Journal of Parallel and Distributing Computing
117 (2018): 180-191

DOI: https://doi.org/10.1016/j.jpdc.2017.09.006

EvoDeep: a new Evolutionary approach for automatic Deep Neural Networks parametrisation

Alejandro Martín[a], Raúl Lara-Cabrera[a], Félix Fuentes-Hurtado[b], Valery Naranjo[b], David Camacho[a,*]

[a] *Computer Science Department, Universidad Autónoma de Madrid, Spain*
[b] *Instituto de Investigación e Innovación en Bioingeniería, Universitat Politècnica de València, Spain*

Abstract

Deep Neural Networks (DNN) have become a powerful, and extremely popular mechanism, which has been widely used to solve problems of varied complexity, due to their ability to make models fitted to non-linear complex problems. Despite its well-known benefits, DNNs are complex learning models whose parametrization and architecture are made usually by hand. This paper proposes a new Evolutionary Algorithm, named *EvoDeep*, devoted to evolve the parameters and the architecture of a DNN in order to maximize its classification accuracy, as well as maintaining a valid sequence of layers. This model is tested against a widely used dataset of handwritten digits images. The experiments performed using this dataset show that the Evolutionary Algorithm is able to select the parameters and the DNN architecture appropriately, achieving a 98.93% accuracy in the best run.

Keywords: Deep Learning, Evolutionary Algorithms, Finite-State Machines, Automated Parametrisation

1. Introduction

In many real life applications experts face problems that are complex enough to be solved with traditional hand-coded algorithms. Furthermore, these problems may change over time, which forces the latter to continuously adapt themselves. That is why learning algorithms are being increasingly adopted in various applications. Among other learning algorithms, such as logistic regression or decision trees, Deep Learning has acquired importance in recent years due to its great success when dealing with pattern recognition problems. Deep Learning methods can be seen as an extension of the classic Artificial Neural Networks

*Corresponding author
Email addresses: alejandro.martin@uam.es (Alejandro Martín), raul.lara@uam.es (Raúl Lara-Cabrera), ffuentes@upv.es (Félix Fuentes-Hurtado), vnaranjo@dcom.upv.es (Valery Naranjo), david.camacho@uam.es (David Camacho)

that take advantage of the current reduced cost of computation, as well as the new paradigms of parallel and distributed computation.

The features of Deep Neural Networks (DNNs) make them a powerful and appealing tool to solve problems of different nature and where there is a common element: complexity. Although there exist many techniques and algorithms designed to solve problems composed of thousand of features, or where the relation between inputs and outputs relies on large non-linear equation systems, these methods have been proved not to be adequate to solve certain kind of problems. These problems are mainly related to recognition processes, and they are usually focused on tasks that are solved intuitively by humans [1].

DNNs are currently being used in a wide range of problems of different nature, mainly due to their capabilities of building strong prediction and classification systems, and its adapting capacity to non-linear spaces. On the other hand, the training process is more difficult when it is compared against other methods as Decision Trees or Support Vector Machines. Another difference with respect to other Machine Learning methods is the fact that the models generated are not self-explanatory, thus disallowing to comprehend the knowledge collected.

As a trade-off to the high precision achieved by Deep Neural Networks, they should be considered as complex learning models: they are made up of complex architectures built by a large number of layers and neurons, each of them being of a different type or having distinct objectives, with many other parameters affecting the architecture as well, such as activation functions or the optimiser applied. Developers usually fix these parameters by hand, following a trial-and-error scheme. There are several approaches in order to speed up and improve this stage in classical Neural Networks training, many of them relying on Evolutionary Computation. However, new features and specifications, as well as an increased complexity, require from new specialised methods related to Deep Neural Networks.

This paper presents an evolutionary approach for optimising both, the architecture and the parameters of Deep Neural Networks in an automated way. Thus, each individual represents a possible network architecture, including global parameters such as the optimiser, or the maximum number of iterations, and a structure where a variable number of different type of layers is defined (that includes several details, as the number of outputs or the activation function for each layer). The main objective of this new evolutionary-based approach is to maximise the accuracy at classifying handwritten digits samples from the MNIST[1] dataset.

Previous literature has also aimed at defining the topology of the network, at defining an initial set of weights or at fixing training parameters. However, and to the best of our knowledge, most of these efforts have been made for classical Artificial Neural Networks. In this research we focus on Evolutionary Deep Learning, where complex layers structures, specific layers, constrains in

[1] `http://yann.lecun.com/exdb/mnist/`

the layers order and new parameters, must be taken into account to define a DNN architecture before starting the training process.

The proposed Evolutionary Algorithm aims to optimize almost all the necessary parameters to train a Deep Neural Network. Its design pursues the extensibility of the model to include new features in the future, avoiding parameter value restrictions, thus allowing individuals to take any possible value within the previously defined range, and without the necessity of fixing any training parameter. At the same time, it is also modelled a Finite-State Machine that determines the possible transitions between different kind of layers, allowing to generate valid sequences of layers, where the output of one layer fits the input requirements of the next one. The model described is tested on a specific problem related to the image recognition field, which consists on detecting the handwritten digit present on a particular image.

The rest of this paper has been structured as follows: the next section (Background) describes the related Work and some recent contributions related to this paper; Section 4 describes the Evolutionary Algorithm, *EvoDeep*, designed to search for the best parameters of a Deep Neural Network; Section 5 provides a detailed description related to the experimental setup, which is later used and analysed in Section 6 (Experimentation); Finally, Section 7 provides some conclusions and future lines of work.

2. Background

Artificial Neural Networks, as a computational model, are vaguely inspired on biological neurons that began to spread in the computer science literature in 1943 when McCulloch and Pitts presented their initial ideas [2]. Since then, high and lows have appeared in the popularity of Neural Networks application [3], mainly due to the report of some deficiencies and limitations of the initial model [4]. In the nineties, Deep Neural Networks emerged as an improvement of the classic model, adding a large number of neurons and layers to the network architecture. In the last years, there has been a rising interest in such improved models [5], primarily due to the development of new programming frameworks focused on Deep Learning, such as TensorFlow [6] or Theano [7].

Deep Neural Networks provide some advantages over other Machine Learning methods, such as classical Artificial Neural Networks, in many fields. For instance, Computer Vision and Image Recognition use currenlty Deep Neural Networks [8, 9], where images pass through several layers in order to extract, or manage, a large number of features. Another field of research where this technique is being increasingly used is at Malware detection, as shown by recent publications such as DroidDetector [10], Droid-Sec [11] and DroidDelver [12], which use Deep Belief Networks to detect Android malicious applications.

Moreover, there are evidences on the use of Deep Neural Networks for audio recognition, as the work by Hinton et al. [13] presents, or the paper by Lee et al. [14]. Deep Neural Networks have been used in other research fields such as time series forecasting [15, 16] and video recognition [17], just to name a few.

Selecting the adequate architecture, parameters and weights of a Neural Network has become a recurring problem that, in turn, is usually solved by using a trial-and-error methodology. The emergence of deep learning techniques in the form of Deep Neural Networks makes it necessary to develop new methods and tools capable of successfully searching combinations of parameters and architectures that lead to a good performance of the network. In this sense, the application of Evolutionary Computation to this matter was studied in detail by Xin Yao [18]. Evolutionary Computation might be applied to many stages of a Neural Network training: weights, learning rules and architecture. Our proposal tackles the layer architecture of the network which is, in fact, the less automated stage of the training process. This issue introduces two conflicting designing criteria: a network with low connectivity and a low number of nodes, which could generate a network architecture with a deficient accuracy to solve a particular problem. But, on the other hand, an excessive complex network architecture (with a high connectivity, and very large number of nodes) can lead to include noise in the model, loosing generalisation ability.

There have been several approaches to tackle the optimization problem of tuning the parameters and architecture of classic Neural Networks [19]. It is possible to use an Evolutionary Algorithm in which the individuals include the number of neurons, however, the number of layers is limited to a specific value [20]. Other approaches include the architecture definition, which may be combined with the adjustment of the weights as in [21]. A hyper-heuristic approach based on Evolutionary Algorithms has also been proposed to adjust the number of layers, the polynomial type and the number of nodes defined in each layer of the network [22]. This kind of algorithms have also been used to improve the efficiency [23] or including connection weights in the evolutionary search [24].

Another research topic focuses on the codification of the individual and how it is possible to speed up the convergence of the algorithm using a grammar graph generator as an individual [25], or adjusting the level of granularity [26]. Another approach is focused in NeuroEvolution of Augmenting Topology (NEAT) methods, consisting of evolving Artificial Neural Networks by employing an encoding that helps to represent and discover large scale ANNs, such as HyperNEAT [27] or SUNA [28]. Evolutionary Algorithms have also been successfully employed in different optimisation problems, such as adjusting weights of a boosting process [29], or to build a classifier for the android malware detection domain [30] among other works focused on this topic [31, 32, 33].

The reader may find in the literature some research specifically related to the application of Evolutionary Algorithms to optimise Neural Networks architectures and parameters, due to the power of these kind of algorithms to leverage the possibilities of Neural Networks [34]. For instance, in [35] an Evolutionary Algorithm was used to optimise the parameters and weights of the Neural Network in a two-step process. In [36], the authors applied the Taguchi method between the mutation and crossover operators and included the initial weights definition.

To our best knowledge, there is little work on studying the use of Evolution-

ary Algorithms to improve Deep Neural Networks beyond using them to evolve the weights of the network [37]. The main contribution of this paper is to design and test a new Evolutionary Algorithm capable of performing an integral optimisation of the parameters and architecture of Deep Neural Networks.

3. Deep Neural Networks

Deep learning leverage computational models composed of a sequence of processing layers to learn representations of data with multiple levels of abstraction [8]. The atomic element of any standard Neural Network (NN) is a simple mathematical processor called neuron. These neurons are organized in form of interconnected groups called layers, and a sequence of these layers composes a Neural Network. A DNN is similar to a standard Neural Network and is able to learn a set of features that will be later used in order to approximate the objective function.

DNNs are named after networks, because they are typically represented by composing together many functions. The model is associated with a directed acyclic graph describing how these functions are composed together.

For instance, a three layer network can be thought of as three functions $f^{(1)}, f^{(2)}, f^{(3)}$ connected in a chain to form $f(\boldsymbol{x}) = f^{(3)}(f^{(2)}(f^{(1)}(\boldsymbol{x})))$. The overall length of the chain gives the **depth** of the model. In this case, $f^{(1)}$ is called the **input layer** of the network, $f^{(2)}$ is called the **hidden layer**, and $f^{(3)}$ is called the **output layer**. Each hidden layer of the network is typically a vector of values, and its dimensionality determines the **width** of the model. In short, a DNN consists of many layers chained in order to approximate some complex objective function f^*, where layers compose a mapping $\boldsymbol{y} = f(\boldsymbol{x}; \theta)$ and learn the value of the parameters θ that result in the best function approximation [1].

3.1. Layers

There are multiple types of layers that can be used to design a Neural Network. Keras [38] implements a wrapper allowing to use TensorFlow layers in a very simple and efficient way [6]. Layers considered in this work (*Dense*, *Dropout*, *Reshape*, *Flatten*, *Convolution2D*, and *MaxPooling2D*) are described in detail in the next subsections.

3.1.1. Dense

Densely connected layers are identical to the layers in a standard multilayer Neural Network. In this kind of layer, every neuron is connected to every neuron in the previous layer. A "neuron" is a computational unit that takes as input $[x_1, x_2, x_3, \ldots, x_n, b]$, where n is the number of inputs of the neuron, b is the bias, and outputs $h_{W,b} = f(W^T x) = f(\sum_{i=1}^{n} W_i x_i + b)$, where $f : \mathbb{R} \mapsto \mathbb{R}$ is called the activation function.

Let l be the number of layers in the network and L_l the set of layers present in a particular network architecture, where $l : 1, 2, 3, \ldots, n_l$. Then, the whole network architecture is defined by $(W, b) = (W^{(1)}, b^{(1)}, W^{(2)}, b^{(2)}, \ldots, W^{(n-1)}, b^{(n-1)})$,

where $W_{ij}^{(l)}$ denotes the parameter or weight associated with the connection between unit j in layer l, and unit i in layer $l+1$ and $b^{(i)}$ is the bias of layer l_i.

The following parameters of Keras functional implementation of Dense layer are used in this work:

- **init**: the initialization mode of the weights of the layer $W^{(l)}$,
- **activation**: the name of the function used as activation function,
- **number of outputs**: the number of outputs of the current layer l.

The possible weight initialization methods are: uniform, LeCun uniform [39], normal, zero, Glorot normal [40], Glorot uniform, He normal [41] and He uniform. Finally, the possible activation functions are: softmax, softplus, softsign, relu, tanh, sigmoid, hard sigmoid and linear.

3.1.2. Dropout

This kind of layer applies dropout to the input. Dropout consists of randomly selecting a fraction rate of input units and setting them to 0 at each update during training time, which helps to prevent overfitting. The units that are kept are scaled by $1/(1-rate)$, so that their sum is unchanged at training and inference time.

The functional Keras implementation of this layer takes as input argument the fraction rate of input units, that will be set to 0.

3.1.3. Reshape

This layer is applied to change the shape of the data. In concrete, it takes as input 1D data and outputs it in the desired shape (1D-data $\mapsto$ ND-data). This is of utmost importance when dealing with layers as Dense and Convolution2D in the same Neural Network, as Dense takes 1D data as input and Convolution2D needs at least 2D data.

3.1.4. Flatten

This layer is complementary to Reshape. It takes multidimensional data (ND data) and *flattens* it to one dimension (ND-data $\mapsto$ 1D-data).

3.1.5. Convolution2D

Convolution is a mathematical operation that is applied on two functions with a real-valued arguments. The result is the integral of the point-wise multiplication of the two functions, as a function of the amount that one of the original functions is translated. It is typically denoted with an asterisk:

$$s(t) = (x * w)(t) = \int x(a)w(t-a)da \tag{1}$$

When using convolution in a Neural Network layer, w needs to be a valid probability density function so that the output is a weighted average. In addition, w needs to be 0 for all negative arguments, otherwise it would be looking into the future.

Convolution layer applies this operator to every element of the input data. In the case of 2D convolution, which is the one that Convolution2D layer applies, elements are pixels of the image. In this case, it is necessary to choose the number of times that the operation will be applied to each pixel ($nb\_filters$) and the size of the neighbourhood (kernel size, specified by nb_row rows and nb_col columns).

The functional Keras implementation of Convolution2D takes as input the following parameters:

- **init**: the initialization mode of the weights of the layer,
- **activation**: the name of the function used as activation function,
- **nb_filter**: the number of filters to apply to each element of the input data,
- **nb_row**: the number of rows of the kernel used by the convolution,
- **nb_col**: the number of columns of the kernel used by the convolution,
- **border_mode**: to stablish the behaviour of the operator at the borders of the image,
- **bias**: wether to include a bias (i.e. make the layer *affine* rather than linear).

The *init* and *activation* parameters are the same as were described in Subsection 3.1.1, *border_mode* establishes how the filter behaves near the borders of the image, and takes two values: "valid" and "same". Whereas, "valid" convolution is applied only when the input and the filter fully overlap, yielding a smaller output than the input, with "same" the area outside the filter when it is placed near an image border is padded with zeros, yielding an output with the same size as the input.

3.1.6. MaxPooling2D

Pooling layer is used to reduce the dimensionality of data. To achieve it, a window is specified and an operator is applied to all elements within this window, which gives a number as output (Figure 1). There exist several operators which can be applied to perform the pooling: max, min, *average*, *median*, ...

In Keras implementation, only *max* and *average* pooling layers are available, from which only the former have been used in this work. The parameters used by this layer are:

- **pool size**: to specify the size of the pooling,
- **strides**: to specify the step taken between poolings,

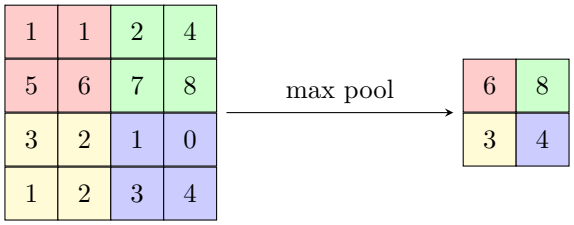


Figure 1: Max pooling layer detailed. In this case, $poolsize = [2, 2]$ and $strides = [2, 2]$, which results in the 2-by-2 matrix shown on the right.

- **border mode**: to stablish the behaviour of the operator at the borders of the image.

4. EvoDeep: Deep Neural Networks parametrisation using Evolutionary Algorithms

The strengths provided by Deep Learning models are counterbalanced by their representative complexity and variable architecture. While training a classical Machine Learning algorithm, such as the well known Random Forest, is a straight task, where typically only the number of trees has to be adjusted, Deep Learning requires to define an architecture which is usually problem dependent. For instance, Deep Learning applied to image recognition, or classification, involves defining several layers in a specified order in charge of applying different transformations to the data.

This architecture is made by a set of layers, which are distributed with the goal of laying out the non-linear relationships that solve a specific problem. Every layer has an arbitrary number of neurons and outputs, as well as distinct initialization methods, and activation functions. These parameters must be fixed before the training process of the Neural Network, hence forming a large search space with the optimal combination of settings being unknown and dependent to the problem to solve.

The design of a multilayer architecture is limited by different restrictions. For example, the input shape of a particular layer must fit the output of the previous one. This can take place when a layer expects a vector as input (i.e. a typical fully connected layer), restricting the layers which may precede. In this case, a Reshape layer, which always delivers an output of at least 2 dimensions, could not be on the left of a fully connected layer. Another restriction to take into account lies in the parametrisation of each new layer added to the model, which must also fulfil certain properties. For instance, a Reshape layer expects a tuple defining the dimensions of the output matrix, which must match the size of the input vector.

Neural Networks parameters and topology, shape a large search space where many possible configurations are defined. Since it is expected to maximise the accuracy of the network in performing a particular task, the selection of the

proper configuration can be seen as an optimisation process. In this paper, an Evolutionary Algorithm has been used to lead a meta-heuristic search to obtain a configuration that maximises the accuracy in a classification task.

Choosing different combinations and configurations of the aforementioned parameters and layer architectures, will lead to a different performance when solving, for instance, a classification problem which, in turn, is the most widely proposed kind of problem when it comes to DNN. The selection of a good set of parameters, and the layer architecture, should be seen as an optimisation problem with improving the classifying accuracy as the main goal.

Due to the huge search space that needs to be explored, we decided to use Evolutionary Algorithms [42], which have shown good performances when dealing with optimisation problems whose search space is very large. In this work, an Evolutionary Algorithm, named **EvoDeep**, has been desigend to perform a meta-heuristic search over the parameters space, looking for the best possible combination that leads to a good accuracy in a classification problem, while reducing the time of a process that is commonly done by hand.

Moreover, the use of an Evolutionary Algorithm might find a combination of parameters and/or a structure of the layers that an expert would not have used due to her biased knowledge of the problem. For instance, an expert would not put two kinds of layers one after another because this combination is not commonly found in the literature. However, it is possible that the Evolutionary Algorithm finds a layer sequence that contains this combination and has a good classifying performance.

In our approach, EvoDeep has been designed and implemented as an Evolutionary Algorithm, where each individual represents a specific network architecture with its respective parameters. The fitness value for each individual in EvoDeep, is calculated as the accuracy of the network when a classification problem is solved. The user only needs to define which parameters are going to make the search space, as well as the range of values they can adopt. We used numerical sequences, with a given step between values, in order to reduce the size of the search space. Regarding categorical parameters, such as the activation function, a list of possible values has to be provided. All the parameters act as inputs to the individual's initialization function, which generates a population of λ individuals, as well as restrictions to the mutation operator in order to perform valid variations.

EvoDeep algorithm has been designed following a $(\mu + \lambda)$ scheme, where λ represents the number of individuals to generate in each generation, and μ indicates the number of individuals selected to generate the next population. Every generation, the recombination and variation (mutation) operators are systematically applied to the individuals with a probability of p_R and p_V, respectively, reproducing them when needed to generate λ individuals which are then evaluated. Once every individual has its fitness updated, a roulette selection method is performed to select individuals proportionally to their fitness values among the new population, plus the μ individuals selected from the previous generation, thus allowing to keep individuals with a good performance while maintaining the diversity of the population, so the algorithm does not stuck into a local

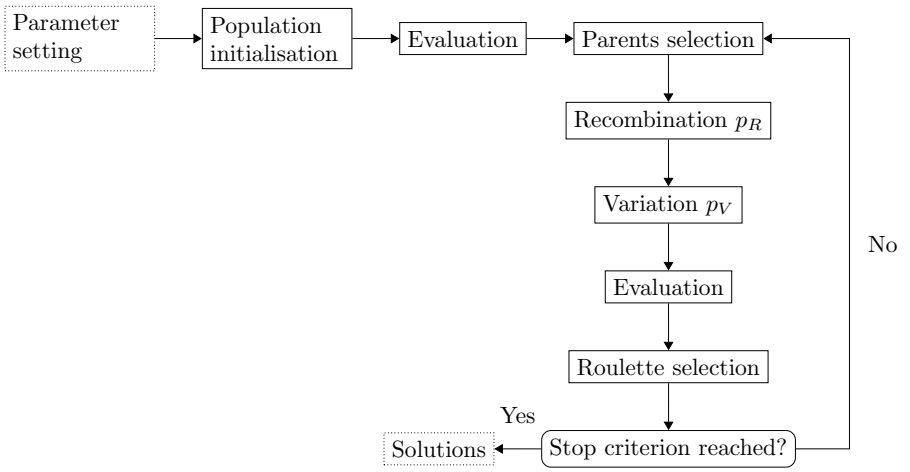


Figure 2: Breeding pipeline in EvoDeep algorithm.

maximum.

4.1. Generation of valid sequences of layers

In order to include all these restrictions and with the aim of building valid layers configurations, we have modelled all the possible transitions through a Finite-State Machine (FSM) in charge of defining possible paths, which are later used to build the individuals. This FSM, as shown in Fig. 3, takes a dictionary where each symbol is related to a particular layer type. When generating individuals, the FSM is used to build all possible paths given a minimum and maximum length, from which a particular one is randomly chosen. When crossing and mutating individuals, the FSM is also employed in order to avoid generating invalid structures or to include new consistent layers in the mutation operation.

The FSM is initially used to generate the initial population, where all the individuals are created with a fixed low number of layers, aiming to generate larger individuals in the course of the evolution if necessary. To this end, an initial and a final state are marked according to the input and output specifications. In Figure 3, an initial and final state have been flagged expecting as input a vector where each position corresponds to a feature, and a vector of length n as output where n is the number of labels to classify (given that a One-hot encoding is used to codify the labels). The initial state 0 defines an empty model where no layer has been added yet. On this basis, only a Dense or a Dropout layer can be added. This is due to the restrictions on the data type layers are able to accept as inputs. In this case, the only layers capable of processing the raw input data (i.e. image as a vector) are Dense, Dropout and Reshape. The final state 1 can only be reached if the last layer of the model is of type Dense.

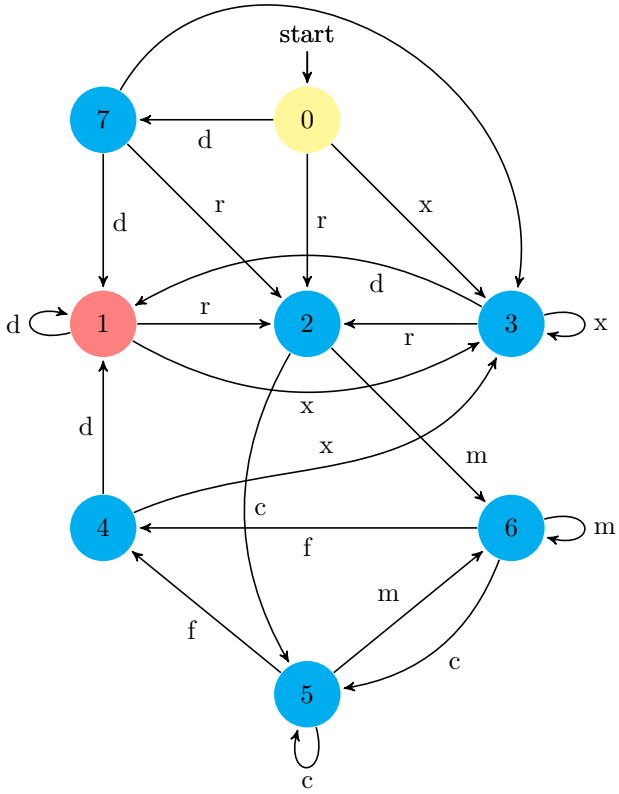


Figure 3: State machine that determines the possible transitions between layers. The alphabet defines each possible layer type: Dense (d), Dropout (x), Reshape (r), Convolutional (c), Flatten (f) and MaxPooling (m)

As it can be seen in Figure 3, there are two states which can be reached by adding a Dense layer, 7 and 1. State 7 forces any path defined by the FSM to contain at least two layers, a restriction that leads to two possible minimum paths: Dense or Dropout followed by a Dense layer.

4.2. Individual encoding

Each individual has been encoded in EvoDeep to allow representing all the parameters needed to train a Deep Neural Network (see Figure 4). Each genome is made of a set of global parameters which defines the general behaviour of the network, and a sequence of layers with an arbitrary number of them. Each one should be of a different type and includes a set of parameters according to its type. Table 2 shows the parameters in the evolutionary search, the scope where

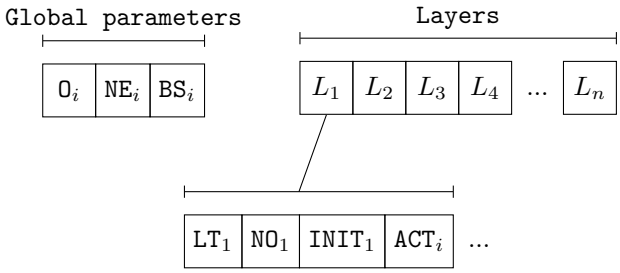


Figure 4: Individual encoding in EvoDeep: any individual is defined by a set of several global parameters and a sequence of layers, each of them having a certain number of layer parameters.

they are defined (as global or layer parameters) and their range of values. Within the set of global parameters, O_i defines the optimizer where six different types have been tested. NE_i indicates the maximum number of epochs or iterations of the training process. However, a dynamic stopping criterion has been applied, fixing a maximum number of iterations while the maximum accuracy obtained is not exceeded. BS_i states the amount of samples that the Neural Network receives at a time, updating its weights according to that set of samples.

Regarding layers, each one belongs to a specific type of layer LT_i, which is linked to different parameters. For instance, a Dense layer in Keras implementation contains a fully connected collection of neurons and implies defining parameters such as the number of outputs NO_i, the initialization function $INIT_i$ and the activation function ACT_i. Another interesting kind of layer defined in Keras is the so called Dropout layer, where a fraction P_i of inputs is set to 0, aiming to avoid overfitting.

It was necessary to design and implement specific evolutionary operators due to the encoding of the solutions in the algorithm. Following there is a detailed description of these operators as well as the fitness function that evaluates every individual.

4.3. Evolutionary operators

The variable size of the individual and the restrictions imposed by a Neural Network architecture led us to develop specific mutation and crossover operators which yield valid individuals that were able to train different network models.

4.3.1. Crossover operator

The crossover operator works at two different levels: at the global parameters level and at the layer level. This is due to the requirements of the encoding designed, where each individual can be composed of a different number of layers and where these layers can consist of different parameters depending on their type. The crossover operation is applied if a cross probability is satisfied.

In the case of the global level, as shown in Figure 5, all global parameters are crossed following an uniform crossover, where each pair is swapped independently based on a probability of 50%. In contrast, a cut and splice crossover

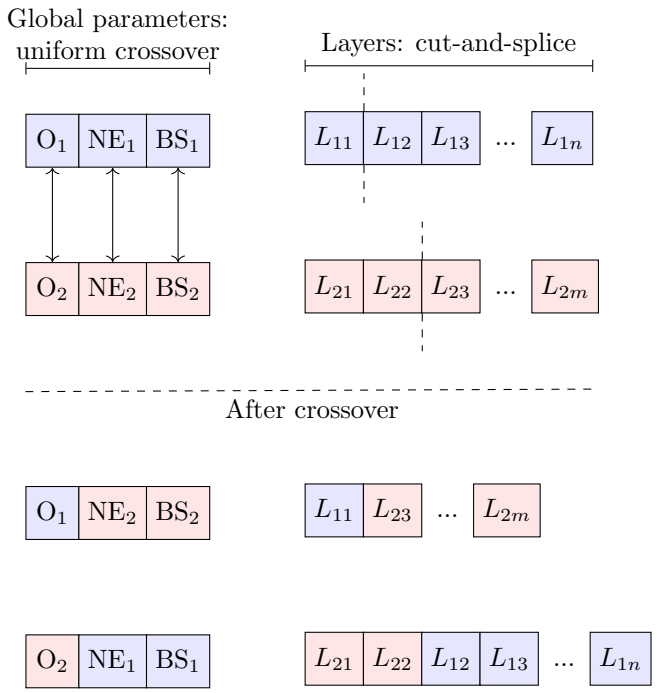


Figure 5: External crossover: global parameters are swapped with a probability of 0.5; layers are crossed using cut-and-splice, that is, one cut point for each individual is randomly selected and then the left part of first individual is concatenated with the right part of the second, and vice-versa.

is applied over the layers section. It allows to build new configurations of different sizes, allowing to decrease or increase the network architecture. In this crossover model, two points p_1 and p_2 are randomly selected while satisfying $1 < p_1 < n$ and $1 < p_2 < m$, where n and m are the number of layers of each individual. These two conditions ensure that the first and last layer are always located in the correct place, since the number of neurons is linked to the number of features and the number outputs respectively for the first and last layer. Moreover, cut points are selected in a way that after swapping parts, the sequence of layers is still valid (see Section 3.1) and the number of layers does not exceed the maximum number of layers set at the beginning. This is achieved taking into account the number of layers left to the maximum number of layers, and restricting the selection of the cut point to be within a safe range according to the aforementioned number. In other words, the operator selects both cut points in a way that the number of layers of the left part for the first individual, plus the number of layers of the right part for the second individual, and vice-versa, do not exceed the maximum number of layers allowed. This can be done because the operator selects first one of those cut points so the second one can be selected fulfilling the aforementioned requirement.

This global crossover method allows to create two new individuals where each global parameter is crossed uniformly and two new layer structures are composed by cutting the parent structures at two random points, provided the probabilities are satisfied. This only move layers between individuals as a whole without entering into their specific layer parameters, though. To build individuals where the internal parameters are also exchanged, a layer level crossover is applied (internal crossover). Starting from the first layer (see Fig. 6), two analogous parameters (p_i and p'_j belonging to two layers L_l and L'_l placed in the same position l of two different individuals) are crossed until the penultimate layer of the shortest individual is reached, while the last layer of each individual is also crossed:

$$l < \min(n, m) \vee l = \max(n, m). \tag{2}$$

This approach aims to cross the maximum number of parameters, even if they come from different layer types, since there is a number of common parameters.

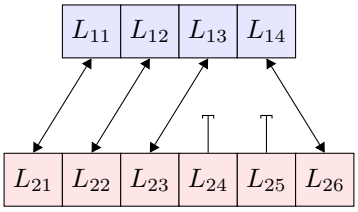


Figure 6: Internal crossover: layer parameters that appear on both layers in the same relative position within the individual might be swapped. Parameters from the last layer of one individual can only be swapped with parameters from the last layer of the other individual. If individuals have a different number of layers, then there are layers not affected by the operator.

4.3.2. Mutation operator

In a same way as the crossover method, the mutation, or variation, operator has been designed with the encoding of the individuals in mind. It also works at the two previously mentioned levels: global parameters and layers. At the first one, every global parameter of an individual is mutated following again a uniform approach, where each parameter is given a new random value with a probability of 50% (if the global mutation probability is met). Each new random value is generated based on the range of values in the case of numerical parameters, or based on the list of values if it is a categorical parameter.

Regarding the layer level mutation, the method randomly selects a point of insertion within the sequence of layers and then inserts a valid sub-sequence of 1, 2 or 3 layers obtained from the FSM defined at the beginning of this section to ensure that the new sequence remains valid. This insertion only takes place if

the final sequence of layers has at most the number of maximum layers defined at the beginning of the algorithm.

The internal parameters of each layer are mutated following the same scheme as global ones, performing an uniform operation parameter by parameter, layer by layer, allocating new values based on the rules defined by the user.

4.4. Fitness function

Finally, the fitness of each individual in EvoDeep represents a measure about how good it is, and in our approach it has been defined by the accuracy achieved after executing a Deep Neural Network whose configuration is determined by the individual phenotype. This value sets an objective to be maximised, as it is pursued to obtain an individual able to reach the maximum accuracy possible. When comparing individuals to detect duplicates, a comparison parameter by parameter is performed, since the fitness is not a reliable measure as it can vary between executions of the Neural Network depending on its weight's initialisation. Finally, although the training process is performed based on the error performance in the training dataset, at the end of its execution the network is evaluated in a validation dataset (a portion of the original data randomly extracted). This allows to calculate the fitness of the individuals based on a set of data different from the one used to evolve the network weights, and allows to provide a more realistic value of the evolutionary process. At the same time, using this validation dataset allows to minimise the possible overfitting effect produced by the DNN operation.

5. Experimental setup

This section describes the Evolutionary Algorithm parametrisation as well as the parameters of the Deep Neural Network composing the search space.

5.1. Keras

We used Keras [38], a high-level library for Artificial Neural Networks with support for both backends Theano and TensorFlow, to execute the Deep Neural Networks defined by the individuals generated by EvoDeep algorithm. Due to the capabilities of the library, it is possible to run the experiments using both Theano and Tensorflow as the execution environment, being the latter the backend selected for our experimentation.

5.2. TensorFlow

TensorFlow [43] is an Application Programming Interface (API) for defining and running Machine Learning algorithms developed by Google and available[2] under the Apache 2.0 license. The system can be used to express a wide variety

[2] `www.tensorflow.org`

Parameter	Description	Value
N_{gen}	Number of generations	20
N_{stop}	Early stop generations	5
μ	Number of individuals selected to generate next population	5
λ	Number of individuals to create in every generation	10
p_R	Recombination probability	0.5
p_V	Mutation probability	0.5
p_L	Probability of adding new layers	0.5

Table 1: Parameters used to run EvoDeep.

of algorithms, such as training and inference algorithms for Deep Neural Network models, and is capable of running them on many kinds of systems such as handheld devices, distributed systems including several machines and GPU cards. TensorFlow has been used broadly for researching and deploying Machine Learning systems across many areas, including computer vision, speech recognition, information retrieval and robotics.

5.3. Evolutionary Algorithm parametrisation

Regarding the parametrisation of the Evolutionary Algorithm, we took into account the time needed to evaluate every individual that, in turn, is pretty high. In order to mitigate this problem, we decided to limit the maximum number of generations to 20, although there is an early stop condition that avoids unnecessary iterations of the algorithm stopping it when the average fitness of the population does not change over the last 5 generations. As defined in Section 4, the algorithm follows a $(\mu + \lambda)$ generational scheme with $\mu = 5$ and $\lambda = 10$. The probability of recombination and mutation has been set as $p_R = 0.5$ and $p_V = 0.5$, respectively, with the probability of adding new layers during the variation stage being 0.5 as well. These parameters were empirically chosen in order to meet a trade-off between the execution time and the accuracy achieved by the algorithm.

5.4. Deep Neural Network parametrisation

In addition to the parameters of the Evolutionary Algorithm, the deep learning system has its own parameters that are, in turn, optimised by the former. Table 2 describes every parameter as well as its range of values or list of categorical values that has been encoded in the algorithm. Global parameters are those that affect the architecture as a whole while those labelled as Layer parameters are different for each layer in the architecture. Some of the latter are common to any kind of layer ($LT_i, INIT_i, ACT_i$) while others are specific to a type of layer ($NO_i, P_i, NF_i, NROW_i, NCOL_i, BIAS_i, BD_i, POOL_i, STR_i$).

Parameter	Definition	Scope	Values
O_i	Optimizer	Global	Adam, SGD, RMSprop, Adagrad, Adamax, Nadam
NE_i	Number of epochs	Global	Min: 2, Max: 20, Step: 2
BS_i	Batch size	Global	Min: 100, Max: 5000, Step: 100
LT_i	Layer type	Layer	Dense, Dropout, Convolution2D, MaxPooling2D, Reshape, Flatten
$INIT_i$	Initialization function	Layer	Uniform, Lecun uniform, Normal, Zero, Glorot normal, Glorot uniform, He normal, He uniform
ACT_i	Activation function	Layer	Relu, Softmax, Softplus, Softsign, Tanh, Sigmoid, Hard sigmoid, Linear
NO_i	Number of outputs of each layer	Layer (Dense)	Min: 10, Max: 500, Step: 20
P_i	Fraction of the input units to drop	Layer (Dropout)	Min: 10%, Max: 80%, Step: 10%
NF_i	Number of filters to apply	Layer (Convolution2D)	Min: 5, Max: 50, Step: 5
$NROW_i, NCOL_i$	Rows and columns of the kernel	Layer (Convolution2D)	Min: 3, Max: 15, Step: 2
$BIAS_i$	Whether to include a bias	Layer (Convolution2D)	True, False
BD_i	Behaviour of the operator at the borders	Layer (Convolution2D, MaxPooling2D)	Valid, Same
$POOL_i$	Size of the pooling	Layer (MaxPooling2D)	Min: 2, Max: 6, Step: 1
STR_i	Step taken between poolings	Layer (MaxPooling2D)	2, 3, 4, 5, 6

Table 2: Parameters involved in the evolutionary search, the scope (Global or Layer) in which they are defined, and range of values evaluated.

5.5. Dataset

In this paper, we have performed several experiments using the well known MNIST dataset[3]. This dataset contains a huge number of handwritten digits, which are centered in a 28x28 pixels pictures. Each pixel is linked to an integer value which ranges from 0 to 255, defining the grey level from white to black. Many works have used this dataset to train and test new classification algorithms.

6. Experimentation

This section aims to analyse the performance of the solution proposed in the previous sections at defining a proper selection of parameters for training a Deep Neural Network. Furthermore, it also provides details related to the most profitable configurations, such as the importance of each particular initialisation function and shows the best individuals found by the algorithm.

6.1. Evolutionary search performance analysis

In first place, different executions of the proposed solution have been run in order to evaluate its ability to maximise the accuracy in a particular problem.

[3] `http://yann.lecun.com/exdb/mnist/`

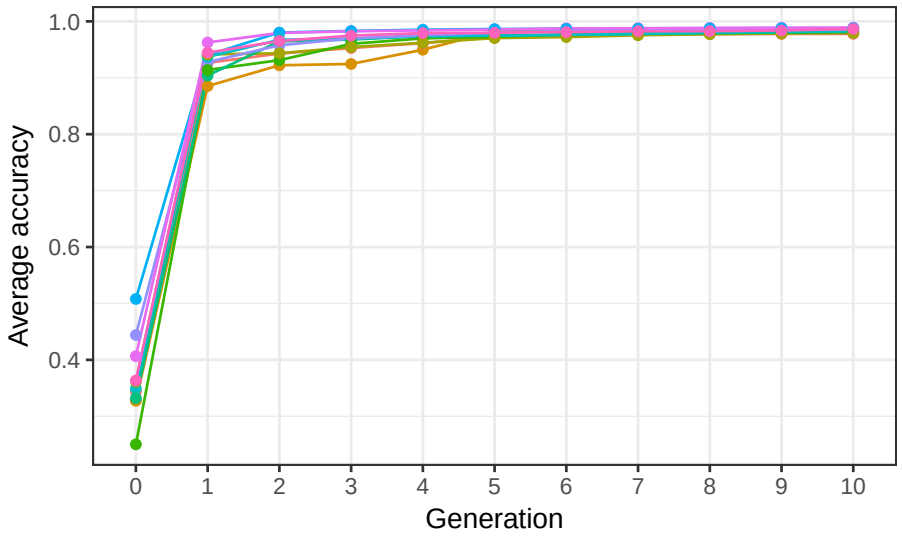


Figure 7: Evolution of the average accuracy achieved by the population for 10 different executions through the evolutionary procedure.

Ten different executions have been performed, delivering a sorted list of individuals by their fitness, measured as the accuracy in the validation dataset. Results are shown in Fig. 7. Each line represents a different execution where each point indicates the average accuracy achieved by the whole population at a particular generation. The evolutionary search produces a fast convergence, requiring just one generation to build promising candidates with 90% of accuracy. Although the maximum number of generations was fixed to 20, in the fifth generation the accuracy is stabilised and few changes are reported in the following ones. This result supports an important goal of this paper: to build accurate models in the shortest possible time.

The results achieved have also been analysed from the perspective of the evolution of the accuracy throughout the different executions. Fig. 8 draws this evolution for the best individual found based on its evaluation with each different dataset. There are two remarkable findings which can be extracted. On the one hand, validation and test results are very close, which is a fact of significant importance, given that the first one is used to lead the evolutionary search, where an improvement in the validation dataset will be linked to an enhanced test accuracy. On the other hand, the accuracy shaped by the test dataset is also near to the results showed by the training dataset. These results can also be seen in Table 3, which shows the summary of the accuracy achieved by the best individual for each execution. The average accuracy in the test dataset was $98.42\% \pm 0.39$, which means just 1.32% of difference. The best value in the test was obtained in an execution with nearly a 99% of accuracy.

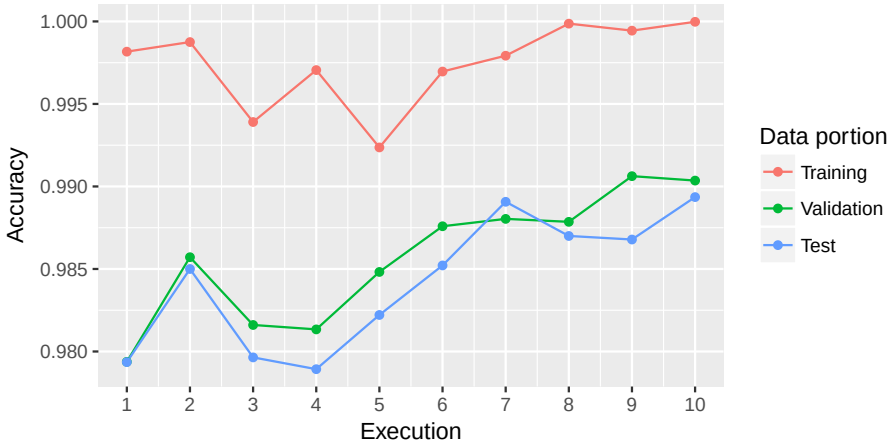


Figure 8: Evolution of the average accuracy achieved by the best individual for each execution in training, validation and test datasets

	Training dataset	**Validation dataset**	**Test dataset**
Minimum	99.23%	97.93%	97.89%
Median	99.80%	98.66%	98.51%
Mean	$99.74\% \pm 0.25$	$98.57\% \pm 0.38$	$98.42\% \pm 0.39$
Maximum	99.99%	99.06%	98.93%

Table 3: Mean, median, minimum and maximum values of the best individual for each execution in the training, validation and test datasets.

6.2. Parameters configurations analysis

Regarding the importance of each parameter in maximising the outcome of the model, it has also been studied their distribution among the best individual delivered by each execution. Fig. 9 shows the distribution for the activation functions (Fig. 9a), the initialisation functions (Fig. 9b) and the layers types defined (Fig. 9c). These figures are made based on the number of occurrences of each possible parameter value in the best individuals, which enables to understand the particularities under the most profitable configurations. Each polygon drawn in these figures represents the best individual for each execution.

By analysing the distribution of the different activation functions employed in the evolutionary search (Fig. 9a), it can be seen that the *hyperbolic tangent*, the *Rectified Linear Unit* (ReLU), which has a threshold located at zero and aims to accelerate the convergence, and the *Softplus* activation function, which is a smooth version of the ReLU, are those preferred by the algorithm as they improve the fitness of the global architecture. There is also a specific individual where the *softsign* activation function, based on quadratic polynomials, plays a key role. In contrast, *linear*, *sigmoid*, *hard sigmoid* and *softmax* activations adopt clearly a secondary role, as they are not as frequent among the best configurations.

In the case of the initialisation functions (see Fig. 9b), the results show a significantly different behaviour from that provided by the activation functions. The individuals distributions are scattered without clear patterns, meaning that there are not clearly differentiated or promising functions, except for the normal initialisation function, which is well represented in a subset of the solutions. This plot also arises that the less helpful for the individuals is the *LeCun* uniform function, based on an uniform distribution taking into account the number of inputs.

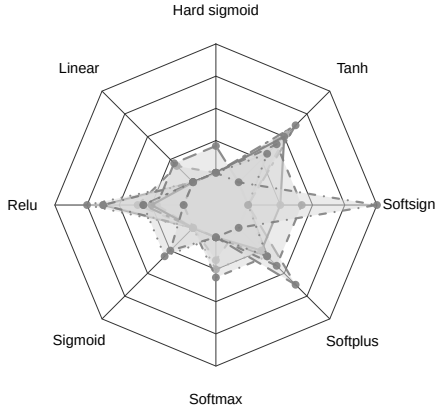
Finally, Fig. 9c represents the level of involvement of the different layer types in building the best individuals. In this case, the fully connected layer (i.e. Dense) is the most representative layer among all the individuals. This is to be expected, since this layer is in charge of defining the relations that connect the inputs with the outputs. Furthermore, there are other two layers that are used more frequently than the rest: Dropout, which randomly sets the output of some neuron to 0, and Convolution2D, which is the keystone of convolutional Deep Neural Networks. In fact, the latter appears on the top 2 architectures, that is, those individuals that achieved the best accuracies, as it is described in the following subsection.

6.3. Analysis of the best architectures found by EvoDeep

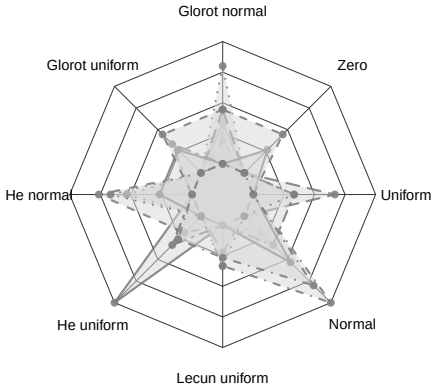
After analysing the DNN architectures that EvoDeep is able to achieve, the best two accuracies values, from the final population of each run altogether, we found that both are mainly made of Dense and Convolution2D layers, which, in addition and taking into account the results shown on the previous subsection, points at the good performance achieved by Deep Neural Network architectures that combines both types of layers.

As shown in Figure 10, best individuals (in terms of accuracy in the validation dataset) have nearly the same number of layers (7 and 8 respectively) and a rather similar architecture: both have an initial sequence of Reshape, Convolution and Flatten as well as two Dense layers at the final zone of the sequence. Although the main difference between them are the Dropout and MaxPooling2D layers, the global functioning of both architectures is rather similar. Initially, data is reshaped as a matrix in order to be processed by the convolution layers, which then extract information and features from the matrix by applying the filters. After that, the matrix is converted to vectors and then there is some kind of information summarising in both architectures: by the Dropout layer in the former, which discard some of the information learned by the previous Dense layer, and also by the MaxPooling2D layer in the latter, which reduces the dimensionality of the data.

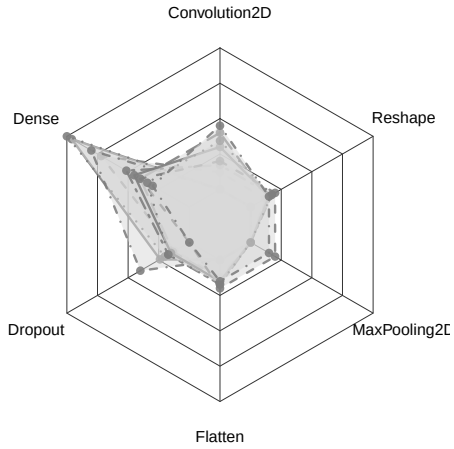
From the perspective of layers' parameters, it is noteworthy the size of the kernel filters applied on both architectures: the former uses a different number of filters which are, in turn, of different sizes while the latter uses the same number of filters with a similar size among the three Convolution layers. In fact, the parameters of the convolution layers in the former architecture are all distinct, which may suggest that combining very different convolution layers is better than using the same parameters for them. Regarding the complexity



(a) Activation functions



(b) Initialisation functions.



(c) Layer types

Figure 9: Probability of occurrence of the different parameter values.

Dense layers, both architectures have a high number of neurons in them (350 and 410 respectively).

7. Conclusions and Future Work

Deep Neural Networks are able to tackle complex classification problems using a large number of features that generate s complicated relationships between them. In contrast to the good performance of DNNs, their specification involves designing large architectures of layers and setting up their parameters. This process is usually made by hand, hence representing a bottleneck in the process of designing DNNs to solve complex problems. To tackle the previous problem,

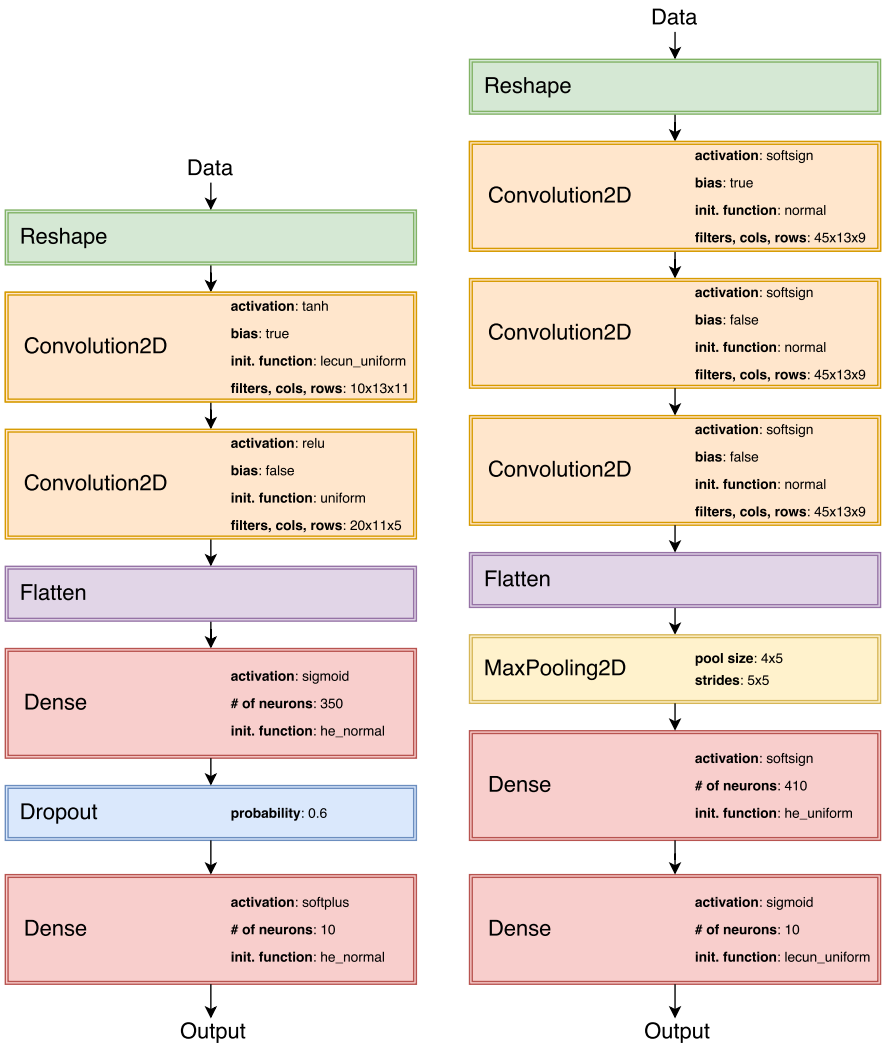


Figure 10: Best two individuals found by the algorithm, achieving accuracies of 98.67% and 98.93% for (a) and (b), respectively.

this paper presents EvoDeep, a new an automated approach that search for the the best architecture of layers, as well as their optimum parameters, by means of an Evolutionary Algorithm. The algorithm is guided by the accuracy achieved by the DNN using one dataset, the MNIST dataset, but it could be changed to another one with ease. Furthermore, the algorithm includes a Finite-State Machine to ensure that all the architectures generated are valid sequences of layers, as they have restrictions on their inputs and outputs types. The experimental

results show that EvoDeep is able to build valid DNNs architectures that, in turn, achieve good accuracies when using the aforementioned dataset.

One of the biggest problems found is the high computational resources needed to train the DNN, so as future work we are planning to study if it would be possible to reduce the training time. In addition, it would be interesting to check the algorithm using additional datasets and study its performance, in a way that it is possible to publish the algorithm as an optimization tool for Keras and other implementations of TensorFlow.

Acknowledgment

This work has been co-funded by the next research projects: EphemeCH (TIN2014-56494-C4-4-P) Spanish Ministry of Economy and Competitivity and European Regional Development Fund FEDER, Justice Programme of the European Union (2014-2020) 723180 – RiskTrack – JUST-2015-JCOO-AG/JUST-2015-JCOO-AG-1, and by the CAM grant S2013/ICE-3095 (CIBERDINE: Cybersecurity, Data and Risks). The contents of this publication are the sole responsibility of their authors and can in no way be taken to reflect the views of the European Commission.

References

[1] I. Goodfellow, Y. Bengio, A. Courville, Deep Learning, MIT Press, 2016, http://www.deeplearningbook.org.

[2] W. S. McCulloch, W. Pitts, A logical calculus of the ideas immanent in nervous activity, The bulletin of mathematical biophysics 5 (4) (1943) 115–133.

[3] B. J. A. Kröse, P. P. van der Smagt, An Introduction to Neural Networks, 4th Edition, The University of Amsterdam, Amsterdam, The Netherlands, 1991.

[4] M. Minsky, S. Papert, Neurocomputing: Foundations of research, MIT Press, Cambridge, MA, USA, 1988, Ch. Perceptrons, pp. 157–169.
URL http://dl.acm.org/citation.cfm?id=65669.104395

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

[6] T. D. Team, TensorFlow: Large-scale machine learning on heterogeneous systems, software available from tensorflow.org (2015).
URL http://tensorflow.org/

[7] Theano Development Team, Theano: A Python framework for fast computation of mathematical expressions, arXiv e-prints abs/1605.02688.
URL http://arxiv.org/abs/1605.02688

[8] Y. LeCun, Y. Bengio, G. Hinton, Deep learning, Nature 521 (7553) (2015) 436–444.

[9] K. He, X. Zhang, S. Ren, J. Sun, Deep residual learning for image recognition, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2016, pp. 770–778.

[10] Z. Yuan, Y. Lu, Y. Xue, Droiddetector: android malware characterization and detection using deep learning, Tsinghua Science and Technology 21 (1) (2016) 114–123.

[11] Z. Yuan, Y. Lu, Z. Wang, Y. Xue, Droid-sec: deep learning in android malware detection, in: ACM SIGCOMM Computer Communication Review, Vol. 44, ACM, 2014, pp. 371–372.

[12] S. Hou, A. Saas, Y. Ye, L. Chen, Droiddelver: An android malware detection system using deep belief network based on api call blocks, in: International Conference on Web-Age Information Management, Springer, 2016, pp. 54–66.

[13] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, et al., Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups, IEEE Signal Processing Magazine 29 (6) (2012) 82–97.

[14] H. Lee, P. Pham, Y. Largman, A. Y. Ng, Unsupervised feature learning for audio classification using convolutional deep belief networks, in: Advances in neural information processing systems, 2009, pp. 1096–1104.

[15] T. Kuremoto, S. Kimura, K. Kobayashi, M. Obayashi, Time series forecasting using a deep belief network with restricted boltzmann machines, Neurocomputing 137 (2014) 47–56.

[16] X. Qiu, L. Zhang, Y. Ren, P. N. Suganthan, G. Amaratunga, Ensemble deep learning for regression and time series forecasting, in: Computational Intelligence in Ensemble Learning (CIEL), 2014 IEEE Symposium on, IEEE, 2014, pp. 1–6.

[17] K. Simonyan, A. Zisserman, Two-stream convolutional networks for action recognition in videos, in: Advances in neural information processing systems, 2014, pp. 568–576.

[18] X. Yao, Evolving artificial neural networks, Proceedings of the IEEE 87 (9) (1999) 1423–1447.

[19] M. Srinivas, L. Patnaik, Learning neural network weights using genetic algorithms-improving performance by search-space reduction, in: Neural Networks, 1991. 1991 IEEE International Joint Conference on, IEEE, 1991, pp. 2331–2336.

[20] F. H.-F. Leung, H.-K. Lam, S.-H. Ling, P. K.-S. Tam, Tuning of the structure and parameters of a neural network using an improved genetic algorithm, IEEE Transactions on Neural networks 14 (1) (2003) 79–88.

[21] J. R. Koza, J. P. Rice, Genetic generation of both the weights and architecture for a neural network, in: Neural Networks, 1991., IJCNN-91-Seattle International Joint Conference on, Vol. 2, IEEE, 1991, pp. 397–404.

[22] J. Gascón-Moreno, S. Salcedo-Sanz, B. Saavedra-Moreno, L. Carro-Calvo, A. Portilla-Figueras, An evolutionary-based hyper-heuristic approach for optimal construction of group method of data handling networks, Information Sciences 247 (2013) 94–108.

[23] X. Yao, Y. Liu, A new evolutionary system for evolving artificial neural networks, IEEE transactions on neural networks 8 (3) (1997) 694–713.

[24] A. Abraham, Meta learning evolutionary artificial neural networks, Neurocomputing 56 (2004) 1–38.

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

[26] V. Maniezzo, Genetic evolution of the topology and weight distribution of neural networks, IEEE Transactions on neural networks 5 (1) (1994) 39–53.

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

[28] D. V. Vargas, J. Murata, Spectrum-diverse neuroevolution with unified neural models, IEEE Transactions on Neural Networks and Learning Systems.

[29] A. Martin, H. D. Menéndez, D. Camacho, Genetic boosting classification for malware detection, in: Evolutionary Computation (CEC), 2016 IEEE Congress on, IEEE, 2016, pp. 1030–1037.

[30] A. Martín, H. D. Menéndez, D. Camacho, Mocdroid: multi-objective evolutionary classifier for android malware detection, Soft Computing (2016) 1–11.

[31] A. Martín, H. D. Menéndez, D. Camacho, Studying the influence of static api calls for hiding malware, in: Conference of the Spanish Association for Artificial Intelligence, Springer, 2016, pp. 363–372.

[32] A. Martín, H. D. Menéndez, D. Camacho, String-based malware detection for android environments, in: International Symposium on Intelligent and Distributed Computing, Springer International Publishing, 2016, pp. 99–108.

[33] A. Martín, A. Calleja, H. D. Menéndez, J. Tapiador, D. Camacho, Adroit: Android malware detection using meta-information, in: Computational Intelligence (SSCI), 2016 IEEE Symposium Series on, IEEE, 2016, pp. 1–8.

[34] J. D. Schaffer, D. Whitley, L. J. Eshelman, Combinations of genetic algorithms and neural networks: A survey of the state of the art, in: Combinations of Genetic Algorithms and Neural Networks, 1992., COGANN-92. International Workshop on, IEEE, 1992, pp. 1–37.

[35] S. Marshall, R. Harrison, Optimization and training of feedforward neural networks by genetic algorithms, in: Artificial Neural Networks, 1991., Second International Conference on, IET, 1991, pp. 39–43.

[36] O. A. Abdalla, A. O. Elfaki, Y. M. Almurtadha, Optimizing the multilayer feed-forward artificial neural networks architecture and training parameters using genetic algorithm, International Journal of Computer Applications 96 (10).

[37] O. E. David, I. Greental, Genetic algorithms for evolving deep neural networks, in: Proceedings of the Companion Publication of the 2014 Annual Conference on Genetic and Evolutionary Computation, ACM, 2014, pp. 1451–1452.

[38] F. Chollet, Keras, https://github.com/fchollet/keras (2015).

[39] Y. LeCun, L. Bottou, G. B. Orr, K. R. Müller, Efficient backprop, Lecture notes in computer science (1998) 9–50.

[40] X. Glorot, Y. Bengio, Understanding the difficulty of training deep feedforward neural networks., in: Aistats, Vol. 9, 2010, pp. 249–256.

[41] K. He, X. Zhang, S. Ren, J. Sun, Delving deep into rectifiers: Surpassing human-level performance on imagenet classification, in: Proceedings of the IEEE international conference on computer vision, 2015, pp. 1026–1034.

[42] A. E. Eiben, J. E. Smith, Introduction to Evolutionary Computing, Natural Computing, Springer-Verlag Berlin Heidelberg, 2003.

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