Address for correspondence: Institute of Computer Science, ul. Nowowiejska 15/19, 00-665 Warsaw

*IOS Press*

# Deep learning optimization tasks and metaheuristic methods

**Rafał Biedrzycki**
*Institute of Computer Science*
*Warsaw University of Technology, Poland*
*rbiedrzy@elka.pw.edu.pl*

**Paweł Zawistowski**
*Institute of Computer Science*
*Warsaw University of Technology, Poland*

**Bartłomiej Twardowski**
*Institute of Computer Science*
*Warsaw University of Technology, Poland*

**Abstract.** In this paper we identify and formulate two optimization tasks solved in connection with training DL models and constructing adversarial examples. This guides our review of optimization methods commonly used within the DL community. Simultaneously, we present findings from the literature concerning metaheuristics and black-box optimization. We focus on well-known optimizers suitable for solving $\mathbb{R}^N$ tasks, which achieve good results on benchmarks and in competitions. Finally, we look into the research connected with utilizing metaheuristic optimization methods in combination with deep learning models.

**Keywords:** Deep learning, metaheuristics, optimization.

## 1. Introduction

Deep learning (DL) methods have been very successful in recent years, with a myriad of models and results presented in the literature. Some of the most prominent DL methods include applications connected with image processing (like object recognition [1, 2] or image segmentation [3, 4]) and natural language processing (like creating word embeddings [5, 6], language modeling [7] or machine translation [8]) to name only a few. As is the case with the majority of machine learning models, the effectiveness of these solutions is tightly coupled with our ability to train them, usually using data sets of significant size. This explains the importance of optimization methods used within the DL ecosystem which made such accomplishments possible. Many optimization methods have been developed alongside modeling techniques and are widely adapted within the community. However, it should be noticed that the spectrum of available optimization techniques is much broader, and many interesting and robust approaches exist, yet are rarely used in connection with DL.

In this paper we identify and define the optimization tasks solved in connection with training predictive models and generating adversarial examples. We then analyze the dimensionalities of such tasks solved during the training of selected successful DL models. Furthermore, we present some of the optimization algorithms often used in the DL domain. This reveals problems connected with local minima and the stability of the obtained solutions. Secondly, we review the available metaheuristics in search for methods suitable for solving optimization tasks similar to the ones found within DL. This allows us to evaluate the feasibility of applying metaheuristics to DL and name the potential gains from such an approach.

The paper is organized as follows: in Section 2 we identify and define common optimization tasks solved in connection with DL and present an overview of the most commonly used methods of solving them. Section 3 introduces the notion of metaheuristics and provides brief descriptions of some important algorithms within that domain. In Section 4 we recognize some current attempts at utilizing metaheuristics within DL, and Section 5 provides a summary and conclusions.

## 2. Optimization in DL

From a very wide perspective, constructing machine learning models involves gathering training data, defining the model architecture and estimating all the necessary parameters. The last part consists of solving an optimization task in which the model's performance is evaluated using some loss function.

More formally, let $D \in \Delta$ denote the training data set within domain $\Delta$. $D$ is used to train model $\hat{f}$ with parameter vector $\theta \in \Theta$. To enable model evaluation, a loss function $\mathcal{L} : \Theta \to \mathbb{R}$ is given which estimates the performance of model $\hat{f}$ on $D$. From an optimization perspective, training model $\hat{f}$ consists of solving the following minimization task:

$$\theta^* = \arg\min_{\theta \in \Theta} \mathcal{L}(\theta; D) , \tag{1}$$

where $\theta^*$ denotes the optimal parameter vector whose values are constrained to the $\Theta$ set.

Such a formulation of model training is very broad and admits both supervised and unsupervised learning tasks. For example, in the case of classification, $\hat{f}$ would denote a parameterized classifier, $D$ would be a set consisting of $(\mathbf{x}, y)$ pairs where $\mathbf{x}$ is the attribute vector and $y$ the class label random variable, and finally $\mathcal{L}$ could be defined using cross entropy between the actual and predicted label distributions. On the other hand, if clustering is to be considered, then the training set $D$ would contain unlabeled objects and $\mathcal{L}$ would be a function evaluating the quality of the obtained clustering.

In order to make this formulation more tangible we identify major groups of DL models and present the challenges connected with training them in Section 2.1.

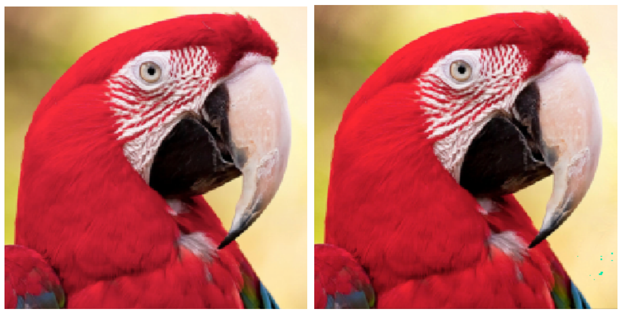

Figure 1. An adversarial example (1b) generated using the DeepFool method [9]. Although the differences from the original (1a) are relatively small, the classifier (ResNet-34 [2]) makes a mistake.

Although model training is a central part of DL, it is not the only application of optimization methods within this domain. One other application is connected with the phenomenon of adversarial examples [10], which are defined as inputs which have been deliberately perturbed in order to induce errors in trained prediction models. This concept may be grasped by analyzing the example in Fig. 1, where a well performing classifier is deceived by a small perturbation applied to the given input. The observation that perturbations which seem irrelevant to a human eye may prove to be disastrous to image classification models has drawn much attention to this subject. In effect, intense studies have been conducted during recent years to develop both the methods to generate adversarial examples ([11, 9, 12]) and algorithms protecting the models from such attacks ([13, 14, 15]).

In the case of classification, constructing adversarial examples may be formalized as follows. Given a classifier $f$ and an input vector $\mathbf{x} \in X$, we define an (untargeted) adversarial example $\tilde{\mathbf{x}}^* \in X$ as a solution to the following minimization task:

$$\tilde{\mathbf{x}}^* = \arg\min_{\tilde{\mathbf{x}} \in X} d(\tilde{\mathbf{x}}, \mathbf{x}) \ , \tag{2}$$

$$\text{s.t.: } f(\tilde{\mathbf{x}}) \neq f(\mathbf{x}) \ , \tag{3}$$

where $d$ denotes a chosen distance measure used to make the perturbations "small". The key difference from tasks connected with parameter training is that the optimization is performed in the input space $X$, not parameter space $\Phi$. There is, however, a method which bridges both these tasks called adversarial training [15]. We present it along with some of the optimization algorithms utilized in the context of adversarial examples in Section 2.3.

## 2.1. Model training tasks

When analyzing the model training optimization task defined in (1), it is clear that the difficulty of the task greatly depends on the parameter space $\Theta$ which is defined by the structure of model $\hat{f}$. Although there are many types of neural network architectures, well outside the scope of this paper, in this section we highlight some key concepts to make the subject more tangible.

Some of the simplest types of neural network architectures are Multilayer Perceptrons (MLP), which in their basic form consist of neurons grouped into layers connected in a feed-forward manner. Specifically, this means that connections are only allowed from neurons in the $i$-th layer to neurons in the $(i + 1)$-th layer. The overall network structure can be therefore described using a directed, acyclic graph and thus individual model parameters are represented as weights on the edges of this graph. An example of such architecture is given in Fig. 2. From the optimization perspective, training an MLP network corresponds to minimizing the loss function in the domain of weights; therefore the dimensionality of $\Theta$ is directly connected to the number of weights in such a network. Such an observation remains true for most, if not all, neural architectures, however, as we discuss below, it is not always a direct one-to-one correspondence.

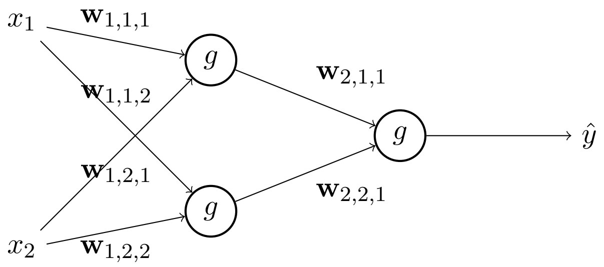

Figure 2. A simple MLP neural network with two-dimensional inputs $\mathbf{x} \in \mathbb{R}^2$, two hidden layers with activation function $g$ and a single output $\hat{y} \in \mathbb{R}$.

Although MLPs provide very robust predictive models, for some tasks the number of parameters may become impractically large. For example, in the case of image processing, the input space $X$ is highly dimensional, and creating a model consisting of multiple fully connected layers leads to enormous numbers of weights that need to be estimated during training. Convolutional Neural Networks (CNNs) are one type of architecture which helps in resolving such issues [16]. These models are feed-forward in principle just like MLPs, however they utilize what are known as convolutional layers, which have a more sophisticated structure, i.e. they use local connections and weight sharing to reduce the number of model parameters. CNNs are well suited to tasks in which the inputs are organized in structures (like sequences, matrices or tensors) in which locality plays an important role. Two prominent examples of such tasks are image classification tasks (where the inputs correspond to pixel values from different color channels and are naturally organized in tensors) and sequence processing tasks (where the temporal dimension or simply the order in the sequence plays a similar role). Training basic CNNs requires a lower number of parameters to be estimated than connections in the network because of the weight sharing property, which causes multiple connections in the network to be controlled by a single parameter. This means that the dimensionality of $\Theta$ may be lower than the number of weights in the given network.

The feed-forward architecture briefly discussed above is static models which have no natural measures to maintain states. Although this might be sufficient for many tasks, there are domains, like natural language processing, which require contextual information to be stored. One suitable group of models for such cases are Recurrent Neural Networks (RNNs) [17], for which the connection graphs are not acyclic. This sets them apart from architectures like MLPs and allows RNNs to maintain some contextual information within the loop-back connections. In practice, such models are often unfolded into feed-forward networks which makes training them an easier task. An example of such an operation is depicted in Fig. 3.

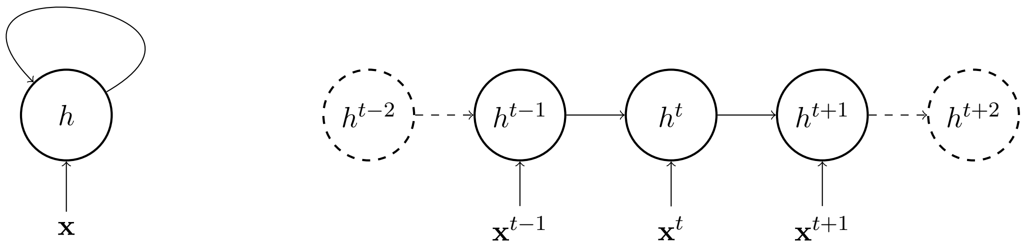

Figure 3. A simple RNN with no outputs (left) along with its unfolded version (right). The superscripts denote the temporal dimension.

One serious drawback of such an operation is however the exploding/vanishing gradient problem. This is a phenomenon recognized in deep networks with many hidden layers, which may make gradient optimization methods unstable and parameter estimation very difficult. A breakthrough has been introduced by the Long Short-Term Memory (LSTM) model [18], which defined a concept of LSTM cells and in principle resolved this issue. Note that this is one of the cases in which the model architecture needed to be modified in order to make applying gradient-based methods possible. Also, regarding the exploding/vanishing gradient problem, recent research suggests that it is possible to train a very deep model (10 000 hidden layers) using gradient methods without using specialized architecture, only careful weight initialization [19].

The concepts highlighted above have become the basis of many types of architectures introduced in recent years. When considering the complexity of optimization tasks solved during the training of these models, the dimensionality of the parameter space $\Theta$ should be considered. To make this more tangible, in Table 1 we present a couple of examples of successful models from various domains in which DL is applied today. It may be noticed that training involves optimization in complicated spaces with many millions of dimensions, however whether this is a global or a local optimization depends on the specific case. From the algorithmic perspective, gradient methods currently seem to be the most widely used tool for training DL models. We provide an overview of specific methods in the following section.

## 2.2. Gradient methods

As stated before, finding the best set of model parameters $\theta$ for $\hat{f}$ is an optimization task that minimizes the value of the defined loss function $\mathcal{L}(\theta; D)$. If the loss function is differentiable, optimization

| # | Model | Parameters | Year | Task |
|---|---|---|---|---|
| 1 | Inception [1] | 6.8M | 2015 | Image classification |
| 2 | ResNet 110 [2] | 1.7M | 2016 | |
| 3 | ResNet 1202 [2] | 19.4M | | |
| 4 | word2vec [5] | 900M[1] | 2013 | Word embeddings |
| 5 | FastText [6] | 600M[1] | 2018 | |
| 6 | FCNV-GG16 [3] | 134M | 2015 | Image segmentation |
| 7 | FCN-GoogLeNet [3] | 6M | | |
| 8 | FRRN [4] | 17.7M | 2017 | |

[1] values estimated on the pretrained model provided by the authors, which consists of 300 dimensional vectors for large numbers of words

Table 1. Model sizes of some recent DL models created for various tasks.

methods based on the objective function gradient w.r.t. model parameters can be used (for neural networks, this gradient is usually calculated using the famous backpropagation algorithm [20]). This allows for iterative updates of the model towards better results. Parameters are updated in the opposite direction of the computed gradient $\nabla_\theta \mathcal{L}(\theta; D)$. This is the merit of the gradient descent method, where the main update step can be defined as:

$$\theta_{t+1} = \theta_t - \lambda \cdot \nabla_{\theta_t} \mathcal{L}(\theta_t; D) \ , \ t = 1, 2, 3, \ldots \tag{4}$$

where $\lambda$ is the learning rate, which defines the step size taken in a single update. After an appropriate number of updates, when the optimization aim is attained, the model is considered to be trained and ready for use. This can be the result of finding either a local or a global optimum. When the model is being trained, the loss function value (which should be a proxy value for solving a task for the trained model, e.g. classification, regression, clustering) becomes minimized.

All the state-of-the-art models presented in Section 2.1 are trained with methods based on gradient computation and use some kind of gradient descent update. These methods are covered in detail in this section. This only proves the current state of optimization tasks for DL models: that gradient methods are considered to be standard and the most common method for learning network parameters. This success is also propelled by the fact that GD methods are first-class optimizers for many popular DL frameworks, used for research in academia and industry, e.g. Tensorflow [21], PyTorch [22].

This situation — that GD methods are the first choice for DL network training or even considered to be the only way — resulted in an interesting phenomenon where the approach for solving ML tasks changed. Initially, a model was chosen for a defined problem and then the optimization method for finding its parameters was selected, and now, the DL network architecture is adjusted to the defined problem and the solid implementation of a widely used optimization method. Thus, the model itself is changed in order to make a training task achievable. Two well-known examples for DL can be recalled here. The first is the special architecture for feed-forward neural networks (FFNNs) called the Highway Network [23]. This is an example of how to cope with very deep networks that are trained using gradient-based methods. The second example is for a recurrent network, when a gradient propagation through time starts to be problematic. In order to solve that problem, gated units have been proposed, e.g. LSTM [18] and GRU [24], and are proven to be successful in tandem with GD training methods.

However, the majority of available optimizers based on GD methods do not solve the task of training DL networks using the presented basic approach. It can be done for some simple models, but for more complicated ones, or when experimenting on different settings, this approach would be insufficient. Thus, some improvements have been proposed over the years. In the following paragraphs, first the strategies of how to calculate the gradient are presented. Then, the remaining problems of using the basic GD method for DL are indicated. Some of these problems are addressed with variations of GD methods. The most commonly used ones in DL are described here. Finally, the methods based on second order information about gradients in the context of DL are presented.

**Batch, mini-batch, stochastic GD, and on-line learning.**

The information about the derivative of the loss function w.r.t. each model parameter is needed for each step of gradient descent. This calculation can be done in different ways while the DL is being trained. The normal way for the basic GD method is to get it calculated over the whole available data set $D$ (either it is an unsupervised task or supervised in the form of $(X, Y)$). This is called *batch gradient descent*. The update is given:

$$\theta_{t+1} = \theta_t - \lambda \cdot \nabla_{\theta_t} \mathcal{L}(\theta_t; D) \tag{5}$$

In each update, the whole data set has to be iterated in order to get the gradient value. In other words, from a machine-learning point of view, the model receives one step of a gradient descent in each epoch. Considering the sizes of modern data sets like ImageNet, MultiSNLI and the number of DL model parameters, this could be computationally infeasible and appropriate only for small-toy examples.

*Stochastic gradient descent* (SGD) does updates for each record. The update equation:

$$\theta_{t+1} = \theta_t - \lambda \cdot \nabla_{\theta_t} \mathcal{L}(\theta_t; d_i) \tag{6}$$

reflects that for each data observation record $d_i \in D$, the gradient is calculated and a descent step is performed. When this method answers the problem of going through the whole dataset for a single update, other shortcomings emerge. The most important one is convergence, even to the local minimum. Because of the very different, and even contradictory, gradients from a single example to another, the model can perform many unnecessary fluctuations. In order to decrease this effect, methods for proper randomization or providing examples in a specific order [25] can be useful for overall training success. The SGD method is a type of *on-line machine learning* where gradient is used. In on-line machine learning, observations from the dataset $D$ are available in sequential order and update the model (in the case of DL – parameters) at each step [26].

Both of the described methods: batch and stochastic, can be considered as going from one extreme to the other — from using a whole dataset to a single-random record per update respectively. In between those two, there is a third method — *mini-batch gradient descent*. This can be presented in a single line update equation:

$$\theta_{t+1} = \theta_t - \lambda \cdot \nabla_\theta \mathcal{L}(\theta_t; d_i, \ldots, d_{i+n-1}) \tag{7}$$

where $d_i \in D$ is a single observation and $n$ is the number of observations in the mini-batch. This method is a trade-off between the computational time of an all-dataset batch and accuracy. The size of the mini-batch is here a hyperparameter for the whole process of training the DL model, which can be adjusted for a specific use case. Currently, this is the most popular method for training DL models.

Even having a few strategies for when and how the gradient is calculated does not answer other problems that are connected with different aspects of training DL models with GD. Some of these problems were identified and described in survey paper [27]. The main ones are:

- choosing the right learning rate [28],
- learning rate schedules — adjusting the learning rate during the training,
- using the same learning rate for all model parameters,
- a highly non-convex error function, with many sub-optimal local minimas. This problem becomes bigger if the situation is connected with saddle points for high-dimensional optimization [29].

In order to answer some of these problems, a number of techniques have emerged. The description below presents the most popular ones.

The first and the simplest of those methods tries to eliminate oscillations of update steps towards the local optimum using a momentum. The momentum is considered as a way of accelerating SGD in the relevant direction. It does this by adding a fraction of the previous update vector to the current one, with $\mu \in (0, 1)$:

$$\begin{aligned} v_t &= \mu v_{t-1} + \lambda \nabla_\theta \mathcal{L}(\theta_t) \\ \theta_{t+1} &= \theta_t - v_t \end{aligned} \tag{8}$$

A similar approach to this, but done in a slightly different way, is taken in the method called Nesterov acceleration gradient (NAG) [30], where at first the jump is done based on the momentum direction and the gradient is calculated. Then the correction is made for the set parameters:

$$\begin{aligned} v_t &= \mu v_{t-1} + \lambda \nabla_\theta \mathcal{L}(\theta_t - \mu v_{t-1}) \\ \theta_{t+1} &= \theta_t - v_t \end{aligned} \tag{9}$$

This approach should alleviate the effect of unnecessary hill climbing when the momentum is big.

Both of these methods presented do not address the problem of setting a proper value for the learning rate itself. Not to mention the strategy how it can change over a time of model training. Considering DL architectures used nowadays this is very important, as the neurons in each layer learn differently and are expected to play a different role in the prediction process. This results in a variation of the gradient magnitude across different layers. In contrast, while training DL models, a vanishing gradient problem can also cause a lot of issues. Both problems can be tackled by properly adjusting the learning rate. In work [31] devoted to adaptive sub-gradient methods for on-line learning and stochastic optimization, the authors proposed the AdaGrad optimization technique, which takes updates of parameters as follows:

$$\theta_{t+1} = \theta_t - \lambda \frac{\nabla_\theta \mathcal{L}(\theta_t)}{\sqrt{G_t + \epsilon}} \tag{10}$$

where $G_t$ is a diagonal matrix, where each diagonal element $i, i$ is the sum of the squares of the gradient of the parameter $\theta_i$ w.r.t to loss function $\mathcal{L}$, $\epsilon$ is a smoothing term and the division is performed elementwise. The adapted learning rate is then applied to the current gradient in a dot product operation in order to calculate the final update.

The AdaGrad method is still used and it is one of the standards in industry [32]. It removes the need to manually tune the learning rate. However, it still has its main drawback that the sum of the gradients is accumulated over time and the learning rate can only be smaller while the model is being trained. This is especially crucial in a situation where the gradient at the very beginning of training is large, as this will influence the rest of the training process with smaller learning rates. The algorithm *Root Mean Square Propagation* (RSMProp) [33] addresses this problem by applying a decaying average of squared gradients. The update step for this algorithm can be presented in:

$$r_t = \mu r_{t-1} + (1-\mu) g_t^2 \tag{11}$$

$$v_t = \sqrt{r_t + \epsilon} \tag{12}$$

$$\theta_{t+1} = \theta_t - \frac{\lambda}{v_t} \circ \nabla_\theta \mathcal{L}(\theta_t) \tag{13}$$

where $\mu$ is the forgetting rate parameter and $E[g^2]_t$ is the decaying average of square gradients, $g_t = diag(G_t)$ for the sake of brevity. However, still the units of such an update do not match the units of the parameters. To tackle this issue a different approach to an adaptive learning rate was proposed in [34] as Adadelta. This method changes the nominator of gradient adjustment components by using the RMS of the squared averages of the updates. The fraction of the current gradient is then taken based the comparison of two RMSs of averages.

The last method that will be described here is an adaptive moment estimation – Adam [35]. This method uses the exponentially decaying average of past squared gradients $p$ (like RMSProp and Adadelta) and the average of gradients $r$ (similar to momentum).

Apart from the first order methods described in this section, some second order approaches are utilized in connection with training neural network models like L-BFGS-B [36] or (usually for smaller models) Levenberg-Marquardt backpropagation [37, 38]. We mention these methods for completeness, but a full description is out of the scope of this paper.

## 2.3. Optimization in adversarial examples

A different niche in which optimization techniques are being used in combination with DL models is the construction of adversarial examples. The task is to create perturbations to a given input vector $\mathbf{x}$ which will induce a desired response from the given model $\hat{f}$. Methods which are designed for that purpose are called attack methods and currently are often developed in the context of classification models. From a security perspective, one important aspect is the threat model under which a given attack or defense technique has been developed [39]. This model explicitly declares what type of information the attacker has access to and what kind of actions are feasible for him. For adversarial attacks, usually the model specifies the following conditions:

- knowledge of the model's architecture and parameters,
- access to the model's predictions,
- information about the randomness source used by the model,
- information about training data used to prepare the model.

On one side of the spectrum of possible attacks are methods which assume full knowledge of the underlying model (known as the white-box setting). One such method, called Deep Fool [9], is presented in Algorithm 1. This approach iteratively searches for an adversarial perturbation by assuming that the attacked model may be approximated by an affine function in the local neighborhood of the input vector $\mathbf{x}_i$. It then performs gradient descent towards a region in which the classifier will change its decision.

**Algorithm 1:** Pseudocode of DeepFool for binary classifiers with classes $\{-1, 1\}$.

```
**Input:** Attacked image $\mathbf{x} \in \mathbb{R}^N$ and classifier $\hat{f} : \mathbb{R}^N \to \mathbb{R}$
**Output:** Adversarially perturbed image
$\mathbf{x}_0 \leftarrow \mathbf{x}$, $i \leftarrow 0$;
**while** $\mathtt{sign}\left(\hat{f}(\mathbf{x}_i)\right) = \mathtt{sign}\left(\hat{f}(\mathbf{x}_0)\right)$ **do**
    $\mathbf{r}_i \leftarrow -\frac{\hat{f}(\mathbf{x}_i)}{||\nabla \hat{f}(x_i)||_2^2} \nabla \hat{f}(\mathbf{x}_i)$;
    $\mathbf{x}_{i+1} \leftarrow \mathbf{x}_i + \mathbf{r}_i$;
**end**
**return** $\mathbf{x}_i$;
```

On the other side of the spectrum, methods exist which both assume no knowledge about the attacked model and have no direct access to it — only the returned class labels are available. One of the most effective versions of such methods is the Boundary Attack [40] presented in Algorithm 2. The method starts from an initial (adversarial) location $\mathbf{x}_0$ and operates in a fashion similar to hill climbing with a dedicated mutation operator — from a given location $\mathbf{x}_{i-1}$, the mutation is performed by drawing a random dislocation vector from a proposal distribution $\mathcal{P}(\mathbf{x}_{i-1})$ and moving it to a new location only if it is adversarial. The key aspect is the construction of the $\mathcal{P}(\mathbf{x}_{i-1})$ distribution, which ensures that the vector $\mathbf{x}_i \sim \mathcal{P}(\mathbf{x}_{i-1})$ is closer to the original image $\mathbf{x}$ than $\mathbf{x}_{i-1}$. It should be noted that because the attack starts from an already adversarial point, it may be applied both in a targeted (where the goal is to force the classifier to assign a specific class to the given image) and an untargeted setting (where the goal is only to force the classifier to change its decision).

**Algorithm 2:** Pseudocode of Boundary Attack method.

```
Input: Attacked image $\mathbf{x}$, classifier's $\hat{f}$ decision $d(\mathbf{x})$ and maximal number of iterations
Output: Adversarially perturbed image
$\mathbf{x}_0 \sim \mathcal{U}(0,1)$ s.t. $\mathbf{x}_0$ is adversarial, $i \leftarrow 0$;
while $i <$ maximal number of iterations do
    $\mathbf{r}_i \sim \mathcal{P}(\mathbf{x}_{i-1})$;
    if $\mathbf{x}_{i-1} + \mathbf{r}_i$ is adversarial then
        $\mathbf{x}_i \leftarrow \mathbf{x}_{i-1} + \mathbf{r}_i$;
    else
        $\mathbf{x}_i \leftarrow \mathbf{x}_{i-1}$;
    end
    $i \leftarrow i + 1$;
end
return $\mathbf{x}_i$;
```

When considering the character of the optimization tasks solved during adversarial attacks, a couple of aspects may be noticed. First of all, the dimensionality of the goal functions depend on the dimensionality of inputs rather than parameter spaces, and effectively these are (usually) lower when considering large DL models like the ones from Table 1. Furthermore, the goal function might be twofold. If white-box access to the predictions is assumed, the attack can observe, for example, the score that the model assigns to a specific class label. The goal is to minimize this score sufficiently. In the black-box scenario, where only predicted class labels are available, the goal function is either flat or simply measures the magnitude of the perturbation (or equivalently the distance between the original and perturbed images). However in this case, the optimization constraints are very problematic as only solutions for which the model makes a desired type of mistake are feasible. All in all, it is a problematic optimization space with constraints that are very expensive to evaluate and usually a limited budget of such evaluations.

## 3. Metaheuristics

The term "metaheuristic" was coined by Glover [41]. It is used for heuristic search methods that have a master strategy that guides and modifies the base heuristic. Nowadays this group includes a plethora of algorithms. Not all of them are of equal importance to the optimization community and for the applications, including DL. Most metaheuristics are nature-inspired algorithms; therefore they can be grouped by the source of the inspiration. What is more, some natural phenomena have attracted so much attention that associated algorithms can be sub-grouped even further. Generally, each such group has its root — a breakthrough concept, which is extended by its followers. Since, according to the "no free lunch" theorem [42], there cannot be a single best performing algorithm, selecting a proper optimization method for a given task is very important, yet difficult.

## 3.1. In search of good methods

A search of good optimization methods can be started from the set of the winners of optimization competitions. There are several families of these. Each competition is connected to a set of benchmark functions and requires a different experimental setup. Even though most benchmarks publish equations of the objective functions that should be optimized, the functions should be treated as black box, i.e., knowledge gained from analysis of the equations cannot be used to design the optimization procedure.

A wide family of optimization competitions is connected with the Congress on Evolutionary Computation (CEC). The experimental procedure and function definitions for both CEC'2017 and CEC'2018 competitions on single objective real-parameter numerical optimization have been described in [43]. The benchmark consists of 30 functions with different characteristics. The experiments are performed in 10, 30, 50 and 100 dimensions. The optimization is finished when an optimal value (within a given error margin) is found or when the budget of $10^4N$ objective function evaluations is exploited, where $N$ is the problem dimensionality.

Another benchmark [44] is used during the workshops on real-parameter black-box optimization benchmarking (BBOB), which take place during the Genetic and Evolutionary Computation Conference (GECCO). The benchmark includes 24 noiseless functions that are optimized in dimensionalities from 2 to 40. The results are presented in the form of plots. The best method can be found by looking for the maximum area under the curve. In BBOB competitions, algorithms do not have a limited budget, but the plot ends at $10^7$ run-length/$N$. Typically participants use the budget of $10^5N$ objective function evaluations in order to not wait too long for the results.

The next benchmark is quite different from the aforementioned ones. The Black Box Optimization Competition (BBComp) [45] implements a truly black-box scenario, i.e., optimized functions are not known to the participants. What is more, the functions cannot be used to carefully tune parameters of the algorithms because each registered participant of the competition has a budget limited to $100N^2$. The participants are not obliged to publish or even describe their methods. There are 100 problems in the benchmark. The problems are solved in dimensionalities from 2 to 64.

In all above benchmarks, the search space is constrained by box constraints, i.e., by the lower and upper bounds of parameter values.

We analyzed the results of the listed benchmarks and, for each algorithm, identified a root concept that was extended or just used. The results of that survey for CEC'2017 and CEC'2018 competitions [46] are shown in Table 2. The best methods according to the overall results in 20 dimensions for BBOB'2018 are shown in Table 3, and the best methods for BBCOMP'2017 and BBCOMP'2018 are shown in Table 4. For BBCOMP, the only methods listed are those for which it was possible to find a document with the description of the underlying algorithm.

From tables 2-4 we can observe that the names of the winners change in time and in the function of used benchmarks, but most of the winning methods are connected to only a few root concepts — the Covariance Matrix Adaptation (CMA) [48] is used for 63% of the listed algorithms and Differential Evolution (DE) [51] is used for 47% (some methods use both CMA and DE). Therefore, we will survey the root concepts and then we will analyze their usefulness for DL.

Table 2. The best algorithms from recent CEC competitions together with their root concepts

| | CEC'2017 | | CEC'2018 | |
|---|---|---|---|---|
| # | alg. name | root | alg. name | root |
| 1 | EBOwithCMAR[47] | CMA[48] | HS_ES[49] | CMA[48] |
| 2 | jSO[50] | DE[51] | LSHADE-ESP[52] | DE[51] |
| 3 | LSHADE-cnEpSin[53] | DE[51] | ELSHADE_SPACMA[54] | DE[51], CMA[48] |
| 4 | LSHADE_SPACMA[54] | DE[51] | EBOwithCMAR[47] | CMA[48] |
| 5 | DES[55] | DE[51], CMA[48] | UMOEAsII[56] | DE[51], CMA[48] |

Table 3. The best algorithms from the BBOB'2018 competition together with their root concepts

| # | alg. name | root |
|---|---|---|
| 1 | BIPOP-CMA-ES[57] | CMA[48] |
| 2 | PSA-CMA-ESwRS[58] | CMA[48] |
| 3 | CMAES-APOP[59] | CMA[48] |

Table 4. The best algorithms from recent BBCOMP competitions together with their root concepts. Only the methods for which it was possible to find a document with the description of the underlying algorithm are listed

| | BBCOMP'2017 | | BBCOMP'2018 | |
|---|---|---|---|---|
| # | alg. name | root | alg. name | root |
| 1 | AS-AC-CMA-ES[60] | CMA[48] | biteopt2017[61] | DE[51] |
| 2 | aDTS-CMA-ES[62] | CMA[48] | PSD-MADS[63] | MADS[64] |
| 3 | two-stage[65] | CMA[48] | GAPSO[66] | PSO[67], DE[51] |

## 3.2. Classical evolutionary algorithm

Evolutionary algorithms (EAs) are a large group of metaheuristics that were inspired by the phenomenon of natural evolution. The first EAs, called genetic algorithms, used sequences of bits to encode solutions [68, 69]. Later it was noticed that real-coded algorithms are better, especially for high-dimensional, multimodal problems [70, 71]. The pseudocode of a general, real-coded EA is provided in Algorithm 3.

**Algorithm 3:** Pseudocode of classical evolutionary algorithm (EA)

```
Input: Initial population $P$;
Output: The best solution found $BestSoFar$
$Objective \leftarrow$EvaluatePopulation($P$);
$BestSoFar \leftarrow \emptyset$;
while StopCriterionNotMet() do
    $Tmp \leftarrow$ Reproduction($P$);
    $Candidates \leftarrow$ GeneticOperations($Tmp$);
    $CandidatesObjective \leftarrow$EvaluatePopulation($Candidates$);
    $BestSoFar \leftarrow$GetTheBestSol($BestSoFar, Candidates, CandidatesObjective$);
    $P, Objective \leftarrow$Succession($P, Candidates, Objective, CandidatesObjective$);
end
return BestSoFar;
```

EAs are usually described using a naming convention borrowed from natural evolution. An encoded single solution of the problem is named the “individual” and a group of individuals that is maintained by the algorithm is called the “population”. The procedure that disturbs values of an individual’s parameters (genes) is called “mutation” and the procedure that creates a new solution by combining features of existing solutions is called “crossover”.

In the approach presented in Algorithm 3, the initial population is provided as an input to the algorithm. The size of the population has to be determined by the user. The initial values of parameters of individuals are usually randomly drawn in the feasible area. The algorithm works until the stop criterion is met. Usually, evolution is stopped after exhausting a given budget, which is expressed in the number of objective function evaluations. Of course, more sophisticated stopping criteria are also used, e.g. the algorithm can stop when it finds an optimum with a given error margin or when stagnation is detected, i.e. when it is not able to improve the average value of an objective function for several generations.

The first step in the algorithm’s loop is reproduction, which creates a new temporary population $Tmp$ by making copies of randomly selected individuals from the current population $P$. Better individuals have a higher chance of being included in $Tmp$. $Tmp$ is the input for genetic operations that produce a candidate population $Candidates$. The basic genetic operation is the mutation which randomly disturbs values of the parameters of individuals (parents) to generate individuals (offspring) in the neighborhood of the parents. The second genetic operator is the crossover, which is not required for the evolution to work, yet it can speed up the process when it is properly defined.

The last step in the EA loop is succession, which decides which individuals from the $P \cup Candidates$ will survive to the next iteration. There are many ways to concertize succession, reproduction and genetic operators. Here the basic but frequently used in practice concretizations will be described.

The reproduction can be implemented as a tournament. The size of the tournament is a user-defined parameter, but typically a tournament of size two is used. In that case, two individuals are randomly drawn (with replacement) from the population. Their objective functions are compared and the better of them is put into the resulting population. In tournament selection, the reproduction probability $p$ of an individual not only depends on its rank $r$ but also on its chance of being selected for the tournament: $p\left(r\right) = \frac{1}{\mu^2}\left(\left(\mu - r + 1\right)^2 - \left(\mu - r\right)^2\right)$.

The mutation is usually implemented by adding Gaussian noise to the features of randomly selected individuals. This approach forces users of the algorithm to provide two parameters: the mutation strength $\sigma$ and mutation probability, which can be perceived as the expected ratio of mutated features.

If a crossover operator is used, usually it is performed before mutation, i.e. the result of the crossover is mutated. One of the forms of that operator is called uniform crossover. In that approach, two individuals (parents) are randomly selected from $Tmp$. Then for each feature, it is randomly decided which parent will be the donor of the feature. As a result, the offspring has about half of the features from the first parent and the rest are from the other parent.

The simplest succession is called "generative". It just replaces $P$ with $Candidates$, i.e. the results of the genetic operations become a current population in the next iteration of the algorithm. The weakness of this approach is that it is possible that a very good individual from population $P$ is forgotten, which slows down the algorithm that will try to rediscover the lost solution. There is no such defect in elitist succession, where $k$ best solutions from $P$ are preserved for the next iteration. The elite size $k$ should be very small to maintain the ability of EA to explore the search space.

In the early days [72], the population size of EA was set between 50 and 100 individuals. It was also popular to set it as a function of problem complexity [73]. Later it was noticed that population size should not only be dependent on problem complexity but also it should change during evolution [74].

The problem of setting population size, mutation strength and probabilities of mutation and crossover is the main disadvantage of classical EA. As an advantage, we can consider that EA is very simple to implement and it has low computational overhead. The more important advantage of EA is that it is quite easy to create problem-specific genetic operators that can bring some domain knowledge to the algorithm.

## 3.3. Evolution strategies

Evolution strategies (ESs) are a German development introduced by Rechenberg in the sixties and further developed by Schwefel; therefore the first publications were in German. The first English paper was about an application of ES [75], then there was a book by Schwefel [76]. Here, a two-membered ES, also called (1+1)-ES, will be discussed. This strategy was described by Schwefel [77] as "the minimal concept for an imitation of organic evolution". It is the first algorithm with automatic adaptation of mutation strength. Its pseudocode is provided in Algorithm 4.

**Algorithm 4:** Pseudocode of two-membered evolution strategy ((1+1)-ES)

```
Input: Initial solution $m$; initial mutation strength $\sigma$; adaptation interval $ai$
Output: The best solution found $BestSoFar$
$Obj \leftarrow$Evaluate($m$);
$BestSoFar \leftarrow \emptyset$;
$SuccessMemory \leftarrow \emptyset$;
$iteration = 1$;
while StopCriterionNotMet() do
    $Mutant \leftarrow m + \sigma \cdot N(0,1)$;
    $MutObj \leftarrow$Evaluate($Mutant$);
    if $MutObj \leq Obj$ then
        AddToSuccessMemory(TRUE);
        $m \leftarrow Mutant$;
        $Obj \leftarrow MutObj$;
        $BestSoFar \leftarrow Mutant$;
    else
        AddToSuccessMemory(FALSE);
    end
    if $iteration \% ai = 0$ then
        if SuccessRatio($SuccessMemory$, ai) $> 1/5$) then
            $\sigma \leftarrow 1.22 \cdot \sigma$;
        end
        if SuccessRatio($SuccessMemory$, ai) $< 1/5$) then
            $\sigma \leftarrow 0.82 \cdot \sigma$;
        end
    end
    $iteration = iteration + 1$
end
return BestSoFar;
```

The strategy creates a mutant by adding a vector of normally distributed random numbers to the current solution. The strength of the mutation is controlled by $\sigma$ parameter, which is also called mutation step size. If the mutant is not worse than the current solution $m$, it replaces $m$ and the success of the strategy is recorded. Otherwise, a failure is recorded. After every $ai$ iteration, $\sigma$ is updated by the use of what is known as the one-fifth success rule. $\sigma$ is increased if the strategy success ratio in the last $ai$ iterations is greater than $1/5$. If the success ratio is less than $1/5$, $\sigma$ is decreased. The $1/5$ success rule is a result of the theoretical investigations on two simple objective functions (quadratic and corridor model). Other parameters are selected experimentally.

Simplicity, low computational overhead and the auto-adaptation mechanism are the advantages of (1+1)-ES. The disadvantage is that the strategy gets relatively easily stuck in local optima. Another disadvantage is that it requires the user to provide initial mutation strength and adaptation interval values.

There are many more variants of ES — interested readers are referred to survey papers, e.g. [78, 79].

### 3.3.1. Covariance Matrix Adaptation Evolution Strategy

The Covariance Matrix Adaptation Evolution Strategy (CMA-ES) [48] is a very successful algorithm. Most of the algorithms listed in Section 3.1 exploited the concept of Covariance Matrix Adaptation (CMA). The pseudocode of CMA-ES is provided in Algorithm 5.

**Algorithm 5:** Pseudocode of Covariance Matrix Adaptation Evolution Strategy (CMA-ES)

```
Input: Initial solution $m$; initial step size $\sigma$
Output: The best solution found $BestSoFar$
$BestSoFar \leftarrow \emptyset$;
$\mu, \lambda, C, p_c, p_\sigma \leftarrow$SetInitialValues();
while StopCriterionNotMet() do
    foreach $i \in 1, \ldots, \lambda$ do
        $P[i] \leftarrow$ SampleMultivariateNormal($m$, $\sigma^2 C$ );
    end
    $Objective \leftarrow$EvaluatePopulation($P$);
    $O \leftarrow$The$\mu$BestSol( $P, Objective$ );
    $BestSoFar \leftarrow$GetTheBestSol($BestSoFar, O, Objective$);
    $mNew \leftarrow$UpdateM(O);
    $p_\sigma \leftarrow$UpdateIsotropicEvolutionPath($p_\sigma$, $\sigma^{-1}C^{-0.5}(mNew - m)$);
    $p_c \leftarrow$UpdateAnisotropicEvolutionPath($p_c$, $\sigma^{-1}(mNew - m)$, $\|p_\sigma\|$);
    $C \leftarrow$UpdateC($C, p_c, \sigma, m, O$);
    $\sigma \leftarrow$UpdateSigma($\sigma$, $\|p_\sigma\|$);
    $m \leftarrow mNew$;
end
return BestSoFar;
```

Unlike classical EA, CMA-ES does not maintain the whole population of individuals. It samples the neighborhood of point $m$ using multivariate normal distribution. The shape of the distribution ellipsoid is controlled by the covariance matrix $C$, while mutation strength is controlled by step size $\sigma$. Using the covariance matrix allows CMA-ES to capture relationships between features. Another benefit is that it also makes the algorithm rotation invariant, i.e. the performance on a given objective function is the same as on its rotated version. The number of samples $\lambda$ generated in each iteration of CMA-ES is set to $4 + \lfloor 3 \cdot \log(N) \rfloor$ by default. This is a very small number when compared to the population size of classical EA, e.g., CMA-ES uses only 18 samples per generation for 120 dimensional problems, whereas classical EA uses 1200 samples or more.

The objective function values calculated for all samples are used only to select and rank $\mu$ best solutions. CMA-ES uses $\mu = \lfloor \lambda/2 \rfloor$ by default. The resulting solutions ($O$) are used to calculate a new distribution mean $m$ for the next iteration. $m$ is moved towards better solutions by setting it to a weighted mean of $O$. As a result, $m$ creates a path in the search space, which is accumulated in $p_c$ and $p_\sigma$ and used to update $C$ and $\sigma$. The covariance matrix $C$ is updated in a way to maximize the probability of making successful steps: $\arg\max\left(P\left(\frac{\mathbf{O_i}-\mathbf{m}}{\sigma}|C\right)\right)$, for all $i = 1, \dots, \mu$. The step size $\sigma$ is controlled by a heuristic called cumulative step-size adaptation. The $\sigma$ is increased when $p_\sigma$ is over the expected value and decreased otherwise. This means that $\sigma$ is increased when $m$ moves down the slope of the landscape of the objective function (shift of $m$ is in more or less one direction) and decreased when the algorithm is near the optimum, where the cumulative path of $m$ is small.

CMA-ES is often considered to be a parameter-less algorithm. In fact, for more complicated problems without standardized ranges of features it is required to set initial sigma to about $0.25 \cdot (upperbound - lowerbound)$. For some problems, changing the default $\lambda$ is also beneficial. Nevertheless, in many cases CMA-ES may be treated as a parameter-less approach which makes it easy to use. Another advantage of CMA-ES is that it is able to capture interactions between features and also adapts mutation strength. It is also invariant to many types of transformations of the objective function (e.g. rotation). When it comes to defects, the base algorithm has polynomial computational complexity. It is not good at crossing saddles, i.e. it will not go to a neighboring global optimum if it locates a wide enough local optimum. When it finds such an optimum it starts to reduce step size to locate it more accurately. From that point, it is not possible to find another optimum even if we wait an infinite amount of time.

## 3.4. Differential evolution

Differential evolution (DE) [51] is a simple but very efficient global optimization method. It was the second most exploited concept by the algorithms listed in Section 3.1. The pseudocode of DE is provided in Algorithm 6.

In classical DE, three distinct individuals ($Base$, $R_1$, $R_2$) are randomly taken from the current population ($P$) and used to generate a mutant. The mutant is a result of the sum of a scaled difference between random individuals and the base individual. The scaling factor $F$ is the parameter of the algorithm. It is usually set to 0.8. The mutant is crossed over with the i-th individual which yields a candidate solution ($Candidate$). The objective function value of the candidate is compared with the objective function value of the i-th individual. When it is not worse, the candidate is included in the next generation. Classical DE uses binomial crossover (bin). In that approach, for each feature of an individual, a standard uniform random number is generated and compared with $CR$, which is a user-specified parameter ($CR \in (0, 1\rangle$). If the generated number is smaller than $CR$, the feature is taken from the mutant, otherwise it is taken from the i-th individual. The $CR$ value is typically set in a range from $0.5$ to $0.9$, whereas setting $CR = 1$ disables crossover.

In classical DE, the exploration-exploitation balance is shifted towards exploration; however, it is easy to change the balance shift towards exploitation. The simplest approach to do so is to use the best individual from the current population as a base individual. Of course, many other strategies of base selection and differential mutation have been proposed to make the balance better. Many of these strategies can be discerned by the naming convention that is used for DE. In that convention, the name

**Algorithm 6:** Pseudocode of classical differential evolution (DE)

```
Input: Initial population $P$; scaling factor $F$; crossover rate $CR$
Output: The best solution found $BestSoFar$
$Objective \leftarrow$EvaluatePopulation($P$);
$BestSoFar \leftarrow \emptyset$;
while StopCriterionNotMet() do
    foreach $i \in 1, \dots, PopSize$ do
        $Base \leftarrow$ RandomIndividual($P$);
        $R_1 \leftarrow$ RandomIndividual($P$);
        $R_2 \leftarrow$ RandomIndividual($P$);
        $Mutant \leftarrow Base + F \cdot (R_1 - R_2)$;
        $Candidate \leftarrow$ BinomialCrossover($Mutant$, $P[i]$, $CR$);
        $Obj \leftarrow$Evaluate($Candidate$);
        if $Obj \leq Objective[i]$ then
            $PNew[i] \leftarrow Candidate$;
            $BestSoFar \leftarrow Candidate$;
            $ObjectiveNew[i] \leftarrow Obj$;
        else
            $PNew[i] \leftarrow P[i]$;
            $ObjectiveNew[i] \leftarrow Objective[i]$;
        end
    end
    $P \leftarrow PNew$;
    $Objective \leftarrow ObjectiveNew$;
end
return BestSoFar;
```

of the DE version has the form: DE/base individual selection/number of pairs of individuals used for mutation/type of crossover. Therefore, classical DE is also named DE/rand/1/bin.

For problems where neighboring features are related, e.g., for designing a semiconductor double-chirped mirror [80], it is better to use a crossover that combines continuous fragments of solutions rather than picking features randomly from both parents. For those kinds of problems, exponential crossover (exp) could be better. Exp starts from the random selection of an index of the feature. From that index, consecutive features are copied from the mutant while a standard uniform random number is less than $CR$. When copying reaches the last feature it is continued from the first feature.

The population size for DE is set to $10N$ as an initial guess, but it shrinks to $0.5N$ for high-dimensional problems [81]. Generally, the optimal population size is problem-dependent [82].

DE has many advantages and the most important of them is auto-adaptation of mutation strength. The DE approach is also simple to implement and its computational overhead is very low. The weakness of the early versions of DE was that their users had to provide $F$ and $CR$ parameters. This weakness was eliminated by introducing additional heuristics to set the parameters, e.g. [83, 84].

There are plenty of papers connected with DE. Interested readers are referred to overview papers [85, 86, 87].

## 3.5. Particle swarm optimization

Particle swarm optimization (PSO) takes its origin from animal behavior, i.e. swarming and flocking, and was first presented by Kennedy and Elberhart in 1995 [67]. Today, it is still considered a good method for many optimization problems and is often found as a top-ranked algorithm in many competitions (see Table 4). PSO maintains a single group of individuals (swarm of particles) and their velocities. During optimization, the positions of these particles are adjusted according to appropriate speed vector combinations. The pseudocode of PSO is provided in Algorithm 7.

**Algorithm 7:** Pseudocode of Particle Swarm Optimization (PSO)

```
Input: Swarm size $\lambda$, Problem dimension $N$, Particle acceleration parameters: $\alpha$, $\beta$, $\gamma$
Output: The best solution found $BestSoFar$
$Particle, ParticleVelocity \leftarrow$ InitialSwarm($\lambda$);
$ParticleBestSoFar \leftarrow Particle$ ;
$BestSoFar \leftarrow \emptyset$ ;
while StopCriterionNotMet() do
    foreach $i \in 1, \dots, \lambda$ do
        $r_1 \leftarrow \beta \cdot$RandomUniform($N$);
        $r_2 \leftarrow \gamma \cdot$RandomUniform($N$);
        $ParticleVelocity[i] \leftarrow \alpha ParticleVelocity[i] + r_1 \circ$
          $(ParticleBestSoFar[i] - Particle[i]) + r_2 \circ (BestSoFar - Particle[i])$;
        $Particle[i] \leftarrow Particle[i] + ParticleVelocity[i]$ ;
        if Evaluate($Particle[i]$) $\leq$ Evaluate($ParticleBestSoFar[i]$) then
            $ParticleBestSoFar[i] \leftarrow Particle[i]$ ;
            if $BestSoFar = \emptyset$ or Evaluate($Particle[i]$) $\leq$ Evaluate($BestSoFar$) then
                $BestSoFar \leftarrow Particle[i]$ ;
            end
        end
    end
end
return BestSoFar;
```

The main hyperparameter influencing the behavior of PSO is $\lambda$, which defines the number of individuals in the swarm. The particle acceleration parameters are $\alpha$, which specifies the proportion of the previous velocity to retain, $\beta$ and $\gamma$, which define how much information is taken from the personal and the global best accordingly. All three of these parameters have an important role in the individual swarm adjustment and influence the overall performance [88]. Thus, in some cases, the meta-optimization layer is used to find the proper values for these hyperparameters [89].

The PSO algorithm starts with initializing each individual particle with a random value, based on the task's dimension $N$ and some prior assumptions about the distribution of parameters. The most common setting is to use a uniform distribution over the search space $\mathbb{R}^N$, taking into account search space boundaries. In the same random way, the initial velocity of particles is set.

There are some variations of PSO based on what information is used to calculate the new particle velocity. Here, for the sake of simplicity, only the global best-so-far solution $BestSoFar$ and the particle's personal $ParticleBestSoFar[i]$ are used. However, there are many approaches based on topology or a social network for the particles, e.g. taking information only from two closest neighbors, forming cliques, etc.

The main algorithm loop goes through all of the particles in each iteration. The number of overall iterations can be specified in the case of a constrained budget or the algorithm can be stopped when the satisfied condition is reached. The processing of each particle in the swarm starts with the adjustment — a direct mutation of the particle. This adjustment changes the position of the particle in the search space based on the previous velocity, the best found solution of this particle and the global best of the whole swarm. How much of the best previous values are mixed into the particle velocity is controlled by input parameters and random values drawn from the uniform distribution. After the particle is updated, its parameters are evaluated in order to update best-so-far values for the current particle and for the whole swarm. The best value for the whole swarm is the solution returned by the algorithm.

## 3.6. Addressing the specificity of DL optimization problems

The optimization tasks that exist in DL were defined in Section 2.1. These tasks can be divided into two groups. The first group contains problems like optimizing hyperparameters or generating adversarial examples, i.e. highly multidimensional problems. The second group contains problems like optimizing the hyperparameters of the neural network, i.e. low dimensional optimization problems with expensive evaluation of the objective function.

The benchmarks mentioned in Section 3.1 assumed neither expensive evaluation nor very high-dimensionality. The largest dimensionality used by the benchmarks is 100 and the budget is quite large — $10000N$ for CEC. Therefore, the winners of discussed benchmarks can perform poorly in the setup required by DL. Fortunately, the problem of expensive objective functions and the problem of high dimensionality have been addressed by groups of researchers for several years.

### 3.6.1. Large-scale global optimization

There are variants of benchmarks and competitions that put stress on highly multidimensional problems. One of them is Large-Scale Global Optimization (LSGO) organized during CEC conferences. The last two realizations of the competition were during CEC'2013 and CEC'2018. Both of them used functions described in [90]. The benchmark consists of 15 functions that are solved in 1000 dimensions. The maximal budget is set to $3 \cdot 10^6$.

The second benchmark was used in the Special Issue of Soft Computing (SOCO) Journal (SOCO'2011 [91]). The benchmark defines 19 functions of dimensionality from 50 to 1000. The budget is set to $5000N$.

We analyzed the results of CEC'2013 and SOCO'2011 and for the best algorithms we identified a root concept that was extended or just used. The results of this survey are presented in Table 5.

Table 5. The best algorithms according to large-scale optimization benchmarks: CEC'2013 and SOCO'2011 together with their root concepts

| | CEC'2013 | | SOCO'2011 | |
|---|---|---|---|---|
| # | alg. name | root | alg. name | root |
| 1 | MOS-CEC2013[92] | DE[51] | MOS-SOCO2011[93] | DE[51] |
| 2 | MA-SW-Chains[94] | EA[71] | jDElscop[95] | DE[51] |
| 3 | 2S-Ensemble[96] | EDA[97] | GaDE[98] | DE[51] |

According to Table 5, the best algorithm is MOS, but in CEC'2018 MOS was beaten by SHADE-ILS [99] which is based on DE. It can be observed that most of the successful algorithms use a concept of DE, although one algorithm uses EA and one EDA. EDA stands for the Estimation of Distribution Algorithm. It is a kind of ES that explicitly calculates sampling distribution (like CMA-ES). Nowadays there is a whole family of algorithms that use the EDA concept.

Classical CMA-ES cannot be used for LSGO because of its computational complexity, but there are many modifications that do not require inversion of the covariance matrix, e.g. [100]. There are also survey papers on large-scale variants of CMA-ES, e.g. [101].

It is worth mentioning that there are online services that facilitate comparisons of LSGO methods. One of them is Midas [102], which uses four benchmarks. The newest service is TFLSGO [103], which uses the CEC'2013 LSGO benchmark and the regular benchmark from CEC'2017.

### 3.6.2. Computationally expensive numerical optimization

The problem of an expensive objective function was also noticed by competition organizers. One of the competitions is the CEC'2015 special session on bound constrained single-objective computationally expensive numerical optimization [104]. The benchmark consists of 15 problems for dimensions 10 and 30, with a budget limited to $50N$ evaluations. The best methods from the benchmark and their root concepts are listed in Table 6.

Table 6. The best algorithms according to the CEC'2015 benchmark for computationally expensive numerical optimization, together with their root concepts

| # | alg. name | root |
|---|---|---|
| 1 | MVMO[105] | MVMO[106] |
| 2 | TunedCMAES[107] | CMA[48] |

According to the Table 6 MVMO is the winner. MVMO stands for Mean Variance Mapping Optimization. It is a variant of EA with a single parent-offspring scheme. MVMO normalizes values of all parameters of the solution into a range $\langle 0, 1\rangle$. The mutation is performed by the use of a special mapping function whose shape depends on the mean and variance of the $n$-th best solutions found so far.

The CMA-ES concept is also good for expensive optimization. It is quite intuitive because even in its default setup, CMA-ES uses small populations. It is also worth mentioning that there are even versions of CMA-ES that generate only one solution per generation, e.g. [108].

#### 3.6.3. Robustness of the solutions

All metaheuristics mentioned in this chapter are global optimization algorithms. However, we have to bear in mind that they neither guarantee to find the global optimum nor find it for complicated objective functions. For large search spaces and a limited budget, the algorithms have a chance to search only a fragment of the search space. The selection of the fragment and the sampling density in that fragment depends on the heuristics employed in each algorithm. Therefore, each metaheuristic introduces a search bias, i.e. results found by different methods may have different properties. One of such interesting properties of population-based EAs is a phenomenon called "survival of the flattest" [109]. The phenomenon is manifested by a preference of individuals whose objective function values are less sensitive to mutations. This means that a narrow optimum is not attractive for evolution because it is hard to generate a good mutant that hits the optimum. On the other hand, a worse but wide optimum is attractive because most individuals will be of good quality. It has even been shown that when classical EA is started in global optimum, it moves to the local but wider optimum where mutation strength is increased [110]. This feature of population-based EAs may be beneficial in DL tasks because EAs naturally generate models that are more robust [111], which can improve their generalization abilities. The possible benefits are especially visible for protecting the classifier from adversarial attacks.

## 4. Trends and perspectives

### 4.1. Current metaheuristic applications

One of the successful applications of metaheuristics in combination with DL models is connected with training FFNNs using the PSO algorithm [112]. The authors claim that PSO trains FFNNs with a performance similar to gradient-based methods, arguing that the latter suffer from local optima and that these algorithms are strongly dependent on problem-specific settings. The experiments were conducted on numerous tasks — small real-life data prediction problems along with some artificial ones. The results show the superiority of PSO compared to a gradient method and EA in cases where a high number of local minima is known to exist. One limitation, however, is that the experiments used a very simple network — one hidden layer with 2-8 neurons, up to seven inputs and a single output. The authors claim that in such a setting the models have minimal complexity which makes the training task difficult. While the reasoning in this publication was sound, it may seem inadequate to even consider it in connection with DL techniques. However, it presents a historical view on how PSO was adopted over a decade ago. Additionally, this work received quite a bit of attention in references from other researchers.

PSO and backpropagation were also compared as training algorithms for neural networks in [113]. Still, the used network with two inputs, three hidden units and a single output is not even close to the usage of neural networks nowadays. More recent results on training RNN models are presented in the paper [114]. The authors claim to get comparable or even better results using PSO in comparison to GD methods. However, there is no information about the architecture of the network, so the reader cannot get information about the number of final parameters that were trained.

An interesting approach of using a hybrid method — PSO with a gradient-based optimizer — was presented in [115]. The idea of this hybrid is to combine the best aspects of two worlds: PSO's global search and the benefits of a quick local search utilizing GD. The hybrid was applied to FFNN training. In the initial phase, the PSO algorithm is used with random distribution in the problem space. Then, after some iterations, the method switches to the backpropagation-based algorithm. This phase is crucial and constitutes a key aspect of the proposed algorithm. The transition is performed when the best fitness value in the history of all particles has stopped improving for a defined number of iterations. At this point, the authors assumed that the particles lost the ability to find a better solution and that GD can start exploiting the local area. Still, the solution was tested on a rather toy-sized network. The authors selected three basic problems for performing their experiments: parity bits, function approximation ($sin(2x)e^{-x}$) and the Iris classification problem. The results show that in less CPU time, the proposed algorithm can get higher training accuracy than GD or PSO alone.

Another metaheuristic which has been successfully applied within the DL ecosystem is called Natural Evolution Strategies (NES) [116, 117]. NES-based algorithms are within the ES group, which was presented in Section 3.3. These methods utilize the natural gradient to update distribution parameters. For large search spaces, variants of NES do not use full covariance matrices, e.g. they limit $C$ to the diagonal only. These desirable properties have been utilized in [117], where a version of NES called Separable-NES is applied to find a controller for non-Markovian double pole balancing, which is a reinforcement learning task solved using neuroevolution. More recent work in the field of deep reinforcement learning has shown that NES algorithms may be a viable solution to optimization problems within that domain [118]. The authors show that such methods parallelize well and provide significant performance gains in comparison to traditionally used approaches, and are better suited to low precision hardware architectures commonly used for DL training. Furthermore, it may be beneficial to incorporate non-differentiable elements in the model's architecture which would be difficult in combination with gradient-based methods.

In the domain of adversarial examples, there are also cases in which NES approaches have been applied. One such example is [119], where the authors present an attack method based on NES strategies and test it in multiple threat models. DE has also been utilized for similar purposes in the one pixel attack [120], which is a method for generating adversarial examples using a single pixel.

### 4.2. General discussion

As large scale optimization of non-convex loss functions for neural networks are commonly handled by gradient methods and backpropagation, many successful applications of this tandem propel such a choice. Thus, works that shed some light on the theory or practical aspects of this approach are useful and getting attention.

In a 2014 paper [29], the authors argued that the proliferation of saddle points in the loss function landscape is more influential on the whole training process than finding the global optimum itself. From the practical point of view, finding a local optimum using existing gradient methods is quite easy and presents satisfactory results. Thus, to improve the application of existing approaches, the authors introduced a new method — a saddle-free Newton approach, that can rapidly escape high dimensional saddle points, unlike GD or quasi-Newton methods.

Historically, neural networks have been trained using methods originating from GD, which are local in their nature, and due to this, substantial research efforts have been put into determining if such practice is sufficient or if searching for the global minimum would lead to obtaining significantly better results. An intriguing property of multi-layer, large neural networks is that while training with SGD, many local minimas can be discovered. However, doing multiple experiments consistently give very similar performance on a test set. This property was explained in work [121], with both a theoretical and experimental approach. The authors used the random matrix theory in order to verify their hypothesis, in which besides the local minima equivalence they prove two peculiar properties. The first claims that the probability of finding a high-valued local minimum is non-zero for small-sized networks and decreases quickly with network size. Second, while struggling to find the global optimum (as opposed to the many good local ones) the generalization of the model is sacrificed. Thus, in practice it is not useful. From the empirical standpoint, the authors conducted experiments with the SGD method over the scaled MNIST data set.

More recent work on this subject shows on one hand that spurious local minima are common in (at least) some types of neural networks [122]. On the other hand, many papers argue that the solutions to which the gradient methods converge are close to or even equal to the globally optimal in the case of other architectures [123, 124, 125, 126]. Considering all the assumptions made by these works we may conclude that the question in the general case seems to be largely open.

The argument about whether optimizing parameters of a given model requires solving a global or only a local optimization task has yet another twist when considering metaheuristic algorithms. Due to the survival of the flattest effect (described in Section 3.6.3), in practice the solutions found using such methods are not global optima but rather robust local optima. An intuition is that narrow peaks of the objective function are vulnerable to negative effects of mutations, i.e., even a small mutation applied to a working point may "push it" out of the peak. Such an effect could explain why some researchers report (e.g. the authors of [118], as discussed in Section 4.1) that solutions found using metaheuristics are more stable in comparison to the ones found using gradient methods.

Another perspective on this matter could be attained by considering the phenomenon of adversarial examples. One possible explanation of the existence of these examples is called evolutionary stalling [127] and seems very similar to the survival of the flattest effect. The intuition given by the authors is that as gradient optimization of a classification model is being performed, the gradients of the correctly classified data points are close to zero and effectively stop contributing to the objective function value. This results in a situation in which most of the training points are close to the decision boundary in the model's hyperspace and thus the model is susceptible to adversarial attacks. The authors of [127] try to overcome this effect by batching the gradients, so they still apply a gradient-based method.

A somewhat similar idea, although formulated from different grounds, has been given by the authors of [15] who propose reformulating the optimization task from Eq. 1 into the following saddle point problem:

$$\theta^* = \arg\min_{\theta \in \Theta} \rho(\theta) \tag{14}$$

$$\rho(\theta) = \mathbb{E}_{(\mathbf{x},y)\sim D} \max_{\mathbf{r}\in S} \mathcal{L}(\theta, \mathbf{x}+\mathbf{r}, y) \ , \tag{15}$$

where the inner $\rho$ function performs adversarial maximization — i.e. tries to find a worst case perturbation $\mathbf{r} \in S$, with $S$ being the subset of perturbations allowed by the assumed threat model. This means that the model parameters must be such that the loss function remains low within a region around vector $\mathbf{x}$. One interesting question regarding this approach and the one from [127] is whether simply applying a metaheuristic would yield similar results.

## 5. Conclusions

From a high-level perspective, metaheuristic optimization and DL appear to be completely different worlds. On one hand, we have a research domain which is in a large part driven by the concept of black-box optimization, i.e. tasks in which little or nothing is known about the structure and properties of the goal function. This is reflected in the way competitions like CEC or BBOB are organized and what kinds of challenges they present to participants. This is also visible in metaheuristic algorithms, which are usually designed with few assumptions about the objective function being optimized. When these methods are evaluated, often very complicated and unpredictable search spaces are utilized. However, the dimensionality of considered problems is not very large compared to the dimensionality of problems emerging in the DL field.

On the other hand, there is DL, a very large ecosystem with countless types of model architectures and a handful of practically applied optimization methods. Although the models implemented by DL architectures are complicated nonlinear functions, they usually have a well-organized structure and are built up from simple, well-defined and differentiable elements. The most popular optimization algorithms deeply rely on these assumptions as they need to be able to perform GD. This means that in some cases (e.g. LSTMs or Highway Networks), compromises have to be made in order to make training possible. Furthermore, larger models seem to be easier to train, as multiple researchers report that over-parametrization is an important aspect which diminishes the problems connected with low-quality local minima [121, 123]. In some sense, the current trends in DL seem to represent a different paradigm than the one from metaheuristic research: instead of trying to create an algorithm which is able to train any given model structure, the structure itself is modified to match the algorithm. Yet the field still struggles with problems which might be connected with overfitting or finding optimization solutions that are simply not robust enough.

The relatively few works that fall into the intersection of these two worlds bring some interesting observations. The published positive results show that it often might be beneficial to apply metaheuristics to DL problems. First of all, the possibility of extending the models with complicated, non-differentiable elements is an advantage. Moreover, a clever implementation of optimizing methods may also result in improved overall training performance. It is also worth noticing that model robustness seems to be positively affected by estimating parameters using metaheuristic algorithms, which seems to be important for demanding real-world environments. As demonstrated by the phenomenon of adversarial examples, the current generation of DL models often demonstrates unstable behavior. In this light, exploring the possibilities and potential benefits that metaheuristics can bring into the DL environment becomes a very interesting and important research subject.

## References

[1] Szegedy C, Liu W, Jia Y, Sermanet P, Reed S, Anguelov D, Erhan D, Vanhoucke V, Rabinovich A. Going deeper with convolutions. In: Proceedings of the IEEE conference on computer vision and pattern recognition. 2015 pp. 1–9. doi:10.1109/CVPR.2015.7298594.

[2] He K, Zhang X, Ren S, Sun J. Deep residual learning for image recognition. In: Proceedings of the IEEE conference on computer vision and pattern recognition. 2016 pp. 770–778. doi:10.1109/CVPR.2016.90.

[3] Long J, Shelhamer E, Darrell T. Fully convolutional networks for semantic segmentation. In: Proceedings of the IEEE conference on computer vision and pattern recognition. 2015 pp. 3431–3440. doi:10.1109/CVPR.2015.7298965.

[4] Pohlen T, Hermans A, Mathias M, Leibe B. Fullresolution residual networks for semantic segmentation in street scenes. *arXiv preprint:1611.08323*, 2017.

[5] Mikolov T, Sutskever I, Chen K, Corrado GS, Dean J. Distributed representations of words and phrases and their compositionality. In: Advances in neural information processing systems. 2013 pp. 3111–3119.

[6] Mikolov T, Grave E, Bojanowski P, Puhrsch C, Joulin A. Advances in Pre-Training Distributed Word Representations. In: Proceedings of the International Conference on Language Resources and Evaluation (LREC 2018). 2018 .

[7] Bowman SR, Vilnis L, Vinyals O, Dai AM, Jozefowicz R, Bengio S. Generating sentences from a continuous space. *arXiv preprint:1511.06349*, 2015.

[8] Bahdanau D, Cho K, Bengio Y. Neural machine translation by jointly learning to align and translate. *arXiv preprint:1409.0473*, 2014.

[9] Moosavi-Dezfooli SM, 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. 2016 pp. 2574–2582.

[10] Goodfellow I, Pouget-Abadie J, Mirza M, Xu B, Warde-Farley D, Ozair S, Courville A, Bengio Y. Generative adversarial nets. In: Advances in neural information processing systems. 2014 pp. 2672–2680.

[11] Papernot N, McDaniel P, Goodfellow I, Jha S, Celik ZB, Swami A. Practical black-box attacks against machine learning. In: Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security. ACM, 2017 pp. 506–519.

[12] Carlini N, Wagner D. Towards Evaluating the Robustness of Neural Networks. In: 2017 IEEE Symposium on Security and Privacy (SP). 2017 pp. 39–57. doi:10.1109/SP.2017.49.

[13] Carlini N, Wagner D. Adversarial Examples Are Not Easily Detected: Bypassing Ten Detection Methods. In: Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security, AISec ’17. ACM, New York, NY, USA. ISBN 978-1-4503-5202-4, 2017 pp. 3–14. doi:10.1145/3128572.3140444.

[14] Athalye A, Carlini N, Wagner D. Obfuscated Gradients Give a False Sense of Security: Circumventing Defenses to Adversarial Examples. *arXiv preprint:1802.00420*.

[15] Madry A, Makelov A, Schmidt L, Tsipras D, Vladu A. Towards Deep Learning Models Resistant to Adversarial Attacks. 2017. doi:10.1227/01.NEU.0000255452.20602.C9. 1706.06083.

[16] LeCun Y, Bottou L, Bengio Y, Haffner P. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 1998. **86**(11):2278–2324.

[17] Goodfellow I, Bengio Y, Courville A. Deep Learning. MIT Press, 2016.

[18] Hochreiter S, Schmidhuber J. Long short-term memory. *Neural computation*, 1997. **9**(8):1735–1780.

[19] Xiao L, Bahri Y, Sohl-Dickstein J, Schoenholz SS, Pennington J. Dynamical Isometry and a Mean Field Theory of CNNs: How to Train 10,000-Layer Vanilla Convolutional Neural Networks. *arXiv preprint:1806.05393*, 2018.

[20] Rumelhart DE, Hinton GE, Williams RJ. Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[21] Abadi M, Barham P, Chen J, Chen Z, Davis A, Dean J, Devin M, Ghemawat S, Irving G, Isard M, et al. Tensorflow: a system for large-scale machine learning. In: OSDI, volume 16. 2016 pp. 265–283.

[22] Paszke A, Gross S, Chintala S, Chanan G, Yang E, DeVito Z, Lin Z, Desmaison A, Antiga L, Lerer A. Automatic differentiation in pytorch. In: 31st Conference on Neural Information Processing Systems (NIPS 2017). 2017 .

[23] Srivastava RK, Greff K, Schmidhuber J. Highway networks. *arXiv preprint:1505.00387*, 2015.

[24] Chung J, Gulcehre C, Cho K, Bengio Y. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint:1412.3555*, 2014.

[25] Bengio Y, Louradour J, Collobert R, Weston J. Curriculum Learning. In: Proceedings of the 26th Annual International Conference on Machine Learning, ICML '09. ACM, New York, NY, USA. ISBN 978-1-60558-516-1, 2009 pp. 41–48. doi:10.1145/1553374.1553380.

[26] Bottou L. Online Algorithms and Stochastic Approximations. In: Saad D (ed.), Online Learning and Neural Networks. Cambridge University Press, Cambridge, UK, 1998. Revised, Oct 2012.

[27] Ruder S. An overview of gradient descent optimization algorithms. *CoRR*, 2016. **abs/1609.04747**.

[28] Darken C, Chang J, Z JC, Moody J. Learning Rate Schedules For Faster Stochastic Gradient Search. In: Neural Networks for Signal Processing [1992] II., Proceedings of the 1992 IEEE-SP Workshop. IEEE Press, 1992 pp. 3–12.

[29] Dauphin YN, Pascanu R, Gulcehre C, Cho K, Ganguli S, Bengio Y. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. In: Advances in neural information processing systems. 2014 pp. 2933–2941.

[30] Sutskever I, Martens J, Dahl G, Hinton G. On the importance of initialization and momentum in deep learning. In: International conference on machine learning. 2013 pp. 1139–1147.

[31] Duchi J, Hazan E, Singer Y. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 2011. **12**(Jul):2121–2159.

[32] Dean J, Corrado G, Monga R, Chen K, Devin M, Mao M, Ranzato M, Senior A, Tucker P, Yang K, Le QV, Ng AY. Large Scale Distributed Deep Networks. In: Pereira F, Burges CJC, Bottou L, Weinberger KQ (eds.), Advances in Neural Information Processing Systems 25, pp. 1223–1231. Curran Associates, Inc., 2012.

[33] Tieleman T, Hinton G. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. Technical report, 2012.

[34] Zeiler MD. ADADELTA: An Adaptive Learning Rate Method. *CoRR*, 2012. **abs/1212.5701**.

[35] Kingma DP, Ba JL. Adam: Amethod for stochastic optimization. In: Proc. 3rd Int. Conf. Learn. Representations. 2014 .

[36] Zhu C, Byrd RH, Lu P, Nocedal J. Algorithm 778: L-BFGS-B: Fortran subroutines for large-scale bound-constrained optimization. *ACM Transactions on Mathematical Software (TOMS)*, 1997. **23**(4):550–560.

[37] Levenberg K. A method for the solution of certain non-linear problems in least squares. *Quarterly of applied mathematics*, 1944. **2**(2):164–168.

[38] Marquardt DW. An algorithm for least-squares estimation of nonlinear parameters. *Journal of the society for Industrial and Applied Mathematics*, 1963. **11**(2):431–441.

[39] Papernot N, McDaniel P, Sinha A, Wellman MP. SoK: Security and Privacy in Machine Learning. In: 2018 IEEE European Symposium on Security and Privacy (EuroS P). 2018 pp. 399–414. doi:10.1109/EuroSP.2018.00035.

[40] Brendel W, Rauber J, Bethge M. Decision-Based Adversarial Attacks: Reliable Attacks Against Black-Box Machine Learning Models. In: International Conference on Learning Representations. 2018 .

[41] Glover F. Future Paths for Integer Programming and Links to Artificial Intelligence. *Comput. Oper. Res.*, 1986. **13**(5):533–549. doi:10.1016/0305-0548(86)90048-1.

[42] Wolpert DH, Macready WG. No Free Lunch Theorems for Optimization. *Trans. Evol. Comp*, 1997. **1**(1):67–82. doi:10.1109/4235.585893.

[43] Awad NH, Ali M, Liang J, Qu B, Suganthan PN. Problem definitions and evaluation criteria for the CEC 2017 special session and competition on real-parameter optimization. Technical report, Nanyang Technol. Univ., Singapore and Jordan Univ. Sci. Technol. and Zhengzhou Univ., China, 2016.

[44] COCO (COmparing Continuous Optimisers). `http://coco.gforge.inria.fr/`. Accessed:2018-02-22.

[45] Loshchilov I, Glasmachers T. Black Box Optimization Competition. `http://bbcomp.ini.rub.de/`. Accessed:2018-01-25.

[46] Suganthan PN. Shared documents. `http://web.mysites.ntu.edu.sg/epnsugan/PublicSite/Shared%20Documents`. Accessed: 2018-06-07.

[47] Kumar A, Misra RK, Singh D. Improving the local search capability of Effective Butterfly Optimizer using Covariance Matrix Adapted Retreat Phase. In: 2017 IEEE Congress on Evolutionary Computation (CEC). 2017 pp. 1835–1842. doi:10.1109/CEC.2017.7969524.

[48] Hansen N, Ostermeier A. Completely Derandomized Self-Adaptation in Evolution Strategies. *Evol. Comput.*, 2001. **9**(2):159–195. doi:10.1162/106365601750190398.

[49] Zhang G, Shi Y. Hybrid Sampling Evolution Strategy for Solving Single Objective Bound Constrained Problems. In: 2018 IEEE Congress on Evolutionary Computation, CEC 2018, Rio de Janeiro, Brazil, July 8-13, 2018. 2018 pp. 1–7. doi:10.1109/CEC.2018.8477908.

[50] Brest J, Maučec MS, Bošković B. Single objective real-parameter optimization: Algorithm jSO. In: IEEE Congr. Evol. Comput. 2017 pp. 1311–1318. doi:10.1109/CEC.2017.7969456.

[51] Storn R, Price K. Differential Evolution – A Simple and Efficient Heuristic for global Optimization over Continuous Spaces. *Journal of Global Optimization*, 1997. **11**(4):341–359. doi:10.1023/A:1008202821328.

[52] Stanovov V, Akhmedova S, Semenkin E. LSHADE Algorithm with Rank-Based Selective Pressure Strategy for Solving CEC 2017 Benchmark Problems. In: 2018 IEEE Congress on Evolutionary Computation (CEC). 2018 pp. 1–8. doi:10.1109/CEC.2018.8477977.

[53] Awad NH, Ali MZ, Suganthan PN. Ensemble sinusoidal differential covariance matrix adaptation with Euclidean neighborhood for solving CEC2017 benchmark problems. In: 2017 IEEE Congress on Evolutionary Computation (CEC). 2017 pp. 372–379. doi:10.1109/CEC.2017.7969336.

[54] Mohamed AW, Hadi AA, Fattouh AM, Jambi KM. LSHADE with semi-parameter adaptation hybrid with CMA-ES for solving CEC 2017 benchmark problems. In: 2017 IEEE Congress on Evolutionary Computation (CEC). 2017 pp. 145–152. doi:10.1109/CEC.2017.7969307.

[55] Jagodziński D, Arabas J. A differential evolution strategy. In: 2017 IEEE Congress on Evolutionary Computation (CEC). 2017 pp. 1872–1876. doi:10.1109/CEC.2017.7969529.

[56] Sallam KM, Elsayed SM, Sarker RA, Essam DL. Improved United Multi-Operator Algorithm for Solving Optimization Problems. In: 2018 IEEE Congress on Evolutionary Computation (CEC). 2018 pp. 1–8. doi:10.1109/CEC.2018.8477759.

[57] Hansen N. Benchmarking a BI-population CMA-ES on the BBOB-2009 Function Testbed. In: Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference: Late Breaking Papers, GECCO '09. ACM, New York, NY, USA. ISBN 978-1-60558-505-5, 2009 pp. 2389–2396. doi:10.1145/1570256.1570333.

[58] Nishida K, Akimoto Y. Benchmarking the PSA-CMA-ES on the BBOB noiseless testbed. In: Proceedings of the Genetic and Evolutionary Computation Conference Companion, GECCO 2018, Kyoto, Japan, July 15-19, 2018. 2018 pp. 1529–1536. doi:10.1145/3205651.3208297.

[59] Nguyen DM. Benchmarking a variant of the CMAES-APOP on the BBOB noiseless testbed. In: Aguirre HE, Takadama K (eds.), Proceedings of the Genetic and Evolutionary Computation Conference Companion, GECCO 2018, Kyoto, Japan, July 15-19, 2018. ACM, 2018 pp. 1521–1528. doi:10.1145/3205651.3208299.

[60] Belkhir N, Dréo J, Savéant P, Schoenauer M. Per Instance Algorithm Configuration of CMA-ES with Limited Budget. In: Proceedings of the Genetic and Evolutionary Computation Conference, GECCO '17. ACM, New York, NY, USA. ISBN 978-1-4503-4920-8, 2017 pp. 681–688. doi:10.1145/3071178.3071343.

[61] Vaneev A. Derivative-Free Optimization Method. `https://github.com/avaneev/biteopt`. Accessed:2018-02-22.

[62] Pitra Z, Bajer L, Holena M. Doubly Trained Evolution Control for the Surrogate CMA-ES. In: Parallel Problem Solving from Nature - PPSN XIV - 14th International Conference, Edinburgh, UK, September 17-21, 2016, Proceedings. 2016 pp. 59–68. doi:10.1007/978-3-319-45823-6_6.

[63] Audet C, Dennis J, Digabel S. Parallel Space Decomposition of the Mesh Adaptive Direct Search Algorithm. *SIAM Journal on Optimization*, 2008. **19**(3):1150–1170. doi:10.1137/070707518.

[64] Audet C, Dennis J. Mesh Adaptive Direct Search Algorithms for Constrained Optimization. *SIAM Journal on Optimization*, 2006. **17**(1):188–217. doi:10.1137/040603371.

[65] Wessing S. BBComp. `https://ls11-www.cs.tu-dortmund.de/staff/wessing/bbcomp`. Accessed:2018-02-22.

[66] Ulinski M, Zychowski A, Okulewicz M, Zaborski M, Kordulewski H. Generalized Self-adapting Particle Swarm Optimization Algorithm. In: Parallel Problem Solving from Nature - PPSN. 2018 doi:10.1007/978-3-319-99253-2_3.

[67] Kennedy J, Eberhart R. Particle swarm optimization. In: Proceedings of ICNN'95 - International Conference on Neural Networks, volume 4. 1995 pp. 1942–1948. doi:10.1109/ICNN.1995.488968.

[68] Holland JH. Adaptation in Natural and Artificial Systems. University of Michigan Press, Ann Arbor, MI, USA, 1975.

[69] Goldberg DE. Genetic Algorithms in Search, Optimization and Machine Learning. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1989. ISBN 0201157675.

[70] Goldberg DE. Real-coded Genetic Algorithms, Virtual Alphabets, and Blocking. *Complex Systems*, 1991. **5**.

[71] Michalewicz Z. Genetic Algorithms + Data Structures = Evolution Programs. Springer Verlag, Ann Arbor, MI, USA, 1992, 1994, 1996.

[72] De Jong KA. An Analysis of the Behavior of a Class of Genetic Adaptive Systems. Ph.D. thesis, Ann Arbor, MI, USA, 1975. AAI7609381.

[73] Goldberg DE, Deb K, Clark JH. Genetic Algorithms, Noise, and the Sizing of Populations. *Complex Systems*, 1992. **6**.

[74] Arabas J, Michalewicz Z, Mulawka J. GAVaPS-a genetic algorithm with varying population size. In: Proceedings of the First IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence. 1994 pp. 73–78 vol.1. doi:10.1109/ICEC.1994.350039.

[75] Klockgether J, Schwefel HP. Two-phase nozzle and hollow core jet experiments. In: Proc. 11th Symp. Engineering Aspects of Magnetohydrodynamics. Pasadena, CA: California Institute of Technology, 1970 pp. 141–148.

[76] Schwefel HP. Numerical Optimization of Computer Models. John Wiley & Sons, Inc., New York, NY, USA, 1981. ISBN 0471099880.

[77] Schwefel HPP. Evolution and Optimum Seeking: The Sixth Generation. John Wiley & Sons, Inc., New York, NY, USA, 1993. ISBN 0471571482.

[78] Bäck T, Hoffmeister F, Schwefel H. A Survey of Evolution Strategies. In: Belew R, Booker L (eds.), Proceedings of the Fourth International Conference on Genetic Algorithms. Morgan Kaufmann, San Francisco, CA, USA, 1991 pp. 2–9.

[79] Beyer HG, Schwefel HP. Evolution strategies – A comprehensive introduction. *Natural Computing*, 2002. **1**(1):3–52. doi:10.1023/A:1015059928466.

[80] Biedrzycki R, Arabas J, Jasik A, Szymański M, Wnuk P, Wasylczyk P, Wójcik-Jedlińska A. Application of Evolutionary Methods to Semiconductor Double-Chirped Mirrors Design. In: Bartz-Beielstein T, Branke J, Filipič B, Smith J (eds.), Parallel Problem Solving from Nature – PPSN XIII, volume 8672 of *Lecture Notes in Computer Science*. Springer. ISBN 978-3-319-10761-5, 2014 pp. 761–770. doi:10.1007/978-3-319-10762-2_75.

[81] Chen S, Montgomery J, Bolufé-Röhler A. Measuring the curse of dimensionality and its effects on particle swarm optimization and differential evolution. *Applied Intelligence*, 2015. **42**. doi:10.1007/s10489-014-0613-2.

[82] Mallipeddi R, Suganthan PN. Empirical study on the effect of population size on Differential evolution Algorithm. In: 2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence). 2008 pp. 3663–3670. doi:10.1109/CEC.2008.4631294.

[83] Qin AK, Huang VL, Suganthan PN. Differential Evolution Algorithm with Strategy Adaptation for Global Numerical Optimization. *IEEE Trans. Evol. Comput.*, 2009. **13**(2):398–417. doi:10.1109/TEVC.2008.927706.

[84] Zhang J, Sanderson AC. JADE: Adaptive Differential Evolution with Optional External Archive. *IEEE Trans. Evol. Comput.*, 2009. **13**(5):945–958. doi:10.1109/TEVC.2009.2014613.

[85] Das S, Mullick SS, Suganthan PN. Recent advances in differential evolution – An updated survey. *Swarm and Evol. Comput.*, 2016. **27**:1 – 30. doi:10.1016/j.swevo.2016.01.004.

[86] Neri F, Tirronen V. Recent advances in differential evolution: a survey and experimental analysis. *Artificial Intelligence Review*, 2010. **33**(1):61–106. doi:10.1007/s10462-009-9137-2.

[87] Al-Dabbagh RD, Neri F, Idris N, Baba MS. Algorithmic design issues in adaptive differential evolution schemes: Review and taxonomy. *Swarm and Evolutionary Computation*, 2018. doi:10.1016/j.swevo.2018.03.008. , in press, 2018.

[88] Cazzaniga P, Nobile MS, Besozzi D. The impact of particles initialization in PSO: Parameter estimation as a case in point. In: Computational Intelligence in Bioinformatics and Computational Biology (CIBCB), 2015 IEEE Conference on. IEEE, 2015 pp. 1–8.

[89] Pedersen MEH, Chipperfield AJ. Simplifying particle swarm optimization. *Applied Soft Computing*, 2010. **10**(2):618–628.

[90] Li X, Tang K, Omidvar MN, Yang Z, Qin K. Benchmark Functions for the CEC2013 Special Session and Competition on Large-Scale Global Optimization. Technical report, School of Computer Science and Information Technology, RMIT University, Melbourne, Australia and School of Computer Science and Technology, University of Science and Technology of China, Hefei, Anhui, China and National University of Defense Technology, Changsha 410073, China, 2013.

[91] Herrera F, Lozano M, Molina D, Lozano Are With M. Test suite for the special issue of soft computing on scalability of evolutionary algorithms and other metaheuristics for large scale continuous optimization problems. Technical report, University of Granada, Spain, 2010.

[92] LaTorre A, Muelas S, Sánchez JMP. Large scale global optimization: Experimental results with MOS-based hybrid algorithms. In: Proceedings of the IEEE Congress on Evolutionary Computation, CEC 2013, Cancun, Mexico, June 20-23, 2013. 2013 pp. 2742–2749. doi:10.1109/CEC.2013.6557901.

[93] LaTorre A, Muelas S, Peña JM. A MOS-based dynamic memetic differential evolution algorithm for continuous optimization: a scalability test. *Soft Computing*, 2011. **15**(11):2187–2199. doi:10.1007/s00500-010-0646-3.

[94] Molina D, Lozano M, Herrera F. MA-SW-Chains: Memetic algorithm based on local search chains for large scale continuous global optimization. In: IEEE Congress on Evolutionary Computation. 2010 pp. 1–8. doi:10.1109/CEC.2010.5586034.

[95] Brest J, Maučec MS. Self-adaptive differential evolution algorithm using population size reduction and three strategies. *Soft Computing*, 2011. **15**(11):2157–2174. doi:10.1007/s00500-010-0644-5.

[96] Wang Y, Li B. Two-stage based ensemble optimization for large-scale global optimization. In: IEEE Congress on Evolutionary Computation. 2010 pp. 1–8. doi:10.1109/CEC.2010.5586466.

[97] Mühlenbein H, Paaß G. From recombination of genes to the estimation of distributions I. Binary parameters. In: Voigt HM, Ebeling W, Rechenberg I, Schwefel HP (eds.), Parallel Problem Solving from Nature — PPSN IV. Springer Berlin Heidelberg, Berlin, Heidelberg. ISBN 978-3-540-70668-7, 1996 pp. 178–187.

[98] Yang Z, Tang K, Yao X. Scalability of generalized adaptive differential evolution for large-scale continuous optimization. *Soft Computing*, 2011. **15**(11):2141–2155. doi:10.1007/s00500-010-0643-6.

[99] Molina D, LaTorre A, Herrera F. SHADE with Iterative Local Search for Large-Scale Global Optimization. In: 2018 IEEE Congress on Evolutionary Computation (CEC). 2018 pp. 1–8. doi: 10.1109/CEC.2018.8477755.

[100] Loshchilov I. A Computationally Efficient Limited Memory CMA-ES for Large Scale Optimization. In: Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation, GECCO '14. ACM, New York, NY, USA. ISBN 978-1-4503-2662-9, 2014 pp. 397–404. doi:10.1145/2576768.2598294.

[101] Varelas K, Auger A, Brockhoff D, Hansen N, ElHara OA, Semet Y, Kassab R, Barbaresco F. A Comparative Study of Large-Scale Variants of CMA-ES. In: Auger A, Fonseca CM, Lourenço N, Machado P, Paquete L, Whitley D (eds.), Parallel Problem Solving from Nature – PPSN XV. Springer International Publishing, Cham. ISBN 978-3-319-99253-2, 2018 pp. 3–15.

[102] LaTorre A, Muelas S, Pena JM. A comprehensive comparison of large scale global optimizers. *Information Sciences*, 2015. **316**:517 – 549. doi:10.1016/j.ins.2014.09.031. Nature-Inspired Algorithms for Large Scale Global Optimization.

[103] Molina D, LaTorre A. Toolkit for the Automatic Comparison of Optimizers: Comparing Large-Scale Global Optimizers Made Easy. In: 2018 IEEE Congress on Evolutionary Computation (CEC). 2018 pp. 1–8. doi:10.1109/CEC.2018.8477924.

[104] Chen Q, Liu B, Zhang Q, Liang J, Suganthan P, Qu B. Problem Definition and Evaluation Criteria for CEC 2015 Special Session and Competition on Bound Constrained Single-Objective Computationally Expensive Numerical Optimization. Technical report, Computational Intelligence Laboratory, Zhengzhou University, Zhengzhou, China, Nanyang Technological University, Singapore, 2014.

[105] Rueda JL, Erlich I. MVMO for bound constrained single-objective computationally expensive numerical optimization. In: 2015 IEEE Congress on Evolutionary Computation (CEC). 2015 pp. 1011–1017. doi: 10.1109/CEC.2015.7257000.

[106] Erlich I, Venayagamoorthy GK, Worawat N. A Mean-Variance Optimization algorithm. In: IEEE Congress on Evolutionary Computation. 2010 pp. 1–6. doi:10.1109/CEC.2010.5586027.

[107] Berthier V. Experiments on the CEC 2015 expensive optimization testbed. In: 2015 IEEE Congress on Evolutionary Computation (CEC). 2015 pp. 1059–1066. doi:10.1109/CEC.2015.7257007.

[108] Igel C, Suttorp T, Hansen N. A Computational Efficient Covariance Matrix Update and a (1+1)-CMA for Evolution Strategies. In: Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation, GECCO '06. ACM, New York, USA. ISBN 1-59593-186-4, 2006 pp. 453–460. doi: 10.1145/1143997.1144082.

[109] Wilke CO, Wang JL, Ofria C, Lenski RE, Adami C. Evolution of Digital Organisms at High Mutation Rates Leads to Survival of the Flattest. *Nature*, 2001. **412**:331–333. doi:10.1038/35085569.

[110] Arabas J, Biedrzycki R. Quasi-Stability of Real Coded Finite Populations. In: Bartz-Beielstein T, Branke J, Filipič B, Smith J (eds.), Parallel Problem Solving from Nature – PPSN XIII, volume 8672 of *Lecture Notes in Computer Science*. Springer. ISBN 978-3-319-10761-5, 2014 pp. 872–881. doi: 10.1007/978-3-319-10762-2_86.

[111] Schonfeld J, Ashlock DA. A comparison of the robustness of evolutionary computation and random walks. In: Proceedings of the 2004 Congress on Evolutionary Computation (IEEE Cat. No.04TH8753), volume 1. 2004 pp. 250–257 Vol.1. doi:10.1109/CEC.2004.1330864.

[112] Mendes R, Cortez P, Rocha M, Neves J. Particle swarms for feedforward neural network training. In: Proceedings of the 2002 International Joint Conference on Neural Networks. IJCNN'02 (Cat. No.02CH37290), volume 2. 2002 pp. 1895–1899 vol.2. doi:10.1109/IJCNN.2002.1007808.

[113] Gudise VG, Venayagamoorthy GK. Comparison of particle swarm optimization and backpropagation as training algorithms for neural networks. In: Proceedings of the 2003 IEEE Swarm Intelligence Symposium. SIS'03 (Cat. No. 03EX706). IEEE, 2003 pp. 110–117.

[114] M Ibrahim A, El-Amary N. Particle Swarm Optimization Trained Recurrent Neural Network for Voltage Instability Prediction. *Journal of Electrical Systems and Information Technology*, 2017. **5**. doi:10.1016/j.jesit.2017.05.001.

[115] Zhang JR, Zhang J, Lok TM, Lyu MR. A hybrid particle swarm optimization–back-propagation algorithm for feedforward neural network training. *Applied mathematics and computation*, 2007. **185**(2):1026–1037.

[116] Wierstra D, Schaul T, Peters J, Schmidhuber J. Natural evolution strategies. In: Evolutionary Computation, 2008. CEC 2008.(IEEE World Congress on Computational Intelligence). IEEE Congress on. IEEE, 2008 pp. 3381–3387.

[117] Wierstra D, Schaul T, Glasmachers T, Sun Y, Peters J, Schmidhuber J. Natural Evolution Strategies. *J. Mach. Learn. Res.*, 2014. **15**(1):949–980.

[118] Salimans T, Ho J, Chen X, Sidor S, Sutskever I. Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint:1703.03864*, 2017.

[119] Ilyas A, Engstrom L, Athalye A, Lin J, Athalye A, Engstrom L, Ilyas A, Kwok K. Black-box Adversarial Attacks with Limited Queries and Information. In: Proceedings of the 35th International Conference on Machine Learning, ICML. 2018 .

[120] Su J, Vargas DV, Sakurai K. One pixel attack for fooling deep neural networks. *IEEE Transactions on Evolutionary Computation*, 2019.

[121] Choromanska A, Henaff M, Mathieu M, Arous GB, LeCun Y. The loss surfaces of multilayer networks. In: Artificial Intelligence and Statistics. 2015 pp. 192–204.

[122] Safran I, Shamir O. Spurious Local Minima are Common in Two-Layer ReLU Neural Networks. *arXiv preprint:1712.08968*, 2017.

[123] LIANG S, Sun R, Li Y, Srikant R. Understanding the Loss Surface of Neural Networks for Binary Classification. 2018. **80**:2835–2843.

[124] Du S, Lee J, Tian Y, Singh A, Poczos B. Gradient Descent Learns One-hidden-layer CNN: Dont be Afraid of Spurious Local Minima. 2018. **80**:1339–1348.

[125] Laurent T, Brecht J. Deep linear networks with arbitrary loss: All local minima are global. In: International Conference on Machine Learning. 2018 pp. 2908–2913.

[126] Du SS, Lee JD, Li H, Wang L, Zhai X. Gradient Descent Finds Global Minima of Deep Neural Networks. *CoRR:1811.03804v2*, 2018.

[127] Rozsa A, Gunther M, Boult TE. Towards Robust Deep Neural Networks with BANG. In: 2018 IEEE Winter Conference on Applications of Computer Vision (WACV). IEEE, 2018 pp. 803–811.