Derivative-Free Global Optimization Algorithms: Population based Methods and Random Search Approaches

Jiawei Zhang JIAWEI@IFMLAB.ORG
Founder and Director
Information Fusion and Mining Laboratory
(First Version: March 2019; Revision: April 2019.)

Abstract

In this paper, we will provide an introduction to the derivative-free optimization algorithms which can be potentially applied to train deep learning models. Existing deep learning model training is mostly based on the back propagation algorithm, which updates the model variables layers by layers with the gradient descent algorithm or its variants. However, the objective functions of deep learning models to be optimized are usually non-convex and the gradient descent algorithms based on the first-order derivative can get stuck into the local optima very easily. To resolve such a problem, various local or global optimization algorithms have been proposed, which can help improve the training of deep learning models greatly. The representative examples include the Bayesian methods, Shubert-Piyavskii algorithm, DIRECT, LIPO, MCS, GA, SCE, DE, PSO, ES, CMA-ES, hill climbing and simulated annealing, etc. This is a follow-up paper of [18], and we will introduce the population based optimization algorithms, e.g., GA, SCE, DE, PSO, ES and CMA-ES, and random search algorithms, e.g., hill climbing and simulated annealing, in this paper. For the introduction to the other derivative-free optimization algorithms, please refer to [18] for more information.

Keywords: Derivative-Free; Global Optimization; Population based Methods; Random Search; Deep Learning

Contents

1. Introduction

This is a follow-up paper of [18]. To make it self-contained, we will briefly introduce the learning settings again as follows. The training set for optimizing the deep learning models can be represented as $\mathcal{T} = \{(\mathbf{x}_1, \mathbf{y}_1), (\mathbf{x}_2, \mathbf{y}_2), \cdots, (\mathbf{x}_n, \mathbf{y}_n)\}$, which involves n pairs of feature-label instances. Formally, for each data instance, its feature vector $\mathbf{x}_i \in \mathbb{R}^{d_x}, \forall i \in \{1, 2, \cdots, n\}$ and label vector $\mathbf{y}_i \in \mathbb{R}^{d_y}, \forall i \in \{1, 2, \cdots, n\}$ are of dimensions d_x and d_y respectively. The deep learning models define a mapping $F(\cdot; \boldsymbol{\theta}) : \mathcal{X} \to \mathcal{Y}$, which projects the data instances from the feature space $\mathcal{X}$ to the label space $\mathcal{Y}$. In the above representation of function $F(\cdot, \boldsymbol{\theta})$, vector $\boldsymbol{\theta} \in \Theta$ contains the variables involved in the deep learning model and Θ denotes the variable inference space. Formally, we can denote the dimension of variable vector $\boldsymbol{\theta}$ as d_θ, which will be used when introducing the algorithms later. Given one data instance featured by vector $\mathbf{x}_i \in \mathcal{X}$, we can denote its prediction label vector by the deep learning model as $\hat{\mathbf{y}}_i = F(\mathbf{x}_i; \boldsymbol{\theta})$. Compared against its true label vector $\mathbf{y}_i$, we can denote the introduced loss for instance $\mathbf{x}_i$ as $\ell(\hat{\mathbf{y}}_i, \mathbf{y}_i)$. Several frequently used loss representations have been introduced in [19], and we will not redefine them here again. For all the data instances in the training set, we can represent the total loss term as

$$\mathcal{L}(\boldsymbol{\theta}) = \mathcal{L}(\boldsymbol{\theta}; \mathcal{T}) = \sum_{(\mathbf{x}_i, \mathbf{y}_i) \in \mathcal{T}} \ell(\hat{\mathbf{y}}_i, \mathbf{y}_i). \tag{1}$$

And the deep model learning can be formally denoted as the following function:

$$\min_{\boldsymbol{\theta} \in \Theta} \mathcal{L}(\boldsymbol{\theta}), \tag{2}$$

which is also the main objective function to be studied in this paper.

We need to add a remark here, the above objective function defines a minimization problem. Meanwhile, when introducing some of the optimization algorithms in the following sections, we may assume the objective function to be a maximization function instead for simplicity. The above objective can be transformed into a maximization problem easily by introducing a new term $\mathcal{L}'(\boldsymbol{\theta}) = -\mathcal{L}(\boldsymbol{\theta})$. We will clearly indicate it when the algorithm is introduced for a maximization problem.

In the following part of this paper, we will introduce the derivative-free optimization algorithms that can be potentially used to resolve the above objective function. To be more specific, this paper covers the introduction to the population based algorithms (e.g., GA, SCE, DE, PSO, ES and CMA-ES) and the random search based optimization algorithms (e.g., hill climbing and simulated annealing). If the readers are interested in other derivative-free optimization algorithms, you can refer to our previous article [18] for more information.

2. Population based Algorithm for Global Optimization

In this section, we will introduce a group of nature-inspired population based meta-heuristic optimization algorithms, including GA (Genetic Algorithm), SCE (Shuffled Complex Evolution), DE (Dierential Evolution), PSO (Particle Swarm Optimization), ES (Evolution Strategy) and CMA-ES (Covariance Matrix Adaption-Evolution Strategy). Different from the algorithms introduced in [18], which starts with one single solution candidate, the algorithms introduced in this part will start with a group of solution candidates candidates instead and propose to update them to improve the learning performance. When the objective variable space is too large to search exhaustively, the population based searches may be a good alternative, which cannot guarantee the optimal solution even though.

2.1 Genetic Algorithm (GA)

Genetic algorithm (GA) [16] is a meta-heuristic algorithm inspired by the process of natural selection in evolutionary algorithms, which has also been widely used for learning the solutions of many optimization problems. In GA, a population of candidate solutions will be initialized and evolved towards better ones. GA has demonstrated its outstanding performance in many learning scenarios, like non-convex objective function containing multiple local optima, objective function with non-smooth shape, as well as a large number of parameters and noisy environments. GA consists of several main steps, including *generation initialization*, *crossover and mutation*, *fitness evaluation and selection*, which can effectively evolve good candidate solutions generation by generation. In this part, we will introduce these three main steps in great detail.

2.1.1 Population Initialization

Given the variable search space Θ, a group of candidate solutions to the objective function can be generated via either random sampling from the space or the output from other existing learning algorithms if Gadam [20] is used as the optimization framework (as introduced in the previous tutorial article [19]). Formally, we can denote the initial set of candidate solutions sampled from Θ as $\mathcal{G}^{(0)} = \{\boldsymbol{\theta}_1^{(0)}, \boldsymbol{\theta}_2^{(0)}, \cdots, \boldsymbol{\theta}_p^{(0)}\}$, where p denotes the population size and the superscript denotes the generation index. These solution vectors are treated as

the chromosome, which can be evolved to achieve better solutions. Traditional GA works well for the binary variable case, and several works also propose to extend GA to the real-number variable scenarios. Depending on the objective variable search space, the variable vectors $\boldsymbol{\theta}_i^{(0)} \in \Theta$ can contain either binary or real codes, and their corresponding sampling approaches can be different as well.

For the binary variable search space, i.e., $\Theta \subseteq \{0,1\}^{d_\theta}$, the entries in vector $\boldsymbol{\theta}_i^{(0)}$ can be sampled via the Bernoulli distribution, i.e., $\boldsymbol{\theta}_i^{(0)}(j) \sim \mathcal{B}(p), \forall j \in \{1, 2, \cdots, d_\theta\}$, where p denotes the probability to sample value 1. Meanwhile, for the real number variable search space, i.e., $\Theta \subseteq \mathbb{R}^{d_\theta}$, the entries in vector $\boldsymbol{\theta}_i^{(0)}$ can be sampled via distributions like the Gaussian distribution, i.e., $\boldsymbol{\theta}_i^{(0)}(j) \sim \mathcal{N}(\mu, \sigma^2), \forall j \in \{1, 2, \cdots, d_\theta\}$, where μ and σ denote the mean and standard deviation parameters of the distribution.

2.1.2 Fitness Evaluation and Selection

Among all the candidate solutions in set $\mathcal{G}$, some of them are good candidates for the objective problem but some of them can be not. GA will evaluate the candidate solutions in $\mathcal{G}$ to pick the good ones as the parents to generate the offsprings. For the objective function mentioned in the Introduction section, we can evaluate the candidate solutions with the objective function $\mathcal{L}(\cdot)$ to be minimized. Formally, by applying the objective function on candidate solution $\boldsymbol{\theta}_i^{(0)}$, we can denote the introduced function value as $\ell_i^{(0)} = \mathcal{L}(\boldsymbol{\theta}_i^{(0)})$. The loss terms introduced by function $\mathcal{L}(\cdot)$ on all the candidate solutions can be denoted as a list $[\ell_1^{(0)}, \ell_2^{(0)}, \cdots, \ell_p^{(0)}]$.

Generally, the solutions leading to smaller function values will have a larger chance to be selected in GA. There exist different ways to define the selection probability of each solution candidate. For instance, we can adopt the softmax equation to define the selection probability for $\boldsymbol{\theta}_i^{(0)}$ as follows:

$$p_i^{(0)} = \frac{\exp(-\ell_i^{(0)})}{\sum_{j=1}^{p} \exp(-\ell_j^{(0)})}. \tag{3}$$

The selection probability of all the candidate solutions can be denoted as a list $[p_1^{(0)}, p_2^{(0)}, \cdots, p_p^{(0)}]$, based on which, $\frac{p}{2}$ pairs of unit model pairs will be selected as the parents for the next generation, which can be denoted as a set $\mathcal{P}^{(0)} = \left\{(\boldsymbol{\theta}_{i_1}^{(0)}, \boldsymbol{\theta}_{j_1}^{(0)}), (\boldsymbol{\theta}_{i_2}^{(0)}, \boldsymbol{\theta}_{j_2}^{(0)}), \cdots\right\}$, where $i_1, j_1, \cdots \in \{1, 2, \cdots, p\}$.

2.1.3 Crossover and Mutation

GA generates the offsprings of the selected parent pairs via the crossover and mutation operations, which imitate the chromosome crossover and mutation of creatures in the natural world. Different kinds of crossover and mutation methods have been proposed already, and in this part we will introduce the classic crossover and mutation operations used in GA respectively.

Crossover: Given a parent variable pair, e.g., $(\boldsymbol{\theta}_{i_k}^{(0)}, \boldsymbol{\theta}_{j_k}^{(0)}) \in \mathcal{P}$, by selecting one or several crossover points, crossover aims at mix the variable values together to generate new children variables. For instance, as shown in Figure 1, we can represent the parent variable pair as

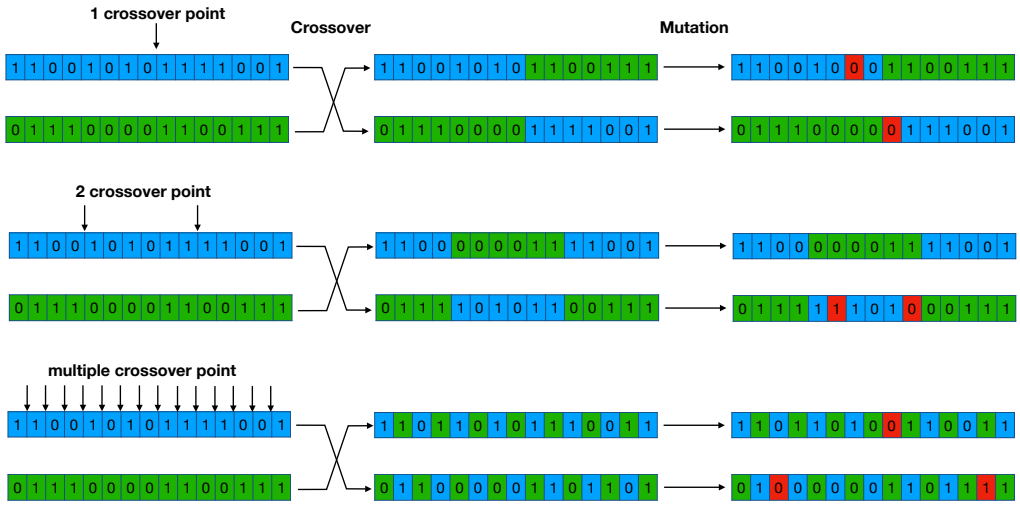

Figure 1: A Example of Crossover and Mutation Operations in GA.

the two sequences on the left-hand side (in blue and green colors respectively). By selecting the crossover point(s), GA will exchange the sub-sequences of the variable values between the parents to generate the children variable sequences, i.e., the ones on the right-hand side. The crossover points are usually selected by random actually. Depending on the number of crossover points finally selected, the crossover operation will create different offspring variables. In the plot, we show the examples with (1) one crossover point, (2) two crossover points, and (3) multiple crossover points, respectively, on the binary variables. Similar operations can be done on real-number variables as well.

Mutation: GA models the gene mutation in the real-world with the mutation operation, which can change a small number of the children variable values with a certain pre-specified probability. Formally, given the mutation probability $\eta \in [0, 1.0]$, GA will enumerate all the variable positions and randomly select a number of them subject to the probability η for mutation. For instance, in Figure 1, for the generated children variables, a number of mutation spots are identified (in red color), the variable values at which are flipped (i.e., 1 changed to 0, and 0 changed 1). Some of the children variables have one single mutation spot, some have none and some may have two mutation spots. When it comes to the real-number variables, the variable mutation can be done in a similar way, where the bit flip operation can be replaced with some real-number sampling operation instead to change the variable values.

With the crossover and mutation operations, GA will generate a new group of candidate variable solutions, which can be denoted as set $\mathcal{G}^{(1)} = \{\boldsymbol{\theta}_1^{(1)}, \boldsymbol{\theta}_2^{(1)}, \cdots, \boldsymbol{\theta}_p^{(1)}\}$. Such an iterative process continues until convergence and the optimal variable in the last generation will be outputted as the solution. The pseudo-code of GA is provided in Algorithm 1.

2.1.4 More Discussions on GA

Before we conclude this subsection on GA, we would like to provide the physical meanings of the crossover and mutation operations. Crossover actually will relocate the variable

Algorithm 1 Genetic Algorithm

Require: Variable Search Space Θ.
Ensure: Model Parameter $\boldsymbol{\theta}$
Initialize a population $\mathcal{G} = \{\boldsymbol{\theta}_1, \boldsymbol{\theta}_2, \cdots, \boldsymbol{\theta}_p\}$
Initialize convergence $tag = False$
while $tag = False$ **do**
  Evaluate the solutions in $\mathcal{G}$ to get the loss function values $[\ell_1, \ell_2, \cdots, \ell_p]$
  Compute the sampling probability $[p_1^{(0)}, p_2^{(0)}, \cdots, p_p^{(0)}]$
  Sample the parent variable pairs $\mathcal{P} = \{(\boldsymbol{\theta}_{i_1}, \boldsymbol{\theta}_{j_1}), (\boldsymbol{\theta}_{i_2}, \boldsymbol{\theta}_{j_2}), \cdots\}$ subject to the probabilities
  Generate the children variables $\mathcal{G}'$ from pairs in $\mathcal{P}$ via crossover
  Update the children variables $\mathcal{G}''$ from $\mathcal{G}'$ via mutation
  Set $\mathcal{G} = \mathcal{G}''$
  if convergence condition holds **then**
    $tag = True$
  end if
end while
Return $\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta} \in \mathcal{G}} \mathcal{L}(\boldsymbol{\theta})$

positions in the search space, which allows the algorithm to explore broadly based on the current feasible solutions. Meanwhile, to enable the algorithm to explore some regions outside the hyper-cubes with current solutions as the vertices, GA adopts the mutation operation which can change a small number of variable values in the learning process.

For instance, in Figure 2, we provide an example on optimization on a search space $\{0, 1\}^3$, where the variables are a binary sequence of length 3 and the space include all the 8 vertices of the cube. Let's assume, we are provided with two random variables 000 and 011, which correspond to the two vertices on the front-side of the cube. By using these two variables as the parents, via the crossover operation, these two vertices can generate the two children variables 010 and 001 (i.e., the remaining two vertices in blue and green colors respectively at the front-side).

Here, we face a dilemma: no matter how we crossover the variables, we cannot explore the remaining 4 vertices at the back-side. However, we discover that the mutation operation allows GA to resolve such a problem actually. For instance, by flipping the first bit of the variables from 0 to 1 in the children variables, GA will successfully reaches the back-side of the cube (i.e., the two red points) and will be able to explore the remaining vertices for learning the optimal solutions to the problem.

Traditional GA is usually very slow, and the learning process may involve a large number of generations. In recent years, some works have been introduced to improve the slow convergence problem of GA, including GADAM [20] and fast GA [2]. GADAM [20] adopts the gradient descent algorithms to help reach the local optimum for each unit variable solutions before the evolution; while fast GA [2] adopts a random mutation rate to enable the GA to achieve a faster convergence rate. GADAM is also introduced in our previous tutorial article [19], and the readers may refer to the cited articles for more detailed information about these two mentioned algorithms.

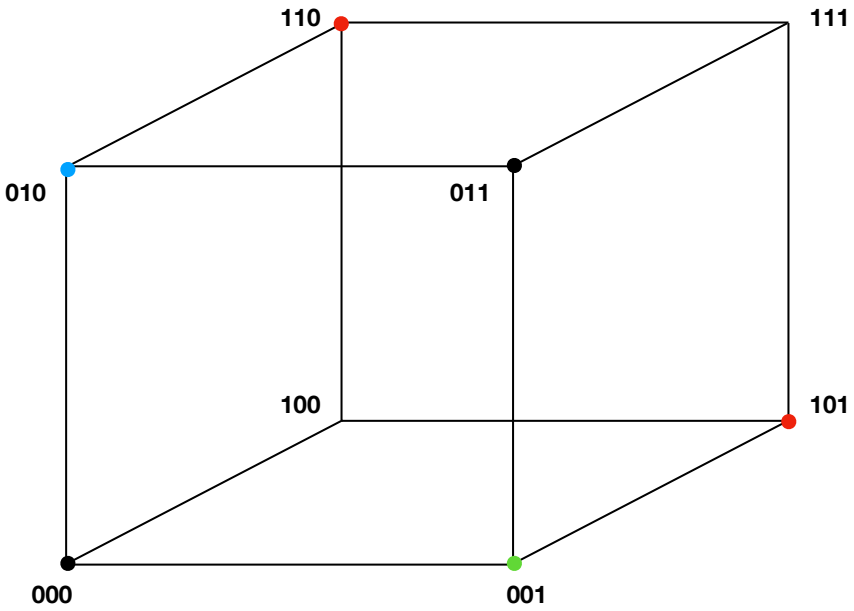

Figure 2: A Example to Illustrate the Physical Meanings of Crossover and Mutation in Optimization.

2.2 Shuffled Complex Evolution (SCE) Algorithm

The Shuffled Complex Evolution (SCE) algorithm [3] to be introduced in this part is an improvement over the genetic algorithm, which is based on a synthesis of four concepts that have been proven to be successful for global optimization, including (1) combination of probabilistic and deterministic approaches, (2) clustering, (3) systematic evolution of a complex of points spanning the search space, and (4) competitive evolution. SCE is shown to be much more effective, efficient and robust for a broad class of problems. In the following part of this subsection, we will introduce the outline of the SCE algorithm, together with the CCE (Competitive Complex Evolution) algorithm which will be used in SCE.

2.2.1 SCE Algorithm Outline

The outline of the SCE algorithm is illustrated in Algorithm 2, which accepts $p \geq 1$, $m \geq d_\theta + 1$ and the search space Θ as the input. The algorithm consists of several important steps, which are listed as follows:

- **Initialization**: In the initialization step (i.e., line 1), SCE will compute the sample size $s = p \times m$.

Algorithm 2 SCE Algorithm

Require: Variable search space Θ; Complex number p; Complex size: m.
Ensure: Model Parameter $\boldsymbol{\theta}$

1: Compute sample size $s = p \times m$
2: Sample s variable points $\{\boldsymbol{\theta}_1, \boldsymbol{\theta}_2, \cdots, \boldsymbol{\theta}_s\}$ from space Θ
3: Evaluate the objective function $\mathcal{L}(\cdot)$ at the variables, and get $\{\ell_i = \mathcal{L}(\boldsymbol{\theta}_i)\}_{i \in \{1,2,\cdots,s\}}$
4: Sort the variables into an array $\mathcal{D} = \{\boldsymbol{\theta}_i\}_{i \in \{1,2,\cdots,s\}}$ in an increasing order of $\{\ell_i\}_{i \in \{1,2,\cdots,s\}}$
5: Initialize convergence $tag = False$
6: **while** $tag = False$ **do**
7: Partition $\mathcal{D}$ sequentially into p equal-sized complexes $\mathcal{A} = \{\mathcal{A}^1, \mathcal{A}^2, \cdots, \mathcal{A}^p\}$
8: **for** each complex $\mathcal{A}^i \in \mathcal{A}$ **do**
9: Evolve $\mathcal{A}^i$ with the CCE algorithm, i.e., $\mathcal{A}^i = CCE(\mathcal{A}^i)$
10: Add all the complexes in $\mathcal{A}$ into $\mathcal{D}$ again, i.e., $\mathcal{D} = \bigcup_{i=1}^{p} \mathcal{A}^i$
11: Evaluate variables in $\mathcal{D}$, and get $\{\ell_i = \mathcal{L}(\boldsymbol{\theta}_i)\}_{\boldsymbol{\theta}_i \in \mathcal{D}}$
12: Resort variables in $\mathcal{D}$ in an increasing order of $\{\ell_i\}_{\boldsymbol{\theta}_i \in \mathcal{D}}$
13: **end for**
14: **if** convergence condition holds **then**
15: $tag = True$
16: **end if**
17: **end while**
18: Return $\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta} \in \mathcal{D}} \mathcal{L}(\boldsymbol{\theta})$

- **Sample Generation**: At line 2, s variable samples will be sampled from the search space Θ (subject to the uniform distribution if no prior information is available).

- **Evaluation and Sorting**: For each of the sampled variable point, SCE will evaluate the objective function at them. Considering our objective function is a minimization problem, the evaluation results will be used to sort the variable points in an increasing order of their function values (as indicated in lines 3-4).

- **Complexes Partition**: The learning process of SCE involves an iterative process, which starts with a complex partition step as indicated in line 7. According to the increasing order of the points in $\mathcal{D}$, SCE sequentially partition these points into p complexes $\mathcal{A} = \{\mathcal{A}^1, \mathcal{A}^2, \cdots, \mathcal{A}^p\}$, each of which contains m points.

- **Complex Evolution**: SCE will call the CCE algorithm to evolve the points in each complex to achieve better offspring variable solutions (i.e., in line 9). The CCE algorithm will be introduced later.

- **Shuffle, Re-evaluate and Re-sort**: Based on the evolved variables, SCE will merge them together and re-evaluate the objective functions at these new points, which will be re-sorted again for the next iteration.

Such a process continues until convergence or the maximum iteration number has been met. The optimal variable in the last iteration will be outputted as the final solution, i.e., $\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta} \in \mathcal{D}} \mathcal{L}(\boldsymbol{\theta})$.

Algorithm 3 CCE Algorithm

Require: Search space Θ; Complex $\mathcal{A}^k$; Parent size q; Evolution iteration parameters α and β; .
Ensure: Evolved complex $\mathcal{A}^k$

for external-evolution iteration index in $\{1, 2, \cdots, \beta\}$ **do**
  Compute weights for points in complex $\mathcal{A}^k$, i.e., $\{p_i\}_{\boldsymbol{\theta}_i^k \in \mathcal{A}^k}$
  Randomly select q parent points $\mathcal{B} = \{\mathbf{u}_1, \mathbf{u}_2, \cdots, \mathbf{u}_q\}$ from $\mathcal{A}^k$ subject to the probabilities
  for inner-evolution iteration index in $\{1, 2, \cdots, \alpha\}$ **do**
    Sort the points in $\mathcal{B}$ in the order of increasing function value
    Computing the centroid $\mathbf{g}$ of points in $\mathcal{B}$ excluding the worst point
    Reflect the worst point with a new point $\mathbf{r} = 2\mathbf{g} - \mathbf{u}_q$
    if $\mathbf{r}$ is not within Θ **then**
      Compute the smallest hypercube $\mathcal{H} \subset \mathbb{R}^{d_\theta}$
      Randomly sample point $\mathbf{z}$ from $\mathcal{H}$
      Set $\mathbf{r} = \mathbf{z}$
    end if
    Evaluate function $\mathcal{L}(\cdot)$ to get $\ell_r = \mathcal{L}(\mathbf{r})$ and $\ell_q = \mathcal{L}(\mathbf{u}_q)$
    if $\ell_r < \ell_q$ **then**
      Set $\mathbf{u}_q = \mathbf{r}$
    else
      Compute $\mathbf{c} = \frac{(\mathbf{g}+\mathbf{u}_q)}{2}$, and evaluate $\ell_c = \mathcal{L}(\mathbf{c})$
      if $\ell_c < \ell_q$ **then**
        Set $\mathbf{u}_q = \mathbf{c}$
      else
        Randomly generate a point $\mathbf{z}$ from $\mathcal{H}$
        Set $\mathbf{u}_q = \mathbf{z}$
      end if
    end if
  end for
  Replace the original points in $\mathcal{A}^k$ with the evolved ones in $\mathcal{B}$
  Resort $\mathcal{A}^k$ according to the re-evaluated function values at points in $\mathcal{A}^k$
end for
Return $\mathcal{A}^k$ as the output

2.2.2 CCE Algorithm Outline

The CCE algorithm called in the SCE algorithm can effectively evolve the variable points in each complex into a new stage in a manner similar to the genetic algorithm, which accepts a complex $\mathcal{A}^k$, parameters $2 \leq q \leq m$, $\alpha \geq 1$ and $\beta \geq 1$ as the inputs. The output of CCE will be the evolved complex in a similar data structure as the input $\mathcal{A}^k$, containing the well-sorted variables. The pseudo-code of the CCE algorithm is provided in Algorithm 3.

The CCE algorithm consists of two evolution iterations. The external iteration of CCE samples the parent variable points, which will be evolved with the internal iteration. The key steps in CCE are introduced as follows:

External Evolution Iteration for β Rounds

- **Weight Computation**: Based on the position indexes of points in complex $\mathcal{A}^k$, CCE will compute the sampling probabilities for these points, which can be denoted as

$$p_i = \frac{2(m+1-i)}{m(m+1)}, \forall i \in \{1, 2, \cdots, m\}, \tag{4}$$

where i here denotes the index of a variable point. In other words, for the points which are in the front (i.e., with smaller function evaluation values) will have a higher sampling opportunity. For instance, for the first point in $\mathcal{A}^k$, its sampling probability will be $\frac{2}{(m+1)}$; while for the last point in $\mathcal{A}^k$, its sampling probability will be $\frac{2}{m(m+1)}$.

- **Parent Selection**: A batch of points will be randomly selected from $\mathcal{A}^k$ as the parents, which is denoted as $\mathcal{B} = \{\mathbf{u}_1, \mathbf{u}_2, \cdots, \mathbf{u}_q\}$ in line 3. Here, we use the notation $\mathbf{u}$ instead of $\boldsymbol{\theta}$ to avoid the confusion about the subscripts in the representation.
- **Internal Evolution Iteration: Offspring Generation for α Rounds**
  - **Centroid Computing**: Variable points in the parent batch $\mathcal{B}$ will be sorted, which will all (except the worst point) be used to compute the centroid of the complex, i.e.,
  $$\mathbf{g} = \frac{1}{q-1}\sum_{i=1}^{q-1}\mathbf{u}_i. \tag{5}$$
  - **Reflection Step**: A new point will be computed based the worst point in $\mathcal{B}$, which can be denoted as $\mathbf{r} = 2\mathbf{g} - \mathbf{u}_q$.
  - **Mutation Step**: If the new point $\mathbf{r}$ is not in the search space Θ, CCE will replace $\mathbf{r}$ with a new randomly selected point $\mathbf{z}$ from the smallest hypercube $\mathcal{H}$ which covers all the points in $\mathcal{A}^k$.
  - **Contraction Step**: In the case if $\mathbf{r}$ is better than $\mathbf{u}_q$, CCE will replace $\mathbf{u}_q$ with $\mathbf{r}$; otherwise, CCE will create a new point $\mathbf{c} = \frac{(\mathbf{g}+\mathbf{u}_q)}{2}$ (i.e., the central point between centroid $\mathbf{g}$ and the worst point $\mathbf{u}_q$).
  - **Mutation Step**: If $\mathbf{c}$ is better than $\mathbf{u}_q$, CCE will replace $\mathbf{u}_q$ with $\mathbf{c}$; otherwise, CCE will replace $\mathbf{u}_q$ with a randomly selected point $\mathbf{z} \in \mathcal{H}$.
- **Parent Update**: All the generated offsprings will be put back into $\mathcal{A}^k$ to update the variables. Complex $\mathcal{A}^k$ will be updated, re-evaluated, and re-sorted for the next iteration.

2.2.3 More Discussions on SCE

The SCE algorithm treats the global search of optimal solutions as a process of natural evolution, where the sampled s points contribute a population. The population will be partitioned into several communities, each of which will evolve independently (which denotes the process to search the space in different directions). After a certain number of generations, the communities will be forced to mix together, and new communities will be created via a shuffling process. This procedure enhances survivability by a sharing of information gained independently by each community.

The evolution process used in SCE is different from that in GA, where the parents are in a batch instead of a pair. A subset of the points will be sampled from a complex subject to the pre-computed probabilities, which serve as the parents in the evolution. The offsprings are introduced at random locations of the feasible search space under certain condition that the evolution will not be trapped by unpromising regions. Therefore, each

mutation will help improve the community slightly in the evolution process, and the newly generated points will replace the worst point in the community. It is very important for the effectiveness of SCE in guiding the search process.

2.3 Differential Evolution (DE) Algorithm

Differential Evolution (DE) algorithm [15] is a new heuristic approach mainly having three advantages; finding the true global minimum regardless of the initial parameter values, fast convergence, and using few control parameters. DE algorithm is a population based algorithm like GA using similar operators; crossover, mutation and selection. Meanwhile, in searching for better solutions, traditional GA heavily rely on crossover operation for local search; while DE relies more on the mutation operation instead. DE utilizes mutation operation for the solution search purposes, and applies the selection operation to direct the search toward the prospective regions in the variable space. In this subsection, we will provide a brief introduction to the DE algorithm. The general algorithm framework of DE is very similar to GA, and we will not provide its pseudo-code here but focus on introducing the three main operations as follows.

2.3.1 Mutation

Formally, let $\mathcal{P}^{(k)} = \{\boldsymbol{\theta}_1^{(k)}, \boldsymbol{\theta}_2^{(k)}, \cdots, \boldsymbol{\theta}_p^{(k)}\}$ denote a set of variables evolved to the k_{th} generation, where p denotes the population size. The variables in the initial generation, i.e., $\mathcal{P}^{(0)}$, are sampled randomly from the search space Θ if no prior knowledge about the problem optimal solution is available.

The crucial idea behind DE is the mutation scheme for generating trial variable vectors. DE generates new variable vectors by adding a weighted difference vector between two population members to a third member. If the resulting vector yields a lower objective function value than a predetermined population member, the newly generated vector will replace the vector with which it was compared in the following generation. The comparison vector can but need not be part of the generation process mentioned above. In addition, the best parameter vector $\boldsymbol{\theta}_{best}^{(k)}$ will be evaluated for every generation in order to keep track of the progress that is made during the minimization process. Several different mutation schemes are introduced in [15], and we will introduce them as follows.

Scheme 1: For each variable vector $\boldsymbol{\theta}_i^{(k)} \in \mathcal{P}^{(k)}$, DE will generate a trial vector $\mathbf{v}_i^{(k)}$ for it as follows:

$$\mathbf{v}_i^{(k)} = \boldsymbol{\theta}_{r_1}^{(k)} + F \cdot (\boldsymbol{\theta}_{r_2}^{(k)} - \boldsymbol{\theta}_{r_3}^{(k)}), \tag{6}$$

where the indexes r_1, r_2, r_3 are randomly selected from $\{1, 2, \cdots, p\}$ and $r_1, r_2, r_3 \neq i$. Term F is a real constant which controls the amplification of the differential variation term.

Scheme 2: Another scheme introduced in [15] is very similar to the above scheme 1, which further considers the best variable in the current generation when generating the trial vector $\mathbf{v}_i^{(k)}$ for the variables. Formally, let $\boldsymbol{\theta}_{best}^{(k)}$ denote the optimal variable in the current generation $\mathcal{P}^{(k)}$, which introduces the lowest objective function value. We can represent the generated vector for variable $\boldsymbol{\theta}_i^{(k)} \in \mathcal{P}^{(k)}$ as

$$\mathbf{v}_i^{(k)} = \boldsymbol{\theta}_i^{(k)} + \lambda \cdot (\boldsymbol{\theta}_{best}^{(k)} - \boldsymbol{\theta}_i^{(k)}) + F \cdot (\boldsymbol{\theta}_{r_2}^{(k)} - \boldsymbol{\theta}_{r_3}^{(k)}), \tag{7}$$

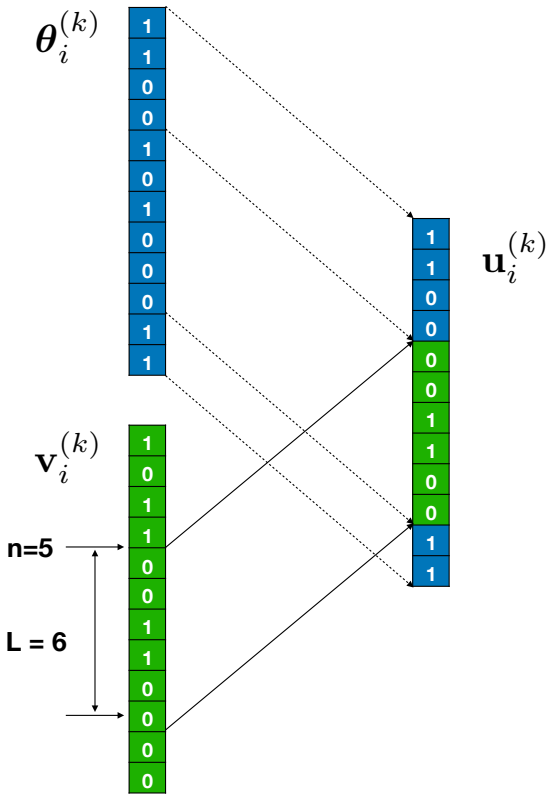

Figure 3: A Example of Crossover Operations in DE.

where λ is an additional introduced control parameter to control the greediness of the scheme by incorporating the current best vector $\boldsymbol{\theta}_{best}^{(k)}$. This new term can be extremely useful for objective functions where the global minimum is relatively easy to find.

2.3.2 Crossover

Based on the generated vector $\{\mathbf{v}_i^{(k)}\}_{i\in\{1,2,\cdots,p\}}$ (with either scheme 1 or scheme 2), DE proposes to crossover it with the original variable vector $\boldsymbol{\theta}_i^{(k)}$. By randomly selecting the crossover index point n as well as the crossover sequence length L, one segment of the variable values from vector $\mathbf{v}_i^{(k)}$ will be used to generate a new vector $\mathbf{u}_i^{(k)}$ of length d_θ. To be more specific, the entries in vector $\mathbf{u}_i^{(k)}$ can be represented with the following equation:

$$\mathbf{u}_i^{(k)}(j) = \begin{cases} \mathbf{v}_i^{(k)}(j), & \text{if } j \in \{n, n+1, \cdots, n+L-1\}, \\ \boldsymbol{\theta}_i^{(k)}(k), & \text{otherwise.} \end{cases} \tag{8}$$

Example 1 *For instance, as shown in Figure 3, given the two variable vectors* $\mathbf{v}_i^{(k)}$ *(in green color) and* $\boldsymbol{\theta}_i^{(k)}$ *(in blue color), by sampling the crossover starting index* n *and the crossover segment length* L, DE *generates a new vector* $\mathbf{u}_i^{(k)}$ *as shown at the right hand*

side. The entries $\mathbf{u}_i^{(k)}(5:10)$ *are from* $\mathbf{v}_i^{(k)}$*, and the remaining entries in* $\mathbf{u}_i^{(k)}$ *are from* $\boldsymbol{\theta}_i^{(k)}$ *instead.*

Algorithm 4 Sampling Approach of L

Require: Crossover probability p; Variable dimension d_θ; Starting index n
Ensure: Crossover segment length L
1: $L = 1$
2: **while** $rand() < p \wedge (n + L) < d_\theta$ **do**
3: $\quad L+ = 1$
4: **end while**
5: Return L

In the DE algorithm, the starting index n is usually sampled from set $\{1, 2, \cdots, d_\theta\}$ subject to the uniform distribution. DE determines the crossover segment length with the following pseudo-code in Algorithm 4.

In the algorithm, p is the crossover probability and $rand()$ denotes a random number sampled in range $[0, 1]$ subject to the uniform distribution. According to the algorithm, the probability to get a crossover segment of length no less than l will be $P(L \geq l) = p^{l-1}$.

2.3.3 Selection

The selection operation in DE is very simple. Based on the original variable $\boldsymbol{\theta}_i^{(k)}$ and the newly generated vector $\mathbf{u}_i^{(k)}$ for it, DE will select which one should be used in the next generation, i.e., generation $k + 1$, with the following equation:

$$\boldsymbol{\theta}_i^{(k+1)} = \begin{cases} \mathbf{u}_i^{(k)}, \text{ if } \mathcal{L}(\mathbf{u}_i^{(k)}) < \mathcal{L}(\boldsymbol{\theta}_i^{(k)}), \\ \boldsymbol{\theta}_i^{(k)}, \text{ otherwise.} \end{cases} \tag{9}$$

In other words, if vector $\mathbf{u}_i^{(k)}$ can lead to a smaller function value, it will be used to replace $\boldsymbol{\theta}_i^{(k)}$ in the next generation; otherwise, $\boldsymbol{\theta}_i^{(k)}$ value is retained in the next generation.

2.3.4 More Discussions on DE

Traditional direction search algorithms, including GA, actually adopt a greedy strategy for selecting good solutions, where variations are created on the solution vectors. Once a variation is generated, the algorithm will evaluate its benefits, which will be accepted as new solutions if it reduces the objective function value (for minimization problems). Such a kind of algorithm can converge fast but may get trapped into the local minima. DE resolves such a disadvantage by adopting a new mutation schema, which makes DE to be self-adaptive. In DE, all the solutions have the same chance to be selected as the parents without dependence of their fitness value. DE also adopts a greedy selection process: the better one of the new solution and its parent wins the competition providing significant advantages of converging performance over genetic algorithm. DE is a stochastic optimization algorithm, which can optimize the objective function and considering the constraints on the variables. Compared with other algorithms, DE has three advantages: (1) identifying true global minimum, (2) fast convergence, and (3) a few control parameters.

2.4 Particle Swarm Optimization (PSO) Algorithm

The PSO (Particle Swarm Optimization) algorithm mimics the social behavior of birds flocking and fishes schooling starting from a randomly distributed set of particles (i.e., the potential solution candidates). Similar to GA, the PSO algorithm will improve the solutions according to a quality measure (i.e., the fitness function) by moving the particles around the search space with a set of simple mathematical expressions. The PSO algorithm shares a lot of common elements with GA:

- Both initialize a population in a similar manner;
- Both use an evaluation function to determine how fit a potential solution is;
- Both are generational by repeating the same set of processes.

The PSO algorithm has two main operators: *velocity update* and *position update*. During each generation, each particle is accelerated toward the particle's previous best position and the global best position. In each generation, a new velocity value for each particle is calculated based on (1) its current velocity, (2) the distance from its previous best position, and (3) the distance from the global best position, which will be used to calculate the next position of the particle in the search space. The pseudo-code of the PSO algorithm is provided in Algorithm 5.

According to the algorithm, in lines 1-5, the algorithm initializes a set of variables, including the particle set, their variable, velocity and optimal variable vectors, respectively. In the learning process, for each particle, the algorithm will evaluate the function at the current variable to update the best seen variable (i.e., lines 9-11) as well as identify the local optimal neighbor index (i.e., lines 13-18), which will be used to update both the velocity vector as well as the variable vector with equations shown in lines 20-21. In the following part, we will introduce the specific representations of the function $\mathbf{v}_i = f(\boldsymbol{\theta}_i, \mathbf{v}_i, \boldsymbol{\theta}_i^*, \boldsymbol{\theta}_g^*)$ and function $\boldsymbol{\theta}_i = h(\boldsymbol{\theta}_i, \mathbf{v}_i)$ proposed in the existing works for different scenarios.

2.4.1 Binary PSO Algorithm

In the case when the search space is binary, i.e., $\Theta = \{0, 1\}^{d_\theta}$, the corresponding PSO used for learning the variable will be called the binary PSO algorithm [8]. In the algorithm, the velocity vector $\mathbf{v}_i$ for particle P_i keep records of the current velocity, whose value is determined by both the velocity in the previous round, the velocity of the historical best seen variable, as well as the optimal neighbor variable value. To differentiate the velocity vector in different iterations, we use $\mathbf{v}_i^{(\tau)}$ to denote the velocity in iteration τ. For vector $\mathbf{v}_i^{(\tau)} = [v_i^{(\tau)}(1), v_i^{(\tau)}(2), \cdots, v_i^{(\tau)}(d_\theta)]$, its j_{th} entry $\mathbf{v}_i^{(\tau)}(j)$ will be updated with the following equation:

$$\mathbf{v}_i^{(\tau)}(j) = f(\boldsymbol{\theta}_i^{(\tau-1)}, \mathbf{v}_i^{(\tau-1)}, \boldsymbol{\theta}_i^*, \boldsymbol{\theta}_g^*) \tag{10}$$

$$= \mathbf{v}_i^{(\tau-1)}(j) + c_1 \cdot \psi_1 \cdot \left(\boldsymbol{\theta}_i^*(j) - \boldsymbol{\theta}_i^{(\tau-1)}(j)\right) + c_2 \cdot \psi_2 \cdot \left(\boldsymbol{\theta}_g^*(j) - \boldsymbol{\theta}_i^{(\tau-1)}(j)\right), \tag{11}$$

where c_1, c_2 denote the two weight parameters (usually set with value 2.0), and terms ψ_1, ψ_2 represent two random number drawn from a uniform distribution between 0.0 and 1.0.

Algorithm 5 PSO Algorithm

Require: Variable search space Θ; Particle population g.
Ensure: Model Parameter $\boldsymbol{\theta}$

Initialize a set of particles $\mathcal{P} = \{P_1, P_2, \cdots, P_g\}$
for each particle $P_i \in \mathcal{P}$ **do**
  Initialize $\boldsymbol{\theta}_i$, $\mathbf{v}_i$ and $\boldsymbol{\theta}_i^*$ from search space Θ for each particle P_i
end for
Initialize convergence $tag = False$
while $tag == False$ **do**
  for each particle $P_i \in \mathcal{P}$ **do**
    /*Evaluate objective function $\mathcal{L}(\boldsymbol{\theta}_i)$ and update the optimal variable $\boldsymbol{\theta}_i^*$*/
    if $\mathcal{L}(\boldsymbol{\theta}_i) < \mathcal{L}(\boldsymbol{\theta}_i^*)$ **then**
      Set the best solution found so far $\boldsymbol{\theta}_i^* = \boldsymbol{\theta}_i$
    end if
    /*Set g to be the index of the optimal neighbor of P_i*/
    Set $g = i$
    for particle P_j in P_i's neighborhood set **do**
      if $\mathcal{L}(\boldsymbol{\theta}_j^*) < \mathcal{L}(\boldsymbol{\theta}_g^*)$ **then**
        Update $g = j$
      end if
    end for
    /*Update terms $\mathbf{v}_i$ and $\boldsymbol{\theta}_i$*/
    Update the velocity $\mathbf{v}_i$ term with the identified g, i.e., $\mathbf{v}_i = f(\boldsymbol{\theta}_i, \mathbf{v}_i, \boldsymbol{\theta}_i^*, \boldsymbol{\theta}_g^*)$
    Update the position $\boldsymbol{\theta}_i$ term, i.e., $\boldsymbol{\theta}_i = h(\boldsymbol{\theta}_i, \mathbf{v}_i)$
  end for
  if Convergence condition holds **then**
    Set $tag = True$
  end if
end while
Return $\boldsymbol{\theta}^* = \arg\max_{i \in \{1,2,\cdots,n\}} \mathcal{L}(\boldsymbol{\theta}_i)$

In many of the cases, there will exist a bound pair $[v_{min}, v_{max}]$ to constrain the possible values of vector $\mathbf{v}_i$ (values exceeding the bound will be smoothen accordingly).

Based on the updated velocity vector $\mathbf{v}_i^{(\tau)}$, the PSO algorithm will update the variable value vector $\boldsymbol{\theta}_i$. Generally, for the larger values in vector $\mathbf{v}_i^{(\tau)}$, the corresponding entry in vector $\boldsymbol{\theta}_i$ will be more likely to have value 1. The binary PSO algorithm will update the variable entry $\boldsymbol{\theta}_i^{(\tau)}(j)$ with the following equation:

$$\boldsymbol{\theta}_i^{(\tau)}(j) = h(\boldsymbol{\theta}_i^{(\tau-1)}, \mathbf{v}_i^{(\tau)}) = \begin{cases} 1, & \text{if } \psi_3 < \frac{1}{1+\exp(-\mathbf{v}_i^{(\tau)}(j))}, \\ 0, & \text{otherwise;} \end{cases} \tag{12}$$

where term ψ_3 is a random number selected with the uniform distribution from range $[0, 1.0]$.

2.4.2 Standard PSO Algorithm

In the case when the objective variables are real numbers, i.e., the search space is $\Theta = \mathbb{R}^{d_\theta}$, the variables will denote a point in the search space, and the PSO algorithm will be called

the standard PSO algorithm [9]. Formally, in the standard PSO algorithm, the velocity and variable vectors will be updated with the following equations respectively:

$$\mathbf{v}_i^{(\tau)}(j) = \mathbf{v}_i^{(\tau-1)}(j) + c_1 \cdot \psi_1 \cdot \left(\boldsymbol{\theta}_i^*(j) - \boldsymbol{\theta}_i^{(\tau-1)}(j)\right) + c_2 \cdot \psi_2 \cdot \left(\boldsymbol{\theta}_g^*(j) - \boldsymbol{\theta}_i^{(\tau-1)}(j)\right); \quad (13)$$

$$\boldsymbol{\theta}_i^{(\tau)}(j) = \boldsymbol{\theta}_i^{(\tau-1)}(j) + \mathbf{v}_i^{(\tau)}(j). \quad (14)$$

Similarly to the binary PSO, to avoid the oscillations of the velocity vector $\mathbf{v}_i^{(\tau)}$, there usually exist a tight lower and upper bounds $[v_{min}, v_{max}]$ for the entries, where the ones exceeding the boundaries will be smoothen effectively with values v_{min} and v_{max} respectively.

2.4.3 PSO with Inertia

In this part, we will introduce the PSO algorithm with inertia [14], which assigns the historical velocity with a weight in the velocity vector updating equation. For a nonzero weight, the algorithm will move the particles in the same direction as the previous iterations. Meanwhile, a decreasing weight over iterations will introduce a shift from the global search to the local search instead. Formally, the velocity and variable updating equation adopted in the PSO algorithm with inertia can be denoted as follows:

$$\mathbf{v}_i^{(\tau)}(j) = w^{(\tau)} \cdot \mathbf{v}_i^{(\tau-1)}(j) + c_1 \cdot \psi_1 \cdot \left(\boldsymbol{\theta}_i^*(j) - \boldsymbol{\theta}_i^{(\tau-1)}(j)\right) + c_2 \cdot \psi_2 \cdot \left(\boldsymbol{\theta}_g^*(j) - \boldsymbol{\theta}_i^{(\tau-1)}(j)\right); \quad (15)$$

$$\boldsymbol{\theta}_i^{(\tau)}(j) = \boldsymbol{\theta}_i^{(\tau-1)}(j) + \mathbf{v}_i^{(\tau)}(j), \quad (16)$$

where term $w^{(\tau)}$ denotes the inertia weight.

Generally, the weight term will be reduced linearly with iteration, from w_{start} to w_{end} (w_{start} and w_{end} are usually set with values 0.9 and 0.4 respectively). The updating equation of term $w^{(\tau)}$ can be represented as follows

$$w^{(\tau)} = \frac{(T_{max} - \tau) \cdot (w_{start} - w_{end})}{T_{max}} + w_{end}, \quad (17)$$

where T_{max} denotes the maximum iteration round allowed in the PSO algorithm. A the algorithm continues, the value of $w^{(\tau)}$ will gradually reduces from w_{start} to w_{end}.

2.4.4 PSO with Constriction Coefficient

Another PSO variant [1] to be introduced in this section incorporates the constriction coefficient into the velocity updating equation, which will lead to particle convergence over iterations, since the amplitude of the particle's oscillations decrease as it focus on the local and neighborhood previous best solutions. Meanwhile, the constriction coefficient also prevents colapse if the right social conditions are in place. The particle will oscillate around the weighted mean of the historical best solution and the optimal neighbor's best solution if they are near each other, which performs a local search in the space. On the other hand, if these solutions are far away from each other, the PSO algorithm will perform a global search instead. The constriction coefficient will balance between local search and global search effectively depending on the constriction coefficient value.

The updating equations of the velocity and variable vectors in the PSO with constriction coefficient algorithm can be represented as follows:

$$\mathbf{v}_i^{(\tau)}(j) = \chi \cdot \left[\mathbf{v}_i^{(\tau-1)}(j) + c_1 \cdot \psi_1 \cdot \left(\boldsymbol{\theta}_i^*(j) - \boldsymbol{\theta}_i^{(\tau-1)}(j)\right) + c_2 \cdot \psi_2 \cdot \left(\boldsymbol{\theta}_g^*(j) - \boldsymbol{\theta}_i^{(\tau-1)}(j)\right)\right]; \tag{18}$$

$$\boldsymbol{\theta}_i^{(\tau)}(j) = \boldsymbol{\theta}_i^{(\tau-1)}(j) + \mathbf{v}_i^{(\tau)}(j), \tag{19}$$

where $\chi = \frac{2k}{|2-\psi-\sqrt{\psi^2-4\psi}|}$ and $\psi = c_1 + c_2$. Normally, variable k is set with value 1 and c_1, c_2 are assigned with value 2 in the algorithm.

2.5 Evolution Strategy (ES) Algorithm

Evolution Strategy (ES) [4], also known as Evolutionary Strategy, is a search paradigm inspired by the principles of the biological evolution, which also belongs to the population based evolutionary algorithms. ES and GA work in a very similar way, involving selection, mutation and crossover (called recombination in ES), which were developed independent by two groups of researchers (ES was developed by the European computer scientists and GA was introduced the the USA computer scientists). In most of the cases, GA adopts a binary code for the solutions, while ES works well for the optimization functions with real-number variables. A more detailed discussions on the differences and similarities between ES and GA is available in [7]. In this subsection, we will first talk about the general algorithm framework of ES, where the detailed parameter control strategies used in ES will be introduced in the following subsection.

2.5.1 Algorithm Outline

The ES algorithm involves an iterative procedure, where new individuals will be created in each generation from the existing ones. Formally, to specify the learning settings of the algorithm, ES can be normally denoted as $(\mu/\rho \overset{+}{,} \lambda)$-ES. In the case where parameter ρ is not specified, the ES algorithm can also be denoted as $(\mu \overset{+}{,} \lambda)$-ES as well. Here, μ, ρ and λ are positive integers, whose physical meanings are denoted as follows:

- μ: the number of individuals in the parent set, i.e., the parent population.
- ρ: the number of parent individuals selected (out of μ parents) for recombination.
- λ: the number of offsprings generated in each iteration.
- $\overset{+}{,}$: the two selection modes of the algorithm. If notation "+" is used (i.e., the "plus" selection mode), the individual age is not considered in selection, and μ best of the $\mu + \lambda$ individuals will be selected as parents in each iteration. If notation "," is used (i.e., the "comma" selection mode), the senior individuals will die out after each iteration, and μ out of the generated λ offsprings will be selected as parents in each iteration.

In the (μ, λ)-ES, $\mu \leq \lambda$ should always hold; while in the $(\mu + \lambda)$-ES, $\mu \leq \lambda$ is not necessary any more and $\lambda = 1$ is also a feasible setting for the algorithm. In some cases, a subscript will

Algorithm 6 Evolution Strategy Algorithm

Require: Variable search space Θ; Variable dimension d_θ; Evolution parameters ρ, μ and λ
Ensure: Model Parameter $\boldsymbol{\theta}$

```
Initialize parent population P = {}
Initialize an instance with variable vector θ ∈ R^{d_θ} with control parameter s
Initialize convergence tag = False
while tag = False do
    for i ∈ {1, 2, ··· , λ} do
        θ_i, s_i = mutate(θ, s)
        P = P ∪ {(θ_i, s_i, L(θ_i))}
    end for
    P = select_by_age(P)
    P = select_μ_best(μ, P)      // ** optional **
    (θ, s) = recombine(select_mate(P, ρ), θ, s)
    if Convergence condition holds then
        Set tag = True
    end if
end while
Return θ* = arg min_{θ∈P} L(θ)
```

be attached to ρ to denote the combination modes: ρ_{I} for intermediate recombination and ρ_{W} for weighted recombination. Intermediate recombination is also the default recombination mode if the subscript is not indicated.

The pseudo-code of the $(\mu/\rho \overset{+}{,} \lambda)$-ES algorithm is provided in Algorithm 6, which accepts d_θ, ρ, μ and λ as the input. At the beginning, the algorithm will initialize an individual $\boldsymbol{\theta}$ at the beginning. The algorithm will also create a parameter variable s for it, which can covers control or endogenous strategy parameters, e.g., the success counter or a step-size that primarily serves to control the mutation. As the algorithm continues, a group of λ offspring instances will be generated via the mutation operations, where the μ good instances will be selected to form the new parent set. The initial variable $\boldsymbol{\theta}$ and parameter s will be updated with the selection and recombination operations. Detailed information about the operations used in the pseudo-code will be introduced in detail as follows.

2.5.2 Mate Selection and Recombination

Prior to the instance recombination, the ES algorithm adopts a mating selection step to pick individuals from the population to become the new parents. Depending on whether the function evaluation is involved in the mate selection or not, the existing mate selection strategies can be divided into two categories:

- **Fitness-based Mate Selection**: Such a mate selection approach utilizes the fitness ranking of the parent individuals to pick the good ones for recombination. The global environment selection step (i.e., function call "select_μ_best()" in Algorithm 6) can be omitted if this mate selection strategy is adopted.
- **Fitness-independent Mate Selection**: This mate selection approach picks the parent individuals doesn't depend on the fitness values of the individuals, which can be either deterministic or stochastic. For such a mate selection strategy, the global

environment selection step (i.e., function call "select_μ_best()" in Algorithm 6) will be necessary and required.

Based on the selected individuals, ES will call the "recombine()" function to combines information from several parents (the parent individual number is usually denoted by the parameter ρ) to generate a single new offspring. There also exist different types of recombination operators in the ES algorithm, and several important ones are introduced as follows respectively:

- **Discrete Recombination**: Such a recombination operator is also called the uniform crossover in GA, which randomly pick a value from the parents' corresponding variable vector for each of the variable entries. Formally, given the ρ parents available for recombination, for each entry $\boldsymbol{\theta}'_i(j), \forall j \in \{1, 2, \cdots, d_\theta\}$ in the offspring variable vector, the discrete recombination approach will select on individual from those ρ parents (e.g., $\boldsymbol{\theta}_k$) and fill in the entry with values from $\boldsymbol{\theta}_k$ (i.e., set $\boldsymbol{\theta}'_i(j) = \boldsymbol{\theta}_k(j)$).
- **Intermediate Recombination**: This recombination operator computes the average value of each variable of the selected ρ parent individuals as the corresponding variable vector for the newly generated offspring. Formally, we can represent the offspring variable vector as $\boldsymbol{\theta}_i = \frac{1}{\rho}\sum \boldsymbol{\theta}_k$, where the summation term will sums all the variable vectors of the ρ selected parent individuals.
- **Weighted Recombination**: The weighted recombination operator is a generalization of the intermediate recombination, which consider the variable vectors from all the ρ parent individuals (usually $\rho = \mu$) via a weighted sum. The weight values are computed based on the fitness ranking of the individuals, where the good individuals will get a weight no less than that of the inferior ones.

Similar to the crossover operator in GA, the recombination operator in ES will create variations in the population, which allows the algorithm to explore different regions of the search space. The discrete recombination works quite similar to the crossover operator, which adjust the search regions located at the vertices of the search region; while the intermediate recombination and weighted recombination allow the algorithm to search along the edges of the hyper-rectangle of the search region instead.

2.5.3 Mutation and Parameter Control

The mutation operation will introduce some "small" variation to the variable vectors, which allows the ES to jump to other search regions for further explorations. ES introduces the perturbation vector from a multivariate normal distribution, e.g., $\epsilon \sim \mathcal{N}(\mathbf{0}, \mathbf{C})$, with zero mean and covariance matrix $\mathbf{C} \in \mathbb{R}^{d_\theta \times d_\theta}$. Formally, as shown in Algorithm 6, given the variable vector $\boldsymbol{\theta}'$ obtained from the recombination, we can represent its corresponding vector after mutation as $\boldsymbol{\theta}'' = \boldsymbol{\theta}' + \epsilon$, which can be treated as a vector draw from distribution $\boldsymbol{\theta}' + \mathcal{N}(\mathbf{0}, \mathbf{C})$ (or the equivalent distribution $\mathcal{N}(\boldsymbol{\theta}', \mathbf{C}^{\frac{1}{2}}\mathcal{N}(\mathbf{0}, \mathbf{I}))$).

Therefore, the covariance matrix $\mathbf{C}$ determines the mutation step in the ES algorithm, and the selection of different type of matrix $\mathbf{C}$ will lead to different types of mutations in ES. In Figure 4, we show several examples of the mutation distribution plots with different covariance matrices $\mathbf{C}$.

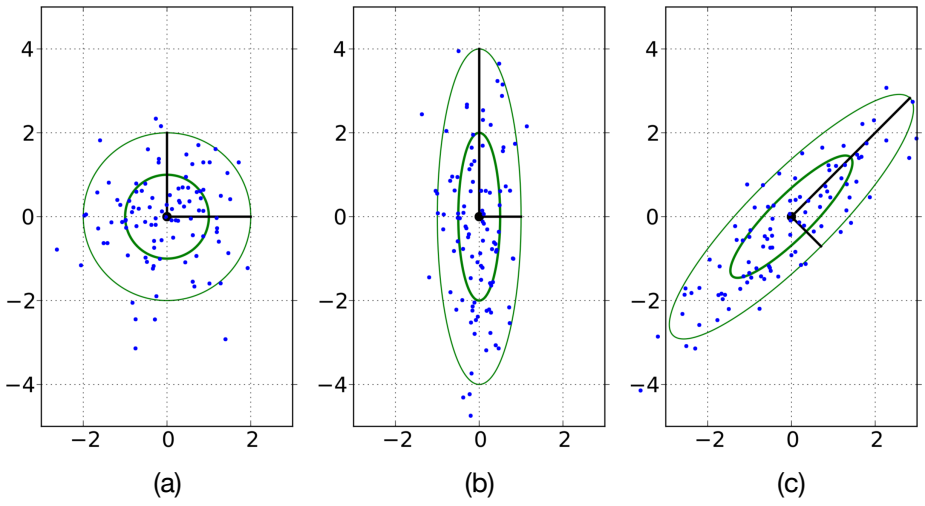

Figure 4: Examples of ES Mutation Distribution.

- If the different dimensions of the mutation distribution are independent but with common variance, i.e., $\mathbf{C} = c \cdot \mathbf{I}$ (which denotes $\mathbf{C}$ is a diagonal matrix with constant c on its diagonal), the corresponding distribution $\mathbf{C}^{\frac{1}{2}}\mathcal{N}(\mathbf{0}, \mathbf{I})$ will be a re-scaled normal Gaussian distribution and its distribution plot will be in a spherical shape (as illustrated in plot (a) of Figure 4).
- If the different dimensions of the mutation distribution are independent but with different variances, i.e., $\mathbf{C} = diag(\sigma^2)$ (here $diag(\sigma^2)$ denotes a diagonal matrix with σ^2 on its diagonal), the distribution region of distribution $\mathbf{C}^{\frac{1}{2}}\mathcal{N}(\mathbf{0}, \mathbf{I})$ will be in a ellipsoid shape with principal axes parallel to the coordinate axes (as illustrated in plot (b) of Figure 4).
- If the covariance matrix is positive definite (i.e., $\mathbf{x}^\top \mathbf{C}\mathbf{x} > 0, \forall \mathbf{x} \in \mathbb{R}^{d_\theta}$), which denotes a general case covering the previous two special case as well, the corresponding distribution plot will be in a spherical shape with principal axes pointing to any directions (as illustrated in plot (c) of Figure 4).

In the ES algorithm, controlling the parameter $\mathbf{C}$ is the key to design the evolution strategy. Several different parameter control approaches have been introduced, including the 1/5 success rule for parameter control [12], self-adaption [13], derandomized self-adaption [10], *cumulative step-size control* (CSA) [11, 6], *covariance matrix adaption* (CMA) [5] and *natural evolution strategies* (NES) [17]. In the following part, we will introduce one of them, i.e., CMA, and the ES algorithm with CMA based parameter control is also called the CMA-ES.

2.5.4 CMA-ES

In this part, we will take the CMA-ES algorithm [5] as an example to introduce a concrete implementation of ES. The key necessary steps involved in CMA-ES include:

Algorithm 7 CMA-ES Algorithm

Require: Variable search space Θ; Variable dimension d_θ; Evolution parameters ρ, μ and λ
Ensure: Model Parameter $\boldsymbol{\theta}$

Set parameters to be used in the algorithm
Set evolution path vectors $\mathbf{p}^{(0)} = \mathbf{0}$ and $\mathbf{q}^{(0)} = \mathbf{0}$, covariance matrix $\mathbf{C}^{(0)} = \mathbf{I}$ and $g = 0$
Initialize mean vector $\mathbf{m}^{(0)} \in \mathbb{R}^{d_\theta}$ with step-size $\sigma^{(0)} \in \mathbb{R}$
Initialize convergence $tag = False$
while $tag = False$ **do**
    Initialize population set $\mathcal{P} = \{\}$
    for $i \in \{1, 2, \cdots, \lambda\}$ **do**
        // ** **Sample population individuals:** **
        $\boldsymbol{\theta}_i^{(g+1)} \sim \mathbf{m}^{(g)} + \sigma^{(g)}\mathcal{N}(\mathbf{0}, \mathbf{C}^{(g)})$
        $\mathcal{P} = \mathcal{P} \cup \{(\boldsymbol{\theta}_i, \mathcal{L}(\boldsymbol{\theta}_i))\}$
    end for

    // ** **Update Mean Vector:** **
    $\mathbf{m}^{(g+1)} = \mathbf{m}^{(g)} + c_m \cdot \sum_{i=1}^{\mu} w_i \cdot (\boldsymbol{\theta}_i^{(g+1)} - \mathbf{m}^{(g)})$

    // ** **Update covariance matrix:** **
    $\mathbf{p}^{(g+1)} = (1 - c_c)\mathbf{p}^{(g)} + \sqrt{c_c(2 - c_c)\mu_{eff}}\frac{\mathbf{m}^{(g+1)} - \mathbf{m}^{(g)}}{\sigma^{(g)}}$
    $\mathbf{C}^{(g+1)} = (1 - c_1 - c_\mu \sum_{i=1}^{\lambda} w_i)\mathbf{C}^{(g)} + c_1 \mathbf{p}^{(g+1)}{\mathbf{p}^{(g+1)}}^\top + c_\mu \sum_{i=1}^{\lambda} w_i(\boldsymbol{\theta}_i^{(g+1)} - \mathbf{m}^{(g+1)})(\boldsymbol{\theta}_i^{(g+1)} - \mathbf{m}^{(g+1)})^\top$

    // ** **Update step-size:** **
    $\mathbf{q}^{(g+1)} = (1 - c_\sigma) \cdot \mathbf{q}^{(g)} + \sqrt{c_\sigma(2 - c_\sigma)\mu_{eff}} \cdot (\mathbf{C}^{(g)})^{-\frac{1}{2}} \frac{\mathbf{m}^{(g+1)} - \mathbf{m}^{(g)}}{\sigma^{(g)}}$
    $\sigma^{(g+1)} = \sigma^{(g)} \exp\left(\frac{c_\sigma}{d_\sigma}\left(\frac{\|\mathbf{q}^{(g+1)}\|}{\mathsf{E}\|\mathcal{N}(\mathbf{0}, \mathbf{I})\|} - 1\right)\right)$

    if Convergence condition holds **then**
        Set $tag = True$
    end if
    $g = g + 1$
end while
Return $\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta} \in \mathcal{P}} \mathcal{L}(\boldsymbol{\theta})$

- **Mutation**: In CMA-ES, a population of new search points will be generated by sampling a multivariate normal distribution. Formally, let $\mathbf{m}^{(g)}$, $\sigma^{(g)}$ and $\mathbf{C}^{(g)}$ denote the updated mean vector, step size and covariance matrix from the g_{th} generation. Based on them, we can generate the variable individuals in the $(g+1)_{th}$ generation with

$$\boldsymbol{\theta}_i^{(g+1)} \sim \mathbf{m}^{(g)} + \sigma^{(g)}\mathcal{N}(\mathbf{0}, \mathbf{C}^{(g)}), \forall i \in \{1, 2, \cdots, \lambda\}. \tag{20}$$

  Therefore, we can represent the obtained population for generation $g + 1$ as a set $\mathcal{P} = \{\boldsymbol{\theta}_1^{(g+1)}, \boldsymbol{\theta}_2^{(g+1)}, \cdots, \boldsymbol{\theta}_\lambda^{(g+1)}\}$.

- **Selection and Recombination**: CMA-ES adopts the weighted recombination to compute the algorithm new mean for generation $g+1$, which can be denoted as

$$\mathbf{m}^{(g+1)} = \mathbf{m}^{(g)} + c_m \cdot \sum_{i=1}^{\mu} w_i \cdot (\boldsymbol{\theta}_i^{(g+1)} - \mathbf{m}^{(g)}), \tag{21}$$

  where $c_m \leq 1$ denotes the learning rate in CMA-ES, term w_i denotes the weight of individual $\boldsymbol{\theta}_i^{(g+1)}$ and we have $\sum_{i=1}^{\mu} w_i = 1$. Here, individuals in the population set $\mathcal{P}$ will be sorted according to their evaluations of the objective function, and the top μ individuals are selected from the population set $\mathcal{P}$. In other words, the mate selection strategy used in CMA-ES is fitness dependent. Different ways can be used to define the weight terms $\{w_1, w_2, \cdots, w_\mu\}$, and setting $w_i = \frac{1}{\mu}, \forall i \in \{1, 2, \cdots, \mu\}$ will reduce the recombination to the intermediate recombination.

- **Covariance Matrix Adaption**: CMA-ES updates the covariance matrix iteratively, and the updating equation considers information from three different perspectives: (1) current covariance matrix, (2) covariance computed with the newly mutated individuals, and (3) covariance computed with the evolution path. Formally, the covariance matrix updating equation can be represented as follows:

$$\mathbf{C}^{(g+1)} = (1-c_1-c_\mu \sum_{i=1}^{\lambda} w_i)\mathbf{C}^{(g)} + c_1 \mathbf{p}^{(g+1)} {\mathbf{p}^{(g+1)}}^\top + c_\mu \sum_{i=1}^{\lambda} w_i (\boldsymbol{\theta}_i^{(g+1)} - \mathbf{m}^{(g+1)})(\boldsymbol{\theta}_i^{(g+1)} - \mathbf{m}^{(g+1)})^\top, \tag{22}$$

  where these three terms denote the covariance matrices mentioned above, and c_1, c_μ represent the weights of the last two terms. Vector $\mathbf{p}^{(g+1)}$ denote the evolution path vector of the $(g+1)_{th}$ generation, and it can be denoted as

$$\mathbf{p}^{(g+1)} = (1-c_c)\mathbf{p}^{(g)} + \sqrt{c_c(2-c_c)\mu_{eff}} \frac{\mathbf{m}^{(g+1)} - \mathbf{m}^{(g)}}{\sigma^{(g)}}, \tag{23}$$

  where $\mu_{eff} = (\sum_{i=1}^{\mu} w_i^2)^{-1}$ is the variance effective selection mass and $c_c \leq 1$ is a weight term. The factor $\sqrt{c_c(2-c_c)\mu_{eff}}$ is a normalization constant for $\mathbf{p}$. For $c_c = 1$ and $\mu_{eff} = 1$, the factor reduces to 1 and $\mathbf{p}^{(g+1)}$ reduces to $\mathbf{p}^{(g+1)} = \frac{\mathbf{m}^{(g+1)} - \mathbf{m}^{(g)}}{\sigma^{(g)}}$.

- **Step-Size Control**: In addition to the covariance matrix adaption, CMA-ES will also update the step-size which controls the variation scale in the mutation. Formally, the updating equation of the step-size can be denoted with the following equation:

$$\sigma^{(g+1)} = \sigma^{(g)} \exp\left(\frac{c_\sigma}{d_\sigma}\left(\frac{\left\|\mathbf{q}^{(g+1)}\right\|}{\mathsf{E}\left\|\mathcal{N}(\mathbf{0}, \mathbf{I})\right\|} - 1\right)\right). \tag{24}$$

  In the above equation, $d_\sigma \approx 1$ is the damping parameter, and vector $\mathbf{q}^{(g+1)}$ denote the conjugate evolution path vector in generation $g+1$, which can be denoted as

$$\mathbf{q}^{(g+1)} = (1-c_\sigma) \cdot \mathbf{q}^{(g)} + \sqrt{c_\sigma(2-c_\sigma)\mu_{eff}} \cdot (\mathbf{C}^{(g)})^{-\frac{1}{2}} \frac{\mathbf{m}^{(g+1)} - \mathbf{m}^{(g)}}{\sigma^{(g)}}, \tag{25}$$

  where $c_\sigma < 1$ is a weight term.

Based on these steps introduced above, we can provide the pseudo-code of the CMA-ES algorithm in Algorithm 7.

Algorithm 8 Hill Climbing Algorithm

Require: Variable search space Θ;
Ensure: Model Parameter $\boldsymbol{\theta}$
Initialize a random variable vector $\boldsymbol{\theta} \in \Theta$
Initialize the stop $tag = False$
while $tag = False$ **do**
  Compute the neighbor set $\Gamma(\boldsymbol{\theta})$ of $\boldsymbol{\theta}$
  Evaluate the function and compute $\{\mathcal{L}(\boldsymbol{\theta}_i)\}_{\boldsymbol{\theta}_i \in \Gamma(\boldsymbol{\theta})}$
  Set $\boldsymbol{\theta}'$ to be the optimal neighbor in $\Gamma(\boldsymbol{\theta})$ with the largest function evaluation value
  if $\mathcal{L}(\boldsymbol{\theta}') \leq \mathcal{L}(\boldsymbol{\theta})$ **then**
    Set $tag = True$
  else
    Set $\boldsymbol{\theta} = \boldsymbol{\theta}'$
  end if
end while
Return $\boldsymbol{\theta}$ as the solution

3. Random Search Algorithms

In this part, we will introduce several other derivative-free optimization algorithms based on generic random search, which don't belong to the above three categories of algorithms that we have introduced before in this paper and in [18]. Many of the algorithms introduced above actually may also belong to the random search algorithm category, e.g., GA and ES. Random search algorithms are useful for many ill-structured global optimization problems with continuous and/or discrete variables. The random search algorithms to be introduced here include Hill Climbing and Simulated Annealing.

3.1 Hill Climbing

The hill climbing algorithm to be introduced here has a close relation with the gradient descent algorithms we introduced in the previous tutorial article [19], which all search for the solutions via an iterative search to maximize/minimize the function evaluation at the current state. Meanwhile, hill climbing is also very different from gradient descent, since it doesn't require any derivative computation in the optimization process. In hill climbing, starting at the base of a hill, we walk upwards until we reach the top of the hill. In other words, hill climbing starts with initial state and keeps improving the solution until reaching its (local) optimum.

The pseudo-code of the hill climbing algorithm is provided in Algorithm 8. The algorithm involves several key steps:

- **Initialization**: The hill climbing algorithm starts with a random point in the search space, which can be denoted as $\boldsymbol{\theta} \in \Theta$.

- **Neighborhood Exploration**: Based on the current state, the hill climbing algorithm iteratively searches the nearby points of $\boldsymbol{\theta}$ to determine the moving direction for the next step. Formally, these nearby points define the neighbor set of $\boldsymbol{\theta}$, which can be denoted as $\Gamma(\boldsymbol{\theta})$. Different ways can be used to define the "neighbor" in hill climbing. For instance, given a vector $\boldsymbol{\theta}$, we can enumerate all the entries in the vector to

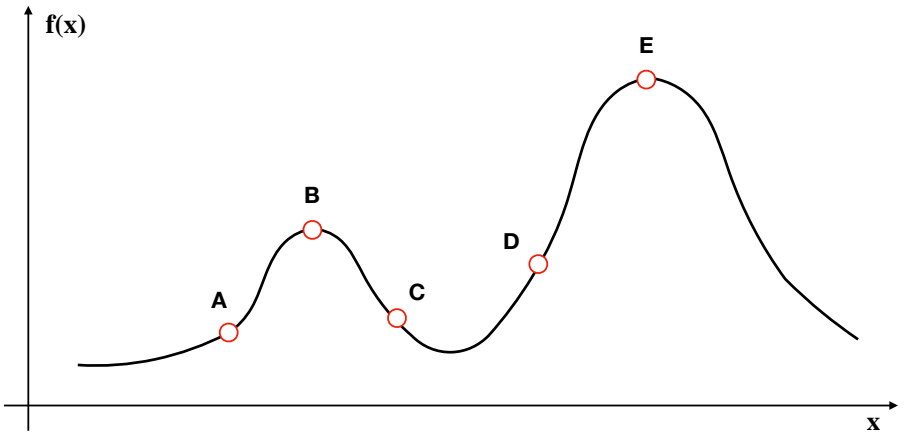

Figure 5: An Example of the Hill Climbing Algorithm.

add/minus 1 to define a group of new nearby vectors as the neighbors, which can be denoted as $\{[\boldsymbol{\theta}(1), \boldsymbol{\theta}(2), \cdots, \boldsymbol{\theta}(i) \pm 1, \cdots, \boldsymbol{\theta}(d_\theta)]\}_{i \in \{1,2,\cdots,d_\theta\}}$.

- **Neighborhood Evaluation**: Hill climbing will evaluate the objective function $\mathcal{L}(\cdot)$ at these neighbor points, and pick the optimal one with the minimum function evaluation, e.g., $\boldsymbol{\theta}'$.
- **Stop Criterion**: If the optimal neighbor variable is better than the current variable, i.e., $\mathcal{L}(\boldsymbol{\theta}') \geq \mathcal{L}(\boldsymbol{\theta})$, the algorithm will moves to that neighbor point. Otherwise, the algorithm will stop and return the current variable as the output.

Example 2 *For instance, in Figure 5, we provide an example to illustrate how the hill climbing algorithm works in learning the optimal variable to the single-variable function $f(x)$. Let A denote the starting point, where optimization process starts. The hill climbing algorithm will select its neighbor point B as the successor to evaluate the function. We know that $f(B) > f(A)$. The algorithm will move the current state to B and the search process will continue to point C. Noticing that $f(C) < f(B)$, the climbing algorithm will stop there and return B as the optimal solution, which is actually a local maximum point of the function.*

3.2 Simulated Annealing

According to the above description as well as Example 2, we can discover that the hill climbing algorithm is a greedy algorithm and can be short-sighted in the optimization process, especially for the non-convex functions. For instance, in Example 2, the algorithm is only able to identify a local maximum B but fail discover the global optima E, which is nearby to B actually. In this part, we will introduce the simulated annealing algorithm, which is a probabilistic technique for approximating the global optimum of a given function. Simulated annealing allows the continuous search in the worse-regions subject to a decreasing probability.

Algorithm 9 Simulated Annealing Algorithm

Require: Variable search space Θ; Temperature decreasing factor $\alpha \in [0, 1]$; Max iteration I_{max}
Ensure: Model Parameter $\boldsymbol{\theta}$

```
Initialize a random variable vector θ ∈ Θ
Initialize iteration count c = 0
while iteration c ≤ I_max do
    Compute the neighbor set Γ(θ) of θ
    Select a point θ' from set Γ(θ)
    if L(θ') ≥ L(θ) then
        Set θ = θ'
    else
        if exp((L(θ') − L(θ))/T) > random(0, 1) then
            Set θ = θ'
        end if
    end if
    Update T = α · T
    Update c = c + 1
end while
Return θ as the solution
```

The name, i.e., "simulated annealing", and inspiration come from annealing in metallurgy, a technique involving heating and controlled cooling of a material to increase the size of its crystals and reduce their defects. The simulation of annealing can be used to find an approximation of a global minimum for a function with a large number of variables. The simulated annealing algorithm allows the search in the region which is worse than the current state, but the probability for such a kind of worse-region search decreases step by step. This notion of slow cooling implemented in the simulated annealing algorithm is interpreted as a slow decrease in the probability of accepting worse solutions as the solution space is explored. Accepting worse solutions is a fundamental property of meta-heuristics because it allows for a more extensive search for the global optimal solution.

In general, the simulated annealing algorithms work as follows.

- The algorithm initializes the search by randomly selecting one point in the search space.
- At each time step, the algorithm randomly selects a solution close to the current one, measures its quality, and then decides to move to it or to stay with the current solution.
- During the search, the temperature is progressively decreased from an initial positive value to zero and affects the worse-region exploration probabilities, i.e., the probability of moving to a worse new solution is progressively changed towards zero.

The pseudo-code of the simulated annealing algorithm is provided in Algorithm 9. According to the pseudo-code, the algorithm accepts the temperature decreasing factor $\alpha \in [0, 1]$ and the maximum iteration I_{max} as the input. Via an iterative search, the algorithm will accept a new neighbor point $\boldsymbol{\theta}'$ iff (1) $\mathcal{L}(\boldsymbol{\theta}') \geq \mathcal{L}(\boldsymbol{\theta})$; or (2) $\mathcal{L}(\boldsymbol{\theta}') < \mathcal{L}(\boldsymbol{\theta})$ but $random(0, 1) < \exp\left(\frac{\mathcal{L}(\boldsymbol{\theta}') - \mathcal{L}(\boldsymbol{\theta})}{T}\right)$. In other words, the algorithm will explore to $\boldsymbol{\theta}'$

if it leads to a better performance, or it will explore a worse region with probability $P(T; \boldsymbol{\theta}', \boldsymbol{\theta}) = \exp\left(\frac{\mathcal{L}(\boldsymbol{\theta}') - \mathcal{L}(\boldsymbol{\theta})}{T}\right)$. The probability term $P(T; \boldsymbol{\theta}', \boldsymbol{\theta})$ depends on both the function evaluations at points $\boldsymbol{\theta}'$ and $\boldsymbol{\theta}$, as well as the temperature term T. Noticing the T term will be updated iteratively with $T = T \times \alpha$ to 0 as the algorithm continues, which will lower down the probability value $P(T)$ for worse-region exploration steadily.

Example 3 *For instance, we can still use the example shown in Figure 5, where A is the starting point for search. The hill climbing algorithm will stop at B since further exploration will lead to worse results. However, the simulated annealing algorithm allows us to take the risk in further searching the region after C with a certain probabilities, e.g., D and E, even though C and D are both worse than B. It will allow the algorithm to reach the global optimum E at the final.*

4. Summary

In this paper, as a follow-up of [18], we have introduce several other categories of derivative-free optimization algorithms, including both the population based algorithms and several other random search based algorithms. Many of these introduced algorithms can be potentially applied to learn the deep neural network models. This tutorial article will also be updated accordingly as we observe more new developments on this topic in the near future.

References

[1] M. Clerc. The swarm and the queen: towards a deterministic and adaptive particle swarm optimization. In *Proceedings of the 1999 Congress on Evolutionary Computation-CEC99 (Cat. No. 99TH8406)*, volume 3, pages 1951–1957 Vol. 3, July 1999.

[2] Benjamin Doerr, Huu Phuoc Le, Régis Makhmara, and Ta Duy Nguyen. Fast genetic algorithms. *CoRR*, abs/1703.03334, 2017.

[3] Q. Y. Duan, V. K. Gupta, and S. Sorooshian. Shuffled complex evolution approach for effective and efficient global minimization. *Journal of Optimization Theory and Applications*, 76(3):501–521, Mar 1993.

[4] Nikolaus Hansen, Dirk V. Arnold, and Anne Auger. Evolution strategies. In *Handbook of Computational Intelligence*, pages 871–898. Springer, 2015.

[5] Nikolaus Hansen and Andreas Ostermeier. Completely derandomized self-adaptation in evolution strategies. *Evol. Comput.*, 9(2):159–195, June 2001.

[6] Nikolaus Hansen, Andreas Ostermeier, and Andreas Gawelczyk. On the adaptation of arbitrary normal mutation distributions in evolution strategies: The generating set adaptation. In *Proceedings of the 6th International Conference on Genetic Algorithms*, pages 57–64, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[7] Frank Hoffmeister and Thomas Bäck. Genetic algorithms and evolution strategies: Similarities and differences. In Hans-Paul Schwefel and Reinhard Männer, editors, *Parallel Problem Solving from Nature*, pages 455–469, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg.

[8] J. Kennedy and R. C. Eberhart. A discrete binary version of the particle swarm algorithm. In *1997 IEEE International Conference on Systems, Man, and Cybernetics. Computational Cybernetics and Simulation*, volume 5, pages 4104–4108 vol.5, Oct 1997.

[9] James Kennedy and Russell C. Eberhart. *Swarm Intelligence*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[10] Andreas Ostermeier, Andreas Gawelczyk, and Nikolaus Hansen. A derandomized approach to self-adaptation of evolution strategies. *Evol. Comput.*, 2(4):369–380, December 1994.

[11] Andreas Ostermeier, Andreas Gawelczyk, and Nikolaus Hansen. Step-size adaptation based on non-local use of selection information. In Yuval Davidor, Hans-Paul Schwefel, and Reinhard Männer, editors, *Parallel Problem Solving from Nature — PPSN III*, pages 189–198, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg.

[12] I. Rechenberg. *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution.* PhD thesis, TU Berlin, 1971.

[13] H.-P Schwefel. *Numerische Optimierung von Computer-Modellen mittels der Evolutionsstrategie*, volume 26 of *ISR*. Birkhaeuser, Basel/Stuttgart, 1977.

[14] Yuhui Shi and Russell C. Eberhart. Parameter selection in particle swarm optimization. In V. W. Porto, N. Saravanan, D. Waagen, and A. E. Eiben, editors, *Evolutionary Programming VII*, pages 591–600, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[15] Rainer Storn and Kenneth Price. Differential evolution – a simple and efficient heuristic for global optimization over continuous spaces. *J. of Global Optimization*, 11(4):341–359, December 1997.

[16] Darrell Whitley. A genetic algorithm tutorial. *Statistics and Computing*, 4(2):65–85, Jun 1994.

[17] Daan Wierstra, Tom Schaul, Tobias Glasmachers, Yi Sun, Jan Peters, and Jürgen Schmidhuber. Natural evolution strategies. *J. Mach. Learn. Res.*, 15(1):949–980, January 2014.

[18] Jiawei Zhang. Derivative-free global optimization algorithms: Bayesian method and lipschitzian approaches, 2019.

[19] Jiawei Zhang. Gradient descent based optimization algorithms for deep learning models training, 2019.

[20] Jiawei Zhang and Fisher B. Gouza. GADAM: genetic-evolutionary ADAM for deep neural network optimization. *CoRR*, abs/1805.07500, 2018.