Evolving Architectures for Convolutional Neural Networks using the Genetic Algorithm

Alessandro Saviolo
Department of Information Engineering
Padova, Italy
alessandro.saviolo.2@studenti.unipd.it

Matteo Terranova
Department of Information Engineering
Padova, Italy
matteo.terranova@studenti.unipd.it

***Abstract*—In this paper, the genetic algorithm (GA) is used to improve the architecture of a given Convolutional Neural Network (CNN) that is used to address image classification tasks.**

Designing the architecture for a Convolutional Neural Network is a cumbersome task because of the numerous parameters to configure, including activation functions, layer types, and hyper-parameters. With the large number of parameters for most networks nowadays, it is intractable to find a good configuration for a given task by hand. Due to the drawbacks of existing methods and limited computational resources available to interested researchers, most of these works in CNNs are typically performed by experts which use new theoretical insights and intuitions gained from experimentation.

The proposed method is based on the genetic algorithm since it does not require a rich domain knowledge. The goal of the algorithm is to help interested researchers to optimize their CNNs and to allow them to design optimal CNN architectures without the necessity of expert knowledge or a lot of trial and error.

***Index Terms*—convolutional neural network, genetic algorithm, neural network architecture optimization, evolutionary deep learning, CIFAR-10 dataset.**

I. Introduction

Convolutional Neural Networks (CNNs) have gained a remarkable success in image recognition tasks in recent years and are applied to various computer vision applications [21]. Since LetNet-5 was proposed [13], various variants of CNNs have been investigated (i.e., ResNet [8], DenseNet [9], AlexNet [12], VGG [15], GoogleNet [17]), each one with a different architecture. These variants significantly improve the classification accuracies of the best rivals in image classification tasks. However, designing such networks requires a rich domain knowledge from both the CNNs and the investigated problems. Because the performance of the CNNs strongly relies on the investigated data, it is expected that there is a major limitation to this design manner.

In this paper, we present and analyze an efficient GA method to automatically improve the architecture of a given CNN to address image classification problems. To achieve this goal, it is necessary to design a flexible gene encoding scheme of the architecture, which does not constrain the maximal length of the building blocks in CNNs. With this gene encoding scheme, the evolved architecture is expected to benefit CNNs to achieve good performance in solving different tasks at hand. Moreover, it is mandatory to develop associated selection, crossover, and mutation operators that can cope with the designed gene encoding strategies of the architectures.

The remainder of this paper is organized as follows. Firstly, the background is presented in Section II. Then, the details of the proposed algorithm are documented in Section III. Next, the experimental designs and experimental results are shown in Sections IV and V, respectively. Finally, conclusions and future works are outlined in Section VI.

II. Background

A. Genetic Algorithm

The genetic algorithm (GA) is an adaptive heuristic search algorithm based on the evolutionary ideas of natural selection and genetics. The method was developed by John Holland and popularized by David Goldberg [4]. It has been recognized that GAs are capable of generating high-quality optimal solutions by using bio-inspired operators such as mutation, crossover and selection [14].

The flowchart of a genetic algorithm is shown in Fig. 2. In particular, a population of individuals (i.e., CNNs with architecture variants) is randomly initialized first, and then the fitness of each individual is evaluated. The fitness is measured by a deterministic function (i.e., the fitness function) that is based on the context of the problem to be optimized (i.e., performance of CNNs on specific image classification tasks). After evaluating the individuals, the ones that have a better fitness are chosen by the selection operation. From the selected parents, a new offspring is generated by the crossover and mutation operators. The generated offspring is evaluated by the fitness function, and compared with the current population. The best individuals among the population and the offspring are kept and form the new population (the population is kept at a static size). Through repeating a series of these operations, it is expected to find the optimal solution from the population when the GA terminates. The termination criterion for the GA is a predefined maximal generation number (e.g., 20 generations).

The convergence of the genetic algorithm mainly depends on the operators used (i.e., selection, crossover and mutation). The operators are employed to create and maintain genetic diversity (mutation operator), combine existing solutions into new solutions (crossover) and select between solutions (selection). Each operator is explained in the following subsections.

1) Selection: The selection operator defines which individuals of the population will be used for generating the new offspring at each generation. There exists many variants of the selection operator. Different methods may choose different solutions as being the best.

2) Crossover: The crossover operator produces a new individual by combining portions of two selected individuals. If the selected individuals have a good fitness, the genetic algorithm is more likely to create a better individual.

As with selection, there are a number of different methods for combining the parent solutions. For example, single-point crossover splits the two selected individuals into two parts and combines the first part of one individual with the second of the other. An example of a single-point crossover is shown in Fig. 1.

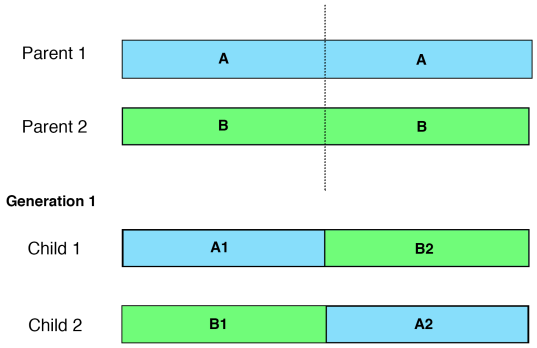


Fig. 1: A single-point crossover.

3) Mutation: The mutation operator encourages genetic diversity amongst individuals and attempts to prevent the genetic algorithm to converge to a local minimum. By applying the mutation operator to an individual, it may change entirely. So, this operator denies the individuals to be too close to one another.

There are many different methods of mutation that may be used. For example, a simple permutation of the encoding of the individual (e.g., swap of two layers in CNNs).

B. Architecture of Convolutional Neural Network

A Convolutional Neural Network takes an image and passes it through a series of layers that are specified by the architecture of the network [10] [22]. An example of a CNN architecture is shown in Fig. 3. Each layer of the architecture is explained in the following subsections, and a detailed schema of the parameters of each layer is proposed in Table I.

1) Convolutional layer: The convolutional layer is a core building block of any CNN architecture. It performs convolutional operations on the input data by employing different filters (i.e., matrices).

Filters are randomly initialized with a predefined size (i.e., the filter width and the filter height). During the convolutional operation, each filter travels from the leftmost to the rightmost of the input data with the step size equal to a stride (i.e., the stride width), and then travels again after moving downward

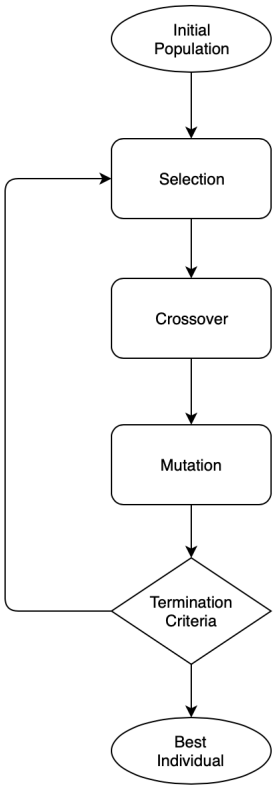


Fig. 2: The flowchart of the genetic algorithm.

with the step size equal to a stride (i.e., the stride height), until it reaches the right bottom of the input image. At each position, the filter is applied to the image by multiplying each filter value with the corresponding pixel value, and summing the results to give the output of the filter. The set of filter outputs form a new matrix called the feature map. In a convolutional layer, multiple filters are allowed to coexist, producing a set of feature maps. The exact number of feature maps used is a parameter in the architecture of the corresponding CNN.

Moreover, depending on whether to keep the same sizes between the feature map and the input data (i.e., if we want to slide the filter to the bordering elements of the input image matrix), two convolutional operations are applied: the VALID (without padding) and the SAME (with padding) convolutional operations. An example of the VALID convolutional operation is illustrated by Fig. 4.

Hence, the parameters of a convolutional layer that need to be specified before applying the convolution are: the number of feature maps, the filter size, the stride size and the convolutional operation type.

2) Pooling layer: The pooling layer has common components of a convolutional layer except that: the filter is called the kernel and it has no value; the output of a kernel is the maximal (i.e., max pooling) or mean (i.e., average pooling) value of the elements where it slides; the spatial size of the input data is not changed through a pooling layer. An example of the pooling operation is illustrated by Fig. 5.

The process of pooling has two main advantages. The first is that the amount of parameters or weights is reduced by 75%, thus lessening the computation cost. The second is that it will control overfitting as it reduces the number of parameters and computations in the network.

Hence, the parameters of a pooling layer are the kernel size, the stride size and the pooling type used.

3) Dropout layer: The dropout layer helps preventing the overfitting of the CNNs. Such characteristic is obtained by randomly setting a fraction rate of input units to 0 at each update during training time, so that all the neurons selected and all its connections are ignored in the update operation of a single iteration. This has the effect of preventing the formation of overspecialized clusters of units, as the trained structure changes constantly.

Hence, the fraction of the input units to drop is a parameter for a CNN's architecture.

4) Fully-connected layer: The fully-connected layer is usually incorporated into the tail of a CNN. It is a traditional Multi-Layer Perceptron layer that uses a softmax activation function in the output layer. The softmax activation function generates the outputs in the range $[0, 1]$, which can be seen as the probability that any of the classes are true.

The number of fully-connected layers and the number of neurons in each fully-connected layer are also parameters for a CNN's architecture.

TABLE I: Layer Parameters

Layer	Parameters
Convolutional	feature maps, filter size, stride size, padding, convolutional type
Pooling	kernel size, stride size, padding, pooling type
Dropout	fraction rate of input units
Fully-connected	number of neurons

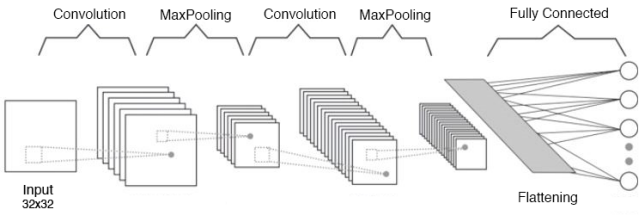


Fig. 3: A general architecture for a Convolutional Neural Network. In particular, the architecture has two convolutional operations, two pooling operations, the resulted four groups of feature maps, and the full connection layer in the tail. The numbers in the figure refer to the sizes of the corresponding layer: the input data is with $24x24$, the output is with $128x1$, and the other numbers denote the feature map configurations (e.g., $4@20x20$ implies there are 4 feature maps, each with the size of $4x4$).

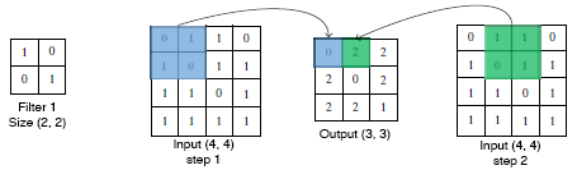


Fig. 4: An example of a convolutional operation. The filter has size 2×2, the stride 1×1, and the input data 4×4. Since padding is set to VALID, the generated feature map has size 3×3. The colored areas in the input data refer to the overlaps with the filter at different positions. The colored areas in the feature map are the resulted convolutional outcomes.

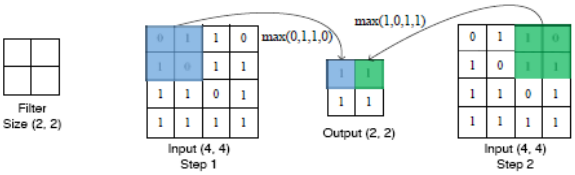


Fig. 5: An example of a max pooling operation. The kernel has size 2×2, the stride 2×2, and the input data 4×4. The colored areas in the input data refer to the overlaps with the filter at different positions. The colored areas in the feature map are the resulted pooling outcomes.

III. The Proposed Algorithm

In this section, we firstly present the framework of the proposed algorithm, and then we detail the main steps.

Fig. 6 shows the flowchart of the proposed algorithm (note the differences with Fig. 2).

A. Algorithm overview

Algorithm 1 outlines the structure of the proposed method. Specifically, by giving the input CNN architecture, the sizes of the population and the offspring and the number of generations, the proposed algorithm begins to work, through a series of evolutionary processes, and finally discovers an improved architecture for the input CNN.

At first, the initial population is computed by strongly mutating the input CNN architecture (lines $1-5$). Then, the evolution begins to take effect for the input number of the generations (lines $6-15$).

At each step of the evolution, a new offspring is computed, evaluated on the given dataset and compared with the population generated the step before. Specifically, each individual of the new offspring, which encodes a particular architecture of the CNN, is generated by first selecting two individuals from the population (line 8), and then combining them by using the genetic operators (lines $9-10$). After that, the new individual is evaluated (line 11) and it is compared with the population (line 12). The population of individuals surviving into the next generation is obtained from the current population (line 14).

In the following subsections, the key steps in Algorithm 1 are described in detail.

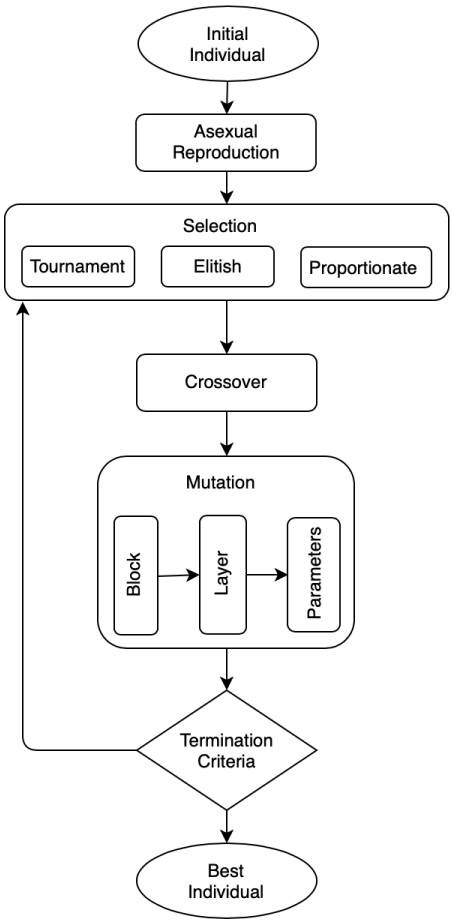


Fig. 6: The flowchart of the proposed genetic algorithm.

Algorithm 1: Framework of The Proposed Algorithm

Input: input CNN architecture, population size P, offspring size O, number of generations G.
Output: optimized input CNN architecture.

for $s \leftarrow 0$ **to** P **do**
  $i_s \leftarrow$ *strongly mutate the input CNN architecture*;
  Evaluate the fitness of i_s;
  $P_0 \leftarrow P_0 \cup \{i_s\}$;
end
for $t \leftarrow 0$ **to** G **do**
  for $k \leftarrow 0$ **to** O **do**
    Select two individuals from P_t;
    Generate a new individual i_k *using crossover*;
    Randomly apply soft mutation to i_k;
    Evaluate the fitness of i_k;
    Evolve P_t *with* i_k;
  end
  $P_{t+1} \leftarrow P_t$
end
Select the best individual from P_t *and decode it to the corresponding CNN architecture.*

B. Encoding strategy

As introduced in Subsection II-B, four different building blocks (i.e., the convolutional layer, the pooling layer, the dropout layer, and the fully-connected layer) exist in the architectures of CNNs. Therefore, they should be encoded in the same individual for evolution.

In the proposed encoding strategy, we design each individual as a sequence of blocks with variable length (Fig. 7). The initial block of every individual is composed by a convolutional layer (conventional decision for CNN architectures) and can only be affected by a parameter mutation (will be introduced in Subsection III-G). The final block contains only the fully-connected layer and, as the initial block, can only be affected by a parameter mutation. Each remaining block of the encoding contains at least a convolutional or a pooling layer, but it can also contain a combination of the two (the maximal size of each block is not constrained).

Furthermore, since the performance of CNNs is highly affected by their depths [15] and the encoding strategy can generate CNN architectures of variable depth (i.e., without constraining the architecture search space), the proposed algorithm can reach the best results.

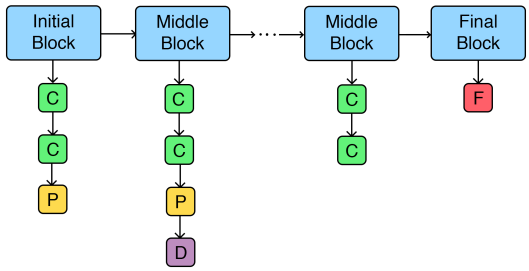


Fig. 7: An example of the variable-lenght encoding of an individual in the proposed algorithm. The initial block always contains a convolutional layer in the first position. The final block contains only the fully-connected layer. The number of middle blocks is random (can also be equal to 0).

C. Population initialization

The initial population is computed by strongly mutating the input CNN architecture (the strong mutation is presented in Subsection III-G). Each time a new individual is computed, it is evaluated by the fitness function and then added to the initial population.

Maintaing the link with natural evolution, this initialization method is analogous to *asexual reproduction*. The effectiveness of the proposed algorithm highly depends by this initialization, since it defines the variability of the system, which is the key for better adaptability.

D. Fitness evaluation

Fitness evaluation aims at giving a quantitative measure of the quality of an individual. Since the proposed algorithm concerns on solving image classification tasks, the classification error is the best strategy to compute the fitness of an individual.

In order to compute the classification error, it is necessary to split the original dataset into a training set D_{train} and a validation set D_{val}, train each individual on D_{train} and then test it on D_{val}. The resulting validation loss is the fitness of the individual. This implies that the best individual is the one with the lowest fitness.

E. Selection

Selection chooses which individuals of the population will be used for generating the new offspring at each iteration. In the proposed algorithm, three different selection techniques have been implemented, namely elitism selection, tournament selection and proportionate selection. In order to maximize the variability of the algorithm (i.e., escaping from local minimum) and to make of the most of the strenghts of such techniques (presented below), the choice among the three is left random at each generation.

Elitism selection is presented by Algorithm 2. It is the simplest type of selection, since it just chooses the best two individuals (lines $1-2$). Elitism selection ensures that the proposed algorithm does not waste time re-discovering previously discarded partial solutions, and this can sometimes have a dramatic impact on performance. The problem with elitism is that it may cause the algorithm to converge on local maximum instead of the global maximum.

Tournament selection is presented by Algorithm 3. It is implemented by choosing the first individual at random from the population (line 2), and then by selecting the other among all the individuals with a lower fitness. The selection of the second individual is made random (lines $3-8$).

Proportionate selection is presented by Algorithm 4. It is implemented by first spreading a probability distribution over the population (lines $1-12$), so that the selection probability of an individual (its selection pressure) is proportional to its fitness, and then by selecting two random individuals from the distribution (lines $13-17$).

The tournament selection and the proportional selection share the same time complexity but offer different advantages and disadvantages: the tournament selection is translation and scale invariant since the selection pressure does not change when every individual's fitness is multiplied and added by a constant; on the other hand, the selection intensity of proportional selection is the only one that is sensitive to the current population distribution.

F. Crossover

Crossover produces a new individual by combining portions of two selected individuals. The crossover operator of the proposed algorithm is inspired by the one-point crossover in traditional genetic algorithms (Subsection II-A2). The proposed crossover operation can be seen in Fig. 8. First each parent individual is randomly split into two parts, and then a new individual is obtained by taking the first part from one parent and the second part from the other. So, the designed crossover differs from the traditional one, since it can produce individuals with variable lengths.

Algorithm 2: Elitism Selection

Input: population P.
Output: two individuals from P.

1 *Sort P on ascending order based on fitness*;
2 *Return the first two individuals in P.*

Algorithm 3: Tournament Selection

Input: population P.
Output: two individuals from P.

1 *Sort P on ascending order based on fitness*;
2 *Randomly pick a number i from $[0, |P|)$*;
3 $j \leftarrow i$;
4 **while** $j < |P|$ **do**
5 | $j \leftarrow j + 1$;
6 | *Return individuals i and j with probability* 0.5;
7 **end**
8 *Return individuals i and* 0.

Algorithm 4: Proportionate Selection

Input: population P.
Output: two individuals from P.

1 $sum \leftarrow 0$;
2 **for** $i \leftarrow 0$ **to** $|P|$ **do**
3 | $fitness_i \leftarrow$ *Fitness of individual i*;
4 | $sum \leftarrow sum + fitness_i$;
5 **end**
6 $percentual \leftarrow$ *Empty list*;
7 **for** $i \leftarrow 0$ **to** $|P|$ **do**
8 | $count \leftarrow (100 \cdot fitness_i)$ / sum;
9 | **for** $j \leftarrow 0$ **to** $count$ **do**
10 | | $percentual$ *append* i;
11 | **end**
12 **end**
13 *Randomly pick two numbers i and j from* $[1, 100]$;
14 **while** $i == j$ **do**
15 | *Randomly pick two numbers i and j from* $[1, 100]$;
16 **end**
17 *Return individuals $percentual[i]$ and $percentual[j]$.*

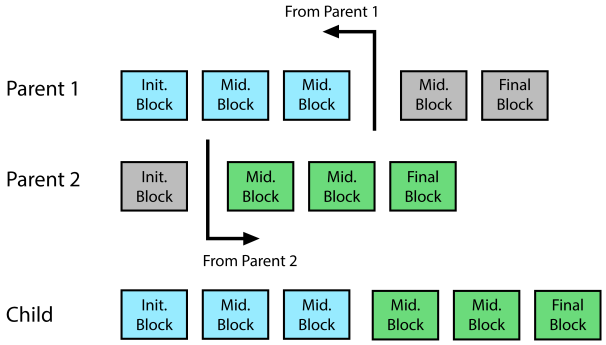


Fig. 8: An example of the crossover process.

G. Mutation

Mutation is the key operator for the proposed algorithm. Two types of mutation are used: strong and soft. The difference between the two is the degree they mutate the architecture. Specifically, a *strong mutation* is obtained by three sequential operations, namely block mutation, layer mutation and parameter mutation, while a *soft mutation* is obtained by a layer mutation and a parameter mutation.

The block mutation selects at random a block among the encoding of the individual (except for the initial and the tail blocks), and then it decides randomly to add a new layer or to remove an existing one (selected at random among the layers in the block). This mutation alters the structure of the architecture at block level. By using this mutation, the CNN architectures are capable of adapting their depth to the problem considered. Algorithm 5 outlines the block mutation.

As the block mutation, the layer mutation selects at random a block among the encoding of the individual. Then, it considers a random layer from the selected block, and it mutates the layer depending on its type. If it is a Convolutional layer, then the operator divides by 2 the number of feature maps of the layer and creates a new layer by applying a deepcopy (i.e., copy that replicates also the parameters). This new layer is then positioned right after the selected one. If it is a Pooling layer, then the operator replaces it with a new Convolutional layer with stride size equal to 2, so that the new layer downsamples the image as the Pooling. This mutation makes the architecture more expressive [16]. For example, by replacing a Pooling layer with a Convolutional, the CNN is capable to learn certain properties that are lost by the pooling layer . This is due to the fact that pooling is a fixed operation, while convolution can be learned. The down side is that it also increases the number of trainable parameters (the pooling layer has no parameters to learn). Algorithm 6 provides the layer mutation pseudocode.

The parameter mutation alters the value of the parameters of the different layers. This mutation is applied to all layers of all blocks, including the first and the final blocks. At each iteration, this operator considers a layer and randomly decides to apply the mutation or not. Algorithm 7 provides the parameter mutation pseudocode.

The mutation applied to the parameter depends on the layer type. If it is a Convolutional layer, then the mutation can multiply or divide by 2 or by 4 the number of feature maps, or it can switch the padding value (e.g., from $VALID$ to $SAME$). If it is a Pooling layer, then the mutation can switch the pooling type (e.g., from MAX to AVG), or it can switch the padding value. If it is a Dropout layer, then the mutation can increase or decrease by 0.10 or 0.20 the dropout rate. Finally, if it is a Fully-connected layer, then the mutation can multiply or divide by 2 the number of neurons. This mutation combined with the layer mutation alters the structure of the architecture at layer level. By doing so, these operators are capable of maximizing the variance of the new individual produced and boosting up the exploration of the architecture search space.

Algorithm 5: Block Mutation

Input: an individual i.
Output: i mutated.

Select randomly a block B from the encoding of i;
Randomly pick a number l from $[0, 1]$;
Randomly pick a number k from $[0, 2]$;
if $l == 0$ **then**
  if $k == 0$ **then**
    Add a Convolutional layer to B;
  else if $k == 1$ **then**
    Add a Pooling layer to B;
  else
    Add a Dropout layer to B;
  end
else
  Remove a random layer from B;
end

Algorithm 6: Layer Mutation

Input: an individual i.
Output: i mutated.

Select randomly a block B from the encoding of i;
Select randomly a layer l from B;
if *l is a Convolutional layer* **then**
  Divide by 2 the number of features maps in l;
  Create a new Convolutional layer l' equal to l;
  Add l' to B right after l.
else if *l is a Pooling layer* **then**
  Remove l from B;
  Create a new Convolutional layer l' with stride 2;
  Add l' to B in the position of l.

Algorithm 7: Parameter Mutation

Input: an individual i.
Output: i mutated.

foreach *block B in the encoding of i* **do**
  foreach *layer l in B* **do**
    Randomly pick a number p from $[0, 1]$;
    if $p == 0$*;*
     then
      Alter the parameters of l;
    end
  end
end

IV. EXPERIMENTAL DESIGN

A. CIFAR-10 dataset

We test our method on the image classification task using the CIFAR-10 dataset [11], in which the number of classes is 10. The numbers of training and validation images are, respectively, 50000 and 10000, and the size of each image is 32×32.

We consider two experimental scenarios: the default scenario and the small-data scenario. The default scenario uses the default numbers of the training and validation images, whereas the small-data scenario assumes that only 5000 images are available. In particular, in the small-data scenario, a randomly sample of 4500 images is used for the training and the remaining 500 images are employed for the validation.

B. Experimental settings

The proposed algorithm and the experiments reported in Section V have been performed with Google Colab [7] using Keras [2]. In particular, each run is excecuted by a computer equipped with a Tesla $K80$ GPU card.

C. Parameter settings

The effectiveness of the genetic algorithm is determined by the choice of the parameters (e.g., population and offspring size). This choice has been intensively studied in literature [5], and it has been proven that it is task dependent [6].

In the proposed algorithm, the choice of the parameters is determined by the quality of the solution one wants to obtain, by the computational resources available and by the size of the dataset considered. The tradeoff between these specifications defines the computational time required by the algorithm to converge to a solution. Specifically, if the computational resources available are only a few (e.g., no GPU accessible for the computation), but one still wants a robust improvement of his CNN architecture, the computational time required by the proposed algorithm may become challenging.

The choice of the population size defines the adaptability of the proposed algorithm. If the population size is too small, then the algorithm is not capable of diversifying enough the search and it tends to stick to a limited portion of the solution space (i.e., a large region of the solution space remains completely unexplored by the search). On the other hand, if the population size is too big, then the computational time required by the algorithm may become unfeasible.

The larger the choice of the population size, the larger should also be the number of generations and the offspring size. These two parameters define the quality of the solution returned by the proposed algorithm. Specifically, if they are set too small, then the algorithm is not capable of intensifying enough the search and it tends to make a shallow exploration of the solution space (i.e., the search is not deepened in promising regions).

V. EXPERIMENTAL RESULTS AND ANALYSIS

In this section, the proposed algorithm is tested on a baseline CNN architecture for the CIFAR-10 dataset, and the resulting CNN is presented and analyzed.

The selected baseline CNN architecture is VGG [15], since it is easy to understand and implement. The VGG architecture involves stacking two Convolutional layers with filters of size 3×3, followed by a Max Pooling layer. The sequence of these three layers is then repeated multiple times, and the number of feature maps in each sequence is increased with the depth of the network (e.g., $32, 64, 128, 256$ for the first four sequences of the model).

The section is organized as follows. In Subsection V-A, the VGG architecture considered for the experiment consists of one sequence (namely VGG-1), whereas in Subsection V-C, the VGG architecture considered for the experiment consists of two sequences (namely VGG-2). In Subsections V-B and V-D, the resulting CNN architectures generated by the proposed algorithm are described and illustrated.

A. VGG-1: Experimental result

The VGG-1 model consists of two Convolutional layers, followed by a Max-Pooling and the Fully-connected layer. Since the model is very simple, for this experiment the small-data scenario has been considered (Subsection IV-A).

The parameters of the proposed algorithm chosen for this experiment are shown in Table II. The choice is based on the computational resources available (Subsection IV-B) and the simple data scenario considered.

The VGG-1 model and the resulting CNN generated by running the proposed algorithm on VGG-1 are illustrated in Fig. 9, and the architectures are presented, respectively, in Table III and Table IV.

The resulting CNN model consists of three blocks. The initial block contains two Convolutional layers, followed by a Max-Pooling. The middle block contains three Convolutional layers which differs by the number of feature maps (i.e., $16, 32, 256$). The tail block contains the Fully-connected layer.

TABLE II: Proposed Algorithm chosen Parameters

Parameter	Value
# Generations	6
Population Size	6
Offspring Size	4
# Epochs	15
Batch Size	32

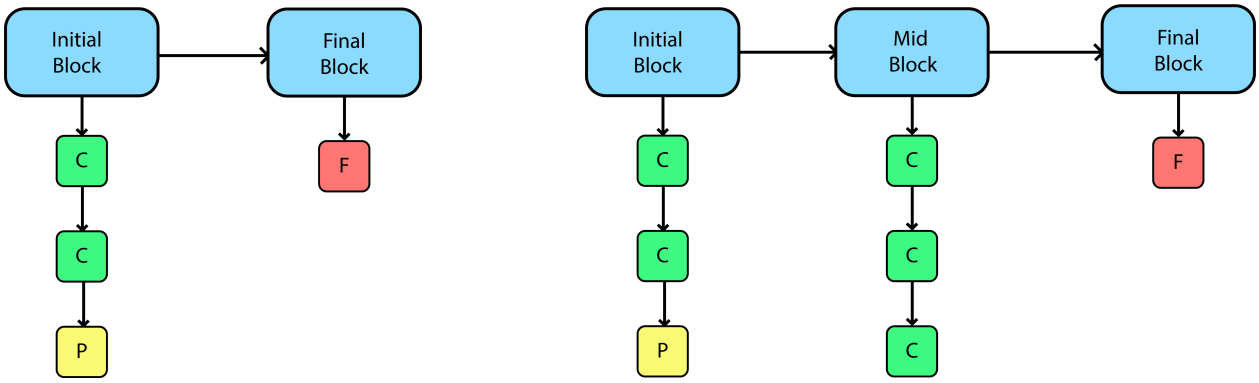


Fig. 9: The diagram on the left represents the VGG-1 model. The diagram on the right represents the resulting CNN model generated by the proposed algorithm from VGG-1. The model on the right is characterized by a greater learning capability thanks to the additional Convolutional layers in the middle block.

TABLE III: VGG-1 Architecture

Layer	Parameters
Convolutional	feature maps = 32, filter size = 3×3, stride size = 1×1, padding = SAME
Convolutional	feature maps = 32, filter size = 3×3, stride size = 1×1, padding = VALID
Max Pooling	kernel size = 2×2, stride size = 2×2, padding = SAME
Fully-Connected	units = 128

TABLE IV: The Resulting CNN Architecture from VGG-1

Layer	Parameters
Convolutional	feature maps = 64, filter size = 3×3, stride size = 1×1, padding = SAME
Convolutional	feature maps = 64, filter size = 3×3, stride size = 1×1, padding = VALID
Max Pooling	kernel size = 2×2, stride size = 2×2, padding = SAME
Convolutional	feature maps = 16, filter size = 3×3, stride size = 1×1, padding = SAME
Convolutional	feature maps = 32, filter size = 3×3, stride size = 1×1, padding = VALID
Convolutional	feature maps = 256, filter size = 3×3, stride size = 1×1, padding = VALID
Fully-Connected	units = 128

B. VGG-1: Experimental analysis

The VGG-1 model achieves an accuracy of 43.6% and a loss of 4.98 on the validation set after being trained for 15 epochs. The accuracy and loss curves are illustrated in Fig. 10.

The resulting CNN model generated by the proposed algorithm achieves an accuracy of 52.3% and a loss of 3.07 on the validation set after being trained for 15 epochs. The accuracy and loss curves are illustrated in Fig. 11.

The results are strictly related to the CNN architecture found. As a matter of fact, the model returned by the proposed algorithm presents a new middle block of three Convolutional layers, which increase the capacity of the model and its capability to learn.

Even if the models are quite different, the training losses and accuracies are similar. This is due to the small-data scenario considered, which causes the overfitting of both the models. Both models start overfitting in epoch 2, as they become more specialized in recognizing the patterns of the few images provided as input. The accuracy curves represent the same trend, except for the fact that the returned CNN converges faster than VGG-1. A similar behavior can be observed in the losses curves. As before the trend is similar, however the returned CNN seems more stable, which can be an indicator of how well the model will behave on a real classification task with a proper training with respect to the original one.

It is observable that as the number of epochs grows, the accuracy does not increase, whereas the loss seems to increase. Such fact can be countered by increasing the number of samples or adding more regularizers, at the price of increasing the computational cost for each epoch.

By comparing the losses and the accuracies obtained by the VGG-1 model and the returned CNN, it seems reasonable to state that the latter is more suitable for the classification task considered, and that, after a complete training, it will perform better than the first.

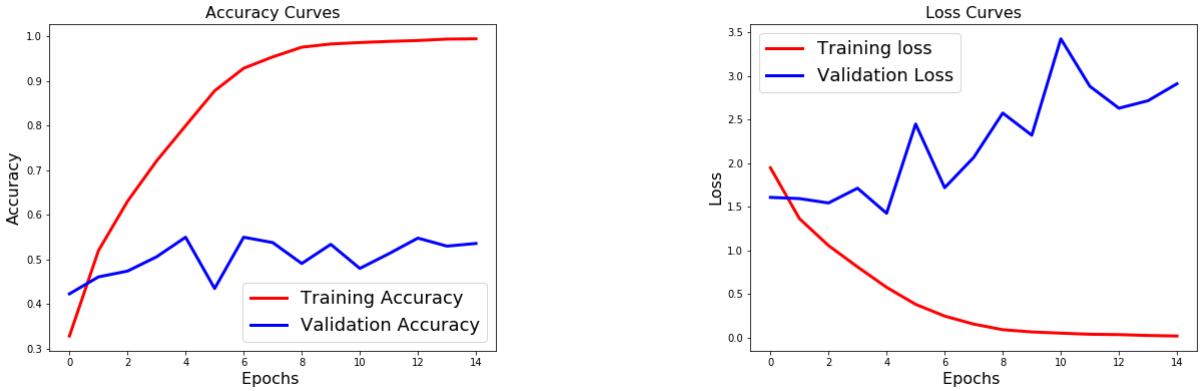


Fig. 10: Accuracy and Loss curves generated by the VGG-1 model after being trained for 15 epochs. The training loss and accuracy are, respectively, 0.0547 and 0.9834, the validation loss and accuracy are, respectively, 4.9833 and 0.4360.

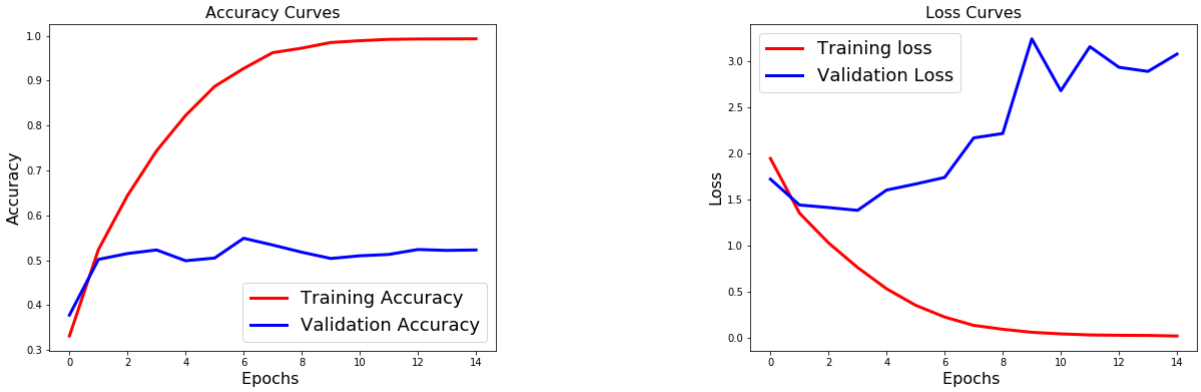


Fig. 11: Accuracy and Loss curves generated by the proposed algorithm on the resulting CNN model. The training loss and accuracy are, respectively, 0.0220 and 0.9930, the validation loss and accuracy are, respectively, 3.0766 and 0.5230.

C. VGG-2: Experimental result

The VGG-2 model extends the VGG-1 by adding a middle block (i.e., two Convolutional layers, followed by a Max-Pooling). For this experiment the default scenario has been considered (Subsection IV-A).

The parameters of the proposed algorithm chosen for this experiment are shown in Table V. The choice is based on the computational resources available (Subsection IV-B). Due to the chosen parameters, the computational time required by running this experiment is significantly high.

The VGG-2 model and the resulting CNN generated by running the proposed algorithm on VGG-2 are illustrated in Fig. 12, and the architectures are presented, respectively, in Table VI and Table VII.

The resulting CNN model consists of three blocks. The initial and the final blocks are equal to the relative blocks of the VGG-2 model. The middle block contains four Convolutional layers which differs by the number of feature maps (i.e., $64, 32, 32, 16$), followed by an Average-Pooling.

Note that the two models are very similar. This is due to the fact that the VGG-2 model is a valuable model for the scenario considered.

TABLE V: Proposed Algorithm chosen Parameters

Parameter	Value
# Generations	10
Population Size	10
Offspring Size	10
# Epochs	50
Batch Size	64

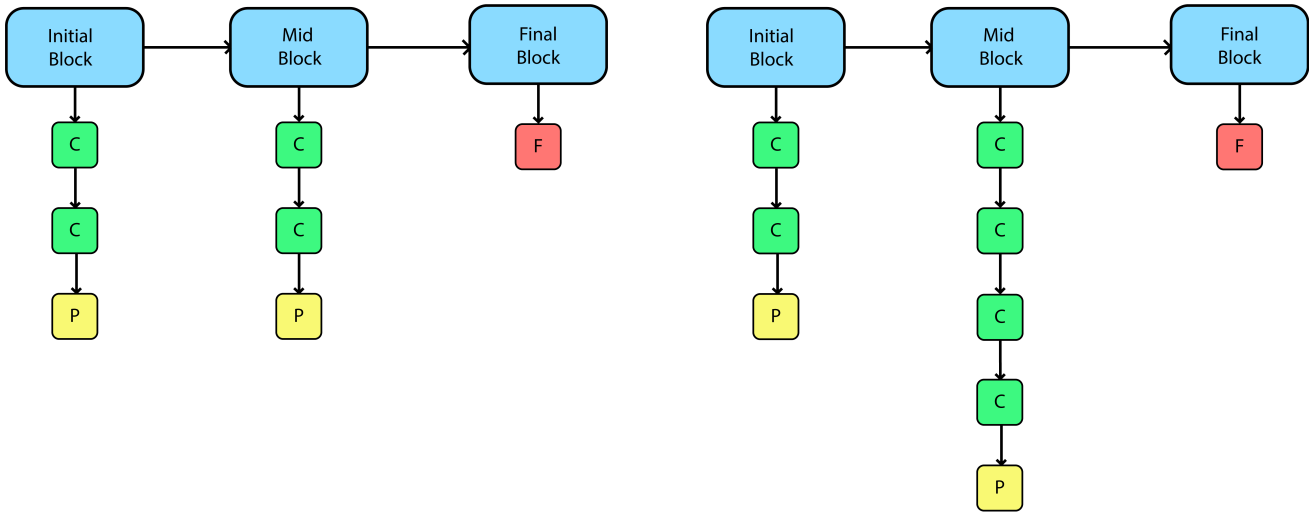


Fig. 12: The diagram on the left represents the VGG-2 model. The diagram on the right represents the resulting CNN model generated by the proposed algorithm from VGG-2. The model on the right is characterized by a greater learning capability thanks to the additional Convolutional layers in the middle block.

TABLE VI: VGG-2 Architecture

Layer	Parameters
Convolutional	feature maps = 32, filter size = 3×3, stride size = 1×1, padding = SAME
Convolutional	feature maps = 32, filter size = 3×3, stride size = 1×1, padding = VALID
Max Pooling	kernel size = 2×2, stride size = 2×2, padding = SAME
Convolutional	feature maps = 64, filter size = 3×3, stride size = 1×1, padding = SAME
Convolutional	feature maps = 64, filter size = 3×3, stride size = 1×1, padding = VALID
Max Pooling	kernel size = 2×2, stride size = 2×2, padding = SAME
Fully-Connected	units = 128

TABLE VII: The Resulting CNN Architecture from VGG-2

Layer	Parameters
Convolutional	feature maps = 32, filter size = 3×3, stride size = 1×1, padding = SAME
Convolutional	feature maps = 64, filter size = 3×3, stride size = 1×1, padding = VALID
Max Pooling	kernel size = 2×2, stride size = 2×2, padding = SAME
Convolutional	feature maps = 64, filter size = 3×3, stride size = 1×1, padding = SAME
Convolutional	feature maps = 32, filter size = 3×3, stride size = 1×1, padding = VALID
Convolutional	feature maps = 32, filter size = 3×3, stride size = 1×1, padding = SAME
Convolutional	feature maps = 16, filter size = 3×3, stride size = 1×1, padding = VALID
Average Pooling	kernel size = 2×2, stride size = 2×2, padding = SAME
Fully-Connected	units = 512

D. VGG-2: Experimental analysis

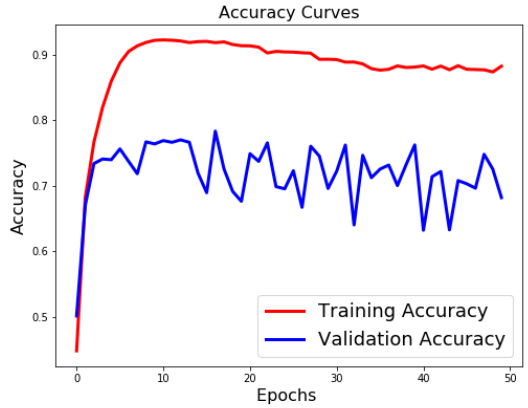
The VGG-2 model achieves an accuracy of 71.08% and a loss of 1.43 on the validation set after being trained for 50 epochs. The accuracy and loss curves are illustrated in Fig. 13.

The resulting CNN model generated by the proposed algorithm achieves an accuracy of 79.86% and a loss of 0.8106 on the validation set after being trained for 50 epochs. The accuracy and loss curves are illustrated in Fig. 14.

In both cases, the validation loss curves show an unstable trend, which suggests that the model, with the chosen parameters, will never converge. This behaviour can be countered by implementing very different techniques, such as adding regularization terms or choosing a different optimizer with distinct parameters. The proposed algorithm counters the aforementioned begaviour by altering the architecture of the model and by tuning the parameters of the layers (i.e., modifying the capacity of the model).

Since the accuracy curves implicitly show the same trend as the loss curves, the same observations stated before still hold.

Fig. 15 illustrates the vacillation of the fitness value and the number of parameters of the best individual of the population (i.e., the one with the lowest fitness). Note that the model returned by the proposed algorithm achieves better results on the validation set, thanks to the higher learning capability of the model, but this traslantes to an increased number of parameters (i.e., higher computational time).



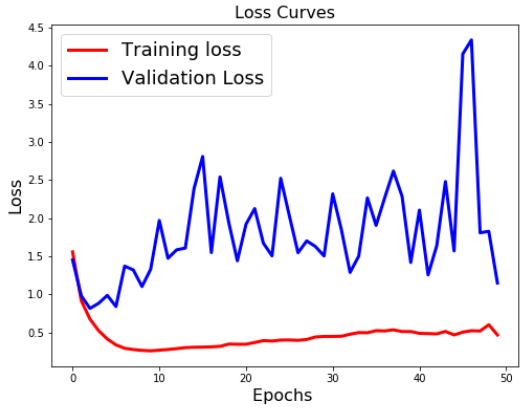
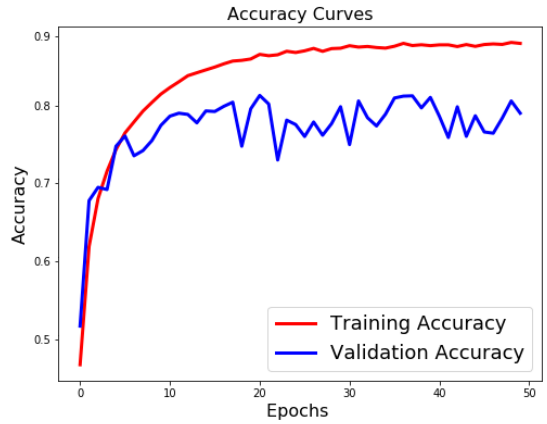


Fig. 13: Accuracy and Loss curves generated by the VGG-2 model after being trained for 50 epochs. The training loss and accuracy are, respectively, 0.5187 and 0.8607, the validation loss and accuracy are, respectively, 1.4304 and 0.7108.



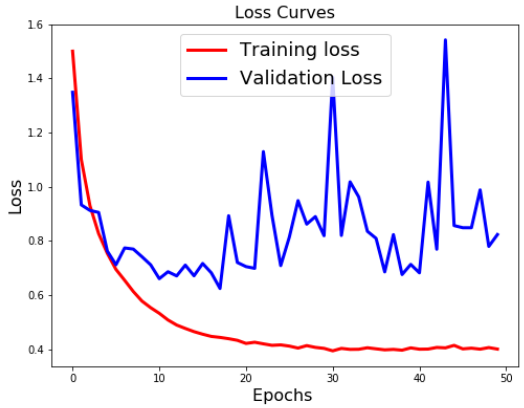
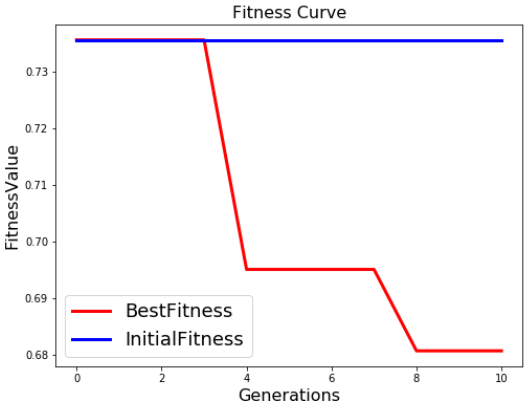


Fig. 14: Accuracy and Loss curves generated by the proposed algorithm on the resulting CNN model. The training loss and accuracy are, respectively, 0.4010 and 0.8962, the validation loss and accuracy are, respectively, 0.8106 and 0.7986.



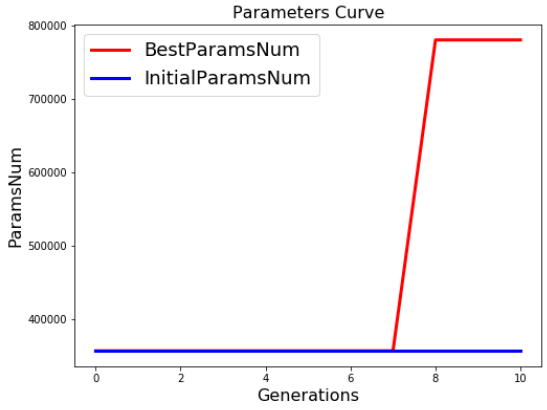


Fig. 15: Vacillation of the fitness value and the number of parameters of the best individual of the population (i.e., the one with the lowest fitness). The fitness value decreases along the generations thanks to the intelligent random search exploited by the proposed algorithm. The number of parameters increases due to the higher learning capability of the found model.

Model	Train Loss	Train Acc	Val Loss	Val Acc	# Params	Time
VGG-1	0.8037	72.31%	1.4923	48.60%	933162	–
Test-1	0.0191	99.58%	3.1979	52.80%	2117962	4082
Test-2	0.0268	99.31%	2.9156	52.60%	933162	2984
Test-3	0.5094	82.13%	1.8984	51.20%	474522	2695
Test-4	0.0341	99.42%	3.6212	48.40%	4216522	4821
Test-5	0.0364	98.98%	2.8467	53.80%	1060138	3617
Test-6	0.0200	99.40%	3.2868	51.60%	3233482	4250
Test-7	0.0380	99.09%	3.7582	48.80%	7413514	5813
Test-8	0.2290	99.49%	3.7564	48.80%	3221994	4177
Test-9	0.0188	98.32%	2.4873	49.00%	6232543	5208
Test-10	0.0189	99.47%	4.3944	51.00%	3727946	4304

TABLE VIII: The model given in input to the proposed algorithm is VGG-1. The algorithm has been run 10 times, and the small-data scenario has been considered. The following parameters have been used for the tests: batch size equal to 32, 15 epochs for training each individual, population size equal to 8, 6 generations, offspring size equal to 6. Due to the randomness of the search, the resulting model produced by each run may be very different from the others. The traning and validation accuracies are expressed in percentual to ease the comparison between the models. Time is espressed in seconds and represents the time required by the proposed algorithm to compute and train the resulting model.

VI. Conclusion

The proposed algorithm represents an intelligent exploitation of a random search. Although randomized, the proposed algorithm is by no means random. Instead, it exploits historical information to direct the search into the region of better performance within the search space.

Over the course of many generations, the algorithm picks out the layers of the CNN architecture. It learns through random exploration and slowly begins to exploit its findings to select higher performing models. It receives the testing accuracy as a means of comparison between architectures and ultimately selects the best architecture. The entire process goes on for many generations until a fully trained suitable CNN model is generated.

The choice of the parameters is crucial for the success of the proposed algorithm. If the parameters are tuned correctly, the resulting CNN has a classification error that is significantly better than the one of the input CNN.

Table VIII reports the results of 10 independent runs of the proposed algorithm, using VGG-1 as initial model. By comparing the validation accuracies of the resulting models with the validation accuracy of VGG-1, it is clear that the proposed algorithm is capable of improving the input CNN at each run. The drawback is the computational time required and the increased amount of parameters of the new model. Moreover, it is observable that the models are overfitting too much the data (i.e., the training accuracy is too high), and this is due to the small-data scenario considered.

Due to the randomness of the proposed algorithm, the resulting CNN models may be very different one from another. As a matter of fact, in Test-1, the proposed algorithm has returned a CNN model which is capable of better learning the classification task, at the cost of a huge amount of additional parameters, whereas in Test-2, the algorithm has return a CNN model which still improves the learning capability of VGG-1, but without affecting the number of parameters.

The goal of this paper was to develop a new evolutionary algorithm to evolve the architecture of a given CNN for an image classification problem. This goal has been successfully achieved by proposing a new encoding scheme of variable-length individuals and a new crossover operator. The proposed algorithm was examined and compared to the state-of-the-art VGG model. The experimental results show that the algorithm significantly improves the VGG architecture in terms of the best classification performance (i.e., the validation accuracy). Therefore, it can be concluded that the proposed algorithm have the potential of evolving successfully CNN architectures.

In this paper, we have investigated the proposed algorithm only on the CIFAR-10 dataset. However, this dataset is small compared to the commonly used datasets for image classification problems. The main bottleneck for the proposed algorithm is the computation of the fitness (i.e. training) of each individual. Since computing the fitness in just one epoch on a largescale dataset would require significant computational resources and a considerable computational time, the proposed fitness evaluation method is not suitable for such scenario (unless a huge amount computational resources are available).

The future work will be aimed at the following implementations: a more efficient fitness evaluation technique, a wider range of mutation for the layers and the parameters (i.e., enlarging the search space to increase the variety of the investigated models), and parallelizing the computation of the offspring at each generation.

REFERENCES

[1] F. Chollet, "Keras cnn example for cifar-10", 2015. Last access Jul. 14, 2019 [Online].
URL: github.com/fchollet/keras/blob/master/examples/cifar10_cnn.py.

[2] F. Chollet, "Keras", 2015. Last access Jul. 14, 2019 [Online].
URL: github.com/fchollet/keras.

[3] L. Davis, "Handbook of genetic algorithms", Taylor & Francis Inc, 1991.

[4] D. Goldberg and J. Holland, "Genetic algorithms and machine learning", Machine Learning, vol. 3, no. 2, pp. 95-99, 1988.

[5] A. Eiben, R. Hinterding, and Z. Michalewicz, "Parameter Control in Evolutionary Algorithms", IEEE Transactions on Evolutionary Computation, vol. 3, no. 2, 1999.

[6] A. Eiben, and S. Smit, "Parameter tuning for configuring and analyzing evolutionary algorithms", Swarm and Evolutionary Computation, vol. 1, pp. 19-31, 2011.

[7] Google, "Google Colab", 2018. Last access Jul. 14, 2019 [Online].
URL: colab.research.google.com.

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

[9] G. Huang, Z. Liu, K. Weinberger, and L. van der Maaten, "Densely connected convolutional networks", Proceedings of 2017 IEEE Conference on Computer Vision and Pattern Recognition, pp. 2261-2269, 2017.

[10] A. Karpathy, "Convolutional Neural Networks for Visual Recognition", 2016.

[11] A. Krizhevsky, "Learning multiple layers of features from tiny images", Technical report, 2009.

[12] A. Krizhevsky, I. Sutskever, and G. Hinton, "Imagenet classification with deep convolutional neural networks", Advances in Neural Information Processing Systems, pp. 1097-1105, 2012.

[13] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition", Proceedings of the IEEE, vol. 86, no. 11, pp. 2278-2324, 1998.

[14] M. Mitchell, "An introduction to genetic algorithms", 1998.

[15] K. Simonyan, and A. Zisserman, "Very deep convolutional networks for large-scale image recognition", Conference paper at International Conference on Learning Representations, 2015.

[16] J. Springenberg, A. Dosovitskiy, T. Brox, and M. Riedmiller, "Striving for Simplicity: The all Convolutional Net", Workshop contribution at International Conference on Learning Representations, 2015.

[17] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, A. Rabinovich et al., "Going deeper with convolutions", in Proceedings of 2015 IEEE Conference on Computer Vision and Pattern Recognition, pp. 1-9, 2015.

[18] Y. Sun, B. Xue, M. Zhang, and G. Yen, "Automatically Designing CNN Architectures Using Genetic Algorithm for Image Classification", 2019.

[19] B. Russell, A. TorralbaKevin, P. Murphy, and W. Freeman, "LabelMe: A Database and Web-Based Tool for Image Annotation", International Journal of Computer Vision, vol. 77, no. 13, pp. 157-173, 2008.

[20] Y. Sun, B. Xue, M. Zhang, and G. Yen, "Evolving Deep Convolutional Neural Networks for Image Classification", 2019.

[21] O. Vinyals, A. Toshev, S. Bengio, and D. Erhan, "Show and tell: A neural image caption generator", Proceedings of the 28th IEEE Conference on Computer Vision and Pattern Recognition, pp. 3156-3164, 2015.

[22] U. Walkarn, "An Intuitive Explanation of Convolutional Neural Networks", Technical report, 2016.

[23] R. Zhang, P. Isola, and A. Efros, "Colorful image colorization", Proceedings of the 14th European Conference on Computer Vision, pp. 649-666, 2016.