Can Differential Evolution Be an Efficient Engine to Optimize Neural Networks?

Marco Baioletti, Gabriele Di Bari, Valentina Poggioni(✉), and Mirco Tracolli

University of Perugia, Perugia, Italy
{marco.baioletti,valentina.poggioni}@unipg.it, dbgabri@gmail.com, mirco.theone@gmail.com

Abstract. In this paper we present an algorithm that optimizes artificial neural networks using Differential Evolution. The evolutionary algorithm is applied according the conventional neuroevolution approach, i.e. to evolve the network weights instead of backpropagation or other optimization methods based on backpropagation. A batch system, similar to that one used in stochastic gradient descent, is adopted to reduce the computation time. Preliminary experimental results are very encouraging because we obtained good performance also in real classification dataset like MNIST, that are usually considered prohibitive for this kind of approach.

1 Introduction

The raise of Deep Learning allowed to Neural Networks (NN) to come back on the crest of a wave since very complex problems have been solved with new architectures and optimization techniques [2,5,15,18]. Moreover this raise has been motivated also by the birth of new computational models using NNs, like Neural Turing Machines [11], Neural Programmer-Interpreters [26] or hybrid models [12].

According to these new trends also neuroevolution has been renewed [4,9, 13,14,23,32]. Several approaches have been proposed both to train the topology and the weigths of the networks. Compared to other neural network learning methods, neuroevolution is highly general, allowing learning without explicit targets, with non differentiable activation functions, and with recurrent networks [10,22]. An interesting analysis on the motivations why backpropagation (and its developments) is still the most used technique to train neural networks and evolutionary approaches are not sufficiently studied is presented in [23]. In that work a simple and efficient method to divide the training set in batches in order to train neural networks with a particular version of Differential Evolution (DE) is presented. Despite the performance are very interesting in terms of accuracy of the predictions the authors do not present experiments with large problems.

The advantages of replacing backpropagation, or other similar methods, with an evolutionary algorithm is clear: the fitness function to be optimized is not required to be differentiable or even continuous.

G. Nicosia et al. (Eds.): MOD 2017, LNCS 10710, pp. 401–413, 2018.
https://doi.org/10.1007/978-3-319-72926-8_33

DE it is a well known evolutionary technique that demonstrated very good performance in several problems [31], has a quick convergence and is robust [25].

The main purpose of this paper is hence to show that DE can be effectively used to train neural networks also in case of quite large problems. We propose an algorithm that uses DE with population elements which encode the weights of the neural network. The system applies a batching system, with restart and elitism for different DE variants and mutation operators.

In order to prove that this mechanism is feasible, we tested it using state of art classification datasets chosen with different numbers of features and different number of records. Although the results are preliminary, they are very encouraging because in all the experiments the system reaches a very good accuracy, always comparable or even better than BPG, also when the network is larger (for instance in MNIST) than the ones presented in literature. Considering also that there is a room for further improvements, we are confident that this idea could be applicable also to larger networks.

The paper is organized as follows. Background knowledge about DE algorithm and neuroevolution are summarized in Sect. 2, related works are presented in Sect. 3, the system is described in Sect. 4 and experimental results are shown in Sect. 5. Conclusions are drawn in Sect. 6 where some ideas for future works are also depicted.

2 Background

2.1 Differential Evolution

Differential evolution (DE) is a metaheuristics that solves an optimization problem by iteratively improving a population of N candidate solutions with respect to a fitness function f. Usually, DE is used to solve continuous optimization problems, where the candidate solutions are numerical vectors of dimension D, but there exist many adaptions to solve combinatorial optimization problems, where the solutions are discrete objects [27]. The population evolution proceeds for a certain number of generations or terminates after a given criterion is met. The initial population can be generated with some strategies, the most used approach is to randomly generate each vector. In each generation, for every population element, a new vector is generated by means of a mutation and a crossover operators. Then, a selection operator is used to choose the vectors in the population for the next generation.

The most important operator used in DE is the *differential mutation*. For each vector x_i in the current generation, called *target vector*, a vector $\bar{y}_i$, called *donor vector*, is obtained as linear combination of some vectors in the population selected according to a given strategy. There exist many variants of the mutation operator (see for instance [7,8]). In our paper we have used DE/rand/1, where

$$\bar{y}_i = x_a + F(x_b - x_c)$$

and DE/current_to_best, where

$$\bar{y}_i = x_i + F(x_{best} - x_i) + F(x_a - x_b)$$

In these formulae, a, b, c are unique random indices different from i, *best* is the index of the best vector in the population and F is a real parameter.

Furthermore we have chosen to implement also DE with Global and Local Neighborhoods (DEGL) [6], indeed it works pretty well in neural networks learning, as explained in [24].

DEGL generates the mutant vector through the combination of two contributors. The first contributor is computed as:

$$L_i = x_i + \alpha(x_{i-best} - x_i) + \beta(x_a - x_b)$$

where x_{i-best} is the individual with best fitness in the neighborhood of target x_i and α, β are two constants with same role of F. The neighborhood of the element x_i contains a fixed number of other population elements, chosen at random.

The second contributor is computed as:

$$G_i = x_i + \alpha(x_{best} - x_i) + \beta(x_a - x_b)$$

where x_{best} is the individual with best fitness in the population. The two contributors are then combined as follow:

$$\bar{y}_i = wG_i + (1 - w)L_i$$

where $w \in [0, 1]$ is the interpolation factor between L_i and G_i.

The crossover operator creates a new vector y_i, called *trial vector*, by recombining the donor with the corresponding target vector by means of a given procedure. The crossover operator used in this paper is the binomial crossover regulated by a real parameter CR.

Finally, the usual selection operator compares each trial vector y_i with the corresponding target vector x_i and keeps the better of them in the population of the next generation.

3 Related Works

The first works applying DE to NN date back to the late '90s and the early 2000s [17,20] where the first applications of DE to train feed-forward NN are presented and analyzed. More recently, several other applications of evolutionary algorithms have been presented in the area of neuroevolution but they are different either for the evolutionary approach used or for the object of evolution [7,8].

In the first case the dominating approach used is the genetic one [10,28,33]. The approach is used also to optimize weights but it is very limited by being a discrete approach so it needs an encoding phase. Several authors proposed a direct representation of the real weights in genes either as a string of real values or as a string of characters, which are then interpreted as real values with a given precision using for example Gray-coded numbers. More adaptive approach has been suggested, for example in [29] or in the more recent [21]. In the first work

the authors proposed a dynamic encoding which depends on the exploration and exploitation phases of the search. In the second one the authors propose a selfadaptive encoding, where the characters of the string are interpreted as a system of particles whose center of mass determines the encoded value. Other approaches have also used a direct encoding that exploits the particular structure of the problem. The methods are not general and cannot be extended to be applicable also to general cases [10]. In [14,16] a floating-point representation of the synaptic weights is used. In these cases the authors use the evolution strategy called CMA-ES for reinforcement learning applied to the pole balancing problem.

Among DE applications to neuroevolution it is worth to cite [9,19,24,32]. These works are different from our approach because they apply the evolution in a different way. In [19] the DE algorithm with a modified best mutation operation is used to enhance the search exploration of a PSO; this PSO is then used to train the NN and the global best value obtained is used as a seed by the BPG. In [9] three different methods (GA, DE and EDA) to evolve neural networks architectures are compared. In particular, the evolutionary methods are implemented to train the architecture of a network with one hidden layer, the learning factor and the seed for the weights initialization. In [24] the author studied the stagnation problem of DE approaches when used to train NN. He proposed to merge the DE with Global and Local neighborhood-based mutation operators algorithm with the Trigonometric mutation operator. In [32] the authors use the Adaptive DE (ADE) algorithm to choose the initial weights and the thresholds of networks. Also in this case the networks are trained by BPG. The authors proved that the system is effective to solve time series forecasting problems.

The paper which have the strongest connection with ours is undoubtely [23], where a Limited Evaluation Evolutionary Algorithm (LEEA) is applied to optimize the weigths of the network. The differences between the two papers are several. First of all, we use DE as evolutionary algorithm, while they employ an ad hoc evolutionary algorithm, similar in some aspects to a genetic algorithm. DE and the other enhancement methods allow our algorithm to train networks much larger than those used in [23]: while we are able to train a feed-forward neural network for MNIST (which has more than 7000 weights), the maximum size handled in [23] is less than 1500 weights. Another difference is the batching system: they use mini-batches which are changed at every generation, while we use larger batches which are changed after a certain number of generations. On the other hand, we use the validation set to compare the networks when the batch is changed, while they use a form of fitness inheritance.

4 The Algorithm

In this section we present our idea of applying Differential Evolution to optimize the weights of the connections in a feed-forward neural network.

Let $\mathcal{P}$ a population of np neural networks with a given fixed topology and fixed activation functions.

Since the DE works with continuous values, we can use a straightforward representation based on a one-to-one mapping between the weights of the neural network and individuals in DE population.

In details, suppose we have a feed-forward neural network with k levels, numbered from 0 to $k-1$. Each network level l is defined by a real valued matrix $\mathbf{W}^{(l)}$ representing the connection weights and by the bias vector $\mathbf{b}^{(l)}$.

Then, each population element x_i is described by a sequence

$$\langle(\hat{\mathbf{W}}^{(i,0)}, \mathbf{b}^{(i,0)}), \ldots, (\hat{\mathbf{W}}^{(i,k-1)}, \mathbf{b}^{(i,k-1)})\rangle,$$

where $\hat{\mathbf{W}}^{(i,l)}$ is the vector obtained by linearization of the matrix $\mathbf{W}^{(i,l)}$, for $l = 0, \ldots, k-1$. For a given population element x_i, we denote by $x_i^{(h)}$ its h–th component, for $h = 0, \ldots, 2k-1$, i.e. $x_i^{(h)} = \hat{\mathbf{W}}^{(i,h/2)}$, if h is even, while $x_i^{(h)} = \mathbf{b}^{(i,(h-1)/2)}$ if h is odd. Note that each component $x_i^{(h)}$ of a solution x_i is a vector whose size depends on the number of neurons of the associated levels.

The population elements are evolved by applying mutation and crossover operators in a componentwise way. For instance, the mutation rand/1 for the element x_i is applied in the following way: three indices a, b, c are randomly chosen in the set $\{1, \ldots, np\} \setminus \{i\}$ without repetition; then, the h–th component $\bar{y}_i^{(h)}$ of the donor element $\bar{y}_i$ is obtained as the linear combination

$$\bar{y}_i^{(h)} = x_a^{(h)} + F(x_b^{(h)} - x_c^{(h)})$$

for $h = 0, \ldots, 2L-1$.

The evaluation of a population element in the selection operator is performed by computing the cross–entropy of the corresponding neural network. The optimization problem is then to find the neural network with the minimum cross–entropy value.

Anyway, this computation is the most time consuming operation in the overall algorithm and it will lead to unacceptable computation time if the cross–entropy considers the whole dataset. For this motivation we have decided to follow a batching method similar to the one proposed in [23].

The dataset D is split in three different sets: a training set TS used for the training phase, a validation set VS used for a uniform evaluation of the individuals selected at the end of each training phase, and a test set ES used to evaluate the performance of the best neural network.

Then, the training set TS is randomly partitioned in K batches of size B. This phase is very important because the records in each batch should follow more or less the same distribution as in TS. Otherwise, the risk is to train specialized networks without generalization ability.

At each generation the population is evaluated against only a limited number of training examples given by the size of the current batch, instead of evaluating the population against the whole training set. This allows to reduce the computational load, particularly on large training sets.

The fitness function used is then

$$H_{z'}(z) = -\sum_{i=1}^{B}\sum_{j=1}^{C} z'_{ij}\log(z_{ij})$$

where z'_{ij} and z_{ij} are respectively the predicted value and the true value for the classification of i-th record in the batch with respect to the j-th class and C is the number of classes.

The batch is changed after s generations (called *epoch*), so that the evolution has enough time to learn from the batch. If the algorithm is required to continue for more than K epochs, the batches are reused in a cyclic way, i.e. after the last batch, the first batch will be used again, and so on.

Since the fitness function depends also on the batch and we need a fixed way to compare the elements, at the end of every epoch the best neural network $best\_net_e$ of the epoch e is selected as the neural network in $\mathcal{P}$ which reaches the highest accuracy in the validation set VS. The best neural network $best\_net_{global}$ found so far is then eventually updated.

At the beginning of each epoch, the fitness of every element in $\mathcal{P}$ is re-evaluated by computing the cross-entropy on the new batch.

To avoid a premature convergence of the algorithm, a reset method is applied, i.e. discard all the current population, except the best element, and continue with

Algorithm 1. The algorithm DENN

```
Initialize the population;
Extract the K = TS/B batches batch_0, ..., batch_{K-1};
h ← 0;
for e ← 1 to tot_gen/s do
    Set the current batch as batch_{e mod K};
    Re-evaluate all the elements (x_1, ..., x_np);
    for g ← 1 to s do
        for i ← 1 to np do
            y_i ← generate_offspring(x_i)
        for i ← 1 to np do
            if y_i is better than x_i in terms of H then
                x_i ← y_i
    best_net_e, best_score_e ← best_score(x_1, ..., x_np);
    Update best_net_global, best_score_global;
    if best_net_global is not changed then
        if h > counter then
            Reset the population;
            h ← 0;
        else
            h ← h + 1
return best_net_global;
```

a new randomly generated population. The reset is performed at the end of each epoch e, if the the score of $best\_net_{global}$ has remained unchanged for a certain number *counter* of epochs.

The DE parameters F and CR have a great impact on the evolution and their values are not easy to be chosen. Therefore, we have decided to adopt the auto–adaptive scheme jDE [3]. This method evolves the values of these two parameters by a process which is strictly related to the selection operator of DE. In this way, the algorithm is able to dynamically select the best values of F and CR for the problem. The complete algorithm, called DENN, is depicted in Algorithm 1.

In the algorithm DENN, the function *generate_offspring* computes the mutation and the crossover operator in order to produce the trial element, while the function *best_score* returns the best network and its score among all the elements in the population.

5 Experimental Results

The main objective of these experiments is to assess the effectiveness of DE algorithm as an alternative to backpropagation, and other similar methods, for neural network optimization also in the case of quite large problems. The size of NNs handled in this paper are larger than those used in the previous works presented in literature. We run two kinds of experiments in order to (i) evaluate which combination of DE variant and mutation operator performs better and (ii) study which setting of algorithm parameters can provide the best results, also considering the computational effort. DENN has been implemented both as a TensorFlow plugin written in C++ and Python and as a stand-alone C++ program[1].

5.1 Datasets

We decided to test the system on recent classification datasets downloaded by the UCI repository[2], and on the well known MNIST[3] dataset. MAGIC, QSAR and GAS are datasets for classification problems that have been chosen because they differ for the number of features and records and therefore are well suited to assess the scalability of the system. Finally, we decided to test the system on the MNIST dataset because it is a classical challenge with well known results obtained by NN classification systems. Moreover, it is considered an interested challenge also in [23].

- MAGIC Gamma telescope: dataset with 10 features, 2 classes and 19020 records.
- QSAR biodegradation: dataset with 41 features, 2 classes and 1055 records.

[1] Source code available at https://github.com/Gabriele91/DENN.
[2] https://archive.ics.uci.edu/ml/datasets.
[3] http://yann.lecun.com/exdb/mnist/.

– GAS Sensor Array Drift: dataset with 128 features, 6 classes and 13910 records.
– MNIST: dataset with 784 features, 10 classes and 70000 records.

MAGIC, QSAR and GAS have been split in this way: the training set is composed of 80% of the records, the validation set is composed of another 10% of the records and the remaining 10% records are in the test set.

The MNIST dataset is already provided as a pair with separated training and test sets (TS, ES). Then we extracted the validation set VS from the training set by a uniform random sampling that preserves the distribution over the classes. Since a too small validation set could have a negative impact on the performance in term of accuracy, we chose, as in the other datsets, $|VS| = |TS| \cdot 10\%$.

5.2 Results

First of all we have analyzed the data in order to understand which are the parameters yielding to the best performance. The system depends from several parameters: some deriving from the use of DE (np, F, CR, the DE-variant and the mutation and crossover operators), other depending from the batching system (B, s) or from the application of the reset mechanism (*counter*).

A systematic battery of test has been run in order to study all the parameters. Due to the space limits, in this work just some data and graphics can be discussed. Other preliminary experiments are discussed in [1,30].

During this experimental phase, we have noted that the most important choice is selecting the DE variant (classical or jDE) and the mutation operator (*rand/1*, *current-to-best*, *DEGL*). Increasing other parameters, like np or B, can have a positive impact on the algorithm performance only when these values are below a certain threshold. When this threshold is overcome, either the computational time becomes too high or the results do not improve. This fact also agrees with *(i)* other traditional results on DE that in general suggest large (but not too large) populations and *(ii)* our initial idea about the batch size, according to which a trade-off between batch size and computational effort is necessary.

The batch size has been chosen to be proportional to the number of classes, in order to have, on average, a given number of examples for each class. Moreover we found that B influnces also the number of population elements np, and the steps s spent on the same batch to train the network. This is unsurprising because with a larger number of records in a batch, the population size (and the number of steps) should be larger as well.

In Fig. 1, for all the datasets analyzed in this work, the accuracy values of a NN without hidden layer trained with different settings of the algorithm are plotted. We compare the six different combinations of the DE variants and mutation operators (*DE+rand/1*; *DE+current_to_best*; *DE+degl*; *jDE+rand/1*; *jDE+current_to_best*; *jDE+DEGL*). Moreover, also the accuracy values obtained by the same NN trained with BPG (*GD*) are reported. The accuracy values plotted in the graphics are computed: (i) training the neural network for a given

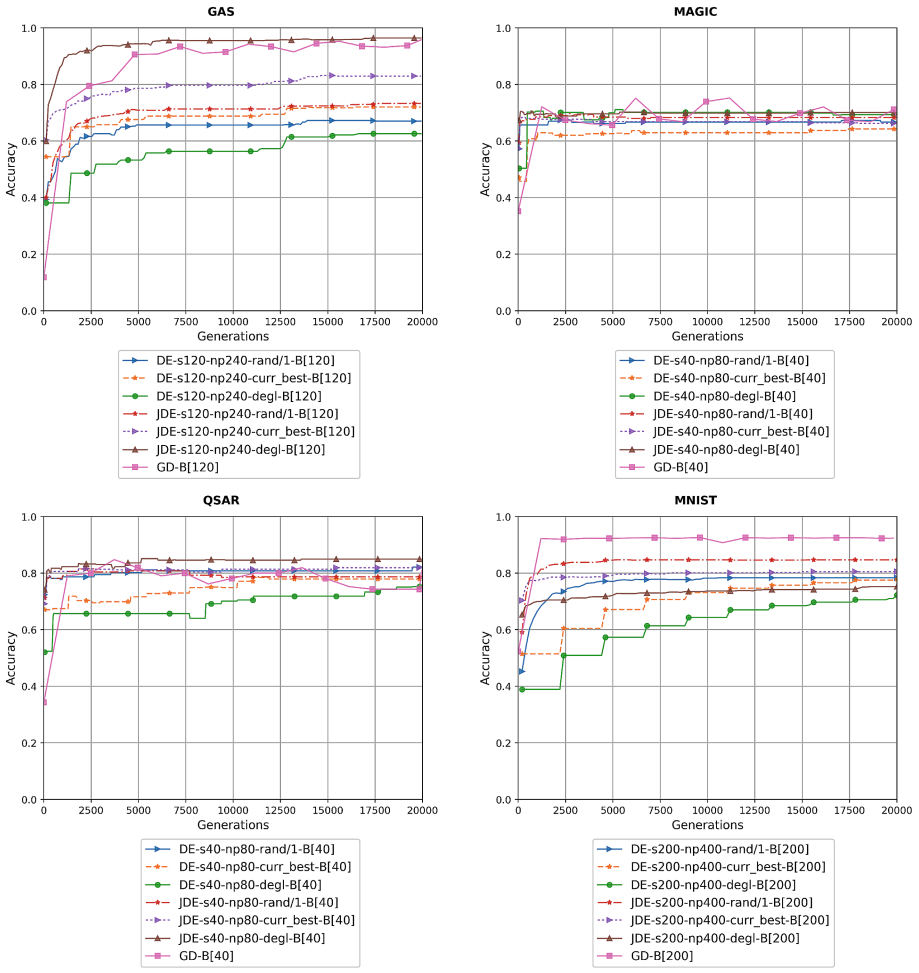

Fig. 1. Comparisons in term of accuracy among the different combinations DEvariant-mutationType on GAS, MAGIC, MNIST and QSAR datasets

number k of generations (on the x-axis from 1 to 20000) and (*ii*) running, on the test set, the NN that obtained the best evaluation on the validation set.

The values of the other parameters are: batch size $B = 20C$, where C is the number of classes in the dataset, population size $np = 2B$, number of training steps in the same batch $s = B$ and $counter = 10$. The values of $F = 0.5$ and $CR = 0.9$ are used only when the classical DE is applied. These data have been chosen after an extended experimental phase, partially showed in [1,30]. From these experiments we noted, for example, that fixing B and setting $s = B$ and $np = 2B$ we can obtain, on average, good results.

From the data plotted in the graphics in Fig. 1 we can conclude that in the most cases jDE performs better than standard DE. Both in the largest datasets

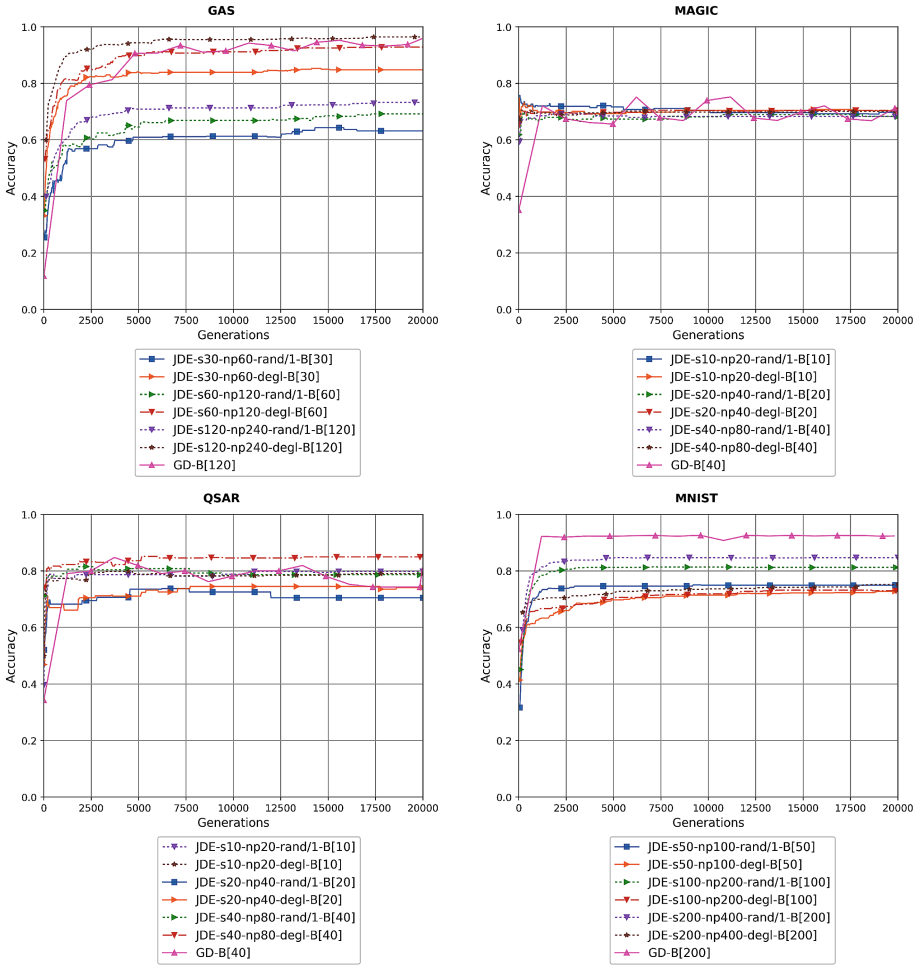

Fig. 2. Comparisons in term of accuracy among different setting of s (and consequently of B and np) on GAS, MAGIC, MNIST and QSAR datasets

(GAS and MNIST) and QSAR the differences are sharp, while in the case of MAGIC dataset the differences are so narrow that it is impossible to distinguish the best algorithm.

While the "winner" variant is undisputable (at least for these experiments), the same is not for the mutation operator: in the GAS dataset the difference between the combination *DE+DEGL* and the second performing *jDE+current_to_best* is clear, but in the MNIST dataset the best performing combination is *jDE+rand/1*, with *DE+DEGL* only in 4th/5th position.

In Fig. 2 data on accuracy for different settings of B, s and np are plotted. In these graphics we can compare different values for the batch size B, set respectively to $B = 5C$, $B = 10C$, $B = 20C$, where C is the number of the classes.

The right setting of B is determinant for our algorithm because a too small value for B does not allow to reach good performance, while a too high value can increase too much the execution time and moreover can cause overfitting.

From the plots we can see that, excluding the cases where the differences are not significant, the best values are obtained with the highest values of B, both for the *rand/1* and *degl* mutation.

6 Conclusions and Future Works

In this paper we presented an algorithm based on Differential Evolution to train the weights of a neural network. This algorithm can be an effective alternative to the backpropagation method because of its intrinsic advantages deriving from the use of an evolutionary algorithm. The experiments presented show how the system is able to solve classification problems also in case of large image datasets, like MNIST, reaching satisfying accuracy very close to the state of the art. These results are very encouraging considering that the algorithm and the implementation could be improved and other enhancements are already under investigation.

The proposed approach allows also to handle computational models based on neural networks which do not need to be fully differentiable and this can lead to simpler models.

Future works include: the implementation of other DE variants and mutation/crossover operators; the application of the system both to other kind of problems like numerical estimation and to larger problems; the application to other computational models based on neural networks like Neural Turing Machines.

References

1. Bari, G.D.: Denn: Differential evolution for neural networks. Master thesis (2017)
2. Bengio, Y., Goodfellow, I.J., Courville, A.: Deep learning. Nature **521**, 436–444 (2015)
3. Brest, J., Boskovic, B., Mernik, M., Zumer, V.: Self-adapting control parameters in differential evolution: a comparative study on numerical benchmark problems. IEEE Trans. Evol. Comput. **10**(6), 646–657 (2006)
4. Cardamone, L., Loiacono, D., Lanzi, P.L.: Evolving competitive car controllers for racing games with neuroevolution. In: Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation, pp. 1179–1186. ACM (2009)
5. Collobert, R., Weston, J.: A unified architecture for natural language processing: deep neural networks with multitask learning. In: Proceedings of the 25th International Conference on Machine Learning, ICML 2008, pp. 160–167. ACM, New York (2008)
6. Das, S., Abraham, A., Chakraborty, U.K., Konar, A.: Differential evolution using a neighborhood-based mutation operator. IEEE Trans. Evol. Comput. **13**(3), 526–553 (2009)

7. Das, S., Mullick, S.S., Suganthan, P.: Recent advances in differential evolution an updated survey. Swarm Evol. Comput. **27**, 1–30 (2016)
8. Das, S., Suganthan, P.N.: Differential evolution: a survey of the state-of-the-art. IEEE Trans. Evol. Comput. **15**(1), 4–31 (2011)
9. Donate, J.P., Li, X., Sánchez, G.G., de Miguel, A.S.: Time series forecasting by evolving artificial neural networks with genetic algorithms, differential evolution and estimation of distribution algorithm. Neural Comput. Appl. **22**(1), 11–20 (2013)
10. Floreano, D., Dürr, P., Mattiussi, C.: Neuroevolution: from architectures to learning. Evol. Intell. **1**(1), 47–62 (2008)
11. Graves, A., Wayne, G., Danihelka, I.: Neural turing machines. arXiv preprint arXiv:1410.5401 (2014)
12. Graves, A., Wayne, G., Reynolds, M., Harley, T., Danihelka, I., Grabska-Barwińska, A., Colmenarejo, S.G., Grefenstette, E., Ramalho, T., Agapiou, J., et al.: Hybrid computing using a neural network with dynamic external memory. Nature **538**(7626), 471–476 (2016)
13. Hausknecht, M., Lehman, J., Miikkulainen, R., Stone, P.: A neuroevolution approach to general atari game playing. IEEE Trans. Comput. Intell. AI Games **6**(4), 355–366 (2014)
14. Heidrich-Meisner, V., Igel, C.: Neuroevolution strategies for episodic reinforcement learning. J. Algorithms **64**(4), 152–168 (2009)
15. Hinton, G., Deng, L., Yu, D., Dahl, G.E., Mohamed, A.R., Jaitly, N., Senior, A., Vanhoucke, V., Nguyen, P., Sainath, T.N., Kingsbury, B.: Deep neural networks for acoustic modeling in speech recognition: the shared views of four research groups. IEEE Signal Process. Mag. **29**(6), 82–97 (2012)
16. Igel, C.: Neuroevolution for reinforcement learning using evolution strategies. In: The 2003 Congress on Evolutionary Computation, 2003. CEC 2003, vol. 4, pp. 2588–2595 (2003)
17. Ilonen, J., Kamarainen, J.K., Lampinen, J.: Differential evolution training algorithm for feed-forward neural networks. Neural Process. Lett. **17**(1), 93–105 (2003)
18. Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. In: Pereira, F., Burges, C.J.C., Bottou, L., Weinberger, K.Q. (eds.) Advances in Neural Information Processing Systems 25, pp. 1097–1105. Curran Associates Inc. (2012)
19. Leema, N., Nehemiah, H.K., Kannan, A.: Neural network classifier optimization using differential evolution with global information and back propagation algorithm for clinical datasets. Appl. Soft Comput. **49**, 834–844 (2016). http://www.sciencedirect.com/science/article/pii/S1568494616303866
20. Masters, T., Land, W.: A new training algorithm for the general regression neural network. In: 1997 IEEE International Conference on Systems, Man, and Cybernetics, vol. 3, pp. 1990–1994 (1997)
21. Mattiussi, C., Dürr, P., Floreano, D.: Center of mass encoding: a self-adaptive representation with adjustable redundancy for real-valued parameters. In: Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation, GECCO 2007, pp. 1304–1311. ACM, New York (2007)
22. Miikkulainen, R.: Neuroevolution, pp. 716–720. Springer, Boston (2010). https://doi.org/10.1007/978-0-387-30164-8_589
23. Morse, G., Stanley, K.O.: Simple evolutionary optimization can rival stochastic gradient descent in neural networks. In: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO) 2016, pp. 477–484. ACM, New York (2016)

24. Piotrowski, A.P.: Differential evolution algorithms applied to neural network training suffer from stagnation. Appl. Soft Comput. **21**, 382–406 (2014)
25. Price, K., Storn, R.M., Lampinen, J.A.: Differential Evolution: A Practical Approach to Global Optimization. Springer, Heidelberg (2006). https://doi.org/10.1007/3-540-31306-0
26. Reed, S., de Freitas, N.: Neural programmer-interpreters. Technical report, arXiv:1511.06279 (2015). http://arxiv.org/abs/1511.06279
27. Santucci, V., Baioletti, M., Milani, A.: Algebraic differential evolution algorithm for the permutation flowshop scheduling problem with total flowtime criterion. IEEE Trans. Evol. Comput. **20**(5), 682–694 (2016)
28. Schaffer, J.D., Whitley, D., Eshelman, L.J.: Combinations of genetic algorithms and neural networks: a survey of the state of the art. In: Proceedings of COGANN 1992: International Workshop on Combinations of Genetic Algorithms and Neural Networks, pp. 1–37 (1992)
29. Schraudolph, N.N., Belew, R.K.: Dynamic parameter encoding for genetic algorithms. Mach. Learn. **9**(1), 9–21 (1992)
30. Tracolli, M.: Enhancing denn with adaboost and self adaptation. Master thesis (2017)
31. Vesterstrom, J., Thomsen, R.: A comparative study of differential evolution, particle swarm optimization, and evolutionary algorithms on numerical benchmark problems. In: Proceedings of the 2004 Congress on Evolutionary Computation (IEEE Cat. No.04TH8753), vol. 2, pp. 1980–1987 (2004)
32. Wang, L., Zeng, Y., Chen, T.: Back propagation neural network with adaptive differential evolution algorithm for time series forecasting. Expert Syst. Appl. **42**(2), 855–863 (2015)
33. Yao, X.: Evolving artificial neural networks. Proc. IEEE **87**(9), 1423–1447 (1999)