Adversarial Perturbations for Evolutionary Optimization

Unai Garciarena, Jon Vadillo, Alexander Mendiburu, and Roberto Santana(✉)

Intelligent Systems Group, University of the Basque Country, San Sebastian, Spain
roberto.santana@ehu.eus

Abstract. Sampling methods are a critical step for model-based evolutionary algorithms, their goal being the generation of new and promising individuals based on the information provided by the model. Adversarial perturbations have been proposed as a way to create samples that deceive neural networks. In this paper we introduce the idea of creating adversarial perturbations that correspond to promising solutions of the search space. A surrogate neural network is "fooled" by an adversarial perturbation algorithm until it produces solutions that are likely to be of higher fitness than the present ones. Using a benchmark of functions with varying levels of difficulty, we investigate the performance of a number of adversarial perturbation techniques as sampling methods. The paper also proposes a technique to enhance the effect that adversarial perturbations produce in the network. While adversarial perturbations on their own are not able to produce evolutionary algorithms that compete with state of the art methods, they provide a novel and promising way to combine local optimizers with evolutionary algorithms.

Keywords: Adversarial perturbations · Model-based EAs · Neural networks · Sampling methods · EDAs

1 Introduction

For years, there has been an active cross-fertilization between the fields of neural networks (NNs) and evolutionary algorithms (EAs). While evolutionary methods are the most used algorithms for neural architecture search, neural networks have also been used to guide the search in model-based EAs [1]. One key question in the design of NN-EA approaches is how to combine the excellent efficiency of gradient based optimizers with the power of evolutionary operators to conduct global search. In this paper we investigate the effect that methods conceived for extracting information of NN have in the behavior of EAs. In particular, we focus on the study of adversarial perturbations applied to surrogates of fitness functions.

Given a machine learning model that classifies examples, an adversarial attack [20] aims to make imperceptible changes to the examples in such a way

G. Nicosia et al. (Eds.): LOD 2021, LNCS 13164, pp. 408–422, 2022.
https://doi.org/10.1007/978-3-030-95470-3_31

that they are incorrectly classified by the model. The effectiveness of the adversarial perturbation added to the example is measured both in terms of its capacity to fool the model and in terms of the amount of distortion required to make the adversarial attack successful. In this paper we investigate the use of adversarial perturbations as sampling method and model enhancer within an evolutionary algorithm. Using solutions already evaluated, we train a NN to classify whether a solution will have a high-fitness or a low-fitness. Then we apply adversarial perturbations to low-fitness solutions until the classifier is "fooled" to classify the solution as high-fitness. The assumption made is that, since the NN model captures information about the features that make solutions poor or good, the perturbations made to the solutions will indeed improve the fitness of solutions. Therefore, in order to make the network classify a poor solution as good, the adversarial perturbation will actually improve the solution in terms of the objective function being optimized.

Our work is in line with ongoing research on the use of different types of models to learn the most characteristic patterns of promising solutions in evolutionary algorithms [1,3,16]. There exists an apparent paradox between the high accuracy that a neural network can achieve at the time of capturing the characteristics of the data and the difficulty of effectively exploiting that information at the time of generating new solutions. None of the current approaches provides a general, satisfactory and efficient solution to sampling from neural networks within the evolutionary search scenario. Therefore, this paper introduces adversarial perturbations as a completely new approach for exploiting this information. This is a promising research direction because there is an active research on the theoretical basics of adversarial perturbations and a variety of approaches have been introduced with this goal.

The goal of our paper is not to introduce a new state of the art evolutionary algorithm for numerical optimization. Our objective is to analyze whether adversarial perturbations can be used as a new way to exploit the information learned by models of the best solutions opening the possibility of creating new hybrid optimization methods. We compare 15 different strategies used to create adversarial examples, most of which use gradient information about the model to create the perturbations. We identify those strategies that have a better performance within the context of evolutionary algorithms, and investigate whether gradient optimizers produce any advantage over black-box attacks (those that do not exploit information about the models to create the perturbations). Our preliminary results do not show a significant improvement in efficiency due to the use of adversarial perturbations, the results of the adversarial perturbation methods vary significantly depending on the problem. However, we identify methods that exploit the gradients of the models as the most effective.

The rest of this paper is organized as follows. In the following section we present adversarial examples and adversarial perturbations with a focus on those methods that will be later investigated in more detail. Section 3 introduces the elements of our proposal to use adversarial perturbation as part of the evolutionary process. Section 4 describes the algorithm and analyzes different aspects

of the interaction between the EA, the NN and the technique for creating the adversarial perturbations that need to be taken into account for enhancing the search. Related work is reviewed in Sect. 5. Section 6 presents the experiments conceived to evaluate the performance of the different variants of adversarial attacks. Finally, conclusions drawn from the whole process are summarized in Sect. 7.

2 Adversarial Examples and Adversarial Perturbations

In this section we present a brief introduction of adversarial examples and adversarial perturbations. Our introduction is focused on the concepts and methods used in this paper. For a more detailed introduction to adversarial examples, we encourage the reader to examine [23].

Adversarial examples are inputs deliberately perturbed in order to produce a wrong response in a target deep neural network (DNN), while keeping the perturbation hardly detectable. The existence of slightly perturbed inputs able to fool state-of-the-art DNNs were first reported by [20], for image classification tasks.

Adversarial attacks can be classified according to different characteristics [23]. First of all, we denominate *targeted* adversarial examples to those inputs modified in order to produce a particular output class, or *untargeted* adversarial examples if the objective is to change the output of the model without fixing the output class we want to obtain. Moreover, we consider that an attack is *individual* if it has effect on just one input, or *universal* if it is able to fool the model for a large proportion of input samples. Regarding the information that an adversary has access to from the target model we can also differentiate between *white-box* or *black-box* attacks. In the former, which is the most common attack type, the adversary has full access to the model, including its weights, logits or gradients. In the latter, contrarily, we assume that it is not possible to obtain any information about the target DNN.

Goodfellow et al. [5] introduce the Fast Gradient Sign Method (FGSM) attack, in which the input x of ground-truth class y is perturbed in a single-step, in the same direction of the gradients of the loss function with respect to x:

$$x' \leftarrow x + \epsilon \text{sign}(\bigtriangledown \mathcal{L}(\theta, x, y)), \tag{1}$$

where $\mathcal{L}$ represents the loss function, θ the parameters of the target model and ϵ the maximum distortion allowed for each value in x. A straightforward extension of the FGSM approach is to perform more than one step, which is known as Basic Iterative Method (BIM) [8]. The *Momentum Iterative Attack* (MI) [2] is a further extension of this attack strategy, in which the momentum [14] of the gradients are considered in order to achieve a more effective attack. Thus, at each step the input sample is perturbed according to the following update rule:

$$x_t \leftarrow x_{t-1} + \epsilon \text{sign}(g_t), \tag{2}$$

where g_t represents the accumulated gradients until step t, with a decay factor μ:

$$g_t \leftarrow \mu \cdot g_{t-1} + \frac{\bigtriangledown \mathcal{L}(\theta, x_{t-1}, y)}{|| \bigtriangledown \mathcal{L}(\theta, x_{t-1}, y)||_1} \tag{3}$$

The Decoupled Direction and Norm attack [17], which also relies on iteratively perturbing the input sample in the direction determined by the gradients of the loss function, provides a different strategy in order to minimize the distortion amount of the perturbation. At each step t, the perturbation δ is updated with a step size α:

$$\delta_t \leftarrow \delta_{t-1} + \alpha \frac{\bigtriangledown \mathcal{L}(\theta, x_{t-1}, y)}{|| \bigtriangledown \mathcal{L}(\theta, x_{t-1}, y)||_2}, \tag{4}$$

and it is projected in the sphere of radius ϵ and centered in x, which constrains the ℓ_2 norm. However, at each step the value of ϵ is decreased by a factor of μ if x_{t-1} is able to fool the model or increased if it is not. In this way, after many updates,

$$x_t \leftarrow x + \epsilon_t \frac{\delta_t}{||\delta_t||_2}, \tag{5}$$

the solution is expected to converge to a valid adversarial example, while minimizing the ℓ_2 norm of the perturbation.

The attack rationale of the DeepFool algorithm [12], another iterative gradient-based approach, is to push an input sample x of class y towards the closest decision boundary of the target model.

$$r* = arg\ min\ ||r||_2 \text{ s.t. } f(x+r) \neq f(x). \tag{6}$$

To approximate $r*$, at step t, the decision boundaries of the model are linearly approximated in the vicinity of x, and for each $k \neq y$, the perturbation needed to reach (under the linear approximation) the decision boundary corresponding to the k-th class is determined by:

$$\delta_t^k \leftarrow \frac{f_k(x_t) - f_y(x_t)}{|| \bigtriangledown f_k(x_t) - \bigtriangledown f_y(x_t)||_2^2} (\bigtriangledown f_k(x_t) - \bigtriangledown f_y(x_t)) \tag{7}$$

being f_k the logits of f corresponding to the k-th class. Finally, the input is pushed towards that direction δ_t^* that requires the minimum distance: $x_{t+1} \leftarrow x_t + \delta_t^*$. This is done until the input finally reaches another decision region, that is, until the output of the model is changed.

Finally, the last method we focus on, the *Blend Uniform Noise Attack* method, relies on the addition of uniform noise to the input sample until it is able to produce a wrong prediction in the target model. Although this strategy may require a higher distortion amount to fool the model than the previous approaches, it can be suitable for black-box scenarios.

3 Adversarial Examples as Promising Solutions

In this section we define the components of the model that eventually is used for generating individuals for, and thus guide, the evolutionary algorithm. The three main components interacting in the whole evolutionary process are:

1. Sub-populations (S): Solutions grouped in classes.
2. Model (M): A supervised (neural network) classifier to classify a solutions.
3. Attack (A): A method for creating adversarial perturbations using the model.

3.1 Learning to Discriminate the Quality of the Solutions

The first step of the algorithm is to group the solutions into three groups according to their fitness values. Solutions are sorted according to their fitness and divided into three groups of similar size $\left(\frac{N}{3}\right)$. These groups are called: *Best*, *Middle*, and *Worst* solutions.

We could also split the population into two parts and consider only *Best* and *Worst* solutions. However, by creating an additional class grouping solutions with an intermediate value of the fitness, we intend to emphasize the difference between the groups comprising the best and worst performing solutions.

We define a prediction task that consists of, given a solution, determine whether it belongs to the *Best* or *Worst* groups (in this part of the procedure, the *Middle* set is ignored). This is a typical binary classification problem where the predictor variables or features are the decision variables of the optimization problem and the binary label can be interpreted as: 1) The solution belongs to the class of *Best* solutions. 0) The solution belongs to the class of *Worst* solutions.

Due to their capacity to fit any problem type, the model chosen for testing the adversarial methods is a multi-layer perceptron (MLP) [13]. In an MLP, every neuron in layer $l-1$ is exclusively connected to every other neuron in the next layer l. These dense connections can be represented as matrix operations:

$$n_l = \sigma_l(w_l \times n_{l-1} + b_l) \tag{8}$$

where, n_l represents data computed in the l-th layer, and $w_l, b_l \in \theta$ are the parameters for that layer. The first layer takes input data ($n_0 = x$), and for the output layer $n_l = \hat{y}$. Commonly, the outputs of the layers are activated by non-linear functions, in this equation represented by σ.

The neural network classifier is learned from scratch each generation using the individuals of the *Best* and *Worst* datasets from the previous generation. We also considered the possibility of starting the learning process from the network weights learned in the previous generation, but this approach led to a decreased ability of the network to learn from new solutions. Learning a new neural network in each generation can increase the computational load for problems with many variables. To cope with this question, starting the learning process from random weights could be triggered only at certain generations during the evolution.

3.2 Generating Promising Solutions with Adversarial Attacks

If the model has been successfully trained and it is able to generalize to unseen data, it will predict with a high accuracy whether solutions are good or poor. Therefore, we can start from a poor solution, which is correctly classified by the model as belonging to the class *Worst* and make adversarial perturbations on it until the model classifies it as belonging to the class *Best*. The assumption here is that the perturbations required for the model to be fooled are indeed perturbations that improve the quality of the solution in terms of the fitness.

Instead of using a low-fitness solution from the previous population, it is also possible to use a random solution, or a solution from the *Middle* class, as long as the model initially predicts it as belonging to the *Worst* class, because otherwise it will not be able to create an adversarial perturbation since the model can be fooled only if it initially predicts the class correctly.

3.3 Weaker Models Make More Adversarial Examples

As discussed in the previous section, it is not possible to improve solutions that belong to the *Best* class and are correctly classified by the model as such. This is a drawback because it means that the model can not further improve the best found solutions.

One partial remedy for this problem is, once the model has been trained, to partially modify it in order to make it incorrectly predict at least some of the good solutions as poor, but keeping the prediction of the poor solutions correct. Such modification will increase the pool of solutions to which adversarial perturbations can be applied, and will likely increase the gain in fitness when solutions in the *Best* class can be perturbed multiple times.

However, we would like the modifications made to the network not to distort the relevant information that it captures about the features that differentiate between good and poor solutions. Therefore, we constrain the modifications made to the network to the weights that connect the last hidden layer with the output layer. These weights can be represented by a matrix of dimension $m \times 2$, where m is the number of units in the last hidden layer. The two units in the output layer correspond to the values generated for the two classes.

Since in the output layer a `softmax` function is applied, and the output of the function is proportional to the two inputs values, we can expect that if we increase the weights that are connected to the unit of the *Worst* class then there will be more examples predicted as *Worst*. That is the type of modification made to the network. Therefore, we increase the weights connected to this unit in a parsimonious way until at least one of the good solutions used for training is incorrectly classified as *Worst*, or a maximum number of trials is reached.

This modification to the network can be beneficial not only because it increases the number of solutions to which adversarial perturbation could be applied to, but also because it can determine an increment in the amount of perturbation added to all solutions modified. The assumption is that, in order to reverse the classification given by the modified network to examples from the

Worst class, the adversarial perturbation method will require to add a higher amount of perturbation to the solution. Otherwise, it will not be able to balance the effect that the change of weights in the last connected layer of the modified network produced in the classification.

4 Surrogate Assisted EA with Adversarial Sampling

In model-based evolutionary algorithms (MEAs) [18], a model is used to discover latent dependencies between variables. In this type of algorithms, the synergism between the model and the method for sampling is one of the keys of the correct optimization workflow. Algorithm 1 shows a basic common MEA in pseudocode form.

Algorithm 1: Pseudo-code for a generic MEA.

```
pop = generate_population();
while halting condition is not met do
    fitness = evaluate_population(pop);
    selected_pop, selected_fit = select_solutions(pop, fitness);
    model = create_model(selected_pop, selected_fit);
    offs = sample_model(model);
    pop = offs + best(pop, fitness);
end
```

Several model types can be used to guide a MEA. A common choice are probably probabilistic graphical models, among which some of the most popular choices are Bayesian or Gaussian networks. MEAs driven by these models are known as estimation of distribution algorithms (EDAs). Nevertheless, a number of researchers [11,15,16] have also proposed the usage of different neural networks for guiding the search in EAs.

When inserted as part of Algorithm 1, adversarial perturbations can be seen as a method for sampling solutions generated by an MLP model of the best solutions. The algorithm can be also considered as a hybrid between local optimization methods used to create the adversarial perturbation and global search as implemented by the EA.

4.1 Inserting Adversarial Perturbation Methods in EAs

We can consider the methods for creating the adversarial perturbations as sampling procedures since we assume that they will produce solutions similar to the ones from the class *Best* used for training. However, it is not possible to know in advance which of the strategies discussed in Sect. 2 will be more effective to improve more the fitness of solutions. Some of the methods for creating the adversarial perturbations, such as FGSM are well known local optimizers, but

other algorithms such as Deepfool exploit completely different strategies that depend on the information contained in the neural network representation.

We expect that adversarial methods, such as the `Blend Attack`, that do not exploit information about the network will be less effective than methods that exploit the gradient or the information about the class decision borders of the network, but this does not have to be the case.

Furthermore, evaluating the effect of the adversarial perturbations in the context of evolutionary optimization is also a difficult task since different criteria can be considered. For example, methods that increase the fitness the most in a single application can be desirable, but the speed of the method is another relevant criterion, as is the capacity of the algorithm to create "diverse" adversarial examples to accomplish a more exploratory search. None of these criteria are usually considered when evaluating the adversarial examples for attacking machine learning models.

5 Related Work

Although we did not find any previous application of adversarial perturbations as a component of an evolutionary algorithm, neural networks have been extensively investigated as surrogates in EAs [6]. They have been applied for estimating the values of the fitness functions and to assist in the application of mutation and crossover operators [7]. The main difference between our proposal and previous applications of neural networks as surrogates is that in our approach, the network information is used at the time of generating the sample, not to predict the quality of the sample.

In EDAs [9], sampling methods play an important role to generate new solutions. There are a number of papers that implement sampling methods to generate solutions from a neural network. Perhaps the best known examples are those algorithms based on variants of the Restricted Boltzmann Machines (RBM) [16,21] and Deep Boltzmann Machines [15]. This type of models keep a latent representation of the data, and exploit this information using Gibbs sampling or other Markov Chain Monte Carlo methods.

A closer relationship to our proposal exists with the method of ANN inversion [10], introduced by Baluja [1] to sample an MLP as part of an evolutionary algorithm (Deep-Opt-GA). The goal of ANN inversion (or back-drive) is, given the possible output value of a network, determine which input values produce that output. It is usually applied for neural networks that solve a regression problem [1,3].

There are important differences between the use of back-drive within the context of evolutionary search for generating solutions as in [1] and our proposal. First, back-drive is a single method to recover the inputs while adversarial perturbation is a diverse set of techniques of different nature. Second, back-drive assumes that the neural network has been trained to solve a regression task and therefore requires more detailed information about the solutions. We use a neural network that solves a classification task and therefore our method only

requires a way to separate promising from poor solutions. Partial evaluation of solutions and other strategies can be used for this goal.

6 Experiments

In this experimental section we address different questions in order to evaluate the performance of the adversarial methods for guiding the evolutionary algorithm. More specifically, the goals of our experiments are:

- To determine whether the strategy used to initialize the solutions before applying the adversarial perturbations has an influence in the performance of the method.
- To investigate the behavior of the different adversarial perturbation methods within the evolutionary computation framework.
- To investigate whether the modification of the network improves the frequency in which adversarial perturbations can be applied.

The experiments were divided into three parts according to the questions stated above. In the first part of the experiments (Sect. 6.2), we investigate the influence of the initialization schemes for a reduced number of functions and a large set of adversarial perturbations methods. In the second part of the experiments (Sect. 6.3), we focus on the analysis of a reduced set of adversarial perturbation methods for a larger set of functions. In the third group of experiments, we investigate the effect of adversarial perturbations when networks have been tricked.

6.1 Problem Benchmark and Parameters of the Algorithm

The extensive problem suite of CEC-2005 [19], as implemented in [22], has been used to determine the performance of the adversarial methods driving an evolutionary algorithm. We select functions F1 to F17, except function F7 that has been discarded because of the range of possible values of the variables being not fixed.

For the first and third groups of experiments, the reduced set of functions comprises F1, F2, F6, F8, F13, and F14. These functions have been chosen as representative of three different levels of difficulty; univariate (F1 and F2), basic multimodal (F6 and F8), and expanded multimodal (F13 and F14).

All the experiments involve the same DNN structure: 2 hidden layers; sigmoid activation function after the two layers; softmax activation function after the output layer; weights (w_l) are initialized employing Xavier initialization [4]; and biases (b_l) are initialized to 0. The network is trained using the Adam optimizer, with a batch size of 50. Population size was $N = 1.000$ and we used truncation selection $T = 0.3$ with the *best elitism* method in which all the selected solutions are kept for the next population. The number of generations was $n_{gen} = 50$. For each combination and function, 5 independent executions have been performed. The algorithm was implemented in Python and the code is available from the authors upon request.

6.2 Initialization Schemes for Adversarial Perturbations

As explained in Sect. 3, the solutions are divided into three groups: *Best*, *Middle*, and *Worst*. The adversarial perturbations are applied to solutions in the third group and some solutions in the *Middle* class. The question we want to investigate is whether using these *poor* solutions as initial values can be beneficial for the workflow of the evolution. To test this hypothesis, we consider three scenarios: 1) Use the worst solutions to initialize the adversarial methods, 2) Use mutated variants of the solutions in the *Middle* class. 3) Use randomly initialized solutions. These scenarios are tested for all combinations of adversarial methods and functions.

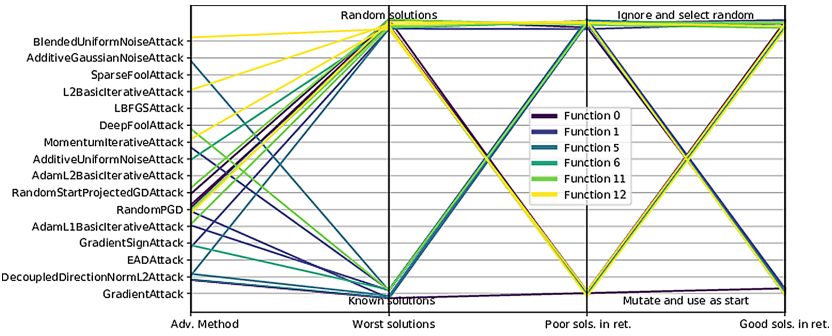

Fig. 1. Parallel coordinates showing the relation between the different components in the preliminary experimentation. For each problem, the top 3 runs are shown, one line for each one.

Figure 1 shows a summary of the results. In the figure, we represent, for each function, which have been the four best combinations of initialization and adversarial perturbation methods out of all the runs executed. It serves as way to identify top-performing configurations. In the figure, each configuration of adversarial method and individual initialization technique is represented by a line.

The analysis of Fig. 1 reveals that the effect of the initialization method is, to some extent, problem-dependent. Simpler problems can be better optimized initializing from the worst solutions, the more complex ones benefit almost exclusively from random solutions. This effect is less noticeable for solutions in the *Middle* set. In this case, initialization from solutions in the *Middle* class can also produce top results for complex functions. Runs that do not use random initialization were only able to achieve a top-3 performance once, for a simple function. These results indicate that the diversity component introduced by the random initialization is, overall, a positive feature.

In addition to clarifying the effects of the different initialization methods, the experiments show that the performance of the adversarial methods is in general problem-dependent. For example, some of the best results obtained for

the expanded multimodal problems (the *clearest* lines), have been obtained when the EA uses adversarial perturbation methods unable to produce top results for any of the basic multimodal or unimodal problems (`BlendedUniformNoise` or `DeepFool`). Regarding these other problems, EAs incorporating adversarial methods that use the gradients and the `DecoupledDirectionNormL2` were the ones producing the best results. This finding is relevant because it points to a possible way to characterize the behavior of adversarial perturbation methods using different classes of optimization problems.

Methods which make use of the gradients (in this case `RandomPGD`, `Decoupled DirectionNormL2`, and `RandomStartProjectedGD`) outperform the others for the simplest type of functions. Other methods which showed good performances in the most difficult functions while still performing considerably well in the simpler ones are `DeepFool` and `BlendedUniformNoise`. As can be observed, the majority of the top-performing methods rely on the gradients of the network. This was to be expected, as these methods take more informed decisions with respect to the model at the time of modifying the individuals.

6.3 Performance of Adversarial Perturbation Methods

In order to perform a more in-depth analysis of the components that can affect the performance of the adversarial example generation technique as a guide of EAs, we enlarge the pool of functions in which the algorithm is tested. We also constrain the set of adversarial perturbation methods to: A method which does not rely on the gradients of the network (and therefore requires no information about the model): `BlendedUniformNoise`, and other four which do `DeepFool`, `DecoupledDirectionNormL2`, `RandomPGD`, and finally `RandomStartProjectedGD`. We evaluate two scenarios for applying the adversarial perturbation methods to solutions in the *Middle* class: R) random initialization and M) mutation of a solution in class *Middle*.

Figure 2 shows a heatmap which encompasses the results of this experiment. All fitness values obtained by every method and combination were scaled to [0, 1] in order to improve the interpretability of the results. For each problem (in the y axis), and method-variant combination (in the x axis), the color represents the mean of the best fitness values, computed from the five runs. The darker (and therefore, lower) the better. Additionally, each of the cells in the heatmap has an overlaid digit. This number is the negative logarithm in base 10 of the variance obtained across all five runs. In other words, the number is the positive version of the exponent of the variance. In this case, a larger number denotes less uncertainty about the final result.

Analyzing this figure row-wise, it is possible to observe that some of the problems have considerably brighter colors across all method combinations. The runs for these functions have produced largely diverse best fitness values (e.g., one *good* run and other four *very poor* runs), which explains the mean being high in the [0, 1] interval and the high variance.

Analyzing the figure as a whole, it becomes apparent that, again, there is not an absolute winner among the adversarial methods. For example, for problems

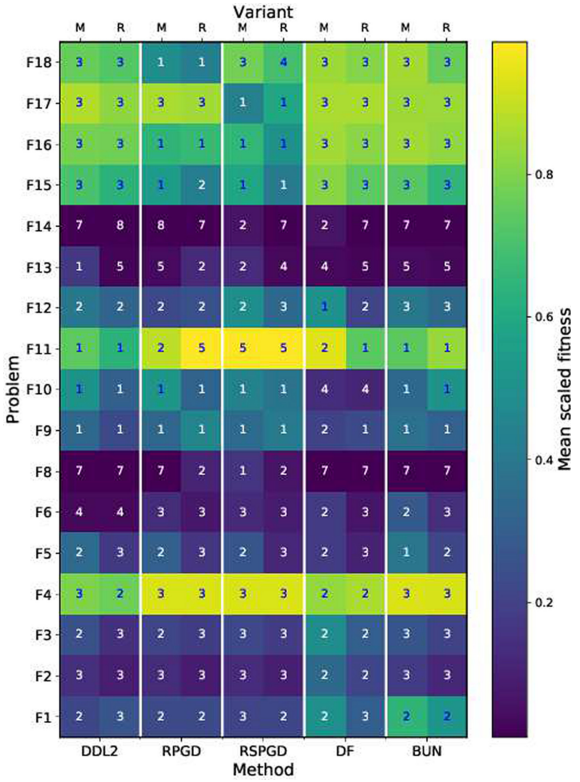

Fig. 2. Heatmap showing the mean values obtained by each method with different treatment of the *poor* solutions in the middle set, for each function. The numbers inside the figures represent the variance of all runs performed. Variants for starting solutions: Mutated versions of the solutions classified as poor in the middle set (M), or random solutions as (R). Methods: Decoupled Direction Norm L2 (DDL2), RandomPGD (RPGD), Random Start Projected GD (RSPGD), DeepFool (DF), Blended Uniform Noise (BUN).

F1-F3, DDL2, RPGD, and RSPGD produce the best mean fitness values. For F5-F8, DDL2 and DF offer the best performance, whereas for F9 and F10, only DF has a good performance. RSPGD and RPGD exhibit the lowest means for F15-F18, and for F4 and F11, DDL2 is the most consistent one. Almost in all cases, using a random initialization is better than applying mutation.

6.4 Network Tricking

Finally, we focus on trying to determine whether tricking the network in such a way that it is more difficult for it to predict a solution as *good* can be beneficial for the EA. If the adversarial perturbation method is forced to further modify

the solutions so that the network recognizes them as good, then there is a chance of the solutions actually improving even more.

To test this hypothesis, we use the reduced set of functions and an adversarial method which has offered consistent performance across functions and parameters (DDL2). Five runs were respectively executed for the EAs that use the ordinary neural network, and the tricked network.

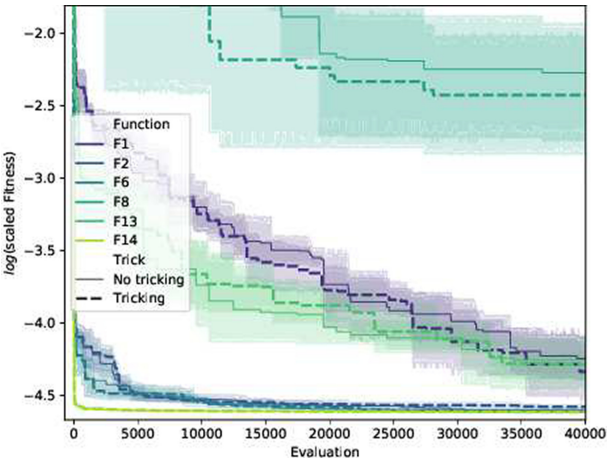

Fig. 3. Best fitness value along generations for EAs that use ordinary and tricked neural network.

Figure 3 shows details of the evolution for the six different functions (in different colors). Each line represents the mean of the best found individual in terms of fitness (y axis) at each point in the evolution (x axis) of each set of five runs. Dashed lines represent runs in which network modification took place. The clear lines accompanying the opaque ones represent the standard deviation of the 5 runs.

As in previous experiments, results are problem-dependent. For functions F1, F6, F8, and F13, performing the network modification produces an improvement in the performance of the algorithm, whereas this was not the case for F2 and F14. What is more, in this case, no pattern about certain characteristics fitting problem particularities can be deduced, since the best results for the unimodal (F1 and F2), and the expanded multimodal functions (F13 and F14) were obtained with different approaches. Taking all into account, however, tricking the network improved the evolutionary process on four of the six sets of runs.

7 Conclusions

In this paper we have proposed the use of adversarial perturbations as a way to guide the search for optimal solutions in an evolutionary algorithm. Our method

combines the use of a neural network that acts as a predictor of promising versus poor solutions, with the application of algorithms originally conceived to deceive neural networks. Our results show that it is indeed possible to use the adversarial perturbations to improve the quality of the solutions from the first generation. However, the perturbations stop improving the results after a relatively small number of generations.

Our results are also of interest for research on neural networks. We have shown how evolutionary optimization can serve as test-bed to evaluate different methods for deceiving the networks. In this sense, our work opens another avenue for the investigation of synergies between neural networks and optimization algorithms.

References

1. Baluja, S.: Deep learning for explicitly modeling optimization landscapes. CoRR abs/1703.07394 (2017). http://arxiv.org/abs/1703.07394
2. Dong, Y., et al.: Boosting adversarial attacks with momentum. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 9185–9193. IEEE Press (2008)
3. Garciarena, U., Mendiburu, A., Santana, R.: Envisioning the benefits of back-drive in evolutionary algorithms. In: 2020 IEEE Congress on Evolutionary Computation (CEC), pp. 1–8. IEEE (2020)
4. Glorot, X., Bengio, Y.: Understanding the difficulty of training deep feedforward neural networks. In: Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics, pp. 249–256 (2010)
5. Goodfellow, I.J., Shlens, J., Szegedy, C.: Explaining and harnessing adversarial examples (2014)
6. Jin, Y.: Surrogate-assisted evolutionary computation: recent advances and future challenges. Swarm Evolut. Comput. **1**(2), 61–70 (2011)
7. Jin, Y., Olhofer, M., Sendhoff, B.: On evolutionary optimization with approximate fitness functions. In: Proceedings of the 2nd Annual Conference on Genetic and Evolutionary Computation, pp. 786–793 (2000)
8. Kurakin, A., Goodfellow, I., Bengio, S.: Adversarial examples in the physical world. CoRR abs/1607.02533 (2016). http://arxiv.org/abs/1607.02533
9. Larrañaga, P., Karshenas, H., Bielza, C., Santana, R.: A review on probabilistic graphical models in evolutionary computation. J. Heuristics **18**(5), 795–819 (2012). https://doi.org/10.1007/s10732-012-9208-4
10. Linden, A., Kindermann, J.: Inversion of multilayer nets. In: Proceedings of the International Joint Conference on Neural Networks, vol. 2, pp. 425–430 (1989)
11. Marti, L., García, J., Berlanga, A., Molina, J.M.: Introducing MONEDA: scalable multiobjective optimization with a neural estimation of distribution algorithm. In: Proceedings of the 10th Annual Conference on Genetic and Evolutionary Computation GECCO-2008, pp. 689–696. ACM, New York (2008). http://doi.acm.org/10.1145/1389095.1389228
12. Moosavi-Dezfooli, S.M., Fawzi, A., Frossard, P.: DeepFool: a simple and accurate method to fool deep neural networks. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 2574–2582 (2016)
13. Pal, S.K., Mitra, S.: Multilayer perceptron, fuzzy sets, and classification. IEEE Trans. Neural Netw. **3**(5), 683–697 (1992)

14. Polyak, B.T.: Some methods of speeding up the convergence of iteration methods. USSR Comput. Math. Math. Phys. **4**(5), 1–17 (1964)
15. Probst, M., Rothlauf, F.: Deep Boltzmann machines in estimation of distribution algorithms for combinatorial optimization. CoRR abs/1509.06535 (2015). http://arxiv.org/abs/1509.06535
16. Probst, M., Rothlauf, F., Grahl, J.: Scalability of using restricted Boltzmann machines for combinatorial optimization. Eur. J. Oper. Res. **256**(2), 368–383 (2017)
17. Rony, J., Hafemann, L.G., Oliveira, L.S., Ayed, I.B., Sabourin, R., Granger, E.: Decoupling direction and norm for efficient gradient-based l2 adversarial attacks and defenses. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 4322–4330 (2019)
18. Stork, J., Eiben, A.E., Bartz-Beielstein, T.: A new taxonomy of global optimization algorithms. Nat. Comput., 1–24 (2020)
19. Suganthan, P.N., et al.: Problem definitions and evaluation criteria for the CEC 2005 special session on real-parameter optimization. Technical report, Nanyang Technological University, Singapore (2005)
20. Szegedy, C., et al.: Intriguing properties of neural networks. CoRR abs/1512.1312.6199 (2015). http://arxiv.org/abs/1312.6199
21. Tang, H., Shim, V., Tan, K., Chia, J.: Restricted Boltzmann machine based algorithm for multi-objective optimization. In: 2010 IEEE Congress on Evolutionary Computation (CEC), pp. 1–8. IEEE (2010)
22. Wessing, S.: Optproblems: infrastructure to define optimization problems and some test problems for black-box optimization. Python package version 0.9 (2016)
23. Yuan, X., He, P., Zhu, Q., Li, X.: Adversarial examples: attacks and defenses for deep learning. IEEE Trans. Neural Netw. Learn. Syst. **30**(9), 2805–2824 (2019)