-Dif with the sonar dataset

α	Loss	Layers and units
0.0000	0.00993972	57, 47, 36, 22, 36, 47, 57
0.0001	0.01062195	37, 32, 8, 32, 37
0.0010	0.01951711	23, 2, 23
0.0100	0.02262429	38, 3, 38
0.1000	0.08206710	1
1.0000	0.10308630	1

The exact MSE values and configuration for each considered α are summarized in Table 9. As indicated above, the performance difference between $\alpha = 0$ and $\alpha = 0.0001$ is almost negligible, but the former uses 7 layers instead of 5 for the latter, and the encoding length is 22 units against only 8. Clearly, in this specific case a small penalization factor returns better AE architectures than higher ones or having no penalization at all.
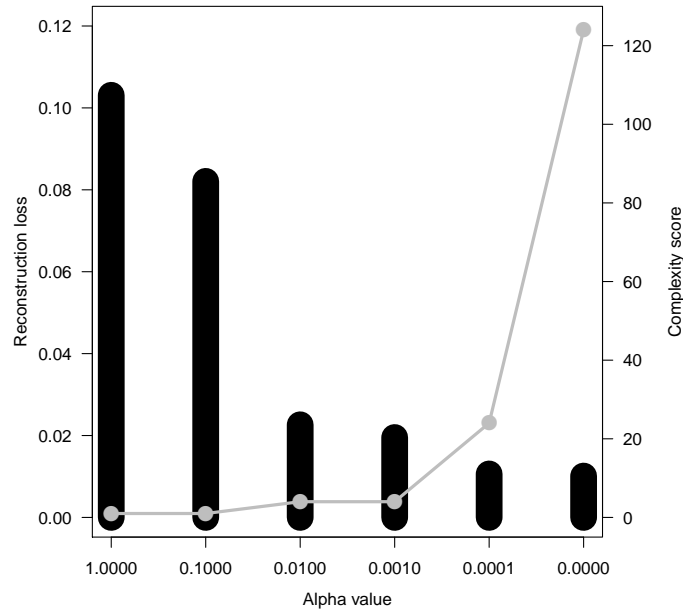
Figure 18: Loss vs AE complexity for different α values using EvoAAA-Dif with the sonar dataset

As a result of this analysis, α was set to 0.0001 for the experiments conducted in this study, as was specified at the beginning of Section 3.

4   Concluding remarks

AEs are a useful tool to face representation learning. However, finding the AE architecture that fits better every case is a difficult task. Internal cross-validation is an usual approach for tuning hyperparameters. However, the solution space is huge if we want to also adjust the AE architecture. Therefore, more powerful methods to face this problem would be needed.

In this paper we have proposed EvoAAA, an evolutionary based approach to find the best AE architecture for each data set. First, a way of encoding the AE architecture within a chromosome has been proposed. It is broad enough to consider most AE variations, including different amounts of layers and units, individual activation functions per layer and several loss functions. Then, three search methods have been planned, one based on differential evolution, another one founded on a genetic algorithm and the other on an evolutionary strategy. Lastly, a thorough experimentation and analysis have been conducted, comparing the results of the three EvoAAA strategies against two different baselines, exhaustive and random search.

Overall, it has been demonstrated that the proposed methodology is able to find a good AE structure for each data set. It may not be the optimal one, as more advanced search algorithms and optimization strategies could improve these results, but it is better than the ones randomly chosen or found through an exhaustive look up if results in a reasonable time are needed. The conducted experiments demonstrate that EvoAAA is a competitive procedure to accomplish the job.

References

[1] C.M. Bishop, *Pattern recognition and machine learning*, Springer, 2006.

[2] T.S. Guzella and W.M. Caminhas, A review of machine learning approaches to spam filtering, *Expert Systems with Applications* **36**(7) (2009), 10206–10222.

[3] S. Bhattacharyya, S. Jha, K. Tharakunnel and J.C. Westland, Data mining for credit card fraud: A comparative study, *Decision Support Systems* **50**(3) (2011), 602–613.

[4] J.B. Schafer, J. Konstan and J. Riedl, Recommender systems in e-commerce, in: *Proceedings of the 1st ACM conference on Electronic commerce*, ACM, 1999, pp. 158–166.

[5] P. Domingos, A few useful things to know about machine learning, *Communications of the ACM* **55**(10) (2012), 78–87.

[6] S. García, J. Luengo and F. Herrera, *Data preprocessing in data mining*, Springer, 2015, pp. 163–194, Chapter 7. ISBN ISBN 978-3-319-10247-4.

[7] M.A. Hall, Correlation-based feature selection for machine learning, PhD thesis, University of Waikato Hamilton, 1999.

[8] H. Peng, F. Long and C.H.Q. Ding, Feature selection based on mutual information criteria of max-dependency, max-relevance, and min-redundancy, *IEEE Transactions on Pattern Analysis and Machine Intelligence* **27** (2005), 1226–1238.

[9] I. Guyon and A. Elisseeff, *An Introduction to Feature Extraction*, in: *Feature Extraction: Foundations and Applications*, I. Guyon, M. Nikravesh, S. Gunn and L.A. Zadeh, eds, Springer Berlin Heidelberg, Berlin, Heidelberg, 2006, pp. 1–25.

[10] D.E. Rumelhart, G.E. Hinton, R.J. Williams et al., Learning representations by back-propagating errors, *Cognitive modeling* **5**(3) (1988), 1.

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

[12] Y. LeCun, Y. Bengio and G. Hinton, Deep learning, *Nature* **521**(7553) (2015), 436.

[13] I. Goodfellow, Y. Bengio and A. Courville, *Deep learning*, MIT press, 2016.

[14] G.E. Hinton and R.R. Salakhutdinov, Reducing the dimensionality of data with neural networks, *Science* **313**(5786) (2006), 504–507.

[15] P. Vincent, H. Larochelle, Y. Bengio and P.-A. Manzagol, Extracting and composing robust features with denoising autoencoders, in: *Proceedings of the 25th international conference on Machine learning*, ACM, 2008, pp. 1096–1103.

[16] P. Vincent, H. Larochelle, I. Lajoie, Y. Bengio and P.-A. Manzagol, Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion, *Journal of machine learning research* **11**(Dec) (2010), 3371–3408.

[17] D. Charte, F. Charte, S. García, M.J. del Jesus and F. Herrera, A practical tutorial on autoencoders for nonlinear feature fusion: Taxonomy, models, software and guidelines, *Information Fusion* **44** (2018), 78–96.

[18] D. Charte, F. Charte, M.J. del Jesus and F. Herrera, A Showcase of the Use of Autoencoders in Feature Learning Applications, in: *From Bioinspired Systems and Biomedical Applications to Machine Learning*, J.M. Ferrández Vicente, J.R. Álvarez-Sánchez, F. de la Paz López, J. Toledo Moreo and H. Adeli, eds, Springer International Publishing, 2019, pp. 412–421. ISBN ISBN 978-3-030-19651-6.

[19] M.R. Garey and D.S. Johnson, *Computers and intractability*, Vol. 29, wh freeman New York, 2002.

[20] T. Bäck and H.-P. Schwefel, An overview of evolutionary algorithms for parameter optimization, *Evolutionary computation* **1**(1) (1993), 1–23.

[21] F.J. Pulgar, F. Charte, A.J. Rivera and M.J. del Jesus, Choosing the proper autoencoder for feature fusion based on data complexity and classifiers: Analysis, tips and guidelines, *Information Fusion* **54** (2020), 44–60.

[22] D. Charte, F. Charte, S. García and F. Herrera, A snapshot on nonstandard supervised learning problems: taxonomy, relationships, problem transformations and algorithm adaptations, *Progress in Artificial Intelligence* **8**(1) (2019), 1–14.

[23] R. Hecht-Nielsen, Theory of the backpropagation neural network, in: *Neural networks for perception*, Elsevier, 1992, pp. 65–93.

[24] H. Robbins and S. Monro, A stochastic approximation method, *The annals of mathematical statistics* (1951), 400–407.

[25] S. Lawrence and C.L. Giles, Overfitting and neural networks: conjugate gradient and backpropagation, in: *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks. IJCNN 2000. Neural Computing: New Challenges and Perspectives for the New Millennium*, Vol. 1, IEEE, 2000, pp. 114–119.

[26] M. Ahmadlou and H. Adeli, Enhanced probabilistic neural network with local decision circles: A robust classifier, *Integrated Computer-Aided Engineering* **17**(3) (2010), 197–210.

[27] N.K. Benamara, M. Val-Calvo, J.R. Álvarez-Sánchez, A. Díaz-Morcillo, J.M. Ferrández-Vicente, E. Fernández-Jover and T.B. Stambouli, Real-Time Emotional Recognition for Sociable Robotics Based on Deep Neural Networks Ensemble, in: *International Work-Conference on the Interplay Between Natural and Artificial Computation*, Springer, 2019, pp. 171–180.

[28] H. Hotelling, Analysis of a complex of statistical variables into principal components, *Journal of educational psychology* **24**(6) (1933), 417.

[29] R.A. Fisher, The statistical utilization of multiple measurements, *Annals of Human Genetics* **8**(4) (1938), 376–386.

[30] L. Cayton, Algorithms for manifold learning, Technical Report, University of California at San Diego, 2005.

[31] J.A. Lee and M. Verleysen, *Nonlinear dimensionality reduction*, Springer Science & Business Media, 2007.

[32] W. Yu, G. Zeng, P. Luo, F. Zhuang, Q. He and Z. Shi, Embedding with autoencoder regularization, in: *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, Springer, 2013, pp. 208–223. doi:10.1007/978-3-642-40994-3_14.

[33] M. Sakurada and T. Yairi, Anomaly detection using autoencoders with nonlinear dimensionality reduction, in: *Proceedings of the MLSDA 2014 2nd Workshop on Machine Learning for Sensory Data Analysis*, ACM, 2014, pp. 4–11. ISBN ISBN 978-1-4503-3159-3. doi:10.1145/2689746.2689747.

[34] S. Park, M. Kim and S. Lee, Anomaly Detection for HTTP Using Convolutional Autoencoders, *IEEE Access* **6** (2018), 70884–70901. doi:10.1109/ACCESS.2018.2881003.

[35] J. Xie, L. Xu and E. Chen, Image denoising and inpainting with deep neural networks, in: *Advances in neural information processing systems*, 2012, pp. 341–349.

[36] X. Lu, Y. Tsao, S. Matsuda and C. Hori, Speech enhancement based on deep denoising autoencoder, in: *Interspeech*, 2013, pp. 436–440.

[37] C. Blum and A. Roli, Metaheuristics in combinatorial optimization: Overview and conceptual comparison, *ACM computing surveys (CSUR)* **35**(3) (2003), 268–308.

[38] H. Adeli and K.C. Sarma, *Cost optimization of structures: fuzzy logic, genetic algorithms, and parallel computing*, John Wiley & Sons, 2006.

[39] E. Aarts and J. Lenstra, Local Search in Combinatorial Optimization Wiley, *New York* (1997).

[40] M. Den Besten, T. Stützle and M. Dorigo, Design of iterated local search algorithms, in: *Workshops on Applications of Evolutionary Computation*, Springer, 2001, pp. 441–451.

[41] R.E. Korf, Real-time heuristic search, *Artificial intelligence* **42**(2–3) (1990), 189–211.

[42] W. Zhang, Algorithms for Combinatorial Optimization, in: *State-Space Search*, Springer, 1999, pp. 13–33.

[43] F. Neri, N. Kotilainen and M. Vapa, *A Memetic-Neural Approach to Discover Resources in P2P Networks*, in: *Recent Advances in Evolutionary Computation for Combinatorial Optimization*, C. Cotta and J. van Hemert, eds, Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 113–129. ISBN ISBN 978-3-540-70807-0. doi:10.1007/978-3-540-70807-0_8.

[44] P.J. Van Laarhoven and E.H. Aarts, Simulated annealing, in: *Simulated annealing: Theory and applications*, Springer, 1987, pp. 7–15.

[45] R. Battiti and G. Tecchiolli, The reactive tabu search, *ORSA journal on computing* **6**(2) (1994), 126–140.

[46] A.A. Freitas, A review of evolutionary algorithms for data mining, in: *Data Mining and Knowledge Discovery Handbook*, Springer, 2009, pp. 371–400.

[47] T. Bäck and H.-P. Schwefel, An Overview of Evolutionary Algorithms for Parameter Optimization, *Evolutionary Computation* **1** (1993), 1–23.

[48] Q. Wang, H.-L. Liu, J. Yuan and L. Chen, Optimizing the energy-spectrum efficiency of cellular systems by evolutionary multi-objective algorithm, *Integrated Computer-Aided Engineering* **26**(2) (2019), 207–220.

[49] C. Kyriklidis and G. Dounias, Evolutionary computation for resource leveling optimization in project management, *Integrated Computer-Aided Engineering* **23**(2) (2016), 173–184.

[50] M. Kociecki and H. Adeli, Two-phase genetic algorithm for topology optimization of free-form steel space-frame roof structures with complex curvatures, *Engineering Applications of Artificial Intelligence* **32** (2014), 218–227.

[51] M. Kociecki and H. Adeli, Shape optimization of free-form steel space-frame roof structures with complex geometries using evolutionary computing, *Engineering Applications of Artificial Intelligence* **38** (2015), 168–182.

[52] C. Blum, Ant colony optimization for the edge-weighted k-cardinality tree problem, in: *Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation*, Morgan Kaufmann Publishers Inc., 2002, pp. 27–34.

[53] R. Poli, J. Kennedy and T. Blackwell, Particle swarm optimization, *Swarm intelligence* **1**(1) (2007), 33–57.

[54] J.A. Foster, Computational genetics: Evolutionary computation, *Nature Reviews Genetics* **2**(6) (2001), 428.

[55] T. Bäck, D.B. Fogel and Z. Michalewicz, *Evolutionary computation 1: Basic algorithms and operators*, CRC press, 2018.

[56] L. Davis, Handbook of genetic algorithms (1991).

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

[58] I. Rechenberg, The evolution strategy. a mathematical model of darwinian evolution, in: *Synergetics—from microscopic to macroscopic order*, Springer, 1984, pp. 122–132.

[59] H. KIM and H. ADELI, Discrete cost optimization of composite floors using a floating-point genetic algorithm, *Engineering Optimization* **33**(4) (2001), 485–501.

[60] F. Friedrichs and C. Igel, Evolutionary tuning of multiple SVM parameters, *Neurocomputing* **64** (2005), 107–117.

[61] S.R. Young, D.C. Rose, T.P. Karnowski, S.-H. Lim and R.M. Patton, Optimizing deep learning hyper-parameters through an evolutionary algorithm, in: *Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments*, ACM, 2015, p. 4.

[62] H. Kitano, Empirical Studies on the Speed of Convergence of Neural Network Training Using Genetic Algorithms., in: *AAAI*, 1990, pp. 789–795.

[63] M. Scholz, A learning strategy for neural networks based on a modified evolutionary strategy, in: *International Conference on Parallel Problem Solving from Nature*, Springer, 1990, pp. 314–318.

[64] L.D. Whitley and T. Hanson, Optimizing Neural Networks Using FasterMore Accurate Genetic Search, in: *Proceedings of the 3rd international conference on genetic algorithms*, Morgan Kaufmann Publishers Inc., 1989, pp. 391–397.

[65] K. Chellapilla and D.B. Fogel, Evolving neural networks to play checkers without relying on expert knowledge, *IEEE transactions on neural networks* **10**(6) (1999), 1382–1391.

[66] D. Floreano, P. Dürr and C. Mattiussi, Neuroevolution: from architectures to learning, *Evolutionary Intelligence* **1**(1) (2008), 47–62.

[67] F. Gruau, Automatic definition of modular neural networks, *Adaptive behavior* **3**(2) (1994), 151–183.

[68] K.O. Stanley, J. Clune, J. Lehman and R. Miikkulainen, Designing neural networks through neuroevolution, *Nature Machine Intelligence* **1**(1) (2019), 24–35.

[69] I. Guyon, L. Sun-Hosoya, M. Boullé, H.J. Escalante, S. Escalera, Z. Liu, D. Jajetic, B. Ray, M. Saeed, M. Sebag et al., Analysis of the AutoML Challenge Series 2015–2018, in: *Automated Machine Learning*, Springer, 2019, pp. 177–219.

[70] C. Ying, A. Klein, E. Christiansen, E. Real, K. Murphy and F. Hutter, NAS-Bench-101: Towards Reproducible Neural Architecture Search, in: *International Conference on Machine Learning*, 2019, pp. 7105–7114.

[71] M. Feurer, A. Klein, K. Eggensperger, J. Springenberg, M. Blum and F. Hutter, Efficient and robust automated machine learning, in: *Advances in neural information processing systems*, 2015, pp. 2962–2970.

[72] B. van Stein, H. Wang and T. Bäck, Automatic Configuration of Deep Neural Networks with Parallel Efficient Global Optimization, in: *2019 International Joint Conference on Neural Networks (IJCNN)*, IEEE, 2019, pp. 1–7.

[73] H. Jin, Q. Song and X. Hu, Auto-Keras: An Efficient Neural Architecture Search System, in: *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, ACM], pages=1946—-1956, year=2019,.

[74] K. Price, R.M. Storn and J.A. Lampinen, *Differential evolution: a practical approach to global optimization*, Springer Science & Business Media, 2006.

[75] A. Krizhevsky, Learning Multiple Layers of Features from Tiny Images (2009).

[76] G. Tsoumakas, I. Katakis and I. Vlahavas, Effective and Efficient Multilabel Classification in Domains with Large Number of Labels, in: *Proc. ECML/PKDD Workshop on Mining Multidimensional Data, Antwerp, Belgium, MMD08*, 2008, pp. 30–44.

[77] H. Xiao, K. Rasul and R. Vollgraf, Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms, 2017.

[78] I.W. Evett and E.J. Spiehler, Rule induction in forensic science, *KBS in Goverment* (1987), 107–118.

[79] V.G. Sigillito, S.P. Wing, L.V. Hutton and K.B. Baker, Classification of radar returns from the ionosphere using neural networks, *Johns Hopkins APL Technical Digest* **10**(3) (1989), 262–266.

[80] L. Deng, The MNIST database of handwritten digit images for machine learning research [best of the web], *IEEE Signal Processing Magazine* **29**(6) (2012), 141–142.

[81] M. Buscema, Metanet*: The theory of independent judges, *Substance use & misuse* **33**(2) (1998), 439–461.

[82] R.P. Gorman and T.J. Sejnowski, Analysis of hidden units in a layered network trained to classify sonar targets, *Neural networks* **1**(1) (1988), 75–89.

[83] L.A. Kurgan, K.J. Cios, R. Tadeusiewicz, M. Ogiela and L.S. Goodenday, Knowledge discovery approach to automated cardiac SPECT diagnosis, *Artificial intelligence in medicine* **23**(2) (2001), 149–169.

[84] J.D. Castro, Arch linux, in: *Introducing Linux Distros*, Springer, 2016, pp. 235–252.

[85] M. Garland, S. Le Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang and V. Volkov, Parallel computing experiences with CUDA, *IEEE micro* **28**(4) (2008), 13–27.

[86] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro and E. Shelhamer, cudnn: Efficient primitives for deep learning, *arXiv preprint arXiv:1410.0759* (2014).

[87] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard et al., Tensorflow: A system for large-scale machine learning, in: *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, 2016, pp. 265–283.

[88] F. Chollet et al., Keras: Deep learning library for theano and tensorflow, *URL: https://keras. io/k* **7**(8) (2015), T1.

[89] D. Charte, F. Herrera and F. Charte, Ruta: implementations of neural autoencoders in R, *Knowledge-Based Systems* **174** (2019), 4–8.

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