Evolutionary Convolutional Neural Networks: an Application to Handwriting Recognition

Article *in* Neurocomputing · March 2018

DOI: 10.1016/j.neucom.2017.12.049

3 authors:

Alejandro Baldominos
University Carlos III de Madrid
53 PUBLICATIONS **191** CITATIONS
SEE PROFILE

Yago Sáez
University Carlos III de Madrid
88 PUBLICATIONS **699** CITATIONS
SEE PROFILE

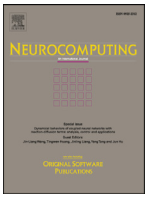
Pedro Isasi
University Carlos III de Madrid
219 PUBLICATIONS **1,593** CITATIONS
SEE PROFILE

Neurocomputing 283 (2018) 38–52

Contents lists available at ScienceDirect

Neurocomputing

journal homepage: www.elsevier.com/locate/neucom



Evolutionary convolutional neural networks: An application to handwriting recognition

Alejandro Baldominos[*], Yago Saez, Pedro Isasi

Computer Science Department, Universidad Carlos III de Madrid, Avenida de la Universidad, 30, Leganes 28911, Madrid

ARTICLE INFO

Article history:
Received 18 September 2017
Revised 6 December 2017
Accepted 20 December 2017
Available online 27 December 2017

Communicated by Jun Yu

Keywords:
Neuroevolution
Evolutionary algorithms
Convolutional neural networks
Automatic topology design
Genetic algorithms
Grammatical evolution

ABSTRACT

Convolutional neural networks (CNNs) have been used over the past years to solve many different artificial intelligence (AI) problems, providing significant advances in some domains and leading to state-of-the-art results. However, the topologies of CNNs involve many different parameters, and in most cases, their design remains a manual process that involves effort and a significant amount of trial and error.

In this work, we have explored the application of neuroevolution to the automatic design of CNN topologies, introducing a common framework for this task and developing two novel solutions based on genetic algorithms and grammatical evolution. We have evaluated our proposal using the MNIST dataset for handwritten digit recognition, achieving a result that is highly competitive with the state-of-the-art without any kind of data augmentation or preprocessing. When misclassified samples are carefully observed, it is found that most of them involve handwritten digits that are difficult to recognize even by a human.

1. Introduction

Convolutional neural networks (CNNs) have received significant attention and had a great impact in recent years, which is partially due to their outstanding behavior in particularly complex supervised learning tasks. These networks have been proven to be particularly powerful when addressing signals and time series of a diverse nature, images, audio and video analysis. In these problems, a fundamental part of the solution is the design of efficient signal preprocessing and feature extraction systems to produce adequate data structures suitable for the classification task. However, CNNs have made preprocessing unnecessary in many domains where it was an integral and essential part of the classification task due to their ability to automatically extract useful features. In many cases, the convolutional layers of CNNs are capable of performing an automatic preprocessing, with an effectiveness in some domains comparable to that which human experts, after thorough and meticulous design processes, are able to achieve. In the case of handwritten character classification, which is the object of study in this work, comparable or even better results can be obtained using CNNs without manual feature engineering or further data transformation.

Nonetheless, the design of CNN architectures remains a meticulous and cumbersome process that requires the participation of specialists in the field. In this work, we attempt to overcome this unwieldy process in such a way that neither the use of preprocessing or other types of *a priori* treatment of the data nor the arduous task of designing and parameterizing the CNN architecture would be required. The main contribution of this paper is that we propose a new procedure based on the use of evolutionary algorithms to automatically discover the most suitable components of the CNN, both for the architecture and the involved hyperparameters, and also introduce an innovative coding scheme of the most relevant parameters of CNN, covering for the first time the evolution of all aspects of design, including the architecture, the activation functions, the learning hyperparameters, etc. This procedure belongs to the field known as "neuroevolution", and although it has been successfully used for almost three decades, its application to every relevant aspect of the design of CNNs is novel.

To validate this proposal and evaluate its performance, we have chosen a very well-known and widely used problem: handwriting recognition. Handwritten character recognition does not pose a real challenge today, even less since the discovery and rise of CNNs, which have been applied extensively to this field over the past years, achieving human-like results for recognition. However, this problem constitutes a good environment for testing the effectiveness of classification methods, as demonstrated by the interest that this problem has aroused in the scientific community over the past years. Recently, the field of CNNs has utilized this domain to

[*] Corresponding author.
E-mail addresses: abaldomi@inf.uc3m.es (A. Baldominos), ysaez@inf.uc3m.es (Y. Saez), isasi@ia.uc3m.es (P. Isasi).

https://doi.org/10.1016/j.neucom.2017.12.049
0925-2312/© 2017 Elsevier B.V. All rights reserved.

evaluate new ideas, to compare different design alternatives, and to test its competitiveness among state-of-the-art solutions.

In particular, we have chosen the MNIST dataset for handwritten digit recognition. The use of this dataset facilitates the replication of the results found in this work, and it allows us to perform an objective comparison with the results reported in the state-of-the-art due to the extensive amount of works that have evaluated their proposals using this domain before, including a large number of original proposals in the field of CNNs.

2. Related work

Neuroevolution has been used for almost three decades to apply evolutionary algorithms for learning weights and topologies of artificial neural networks. Some of the most remarkable techniques developed in the field of neuroevolution include EPNet [1], NEAT [2] or EANT [3]. In this time, neuroevolution has been proven to be successful for finding suitable topologies and weights in neural networks to solve a wide range of problems. However, these networks were often very simple and involved only one hidden layer with few units.

Only since 2014 has the availability of hardware resources enabled the application of neuroevolution to deep and convolutional neural networks, with Koutník et al. [4] publishing the first work in this field[1]. However, in this work, the topology of the neural network is not encoded; rather, the weights of a fixed architecture are evolved, consisting of four convolutional layers with max-pooling and finally a small recurrent network with three hidden units.

In 2015, Verbancsics and Harguess [5] proposed a modification of HyperNEAT [6] to support the evolution of CNNs by adding a new substrate capable of representing this type of network. Unfortunately, their experiments using the evolved CNN MNIST led to a test error rate of 7.9%, one order of magnitude higher than the performance of most CNN-based works. Also in that year, Young et al. [7] introduced MENNDL, a framework for optimizing the hyperparameters of a neural network using GAs; however, their proposal only considered six hyperparameters, the number of filters and the filter size for a fixed 3-layer CNN architecture, resulting in a very limited search space.

In 2016, Loshchilov and Hutter [8] proposed using CMA-ES to evolve 19 hyperparameters of a deep neural network, including optimizer parameters (learning rate, momentum, and so on), batch size, dropout rate, number of filters in the convolutional layers or number of units in the fully connected layer. Nevertheless, their proposal only considered a fixed number of layers and did not evolve most convolutional parameters (e.g., filter sizes, activation functions, and so forth). Although they reported the performance on the MNIST dataset, they appeared to be using a validation set that is different from the standard test set; thus, a fair comparison is not feasible. Also in 2016, Fernando et al. [9] proposed the creation of a differentiable version of a CPPN, called the DPPN. These DPPNs are created using microbial GAs, and they are eventually able to replicate CNN topologies.

Several works aiming at the automatic design of CNN topologies have been published during 2017. Although most of these works rely on neuroevolution, an exception is the paper by Baker et al. [10] introducing MetaQNN, where reinforcement learning is used to search within the space of CNN architectures. Their proposal's performance was tested over MNIST, attaining a test error rate of 0.44%. It is remarkable that MetaQNN does not consider the optimization of learning hyperparameters, the optimization of activation functions or the inclusion of recurrent layers. Related works have recently been published, such as those by Zoph and Le [11], which includes recurrent connections, or by Yu et al. [12], which integrates a tree structure within the deep network to identify more effectively the inter-related learning tasks improving sensitive objects identification with real-world images.

Xie and Yuille [13] have worked on a GA to evolve CNN topologies to perform visual recognition. The authors considered a constrained case with a limited number of layers, with already predefined building blocks, such as convolution or pooling; however, they did not evolve the fully connected or recurrent part of the CNN.

Miikkulainen et al. [14] presented CoDeepNEAT, an automated method for evolving deep neural networks that follows the same working principles as NEAT. CoDeepNEAT allows learning very complex networks involving convolutional, feed-forward and recurrent layers; however, it strongly relies on mutation of these parameters.

Desell [15] recently introduced EXACT, which is mostly focused on describing how the neuroevolutionary algorithm is supported by a largely distributed architecture using volunteer computing. EXACT does not evolve pooling operators, activation functions, fully connected or recurrent layers or other hyperparameters (such as the learning rate). When testing his proposal on the MNIST database, Desell reported an error rate of 1.68%, which is significantly higher than that of most CNN-based approaches.

Additionally, Davison has recently created an open-source project called DEvol [16] for automated deep neural network design using genetic programming. The genome connects several nodes sequentially, with each node representing a layer. The parameters for each layer are also evolved, including the number of filters, the dropout rate, the activation function, and so forth. From the code documentation, it can be inferred that DEvol supports a variable number of convolutional and deep layers. When tested over the MNIST dataset, they have achieved a test error rate of 0.6%, which is fairly good yet not state-of-the-art.

Finally, Suganuma et al. [17] recently published a work using Cartesian genetic programming to optimize CNN architectures; however, they only focus on convolutional layers, and they do not consider fully connected or recurrent layers or the optimization of hyperparameters.

3. Convolutional neural networks

This section does not intend to be an exhaustive study about convolutional neural networks (CNNs); rather, we aim to introduce some key concepts for better understanding this paper's proposal. More detailed works can be found on this subject, such as the paper by Krizhevsky et al. [18] describing AlexNet (the CNN that won 2012's edition of the ImageNet Challenge), or the recent book on deep learning by Goodfellow et al. [19].

CNNs were first introduced by LeCun et al. in 1998 [20,21]. The idea behind CNNs is to combine a feature learning module with a trainable classifier, which often consists of a fully connected network. The feature learning module would replace a prior feature engineering stage, often performed by hand, to reduce data processing to a minimum. In fact, CNNs are intended to work with raw data (or data with very little preprocessing). After features have been learned from raw data, they are introduced to a trainable classifier.

CNNs are interesting for solving many different problems since they provide invariance to translations or local distortions of the input. One of the fields in which this type of network has been most successful is that of computer vision and image understanding [22], but applications to human-based activities are also relevant, such as gesture recognition [23] or activity recognition [24]. Additionally, CNNs rely on the topology and structure of the input data for extracting features. For example, images are 2

[1] According to the authors and also to the best of our knowledge.

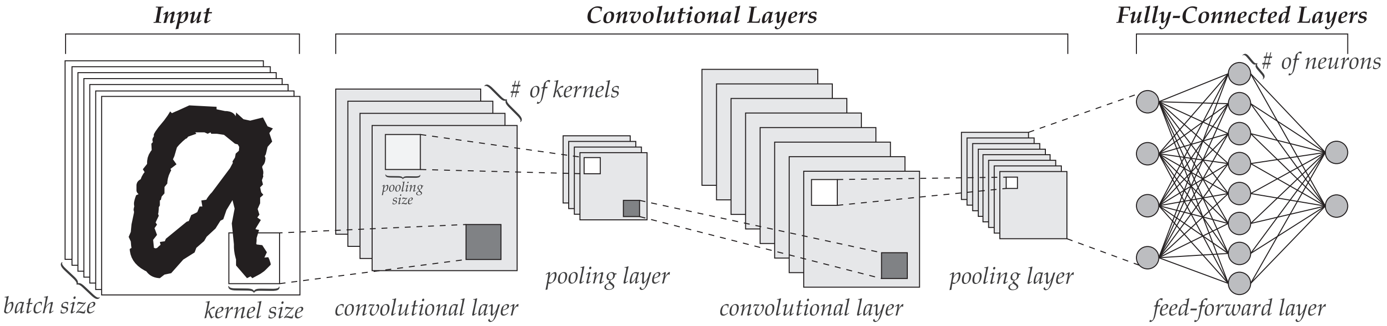


Fig. 1. Typical structure of a sequential convolutional neural network.

dimensional, and CNNs will take advantage of this structure to compute local features, providing a major advantage over traditional feed-forward networks with 1-dimensional inputs.

The typical anatomy of a CNN is shown in Fig. 1 (elements are not to scale), showing some of the key parameters that can be determined. In this work, we have considered networks whose layers are stacked sequentially one after the other, although some authors have explored non-sequential CNNs [25].

3.1. Convolutional layers

The CNN first involves a sequence of one or more convolutional layers. These layers are responsible for performing feature extraction, thus learning relevant features from raw data.

First, raw data will be directly introduced as an input to the first convolutional layer. This layer will output "*feature maps*" from the input, which will then be introduced as the input for the subsequent layer. This process is repeated until there are no more convolutional layers. Because convolutional layers compute feature maps over their input, the greater the number of layers that the network has, the more abstract (or high level) features it will be able to extract.

Each convolutional layer will contain several kernels, also known as filters or patches, which *convolve* the input to generate a feature map as an output. The input data and the kernels will be structured as multidimensional arrays, also known as "*tensors*". For example, if we were working with images, then we would have 2-dimensional inputs and kernels. Whereas the input size is given by the domain or by the output size of the previous layer, the kernel size is defined as a parameter of the network topology.

$$S(i,j) = (K*I)(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n)$$

After feature maps are computed (its number will match the number of kernels), they can be transformed by applying a function element-wise. If the function in this case was non-linear, such as sigmoid, hyperbolic tangent or ReLU, then this process would compute a non-linear transform of the feature map, potentially enabling the extraction of more complex features.

Finally, the output of the layer will be passed as the input for the next layer in the sequence.

3.2. Pooling

Optionally, a pooling operator can be found after a convolutional layer. The purpose of pooling is to reduce the dimensions of the input by performing down-sampling, replacing part of a feature map in a certain location with a statistical summary of the nearby locations. The most common example is max-pooling, where the input is reduced by taking a subtensor of the feature map and replacing it by its maximum value.

Additionally, pooling has been proven to introduce certain invariance to translation, meaning that if the input is slightly translated, most of the pooling output will remain unchanged.

3.3. Fully connected layers

As we previously described, convolutional layers aim to extract relevant features from raw data. Once features are extracted, they can be introduced to a classifier. In most cases, a fully connected network is used for classification, although some works have explored different approaches, such as GoogLeNet using average pooling [25]. To introduce the output tensor of feature maps to the fully connected network, this tensor must be flattened or unrolled, i.e., reshaped into a vector.

The fully connected network can consist of several layers of different types, e.g., feed-forward or recurrent. Fully connected layers are also occasionally called dense layers.

Dense layers are composed of neurons or units. These neurons will process the input and generate an output. In the case of feed-forward layers, they will receive input from neurons in the previous layer through connections (resembling biological synapses) with assigned weights. Regarding recurrent layers, they can be of different types. A basic version of a recurrent layer would match a feed-forward layer but with the input not only coming from the previous layer but also by itself, and a matrix of weights W_h must be considered for these new connections.

Because recurrent layers have connections to themselves, they receive the output from previous time steps, and they are able to learn patterns or functions that depend on temporal context. However, if this context goes back long into the past, then recurrent networks do not appear to be able to learn patterns properly. To solve this issue, LSTMs (long short-term memory) were introduced by Hochreiter and Schmidhuber in 1997 [26]. An interesting variation of LSTM is called GRU (gated recurrent unit), which was introduced by Cho et al. in 2014 [27]. Although there are many more implementations for recurrent cells, the selection of one over the other often has little impact on the outcome [28].

Most relevant parameters of fully connected layers are the number of neurons in the layer and the activation function. The most significant difference arises between linear and non-linear activation functions. There are many different non-linear functions that can be used, and while sigmoid or hyperbolic tangent functions have been extensively used for years, the ReLU function, $f(x) = \max(0, x)$, is currently the most common since it is cheaper to compute and enables faster learning [18].

Finally, a regularization term can be introduced in each layer's weights to prevent overfitting. Common approaches for regularization involves adding either L1 (lasso) or L2 (ridge regression) norms to the loss function. Additionally, a more recent technique called "dropout", which consists of removing a random set of connections during training, has been proven successful [29] for addressing overfitting.

3.4. Learning rules

Once the topology is determined, the network parameters or weights must be learned. This process is called “training” of the neural network.

To train the network, data from a training set will be introduced. In CNNs, raw data will be introduced as input to the first convolutional layer. It is common to introduce a small set of samples, called a “minibatch”, as this approach has been shown to provide faster convergence and more efficient computing of the gradient compared to full-batch gradient descent.

The process for learning the weights is called “learning rule” or “optimizer”, and it uses the value of a loss function computed over the output of the network and the expected output to modify the weights in a certain direction (gradient) to reduce the value of the loss function. The modification of weights is controlled by a parameter called “learning rate”, which has an important impact on how much the weights are modified in each training epoch.

A common learning rule is stochastic gradient descent (SGD), which slightly updates weights in the direction of the gradient. A momentum, also known as Nesterov momentum, can be introduced to control the velocity at which weights are updated. Additionally, some learning rules automatically scale the learning rate during training, such as *adagrad* [30], *adadelta* [31] or *rmsprop* [32]. Recently, *adam* and *adamax* [33] have been proposed, claiming to combine some of the advantages of *adagrad* and *rmsprop*.

The choice of a suitable learning rule and learning rate is important to achieve convergence when optimizing the network weights.

4. Evolution of convolutional neural networks

As shown above, the design of a CNN-based neural system is complex and involves a large number of parameters that can determine the effectiveness of the network to solve a given task. In this work, we attempt to facilitate this task, developing a procedure that is capable of automatically generating a complete design of convolutional neural networks specifically generated to solve a specific problem, in this case, the classification of handwritten text. This implies the ability of evolving many aspects of the architecture, such as the number of layers, connectivity, and so forth, as well as the network operational parameters, such as activation functions, learning rates, and so on.

4.1. Aspects of optimization

However, as noted in the previous section, the number of parameters to consider is too high to be efficiently covered. The only way to perform an effective search is to make some simplifications that do not reduce the chances of finding good solutions. Following this idea, we have organized the parameters into three groups: *convolutional architecture, dense architecture* and *general hyperparameters*.

The convolutional layers must allow us to create any type of architecture following the rules generally established in their design, i.e., sequential layers with neurons of a layer partially connected to a spatially clustered group of the contiguous layer. Each convolutional layer receives values from the preceding layer or from the input if it is the first layer. The values are grouped by one or more kernels, and then a pooling process can be performed.

The most relevant parameters to determine the architecture of the convolutional stages are the number of layers, the number of kernels, their size and activation function in each layer, and the pooling size in the case that max-pooling is performed after one convolutional layer. Regarding the activation function, we only consider two options, linear and ReLU, since the computation of ReLU is more efficient and enables faster learning when compared to alternative non-linear functions such as sigmoid or hyperbolic tangent [18] and is considered a *de facto* standard in deep learning. Moreover, including additional activation functions would increase the search space significantly, thus cancelling the punctual possible advantages of having more choices for activation functions.

Dense levels are less complex. In general, a fully connected feed-forward network can be used along with a backpropagation mechanism for learning the network weights. In this work, we have considered it convenient to also support the use of recurrent architectures; therefore, more powerful and effective models for classification can be generated.

Thus, we will not only parameterize the number of layers and neurons in each layer, which are two critical parameters, but also the connectivity pattern in each layer (feed-forward, recurrent, LSTM, and GRU). This will enable the creation of hybrid networks, where each layer may have a different structure, allowing greater richness and complexity in the alternatives considered, and the creation of new models that are particularly suitable for certain problems, if needed. The activation function is also a parameter to be optimized, and as in the case of convolutional layers, it can be a linear function or ReLU. Additionally, we have included the possibility of L1 and/or L2 and dropout regularization with a rate of 50% after each layer because it is a very common and profitable mechanism to avoid overfitting in CNNs.

Some general hyperparameters, which are not directly related to the topology but rather to the learning process, will also be optimized. The first of these parameters is the batch size: modifying the size of training minibatches can alter the convergence behavior of the process. Finally, we will also optimize the learning rule and the learning rate, which will also affect the way in which weights are learned.

In summary, in our proposal, we have considered the following parameters:

- Convolutional layers
  - Number of convolutional layers
  - Number of kernels of each convolutional layer
  - Kernel size in each convolutional layer
  - Activation function in each convolutional layer
  - Pooling size (if any) after each convolutional layer
- Dense layers
  - Number of dense layers
  - Connectivity pattern of each dense layer
  - Number of neurons of each dense layer
  - Activation function in each dense layer
  - Weight regularization in each dense layer
  - Dropout (none or 50%) in each dense layer
- General hyperparameters
  - Batch size
  - Learning rule
  - Learning rate

Once the main parameters for the CNN design have been identified, an efficient search procedure for these parameters is required.

4.2. Neuroevolution procedure

In our approach, we will use evolutionary algorithms to optimize the previously described parameters of convolutional neural networks. The use of evolutionary computation with the purpose of evolving any aspect of neural networks is known as “neuroevolution” in the literature.

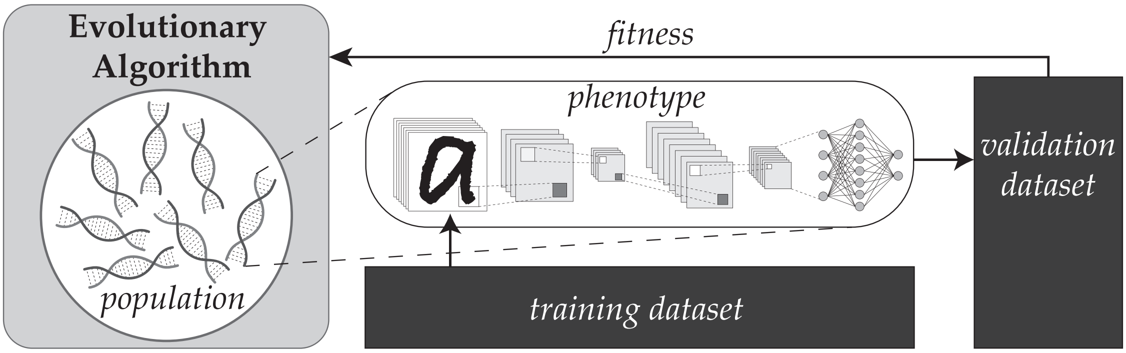


Fig. 2. General framework for neuroevolution of CNNs.

A general framework for neuroevolution in the context of this paper is presented in Fig. 2. We have designed this framework to be general to fit any evolutionary computation technique, and later in the section, we propose two specific instances of evolutionary algorithms: genetic algorithms and grammatical evolution.

In general, the evolutionary algorithm consists of a population of individuals, represented by a genotype. The genotype encoding depends on the specific technique, and in some cases, it must be manually designed. Additionally, there must be a mechanism that is capable of translating the genotype into a phenotype, which will be a definition of a CNN topology by means of the previously specified parameters.

Once we have a phenotype, we can evaluate its performance. For this purpose, we will train a neural network model with the given topology using a training dataset. Then, once the model has been learned, we will stop the training and will test its performance over a validation set, which is disjoint with the training set using a given metric (e.g., accuracy, precision, F1 score, and so on). The value of this metric will be the fitness value of the individual, and this fitness will be provided as feedback to the evolutionary algorithm.

In brief, evolutionary algorithms work as follows: an initial population is randomly generated, and the performance of its individuals is computed following the procedure described above. Then, a sequence of evolutionary operations (e.g., selection, crossover, mutation, and so on) are performed, which rely on the fitness of the individuals to promote the survival of the fittest. The execution of all these operators is called a “generation”, and upon completion, it will lead to a new generation of individuals. Following the Darwinian mechanism of evolution, subsequent populations will have fitter individuals, and it is eventually likely that we will find very good individuals, which in this case translate to successful topologies that perform well on the chosen dataset.

However, such a procedure will have to address two fundamental problems:

- The large (potentially infinite) range in which parameters can be tuned.
- The considerable amount of time needed to evaluate each design.

To solve the first problem, we have decided to discretize the range of possible values that each parameter can have. This decision reduces the search space while still making finding good solutions feasible, without reducing their potential quality too much. In most cases, intermediate values do not make significant differences in the design of the network or in its effectiveness. For instance, regarding the number of kernels, we have chosen a maximum number of 256. However, we do not believe that small differences in the number of kernels would have a significant impact on the network performance. Therefore, intermediate values of the number of kernels have been ignored, thus providing the possibility to choose only between more significant values as 2, 4, 8, 16, 32, 64, 128 or 256. In most cases, only some few significant values have been chosen as an alternative, regardless of whether finer adjustments can be performed in successive stages, if necessary.

The time required for evaluating each of the alternative solutions is one of the major disadvantages for considering a search procedure, even if it is very efficient. When working with population-based search methods, which are almost unanimously used in evolutionary computing, there cannot be considerable improvement unless a large number of alternatives are taken into account, which implies a large number of evaluations. In the present case, each evaluation involves the complete training of a convolutional neural network and its exploitation, making it unfeasible to handle a sufficient number of alternatives for the system to produce significant improvements.

To overcome this drawback, we have decided to make estimations of the effectiveness of the networks rather than to conduct a thorough evaluation. Therefore, networks will be trained using only a reduced sample of existing data and for a small number of training epochs. In this way, the time required by the evaluation process is greatly reduced at the expense of obtaining less accurate evaluations, an approach known as “fitness approximation”. Nevertheless, estimating the effectiveness of networks with few samples and few iterations, although providing poorer results, will rarely affect networks in an irregular manner. It is reasonable to assume that such estimations will not include significant biases toward particular architectures or penalize in a particular way certain others. Furthermore, when using a selection operator based on the principles of natural selection, a common property of most evolutionary computing techniques, accurate evaluations are not a requirement. Rather, what is needed is a fair comparison among solutions to know which ones are better than the remaining solutions. Hence, the ways in which estimations are performed in this work are valid and effective in the scope of evolutionary searching.

To check whether this scheme is invariant to the evolutionary procedure involved, two different evolutionary methods have been used: genetic algorithms (GAs) and grammatical evolution (GE). We have decided to first use GAs as the evolutionary procedure for evolving CNN architectures, given its popularity and since it is a well-known evolutionary computation technique, which can be used as a baseline. Later, we have addressed the problem using GE to reduce the redundancy present in the GA binary encoding and to provide a more flexible definition of the CNN topology.

4.3. Genetic neuroevolution

GAs were first described by John H. Holland in 1975 [34] and have since been extensively used to solve a variety of search and optimization problems.

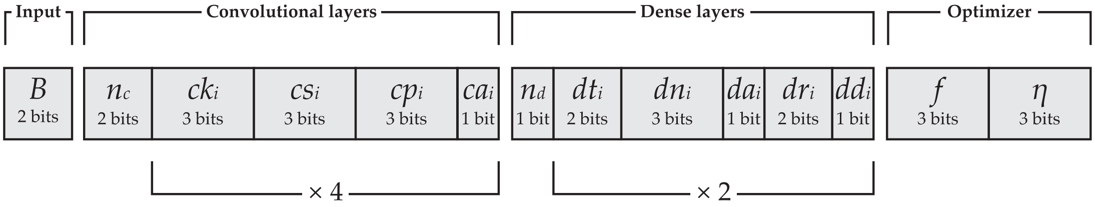


Fig. 3. Definition of the chromosome in the genetic algorithm for the MNIST dataset.

In GAs, binary encoding is often used for the genotype, although different types can be used in practice. The encoding must be designed by the researcher, as well as a mapping function for translating the genotype into the phenotype.

In our GA, a 69-bit binary string has been conceived for the genotype using Gray encoding. A brief summary of the genotype's structure is shown in Fig. 3. Next, we will explain this structure in further detail, as well as how the genotype is processed to be converted into a phenotype.

The chromosome encodes the following parameters, where γ is the integer corresponding to the Gray binary substring. The first parameter defines the input configuration:

- B: the batch size (2 bits), computed as $B = 25 \cdot 2^{\gamma}$, thus taking values $B \in [25, 50, 100, 200]$.

The following five parameters define the setup of the convolutional layers:

- n_c: the number of convolutional layers (2 bits), computed as $n_c = 1 + \gamma$, thus taking values between $n_c = 1$ and $n_c = 4$.
- ck_i: the number of kernels in the ith convolutional layer (3 bits), computed as $ck_i = 2^{(\gamma+1)}$, thus taking values $ck_i \in [2, 4, 8, 16, 32, 64, 128, 256]$.
- cs_i: the kernel size of the ith convolutional layer (3 bits), computed as $cs_i = 2 + \gamma$, thus guaranteeing that the minimum value is $cs_i = 2$ and that the maximum value is $cs_i = 9$. Squared kernels are enforced; thus, cs_i refers to the numbers of both rows and columns.
- cp_i: the pooling of the ith convolutional layer (3 bits), computed as $cp_i = 2 + \gamma$, thus guaranteeing that the minimum value is $cp_i = 2$ and that the maximum value is $cp_i = 9$. Squared pooling is enforced.
- ca_i: the activation function of the ith convolutional layer (1 bit), which can be either *rectify* (ReLU, $\gamma = 0$) or *linear* ($\gamma = 1$).

Because there will be at most 4 convolutional layers, the chromosome repeats 4 times the genes for ck_i, cs_i, cp_i and ca_i. However, the network will only consider the setup for only the first n_c layers, and it will ignore the remaining. The following six parameters define the setup of the dense layers:

- n_d: the number of dense layers (1 bit), computed as $n_d = 1 + \gamma$, thus taking values between $n_d = 1$ and $n_d = 2$.
- dt_i: the type of the ith dense layer (2 bits), which can be either *rnn* ($\gamma = 0$), *lstm* ($\gamma = 1$), *gru* ($\gamma = 2$) or *dense* (non-recurrent, $\gamma = 3$).
- dn_i: the number of neurons in the ith layer (3 bits), computed as $dn_i = 2^{(3+\gamma)}$, thus taking values $dn_i \in [8, 16, 32, 64, 128, 256, 512, 1024]$.
- da_i: the activation function of the neurons in the ith layer (1 bit), which can be either *rectify* (ReLU, $\gamma = 0$) or *linear* ($\gamma = 1$).
- dr_i: the regularization applied to the weights of the i-th layer (2 bits), which can be either *none* ($\gamma = 0$), *l1* ($\gamma = 1$), *l2* ($\gamma = 2$) or *l1l2* ($\gamma = 3$).
- dd_i: the dropout probability for the weights in the ith layer (1 bit), which is computed as $dd_i = \gamma/2$, thus taking values $dd_i = 0$ (no dropout) or $dd_i = 0.5$.

Because there can be up to 2 dense layers, the chromosome repeats twice the genes for dt_i, dn_i, da_i, dr_i and dd_i. However, the network will only use the parameters for the first n_d layers, and it will ignore the others.

Finally, the last 2 parameters store the configuration of the learning process:

- f: the optimizer or gradient descent update function (3 bits), which can be either *sgd* ($\gamma = 0$), *momentum* ($\gamma = 1$), *nesterov* ($\gamma = 2$), *adagrad* ($\gamma = 3$), *adamax* ($\gamma = 4$), *adam* ($\gamma = 5$), *adadelta* ($\gamma = 6$) or *rmsprop* ($\gamma = 7$).
- η: the learning rate (3 bits), which can be either $1 \cdot 10^{-5}$ ($\gamma = 0$), $5 \cdot 10^{-5}$ ($\gamma = 1$), $1 \cdot 10^{-4}$ ($\gamma = 2$), $5 \cdot 10^{-4}$ ($\gamma = 3$), $1 \cdot 10^{-3}$ ($\gamma = 4$), $5 \cdot 10^{-3}$ ($\gamma = 5$), $1 \cdot 10^{-2}$ ($\gamma = 6$) or $5 \cdot 10^{-2}$ ($\gamma = 7$).

4.4. Grammatical neuroevolution

In addition to genetic algorithms, we have decided to test our framework using an additional evolutionary computation technique, namely, grammatical evolution (GE), which was introduced by Ryan, Collins and O'Neill in 1998 [35].

Unlike GAs, GE uses an integer-based encoding that is provided by the technique, and it does not need to be designed by researchers. Moreover, the genotype-phenotype mapping mechanism is also provided by the algorithm; however, it relies on the definition of a formal grammar, which must be provided by the researchers. This formal grammar will generate a language such that the set of strings belonging to such language must be all the valid phenotypes.

The reason to use GE is twofold: it provides a more flexible definition of the phenotype and simultaneously prevents some of the redundancies present in the GA encoding. Additionally, GE could potentially generate infinite languages; however, we will not exploit this possibility since we want to keep the search space within boundaries.

Regarding our proposal for the GE, Fig. 4 shows the definition of the grammar used for generating individuals in Backus–Naur form (BNF).

We can observe how the grammar can generate phenotypes that are very similar to those encoded in the GA. However, in GE, we have more freedom to specify a variable number of values for each parameter. This is due to how the GA encoding works: to avoid redundancy, we decided that each parameter would have a set of values whose cardinality is always a power of 2. In GE, we no longer need to enforce this condition; thus, we can feel free to set any number of values. For instance, the number of units in the dense layers can take seven values, there are five possible values for the learning rate, six possible values for the number of kernels and the kernel size, and so forth.

Consequently, we have decided to remove values that were uncommon in the winning individuals of the GA, thereby considerably reducing the search space. Moreover, following the same logic, the maximum number of convolutional layers has been reduced to three.

Additionally, while the GA had some redundancy in the number of convolutional and dense layers (if not all the layers were used,

```
<dnn>       ::= <conv_lys> <dense_lys> <opt_setup>

<conv_lys>  ::= <conv> | <conv> <conv> |
                <conv> <conv> <conv>
<conv>      ::= <n_kernels> <k_size> <act_fn> <pool>
<n_kernels> ::= 8 | 16 | 32 | 64 | 128 | 256
<k_size>    ::= 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<pool>      ::= null | <p_size>
<p_size>    ::= 2 | 3 | 4 | 5 | 6

<dense_lys> ::= <dense> | <dense> <dense> |
                <dense> <dense> <dense>
<dense>     ::= <d_type> <n_units> <act_fn>
                <reg_fn> <dropout_r>
<d_type>    ::= rnn | lstm | gru | dense
<n_units>   ::= 32 | 64 | 128 | 256 | 512 | 1024
<act_fn>    ::= rectify | linear
<reg_fn>    ::= null | l1 | l2 | l1l2
<dropout_r> ::= 0 | 0.5

<opt_setup> ::= <opt_type> <learn_rt> <batch>
<opt_type>  ::= sgd | nesterov | momentum | adamax |
                adagrad | adam | adadelta | rmsprop
<learn_rt>  ::= 5E-1 | 1E-1 | 5E-2 |
                1E-2 | 5E-3 | 1E-3
<batch>     ::= 25 | 50 | 100
```

Fig. 4. Definition of the grammar in Backus–Naur form for the MNIST dataset.

then the chromosome contained genetic information that was ignored when building the phenotype), this redundancy is naturally removed when using GE. Consequently, we expect GE to converge faster than GA.

4.5. Accelerating fitness computation

As previously mentioned, we have performed some simplifications to reduce the training time. When dealing with deep neural networks, the training process often requires large amounts of time to provide robust and competitive results. This is an important issue when this training process takes part of the fitness computation in an evolutionary computation scheme. In this case, a full training in the dataset with the current setup takes an average of 15 minutes. While this is an affordable time for training a single network, a neuroevolutionary process comprising a population of 50 individuals evolving for 100 generations would take 53 days, which is an impractical amount of time.

To tackle this problem, in each fitness computation, we have trained the network with only 5 epochs and using a random 50% sample of the training set in each epoch. The obtained result is used as a proxy of a more exhaustive training, taking only a fraction of the time required to train a network with the entire training set. By doing so, we have reduced the time required by the evolutionary process to complete, to slightly more than 2 days in our hardware setup.[2]

This simplification results in a pessimistic estimation of the classification capacity of the network, but it will not affect the evolutionary process since nothing suggests that the proposed mechanism may introduce some bias or preferences in the evaluation of some individuals over others. What is important in the evolutionary process is not the precision of the evaluations but rather to allow a fair comparison between the different alternatives. In addition, in this case, to avoid any possible bias, the tournament selection operator has been used, in which the probability of selecting individuals to generate successors does not depend on the difference of the values of the evaluations but rather on the position in a ranking.

4.6. Preserving diversity

Another problem that could arise is the convergence of individuals to similar solutions early in the evolutionary process. This could have a negative impact on the outcome because suboptimal solutions could be found. Meanwhile, it is desirable that the system explore different regions of the search space as a priority over the exploitation of well-known regions because the existence of better but similar solutions does not imply a considerable increase in quality. However, taking less conventional architectures into account can produce larger quality jumps, for which it is necessary to maintain the genetic diversity of the population during the entire process.

Two measures were taken to guarantee genetic diversity across generations. First, note that some CNN architectures are invalid (cannot be executed because the number or size of the convolutional or pooling layers would require a larger input). When the population is randomly initialized, there is a high probability of creating these invalid individuals, thus significantly reducing the effective population, i.e., the one containing valid individuals. We have prevented them from appearing in the initial population.

Second, we have implemented a niching strategy because we found that otherwise many individuals rapidly converge to a single solution, an issue that could lead to local suboptima. We address the niching scheme by having two separate fitness values for each individual: the nominal fitness and the adjusted fitness. The nominal fitness is the fitness as we have described it thus far: the classification error of the trained CNN. The adjusted fitness is computed from the nominal fitness and will worsen if the individual is “very similar” to the remaining of individuals of the current population.

For this purpose, we first have to define an affinity metric that is capable of computing how similar two CNN architectures are. Eq. (1) provides a formal definition of the similarity between two individuals, i_i and i_j:

$$sim(i_i, i_j) = \begin{cases} \frac{|i_i \cap i_j|}{|i|}, & \text{if } n_c^{(i)} = n_c^{(j)} \text{ and } n_d^{(i)} = n_d^{(j)} \\ 0, & \text{otherwise} \end{cases} \qquad (1)$$

Let us explain Eq. (1) in further detail. First, we will only consider two individuals to have a certain degree of similarity when they have the same number of both convolutional and dense layers; otherwise, they will be considered as completely different. In the case that both n_c and n_d match for the two individuals, we will check how many properties that they have in common. A property is a certain configuration parameter, e.g., the batch size (B), the learning function (f), the number of neurons in the first dense layer (dn_1) or the learning rate (η), to mention a few. In Eq. (1),

[2] Time spent per generation: average of 31 min and 40 s, median of 29 min and 26 s, standard deviation of 10 min and 38 s.

$|i_i \cap i_j|$ refers to the cardinality of the intersection of the properties set of both individuals, i.e., the number of properties that both have in common, while $|i|$ is the total number of properties, which will be equivalent for both i_i and i_j. Note that because both individuals have the same number of layers, they will also have the same number of properties, and the similarity metric will lie in the range [0, 1].

Once the similarity function is defined, the adjusted fitness (f_a) is computed from the nominal fitness (f_n) following Eq. (2).

$$f_a(i_i) = f_n(i_i) \times \left(1 - \frac{\sum_{i_j \in P, j \neq i} sim(i_i, i_j)}{|P| - 1}\right) \qquad (2)$$

In summary, the adjusted fitness is computed just as the product of the nominal fitness by a factor that is inverse to the average similarity of the individual with the remaining of individuals of the population. In Eq. (2), P is the set of individuals of the current population. We can observe how if the individual were completely different from the remainder of the population (i.e., its similarity with all other individuals was always zero), then the adjusted fitness and the nominal fitness would be equivalent. Conversely, if all individuals of the population were identical, then all adjusted fitness values would be zero regardless of the nominal fitness values. Both extreme cases are very unlikely to occur in a real-world scenario.

Note that if the fitness is a metric to be minimized (e.g., error), Eq. (2) can be adjusted simply by replacing the product by a division, considering the unlikely case of a division by zero as an infinite value.

5. Evaluation

We will now first describe the environment and setup with enough detail to enable reproducibility of the results reported in this paper, and then we will carefully examine the results and discuss them in depth.

5.1. MNIST database for handwritten recognition

To evaluate the ability of our approach to competitively produce CNN architectures, we will be using a well-known domain: handwriting recognition. This domain has been a choice for applying convolutional neural networks due to their ability to automatically extract local features from multidimensional data, providing state-of-the-art results. Specifically, we will use the MNIST database for handwritten digit recognition, which was introduced two decades ago and has been extensively reviewed in the literature.

5.1.1. Database characteristics

The MNIST (Mixed National Institute of Standards and Technology) database was introduced in 1998 by Yann LeCun, Corinna Cortes, and Christopher J.C. Burges [20]. Since then, MNIST has been extensively used to test machine learning applications and pattern recognition techniques. MNIST contains 60,000 training samples and 10,000 test samples of gray-scale handwritten digits. Half of the data are from NIST's Special Database 1, which was collected from high-school students. The other half of the data are from NIST's Special Database 3, which was collected from Census Bureau employees. The set of writers of the training set and the test set is disjoint, and the training set contains samples from over 250 writers.

The original images were normalized to fit in a 20 × 20 pixel box while preserving their aspect ratio. The images were originally black and white, although they were converted into gray-scale after applying an anti-aliasing filter during the normalization process. Finally, padding was added to fit the images into a larger 28 × 28 pixel figure such that the center of mass of the pixels matched the center of this 28 × 28 box. Fig. 5 presents an example of one sample retrieved from the MNIST training set corresponding to the digit '7' and displaying the 28 × 28 pixel grid. Meanwhile, Fig. 6 displays 10 samples for each digit between 0 and 9 retrieved from the MNIST training set. Although the task of guessing a handwritten digit may appear easy for a human, some particular samples can easily be confused: for example, a '4' can be mistaken for a '9', and some digits can be difficult to recognize (examples can be found in the figure, such as the ninth '2' or the ninth '7').

In this paper, we have not performed any further preprocessing or augmentation of the MNIST database. Improvements in the

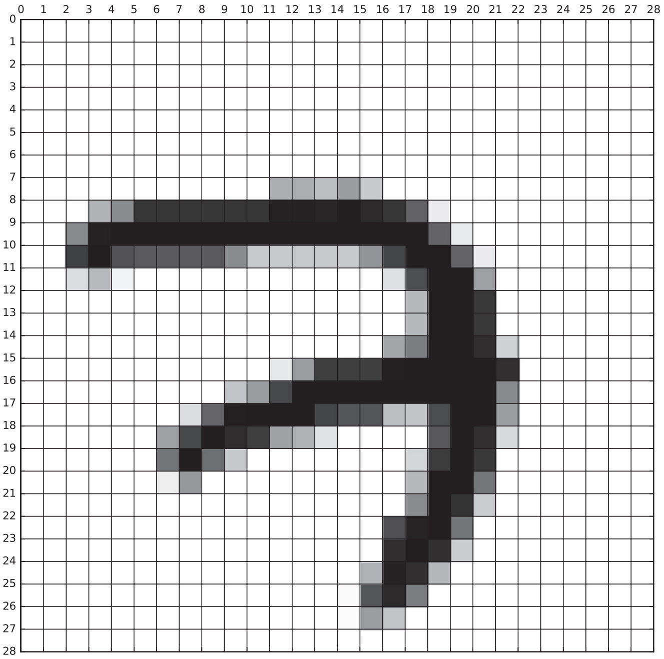


Fig. 5. MNIST sample corresponding to the digit '7'.

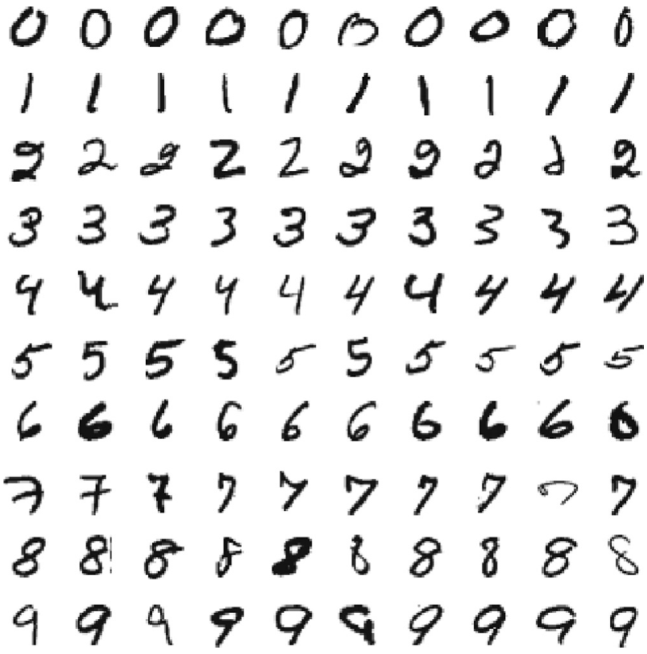


Fig. 6. Ten samples for each digit in the MNIST dataset.

Table 1
Side-by-side comparison of the most competitive (error rate $<$ 1.0) results found in the state-of-the-art for the MNIST database without data augmentation or preprocessing.

Technique	Error Rate (%)
HOPE+DNN with unsupervised learning features [36]	0.40
Deep convex net [37]	0.83
CDBN [38]	0.82
S-SC + linear SVM [39]	0.84
2-layer MP-DBM [40]	0.88
DNet-kNN [41]	0.94†
2-layer Boltzmann machine [42]	0.95
Batch-normalized maxout network-in-network [43]	0.24†
CNN with gated pooling function [44]	0.29
Inception-Recurrent CNN + LSUV + EVE [45]	0.29†
Recurrent CNN [46]	0.31
CNN with norm. layers and piecewise lin. act. units [47]	0.31†
CNN (5 conv, 3 dense) with full training [48]	0.32
Fractional max-pooling CNN with random overlapping [49]	0.32†
CNN with competitive multi-scale conv. filters [50]	0.33†
Fast-learning shallow CNN [51]	0.37
CNN FitNet with LSUV initialization and SVM [52]	0.38
Deeply supervised CNN [53]	0.39
Convolutional kernel networks [54]	0.39
CNN with multi-loss regularization [55]	0.42
CNN (3 conv maxout, 1 dense) with dropout [56]	0.45
Convolutional highway networks [57]	0.45
CNN (5 conv, 3 dense) with retraining [48]	0.46
Network-in-network [58]	0.47
CNN (3 conv, 1 dense), stochastic pooling [59]	0.49†
CNN (2 conv, 1 dense, ReLU) with dropout [60]	0.52
CNN, unsup pretraining [56]	0.53
CNN (2 conv, 1 dense, ReLU) with DropConnect [60]	0.57
SparseNet + SVM [61]	0.59
CNN (2 conv, 1 dense), unsup pretraining [62]	0.60
CNN (2 conv, 2 dense) [63]	0.62
Boosted Gabor CNN [64]	0.68
CNN (2 conv, 1 dense) with L-BFGS [65]	0.69
Fastfood 1024/2048 CNN [66]	0.71
Feature Extractor + SVM [67]	0.83
Dual-hidden Layer Feedforward Network [68]	0.87
CNN LeNet-5 [20]	0.95

results could potentially be achieved by performing data augmentation; however, specific preprocessing or augmentation techniques would introduce another variable to include in the overall comparison with the state-of-the-art.

5.1.2. State-of-the-Art in MNIST database

Throughout this section, we will discuss the current state-of-the-art in terms of the MNIST dataset. We are not interested in discussing the specifics of every work reporting results on this database; rather, we want to highlight the performance of the best performing models, and the reader is referred to the references for further detail.

The MNIST database has been extensively used to evaluate the performance of machine learning techniques. For this reason, several rankings have been published in the past, using MNIST to perform a comparative evaluation. Most of the existing literature on MNIST uses “test error rate” (in %) as the metric for evaluating the performance of different classifiers. This metric is computed as the ratio between the number of incorrectly classified instances and the total number of instances in the test set; thus, it is equivalent to $1 - a$, where a is the classification accuracy.

Table 2
List of parameters used for the genetic algorithm and grammatical evolution.

Parameter	Symbol	Value
Population size	$\|P\|$	50
Maximum number of generations	G	100
Generations without improvements (stop cond.)	G_s	30
Tournament size	τ	3
Minimum number of points in crossover (GA)	x_{min}	3
Maximum number of points in crossover (GA)	x_{max}	10
Crossover rate (GE)	β	0.7
Mutation rate	α	0.015
Elite size	e	1

The most competitive models, namely, those with a test error rate smaller than 1%, can be found in Table 1. To enable a fair comparison, only works that do not use any preprocessing or data augmentation are reported. The upper side of the table shows the performance of classical machine learning approaches (including SVMs and non-convolutional NNs), while the lower side presents the results of convolutional neural networks. When the authors reported different results using the same technique, only the best result is shown. Additionally, to achieve a very exhaustive comparison, some works published in academic repositories but not subject to peer review have been included and are displayed in the table along with the † symbol.

As shown, CNNs provide considerably better results than other techniques. The test error rate is as low as 0.24% when using maxout network-in-network [43]. These results have been improved even further with data augmentation, although they are not considered in this paper to enable a fair comparison.

5.2. Environment

We have performed all the experiments in two compute nodes with two GPUs, both NVIDIA GeForce GTX1080. With this configuration, we could train up to 4 CNNs in parallel. Each node features an Intel Core i7-6700 CPU; however, the CPU is not used for CNN training or evaluation.

Regarding the software stack, we have used Ubuntu Linux 16.04 LTS as the operating system and version 375.66 of the NVIDIA proprietary drivers, along with CUDA Toolkit 8.0 and cuDNN 6, which is a GPU-accelerated library of primitives for deep neural networks from which most deep learning frameworks can benefit.

For developing our proposal, we have used Python 2.7.12, along with NumPy 1.12.1, SciPy 0.19.0, Scikit-learn 0.18.1, Cython 0.25.2 and Pygpu 0.6.5.

Theano 0.9.0 is chosen as the deep learning framework due its versatility. Moreover, Lasagne 0.2.dev1 was used to simplify the design of CNN topologies because it is a very convenient abstraction for our system, with little overhead and impact on performance.

5.3. Experimental setup

In this section, we will describe the experimental setup, both for the GA and GE. Some of the most remarkable parameters are displayed in Table 2.

To determine the population size, the maximum number of generations and the stop condition, we needed to establish a tradeoff that would guarantee an acceptable level of convergence while reducing the time required by the evolutionary algorithm. After some preliminary experiments, we found that it was unlikely that an improvement was achieved after 30 generations without improvements, and given this stop criterion, the algorithm would in most cases finish before 100 generations occurred. Convergence is clearly not guaranteed under these terms, but it remains a good approximation that establishes an acceptable upper bound for the time required by the algorithm to complete. The remaining parameters have been chosen after a preliminary evaluation, concluding that they are acceptable for guiding the evolution toward good individuals.

As we already described earlier in the paper, since fitness evaluations are expensive because they require training and evaluating the CNN, we have performed some simplifications to reduce the training time. In particular, in each fitness computation, the network is trained with only 5 epochs and using a random 50% sample of the training set in each epoch. The obtained result is used as a proxy of a more exhaustive training, taking only 12.5% of the time required to train a network with the entire training set over 20 epochs.

To obtain significant results and avoid bias caused by the random initialization of the initial population, each experiment will be repeated 10 times.

The complete procedure for evaluating each candidate solution is as follows:

1. Translate the genotype into a phenotype by creating a CNN topology with the parameters specified by the genotype.
2. Randomly initialize the network's weights.
3. Train the network during 5 epochs, using a random 50% of the data in each epoch.
4. Compute the classification error of the network on a set that is different from the training set and assign the obtained error as the individual fitness, which must be minimized.

Note that this fitness function is stochastic since the network weights are randomly initialized in each computation.

Throughout the execution of the evolutionary system, the best individuals found thus far are stored such that at the end of the process we will have a set called *hall-of-fame*, with the 20 best architectures found throughout the evolutionary process. Note that the evaluations of individuals are just estimations of the effectiveness of the architectures that they represent and will often constitute a lower bound of this effectiveness due to the use of sampling and a small number of epochs.

The evolutionary process is repeated 10 times, each of them with a different initial population randomly generated, thereby avoiding possible biases due to initialization. In addition, the hall-of-fame set is shared along all the evolutionary processes of the system, and in this way, we help to preserve the genetic diversity of the solutions. This will also increase the variability among the individuals in the hall-of-fame set given that each evolutionary process will be initialized with a new random population.

Upon completion of the entire process, we will end up with the best 20 CNN topologies in the hall-of-fame set. All these topologies correspond to the best estimated individuals found thus far, but their performance is far from that which can be achieved with full training. Thus, we need to perform a more exhaustive learning process on each topology using the complete training set without sampling during 30 epochs. Since this process is also stochastic, we will repeat this full training stage 20 times for each topology, each time with a new set of randomly initialized weights.

Eventually, this will lead to 20 models for each topology, which will be evaluated over the test set and will likely have a better error rate than the individuals resulting from the evolutionary process. In the next section, we will study the effectiveness of these models and will discuss the best topology and the error provided by the best-performing model.

5.4. Results

First, we will discuss the results obtained for both GA and GE with the MNIST dataset, explaining how competitive individuals are obtained by thoroughly training the fittest individuals and discussing the obtained results.

5.4.1. Genetic neuroevolution

After the evolutionary process, the best individual had a fitness of 0.68%, which is significantly better than the fitness of the second individual of 0.84%. The following individuals' fitnesses barely vary, reaching as high as 0.87% for the tenth individual.

Table 3
Summary of errors (in %) of the best 20 individuals of the genetic algorithm after full training in MNIST.

#	Mean	Std. Dev.	Median	Minimum	Maximum
1	0.5130	0.037290	0.515	0.41	0.58
2	0.5380	0.031221	0.535	0.49	0.60
3	0.5485	0.035582	0.545	0.49	0.65
4	0.5875	0.045291	0.580	0.50	0.67
5	0.6115	0.049340	0.620	0.46	0.67
6	0.5010	0.031772	0.505	0.45	0.56
7	0.5760	0.035004	0.575	0.50	0.64
8	0.6085	0.038289	0.605	0.53	0.70
9	0.4795	0.031702	0.485	0.40	0.53
10	0.6090	0.032428	0.600	0.57	0.67
11	0.5600	0.040262	0.570	0.49	0.63
12	0.5850	0.034259	0.570	0.54	0.65
13	0.7095	0.045361	0.720	0.60	0.78
14	0.6045	0.037902	0.615	0.54	0.66
15	0.5780	0.036216	0.580	0.47	0.64
16	0.6265	0.056965	0.625	0.51	0.72
17	0.6010	0.035968	0.600	0.51	0.67
18	0.6815	0.033604	0.685	0.62	0.73
19	0.4805	0.036343	0.485	0.40	0.53
20	0.5930	0.037290	0.585	0.52	0.66

After performing the full training for each of the 20 topologies in the hall-of-fame set, we have obtained 20 models for each of them. Recall that in each run, the architecture is trained from scratch and evaluated over the test set. Table 3 provides a statistical summary of the errors' distribution obtained for each of these individuals expressed as a percentage, including the mean, standard deviation, minimum, median and maximum. Additionally, Fig. 7 depicts this distribution as a boxplot, including the mean, which is shown as a small triangle within each box.

The minimum error found after the entire process is 0.40%. This error was achieved with a CNN topology consisting of 2 convolutional layers with 64 and 256 kernels, respectively, of size 8 in the first layer and 5 in the second layer. Pooling is performed after the two convolutional layers: in the first case, the pooling size is 2, and in the second, the pooling size is 3. There is only one feedforward layer with 256 neurons, receiving a dropout regularization of 50%. The activation function is ReLU for both the convolutional layers and the dense layer. The network has been trained using the *adagrad* algorithm with a learning rate of 0.01 and minibatch size of 25 samples.

5.4.2. Grammatical neuroevolution

After completing the evolutionary process with GE, we observed that compared with the winning architectures of GA, the fitness was consistently better. This result makes sense because we designed an encoding that removed redundancy and reduced the search space. In this case, the best individual had a fitness of 0.66%, and all individuals in the top-20 had a higher fitness than the second-best individual from GA.

As before, full training of the 20 topologies in the hall-of-fame set occurred after the evolutionary process. The statistical summary of the performance for each topology is shown in Table 4. Additionally, the error distribution for each individual is depicted as a boxplot in Fig. 8.

Again, we can observe how the results for each individual are consistently better than when using GA. For example, there are test error rates smaller than 0.4%, whereas no error rate is higher than 0.7% for any of the top-20 topologies. We attribute this improvement in the performance to the fact that GE removes redundancy in the encoding and provides higher flexibility to remove

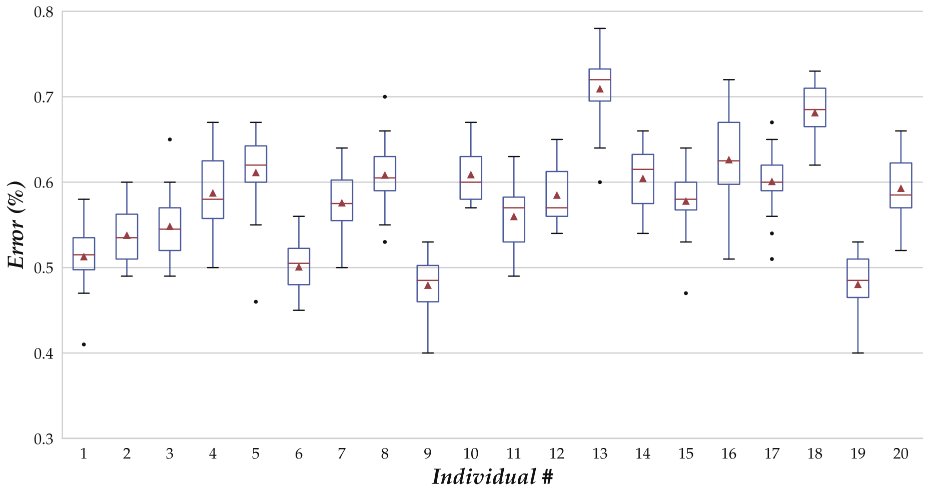


Fig. 7. Boxplot showing the distribution of errors of the best 20 individuals of the genetic algorithm after full training in MNIST.

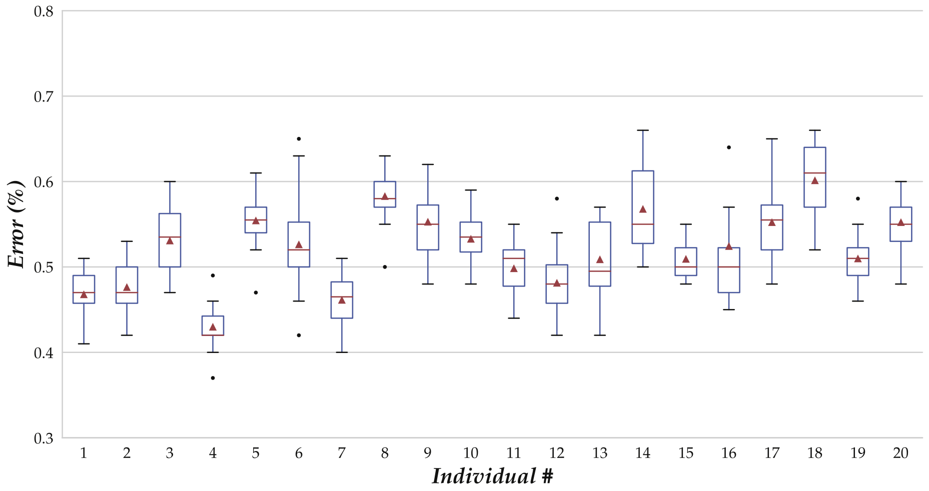


Fig. 8. Boxplot showing the distribution of errors of the best 20 individuals of the grammatical evolution after full training in MNIST.

Table 4
Summary of errors (in %) of the best 20 individuals of the grammatical evolution after full training in MNIST.

#	Mean	Std. Dev.	Median	Minimum	Maximum
1	0.4680	0.026675	0.470	0.41	0.51
2	0.4765	0.032650	0.470	0.42	0.53
3	0.5310	0.040510	0.535	0.47	0.60
4	0.4300	0.025752	0.420	0.37	0.49
5	0.5545	0.030517	0.555	0.47	0.61
6	0.5265	0.054219	0.520	0.42	0.65
7	0.4615	0.032971	0.465	0.40	0.51
8	0.5830	0.030279	0.580	0.50	0.63
9	0.5530	0.037850	0.550	0.48	0.62
10	0.5330	0.031473	0.535	0.48	0.59
11	0.4985	0.030997	0.510	0.44	0.55
12	0.4815	0.041584	0.480	0.42	0.58
13	0.5090	0.046668	0.495	0.42	0.57
14	0.5680	0.049161	0.550	0.50	0.66
15	0.5095	0.021879	0.500	0.48	0.55
16	0.5245	0.105555	0.500	0.45	0.93
17	0.5525	0.042904	0.555	0.48	0.65
18	0.6015	0.040036	0.610	0.52	0.66
19	0.5100	0.028470	0.510	0.46	0.58
20	0.5525	0.032098	0.550	0.48	0.60

some of the uncommon values for certain parameters of the topology, thereby reducing the search space.

With GE, the best error found was 0.37%. This error was achieved with a CNN topology consisting of 3 convolutional layers with 64, 256 and 256 kernels of sizes 4, 2 and 7, respectively. The first and third layers implement a linear activation function, whereas the second layer implements ReLU. Pooling is only performed after the last convolutional layer, with a size of 6. There is only one feed-forward layer with 1024 neurons with a ReLU activation, receiving a dropout regularization of 50%. The network has been trained using the *adamax* algorithm with a learning rate of 0.001 and minibatch size of 100 samples.

We can observe how this best result of 0.37% achieves a very good position in the state-of-the-art ranking for MNIST, positioning in the top-10 (see Table 1). More interestingly, individuals are consistently good: even the worst-performing models after full training attain a test error rate lower than 1%. This consistency in providing competitive results for all the models resulting from topologies in the hall-of-fame set shows the robustness of our proposal.

5.4.3. Comparison with related work

Table 5 presents a brief comparison of the proposal in this paper against the related works that were presented in Section 2. In this table, a dagger (†) is depicted next to two of the works to indicate that these two works have been included due to their relevance but do not use evolutionary computation techniques to

Table 5
Brief comparison of this papers's features with related works.

Work	Var. Ly.	Conv.	FC	Rec.	Act. Fn.	Opt. HP	MNIST (%)
Koutník et al. [4]							–
Verbancsics et al. [5]		•		•			7.90
MENNDL [7]		•					–
Loshchilov et al. [8]		•	•			•	–
MetaQNN [10] †	•	•	•				0.44
Zoph et al. [11] †	•	•	•	•	•		–
GeNet [13]	•	•					–
CoDeepNEAT [14]	•	•	•	•	•	•	–
EXACT [15]	•	•					1.68
Real et al. [69]	•	•					–
DEvol [16]	•	•	•		•		0.60
Suganuma et al. [17]	•	•					–
This paper	•	•	•	•	•	•	0.37

search for optimal CNN topologies. The abbreviations shown in the table header stand for the following criteria:

- **Var. Ly.**: whether the proposal supports a variable number of layers (either convolutional, fully connected, recurrent, and so on).
- **Conv.**: whether the proposal evolves the convolutional layers or some of their parameters.
- **FC**: whether the proposal evolves fully connected layers or some of their parameters.
- **Rec.**: whether the proposal observes the inclusion of recurrent layers or LSTM cells.
- **Act. Fn.**: whether the proposal evolves the activation function rather than hardcoding it.
- **Opt. HP**: whether the proposal supports the evolution of optimized hyperparameters (learning rate, momentum, batch size, and so on).

When comparing our proposal with the related works shown in the table, we find that our approach is the only one, along with CoDeepNEAT, to cover the full spectrum of the topologies of CNNs: convolutional layers, fully connected layers (including recurrent), activation functions and optimized hyperparameters. Most of the other works focuses only on some aspects of the network topology, most commonly on the evolution of convolutional layers. Whereas CoDeepNEAT has a similar coverage as this work, the approaches are far from equivalent, as the former strongly relies on mutation and involves a very different procedure. We have also found that in those cases where the related works report a result on MNIST's standard test set, our result (0.37%) is more competitive.

5.5. Discussion

We will first provide some critical discussion of the best topologies obtained during the evolutionary process.

If we look for some common patterns in the topologies of the top-10 individuals, we find that the topologies differ among them, and this effect can be attributed to two causes: (1) The use of niching to preserve genetic diversity and (2) the execution of 10 different runs with different initial populations. Nevertheless, some common patterns arose when taking a closer look at some of the values. For instance, none of the top-10 individuals had either one or four convolutional layers in the GA, resulting in all of them having either two or three layers. This can occur because one layer is insufficient to extract meaningful features from the data, and four layers may be too much given the small size of the input. This also occurs in GE when no individual had only one convolutional layer, which appears to reinforce the idea that one layer is insufficient to build useful features from raw data. However, unlike in the case of the GA, all of these individuals except for one had three convolutional layers, with the other one having two layers. Thus, it appears that three convolutional layers are the most convenient setup for achieving the best results in the MNIST database.

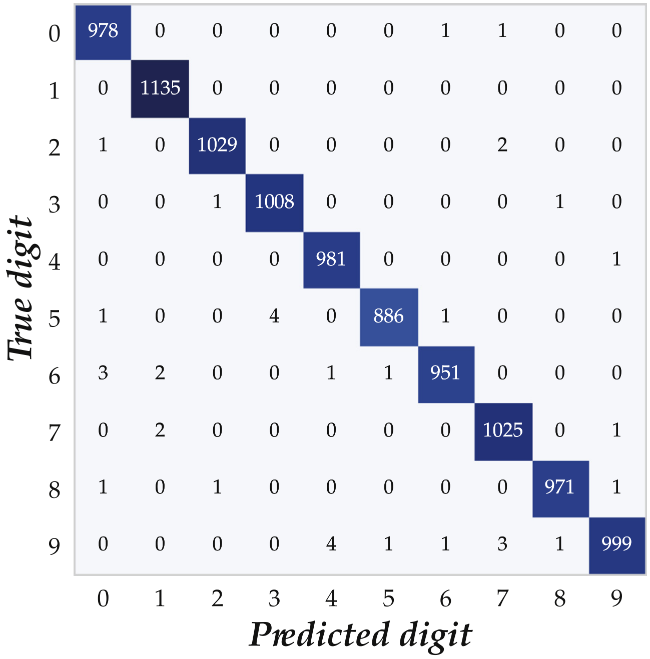
Also regarding the convolutional setup, all individuals had the maximum number of kernels (256) and a non-linear activation function (ReLU) in at least one of their layers. This result can be due to linear transformations not being sufficient to extract valid features from raw data. In GE, we observed that while many individuals implemented pooling, they did not apply this reduction in more than one layer, perhaps because the network structure would otherwise be invalid.

Another common pattern that can be found is the lack of recurrent layers along the fully connected architecture, which does not appear to be required for properly solving the problem at hand. This result makes sense because the input is fairly small and does not involve a temporal dimension. In GE, all the best individuals of them contained only one dense layer, except for one that consisted of two layers, and the number of neurons in the feed-forward layers was always larger than 128.

We could also observe that only one individual applies L2 regularization to one of its dense layers. This result would mean that L1 or L2 regularization is not useful for this domain. The same behavior is not found in dropout, a different form of regularization, as some of the architectures involved a dropout of 50% in one of their dense layers, thus proving useful in certain cases.

Regarding the learning rate, it was never smaller than $\eta = 1 \cdot 10^{-3}$, even if the encoding allowed values as low as $\eta = 1 \cdot 10^{-5}$ in the GA. It is quite likely that such small learning rates are unable to provide an accurate model in as few as five epochs. Finally, in GE, all the optimizers in the best 10 individuals were always either *adagrad, adamax* or *adadelta*, and the learning rate was always 0.5 for *adadelta*, 0.001 for *adamax*, and between 0.005 and 0.01 for *adamax*.

The best result found for MNIST is a test error rate of 0.37% when evolving the topologies using GE. This is a very competitive result considering that the objective was not to beat the best result of the state-of-the-art but to easily and automatically obtain competitive architectures. In this line, the difference in error rate between the result obtained and the best one reported to date is minimal, reducing in practice to eight errors of difference in the set of 10,000 of the test. Meanwhile, another detail that supports the quality of our result is the fact that there is no one technique, other than those involving CNNs, that has better results, as we can find in the upper side of Table 1. Aditionally, because several constraints were placed on the evolutionary process due to the lack of resources, we believe that the results could be further improved by performing a fine-tuning of the CNN architecture.



	0	1	2	3	4	5	6	7	8	9
0	978	0	0	0	0	0	1	1	0	0
1	0	1135	0	0	0	0	0	0	0	0
2	1	0	1029	0	0	0	0	2	0	0
3	0	0	1	1008	0	0	0	0	1	0
4	0	0	0	0	981	0	0	0	0	1
5	1	0	0	4	0	886	1	0	0	0
6	3	2	0	0	1	1	951	0	0	0
7	0	2	0	0	0	0	0	1025	0	1
8	1	0	1	0	0	0	0	0	971	1
9	0	0	0	0	4	1	1	3	1	999

Fig. 9. Confusion matrix of the best model using the grammatical evolution individuals with MNIST.

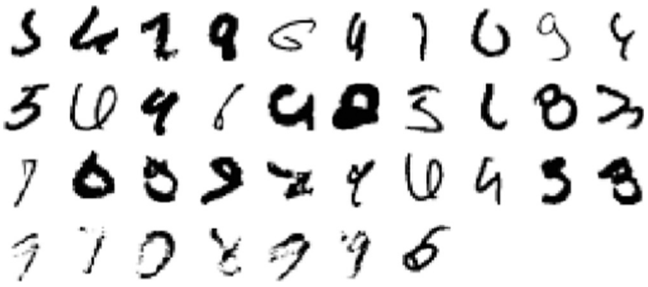


Fig. 10. The 37 misclassified images in the MNIST test set with the best model obtained with grammatical evolution.

Regarding the difference between the GA and GE, the latter has shown a slightly better behavior, which can be attributed to the fact that the encoding has less redundancy and, as a consequence, the search space is reduced.

An error of 0.37% translates into 37 misclassified digits. Fig, 9 shows the confusion matrix for the MNIST dataset with the best model. As shown, the confusion matrix is almost perfect, having almost all of their values in the main diagonal. Moreover, the main errors involve four digits ‘9’ classified as the digit ‘4’ and four digits ‘5’ classified as ‘3’s. The entire set of misclassified digits can be found in Fig, 10. It is noticeable how some ‘9’s are drawn with an open circle at the top, resembling a ‘4’. Additionally, some ‘6’s can be easily confused with ‘0’s, and in general, it can be acknowledged that those digits are poorly written and difficult to recognize even for humans.

6. Conclusions and future work

Convolutional neural networks have currently become the best solution for tackling many different types of artificial intelligence problems, including many applications developed by key players such as Google or Facebook. In recent years, one of the domains that has received substantial benefits from research advances in CNNs is image recognition, given the ability of this type of network to automatically extract local features from raw data while remaining aware of the data structure.

However, CNNs involve a vast number of parameters. Even when only sequential setups are considered, where data are passed through layers one after the other, there are many different key decisions regarding the topology of the network. To mention a few, some important aspects of the structure involve the number of convolutional layers, the number of filters in each layer, the filters’ sizes, the use of pooling after each layer, the number of fully connected layers, the type of neurons in these layers (feed-forward or recurrent), the number of neurons, the activation functions, and some hyperparameters of the learning process, such as the learning rule or the learning rate.

Until now, most works have provided handcrafted topologies for CNNs, specifically designed toward solving a specific problem. This manual design of the topology is expensive and requires trial and error to determine the quality of a solution. In addition, in real-world scenarios, the inputs used by the CNNs are dynamic, and topologies that worked fine in the past can worsen over time. In this paper, we have proposed an approach for the automated neuroevolution of convolutional neural networks, i.e., the application of evolutionary computation to determine their optimal topologies. This proposal can be used both for designing a CNN topology from zero and for optimizing existing ones. For our proposal, we first described a general-purpose framework for how such a system should be developed. In this framework, an evolutionary algorithm is responsible for evolving a population of individuals, where each individual involves the definition of a CNN topology. The quality of these individuals will be approximated by learning the CNN weights using a training set and then evaluating the already learned model in a different dataset with a certain quality metric (e.g., accuracy, F-score, and so on). Then, this quality metric is assigned as the fitness of the individual, which is required by the evolutionary algorithm to evolve the population of individuals.

In our paper, we have used two different evolutionary computation techniques: genetic algorithms and grammatical evolution. The former was chosen because it is a well-known technique that has extensively been used to solve many different optimization problems, and a Gray binary encoding was designed observing some of the most relevant aspects of CNN topologies. The latter was chosen since the definition of a grammar would enable us to further improve the individual encoding by allowing more flexibility and reducing redundancy. We have simplified the fitness computation by approximating the quality function with a reduced sample of the training data and using few training epochs, and we have introduced a niching scheme to preserve the diversity of the population.

For evaluating our proposal, we have used a well-known domain: handwritten digit recognition. This domain has been the choice for the application of CNNs for many years due to their ability to automatically extract local features from multidimensional data, providing state-of-the-art results. In particular, we have used the MNIST dataset, which has been used before in many papers and thus constitute a good baseline for comparison with our approach.

After evolving the topologies and training them with the entire training set, we have attained a test error rate of 0.37% without any type of data augmentation or preprocessing, which is very competitive and would place our proposal among the top results in the state-of-the-art. This result was obtained using grammatical evolution, which behaved slightly better than the genetic algorithm, a consequence that can be attributed to the more flexible and less redundant encoding. When the misclassified samples are observed, it can be concluded that in most cases they involve handwritten digits that would be hard to determine by humans. It is remarkable that such a good result was obtained automatically without spending manual effort on designing the CNN.

In conclusion, this paper shows that neuroevolution is an interesting field to explore the automatic design of CNN topologies, which can compete with handcrafted models requiring a fraction of the time and no manual intervention.

Nevertheless, there are some aspects in which this work could be continued to further study this unexplored research field. For example, to reduce the search space, we have used a discrete encoding that remains fixed along the entire evolutionary process. It could be interesting, however, to perform a fine-tuning of the resulting topologies by changing the encoding in a way that the resolution is increased in those areas of the search space where good solutions are found. As an example, we could include into the encoding an adaptive function that is able to combine different activation functions in a data-driven way as proposed by Qian et al. [70], with the possibility of evolving the combination coefficients of such functions; or optimizing stereo matching for computing vision problems as done by Yang et al. [71]. Another interesting idea worth exploring is the implementation of unified discrete state transition algorithm (DST). This algorithm, proposed by Li et al. [72], will reduce memory and computation bottleneck during DNNs training stage improving the overall performance of the neuroevolutionary process. In the same way, and following one of the latest proposals, we could include distance metric learning such as the one proposed in the Deep-MDML method [73].

Moreover, it could be interesting to explore the construction of CNN committees (or ensembles) from the models found in this work. Committees involve a set of models that “work” together to provide an aggregated prediction. Some papers have explored the performance of CNN committees in the past; however, an approach to evolved committees would be novel.

Finally, it is worth exploring the performance of the proposed neuroevolutionary solution in different domains. When domains are similar, one option is to see if there is any transfer learning that could be studied to reuse evolved topologies in similar yet different domains. In other cases where domains are completely different, for example, in signal classification, then the evolutionary system could be restarted from scratch, designing the topology from the beginning.

Acknowledgment

This research is partially supported by the Spanish Ministry of Education, Culture and Sport under FPU fellowship with identifier FPU13/03917.

References

[1] X. Yao, Y. Liu, A new evolutionary system for evolving artificial neural networks, IEEE Trans. Neural Netw. 8 (3) (1997) 694–713.
[2] K.O. Stanley, R. Miikkulainen, Evolving neural networks through augmenting topologies, Evol. Comput. 10 (2) (2002) 99–127.
[3] Y. Kassahun, G. Sommer, Efficient reinforcement learning through evolutionary acquisition of neural topologies, in: Proceedings of the 13th European Symposium on Artificial Neural Networks, 2005, pp. 259–266.
[4] J. Koutník, J. Schmidhuber, F. Gomez, Evolving deep unsupervised convolutional networks for vision-based reinforcement learning, in: Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation, 2014, pp. 541–548.
[5] P. Verbancsics, J. Harguess, Image classification using generative neuroevolution for deep learning, in: Proceedings of the 2015 IEEE Winter Conference on Applications of Computer Vision, 2015, pp. 488–493.
[6] K.O. Stanley, D.B. D'Ambrosio, J. Gauci, A hypercube-based encoding for evolving large-scale neural networks, Artif. Life 15 (2) (2009) 185–212.
[7] S.R. Young, D.C. Rose, T.P. Karnowsky, S.-H. Lim, 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, 2015.
[8] I. Loshchilov, F. Hutter, CMA-ES for hyperparameter optimization of deep neural networks, (2016) arXiv:abs/1604.07269.
[9] C. Fernando, D. Banarse, M. Reynolds, F. Besse, D. Pfau, M. Jaderberg, M. Lanctot, D. Wierstra, Convolution by evolution: differentiable pattern producing networks, in: Proceedings of the 2016 Genetic and Evolutionary Computation Conference, 2016, pp. 109–116.
[10] N.N.R.R. B. Baker O. Gupta, Designing neural network architectures using reinforcement learning, in: Proceedings of the 5th International Conference on Learning Representations, 2017.
[11] B. Zoph, Q.V. Le, Neural architecture search with reinforcement learning, arXiv abs/1611.01578 (2017).
[12] J. Yu, B. Zhang, Z. Kuang, D. Lin, J. Fan, iPrivacy: image privacy protection by identifying sensitive objects via deep multi-task learning, IEEE Trans. Inf. Forensics Secur. 12 (5) (2017) 1005–1016.
[13] L. Xie, A. Yuille, Genetic CNN, arXiv abs/1703.01513 (2017).
[14] R. Miikkulainen, J. Liang, E. Meyerson, A. Rawal, D. Fink, O. Francon, B. Raju, H. Shahrzad, A. Navruzyan, N. Duffy, B. Hodjat, Evolving deep neural networks, arXiv abs/1703.00548 (2017).
[15] T. Desell, Large scale evolution of convolutional neural networks using volunteer computing, arXiv abs/1703.05422 (2017).
[16] J. Davison, DEvol: automated deep neural network design via genetic programming, 2017, https://www.github.com/joeddav/devol; last visited on 2017-07-01.
[17] M. Suganuma, S. Shirakawa, T. Nagao, A genetic programming approach to designing convolutional neural network architectures, arXiv abs/1704.00764 (2017).
[18] A. Krizhevsky, I. Sutskever, G.E. Hinton, Imagenet classification with deep convolutional neural networks, in: Proceedings of Advances in Neural Information Processing Systems 25, 2012, pp. 1097–1105.
[19] Y.B. Ian Goodfellow, A. Courville, Deep Learning, MIT Press, 2017.
[20] Y. LeCun, L. Bottou, Y. Bengio, P. Haffner, Gradient-based learning applied to document recognition, Proc. IEEE 86 (11) (1998) 2278–2324.
[21] Y. LeCun, Y. Bengio, Convolutional networks for images, speech, and time series, in: The Handbook of Brain Theory and Neural Network, MIT Press, 1998, pp. 255–258.
[22] Y. Guo, Y. Liu, A. Oerlemans, S. Lao, S. Wu, M.S. Lew, Deep learning for visual understanding: a review, Neurocomputing 187 (2016) 27–48.
[23] E. Tsironi, P. Barros, C. Weber, S. Wermter, An analysis of convolutional long short-term memory recurrent neural networks for gesture recognition, Neurocomputing 268 (2017) 76–86.
[24] F.J.O. nez, D. Roggen, Deep convolutional and LSTM recurrent neural networks for multimodal wearable activity recognition, Sensors 16 (1) (2016) 115.
[25] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, A. Rabinovich, Going deeper with convolutions, in: Proceedings of the 2015 IEEE Conference on Computer Vision and Pattern Recognition, 2015, pp. 1–9.
[26] S. Hochreiter, J. Schmidhuber, Long short-term memory, Neural Comput. 9 (8) (1997) 1735–1780.
[27] K. Cho, B. Van Merriënboer, D. Bahdanau, Y. Bengio, On the properties of neural machine translation: encoder-decoder approaches, arXiv abs/1409.1259 (2014).
[28] K. Greff, R.K. Srivastava, J. Koutník, B.R. Steunebrink, J. Schmidhuber, LSTM: a search space odyssey, IEEE Trans. Neural Netw. Learn. Syst. PP (2016).
[29] N. Srivastava, G.E. Hinton, A. Krizhevsky, I. Sutskever, R. Salakhutdinov, Dropout: a simple way to prevent neural networks from overfitting, J. Mach. Learn. Res. 15 (1) (2014) 1929–1958.
[30] J. Duchi, E. Hazan, Y. Singer, Adaptive subgradient methods for online learning and stochastic optimization, J. Mach. Learn. Res. 12 (2011) 2121–2159.
[31] M.D. Zeiler, ADADELTA: an adaptive learning rate method, arXiv abs/1212.5701 (2012).
[32] T. Tieleman, G. Hinton, Neural networks for machine learning, lecture 6.5 – RMSProp, 2012, Coursera, video available in http://www.youtube.com/watch?v=O3sxAc4hxZU.
[33] D. Kingma, J. Ba, Adam: a method for stochastic optimization, arXiv abs/1412.6980 (2014).
[34] J.H. Holland, Adaptation in Natural and Artificial Systems: an Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence, MIT Press, 1975.
[35] C. Ryan, J.J. Collins, M. O'Neill, Grammatical evolution: evolving programs for an arbitrary language, in: Proceedings of the 1st European Workshop on Genetic Programming, in: Lecture Notes in Computer Science, 1391, 1998, pp. 83–96.
[36] S. Zhang, H. Jiang, L. Dai, Hybrid orthogonal projection and estimation (HOPE): a new framework to learn neural networks, J. Mach. Learn. Res. 17 (2016) 1–33.
[37] L. Deng, D. Yu, Deep convex net: a scalable architecture for speech pattern classification, in: Proceedings of the 12th Annual Conference of the International Speech Communication Association, 2011, pp. 2285–2288.
[38] H. Lee, R. Grosse, R. Ranganath, A.Y. Ng, Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations, in: Proceedings of the 26th Annual International Conference on Machine Learning, 2009, pp. 609–616.
[39] J. Yang, K. Yu, T. Huang, Supervised translation-invariant sparse coding, in: Proceedings of the 2010 IEEE Conference on Computer Vision and Pattern Recognition, 2010, pp. 3517–3524.
[40] I.J. Goodfellow, M. Mirza, A. Courville, Y. Bengio, Multi-prediction deep Boltzmann machines, in: Proceedings of Advances in Neural Information Processing Systems 26, 2013, pp. 548–556.
[41] R. Min, D.A. Stanley, Z. Yuan, A. Bonner, Z. Zhang, A deep non-linear feature mapping for large-margin kNN classification, arXiv abs/0906.1814 (2009).

[42] R. Salakhutdinov, G. Hinton, Deep Boltzmann machines, in: Proceedings of the 12th International Conference on Artificial Intelligence and Statistics, in: JMLR Proceedings, 5, 2009, pp. 448–455.
[43] J.R. Chang, Y.S. Chen, Batch-normalized maxout network in network, arXiv abs/1511.02583 (2015).
[44] C.Y. Lee, P.W. Gallagher, Z. Tu, Generalizing pooling functions in convolutional neural networks: mixed, gated, and tree, in: Proceedings of the 19th International Conference on Artificial Intelligence and Statistics, in: JMLR Proceedings, 51, 2015, pp. 464–472.
[45] M.Z. Alom, M. Hasan, C. Yakopcic, T.M. Taha, Inception recurrent convolutional neural network for object recognition, arXiv abs/1704.07709 (2017).
[46] M. Liang, X. Hu, Recurrent convolutional neural network for object recognition, in: Proceedings of the 2015 IEEE Conference on Computer Vision and Pattern Recognition, 2015, pp. 3367–3375.
[47] Z. Liao, G. Carneiro, On the importance of normalisation layers in deep learning with piecewise linear activation units, arXiv abs/1508.00330 (2015).
[48] L. Hertel, E. Barth, T. Käster, T. Martinetz, Deep convolutional neural networks as generic feature extractors, in: Proceedings of the 2015 International Joint Conference on Neural Networks, 2015.
[49] B. Graham, Fractional max-pooling, arXiv abs/1412.6071 (2015).
[50] Z. Liao, G. Carneiro, Competitive multi-scale convolution, arXiv abs/1511.05635 (2015).
[51] M. McFonnell, T. Vladusich, Enhanced image classification with a fast-learning shallow convolutional neural network, in: Proceedings of the 2015 International Joint Conference on Neural Networks, 2015.
[52] D. Mishkin, J. Matas, All you need is a good init, in: Proceedings of the 4th International Conference on Learning Representations, 2016.
[53] C.Y. Lee, S. Xie, P.W. Gallagher, Z. Zhang, Z. Tu, Deeply-supervised nets, in: Proceedings of the 18th International Conference on Artificial Intelligence and Statistics, in: JMLR Proceedings, Vol. 38, 2015, pp. 562–570.
[54] J. Mairal, P. Koniusz, Z. Harchaoui, C. Schmid, Convolutional kernel networks, in: Proceedings of the 27th International Conference on Neural Information Processing Systems, 2014, pp. 2627–2635.
[55] C. Xu, C. Lu, X. Liang, J. Gao, W. Zheng, T. Wang, S. Yan, Multi-loss regularized deep neural network, IEEE Trans. Circuits Systems Video Technol. 26 (12) (2015) 2273–2283.
[56] M.A.R. K. Jarrett K. Kavukcuoglu, Y. LeCun, What is the best multi-stage architecture for object recognition? in: Proceedings of the 2011 International Conference on Computer Vision, 2009, pp. 2146–2153.
[57] R.K. Srivastava, K. Greff, J. Schmidhuber, Training very deep networks, in: Proceedings of the 28th International Conference on Neural Information Processing Systems, 2015, pp. 2377–2385.
[58] M. Lin, Q. Chen, S. Yan, Network in network, in: Proceedings of the 2nd International Conference on Learning Representations, 2014.
[59] M.D. Zeiler, R. Fergus, Stochastic pooling for regularization of deep convolutional neural networks, arXiv abs/1301.3557 (2013).
[60] L. Wan, M. Zeiler, S. Zhang, Y. LeCun, R. Fergus, Regularization of neural networks using DropConnect, Proceedings of the 30th International Conference on Machine Learning, JMLR Proceedings, Vol. 28, 2013 , pp. 3-1058–3-1066..
[61] K. Labusch, E. Barth, T. Matinetz, Simple method for high-performance digit recognition based on sparse coding, IEEE Trans. Neural Netw. 19 (11) (2008) 1985–1989.
[62] M.A. Ranzato, C. Poultney, S. Chopra, Y. LeCun, Efficient learning of sparse representations with an energy-based model, in: Advances in Neural Information Processing Systems 19, NIPS Proceedings, 2006, pp. 1137–1144.
[63] M.A. Ranzato, F.J. Huang, Y.L. Boureau, Y. LeCun, Unsupervised learning of invariant feature hierarchies with applications to object recognition, in: Proceedings of the 2007 IEEE Conference on Computer Vision and Pattern Recognition, 2007.
[64] A. Calderón, S. Roa-Valle, J. Victorino, Handwritten digit recognition using convolutional neural networks and Gabor filters, in: Proceedings of the 2003 International Conference on Computational Intelligence, 2003.
[65] Q.V. Le, J. Ngiam, A. Coates, B. Prochnow, A.Y. Ng, On optimization methods for deep learning, in: Proceedings of the 28th International Conference on Machine Learning, 2011.
[66] Z. Yang, M. Moczulski, M. Denil, N. de Freitas, A. Smola, L. Song, Z. Wang, Deep fried convnets, in: Proceedings of the 2015 IEEE International Conference on Computer Vision, 2015.
[67] F. Lauer, C.Y. Suen, G. Bloch, A trainable feature extractor for handwritten digit recognition, Pattern Recogn. 40 (6) (2007) 1816–1824.
[68] M. McFonnell, M.D. Tissera, T. Vladusich, A. van Schaik, J. Tapson, Fast, simple and accurate handwritten digit classification by training shallow neural network classifiers with the 'extreme learning machine' algorithm, PLoS ONE 10 (8) (2015).
[69] E. Real, S. Moore, A. Selle, S. Saxena, Y. Leon-Suematsu, J. Tan, Q.V. Le, A. Kurakin, Large-scale evolution of image classifiers, arXiv abs/1703.01041 (2017).
[70] S. Qian, H. Liu, C. Liu, S. Wu, H. San-Wong, Adaptive activation functions in convolutional neural networks, Neurocomputing 272 (2018) 204–212.
[71] M. Yang, Y. Liu, Z. You, The Euclidean embedding learning based on convolutional neural network for stereo matching, Neurocomputing 267 (2017) 195–200.
[72] G. Li, L. Deng, L. Tian, H. Cui, W. Han, J. Pei, L. Shi, Training deep neural networks with discrete state transition, Neurocomputing 272 (2018) 154–162.
[73] J. Yu, X. Yang, F. Gao, D. Tao, Deep multimodal distance metric learning using click constraints for image ranking, IEEE Trans. Cybern. 47 (12) (2017) 4014–4024.

Alejandro Baldominos is Computer Scientist and Engineer since 2012 from Carlos III University of Madrid, and got his Master degree in 2013 from the same university. He is currently working as a researcher in the Evolutionary Computation, Neural Networks and Artificial Intelligence research group (EVANNAI) of the Computer Science Department at Universidad Carlos III de Madrid, where he is currently working in his Ph.D. thesis with a studentship granted by the Spanish Ministry of Education, Culture and Sport. He also works as Professor of Artificial Intelligence, Evolutionary Computation and Big Data Engineering.

Yago Saez received the degree in computer engineering in 1999. He got his Ph.D. in Computer Science (Software Engineering) from the Polytechnic University of Madrid, Spain, in 2005. Since 2007 till 2015 he was vice-head of the Computer Science Department from the Carlos III University of Madrid, where he got a tenure and is nowadays associate professor. He belongs to the Evolutionary Computation, Neural Networks and Artificial Intelligence research group (EVANNAI) and member of the IEEE Computational Finance and Economics Technical committee.

Pedro Isasi is Graduate and Doctor in Computer science by the Polytechnic University of Madrid since 1994. Currently, he is University professor and head of the Evolutionary Computation and Neural Networks Laboratory in the Carlos III University of Madrid. Dr. Isasi has been Chair of the Computational Finance and Economics Technical Committee (CFETC) of the IEEE Computational Intelligence Society (CIS), Head of the Computer Science Department and Vice-chancellor in the Carlos III University among others. His research is centered in the field of the artificial intelligence, focusing on problems of Classification, Optimization and Machine Learning, fundamentally in Evolutionary Systems, Metaheuristics and artificial neural networks.