Vrije Universiteit Amsterdam

Research Paper Business Analytics

Effectiveness and efficiency of the CMA-ES algorithm

Author:
Ronald van Zwienen

Supervisor:
Prof. Dr. A.E. Eiben

November 16, 2018

Abstract

This research paper discusses the Evolutionary Computing algorithm CMA-ES. It is investigated how the CMA-ES algorithm performs relative to a Bayesian Optimizer. Two measures are used to assess the performance: the speed of convergence and the number of function evaluations.

The optimizers are used to optimize the controller of robots. The characteristics of such controller functions are typically rugged, noisy and ill-conditioned. Therefore, a test suite is made up that consists of nine functions who have the characteristics mentioned.

From the testing results it can be concluded that in general the CMA-ES algorithm is more effective and the BO algorithm is more efficient for a few function evaluations. As more function evaluations are acceptable, CMA-ES outperform BO in terms of efficiency.

Contents

1 Introduction

This research paper aims to provide insight into the effectiveness and efficiency of the CMA-ES optimizer. The degree of effectiveness is expressed in the distance from the optimum given the number of iterations. The closer the optimizer has reached the optimum, given a number of iterations, the better the solution quality. The efficiency is expressed in the number of function evaluations that are needed to reach the optimum of an objective function. Since the number of function evaluations is a relative term, the results are compared with a Bayesian Optimization algorithm. The results have been compared with a Bayesian Optimization algorithm because, like CMA-ES, it is suitable for optimizing black box functions and does not require derivatives. The research question that is answered in this paper is:

How effective and efficient is the CMA-ES algorithm compared to the Bayesian Optimization algorithm?

The concrete problem behind this research lies in evolutionary robotics where the morphologies (bodies) of the robots are evolvable. In such a system a newborn robot has a new morphology and it needs to learn how to move this body quickly after birth. That is, it has to learn (optimize) its own controller (brain) for an objective function that measures the ability to locomote. The learner / optimizer should be fast in the first place. That is, it is acceptable to sacrifize efficacy for efficiency.

Since this paper focuses specifically on the optimization part of the problem, the evolutionary robotics part of the problem will be neglected. Hence, the concrete problem outlined has been transformed into a more abstract problem. The abstract problem amounts to optimizing a noisy (stochastic) objective function.

2 Sample-Efficient Optimizers

There are many types or classes of optimizers. However, they all have in common that they use the insight that not all function values need to be known to reach an optimum. Typically an optimizer uses a so-called surrogate model. A surrogate model is a mathematical data-driven model that mimics the behavior of another model, the objective function, as closely as possible while being computationally cheap(er) to evaluate [1].

Since there are many types of optimizers, we can not mention them all. To restrict ourselves, we start with the No Free Lunch (NFL) theorem. This theorem is well described by Wolpert [2] and states that the performance of any pair of algorithms a_1 and a_2 over all possible objective functions is the same. As a consequence, if an algorithm performs relatively better than other algorithms on one class of problems its performance is necessarily offset on the remaining problems. Therefore, it is important to first examine the characteristics of the objective function or the class of objective functions one wants to optimize. Thereafter, an optimizer can be selected that is capable to mimic this characteristics.

Characteristics of the objective function
Objective functions in evolutionary robotics have the following characteristics:

- Dimensionality. Objective functions in evolutionary robotics generally have several dozen dimensions.
- Ill-conditioned. In mathematics, the condition number of a function with respect to an argument measures how much the output value of the function can change for a small change in the input argument. A function with a high condition number is said to be ill-conditioned.
- Non-separable. Functions that have dependencies between the objective variables have this characteristic. As a result, the variance can be high between different estimates of the optimum.
- Rugged. This term refers to functions that are non-smooth, discontinuous, multi-modal and / or noisy. Multi-modality means that there are several, at least locally, optimal solutions. A discontinuous function has at least one point in its domain where it has no derivative.

Based on these characteristics, derivative-free sample-efficient stochastic optimizers are the class of optimizers we are interested in. An optimizer is sample efficient when it learns fast by exploring the important parts of the domain of an objective function while ignoring the less relevant parts of the domain. Two main strategy types will be further described in this chapter:

1. Bayesian Optimization algorithms
2. Evolutionary algorithms

2.1 Bayesian Optimization (BO) algorithms

A BO optimizer uses a Gaussian Process (GP) to find the optimum of an objective function f. A GP is a collection of random variables, any finite number of which have (consistent) joint Gaussian distributions [3]. The difference between a Gaussian distribution and a GP is that a Gaussian distribution is over vectors and a GP is over functions. The surrogate model in case of BO can be written as:

$$f \sim \mathcal{GP}\left(m, k\right), \tag{1}$$

which means that the function f is distributed as a GP with mean function m and covariance or kernel function k. A BO algorithm typically starts with two points, x_1 and x_2, in the domain of the objective function, f, and fits a Gaussian Process (GP) over these two points. Then the GP is updated with an iterative procedure until the GP reaches the optimum of f or a certain stop condition. This iterative procedure is given in Algorithm 1.

Algorithm 1 Bayesian Optimization algorithm

1: **procedure** UPDATEGAUSSIANPROCESS
2: **for** $t = 1, 2, ...$ **do**
3: Find x_t by combining attributes of the posterior distribution in an utility function u and maximizing: $x_t = argmax_x\, u\left(x|\mathcal{D}_{1:t-1}\right)$.
4: Sample the objective function f: $y_t = f\left(x_t\right) + \epsilon_t$.
5: Augment the data $\{\mathcal{D}_{1:t} = \mathcal{D}_{1:t-1}, (x_t, y_t)\}$ and update the GP.

The interesting part in Algorithm 1 is the utility function u. The utility function u determines which point x will be evaluated at iteration t. Regarding the utility function u, there is a trade-off between exploration and exploitation. When a utility function focuses on exploration, places in the objective function domain with high variance are selected. When a utility function focuses on exploitation, places with a low mean (in case of minimization) are selected. Therefore, the optimization result depends to a large extent on which utility function and which parameters are chosen.

2.2 Evolutionary algorithms

Evolutionary algorithms are inspired by biological evolutionary principles, such as reproduction, mutation, recombination and selection. The Evolutionary Computing terminology and his equivalent optimization terminology

is given in Table 1. A main difference between BO algorithms relative to evolutionary algorithms is that BO algorithms examine one new point x per iteration whereas evolutionary algorithms simultaneously consider multiple candidate solutions per iteration.

Table 1: Evolutionary Computing terminology

Evolutionary Computing		**Optimization**
individual, parent	$\Leftrightarrow$	candidate solution, decision variables
population	$\Leftrightarrow$	set of candidate solutions
fitness function	$\Leftrightarrow$	objective function
generation	$\Leftrightarrow$	iteration

Pseudo-code of the main steps of an evolutionary algorithm is given in Algorithm 2.

Algorithm 2 Evolutionary Algorithm Scheme

```
Initialize population with random candidate solutions
Evaluate each candidate
while Termination condition is not satisfied do
    Select parents
    Recombine pairs of parents
    Mutate the resulting offspring
    Evaluate new candidates
    Select individuals for the next generation
```

Typically, the starting point of an evolutionary algorithm is a population with random candidate solutions. These candidate solutions $f(x)$ are obtained by drawing random x_i's from the x-domain of the objective function. After making the initial population, each candidate solution is evaluated and evolution operators are applied. There are three evolution operators:

1. Selection,
2. Recombination or crossover and
3. Mutation.

The selection operator acts on a population level. Based on their fitness, the μ best candidates are selected and serve as parents for the next generation. The recombination and mutation operator act on an individual level. Recombination means that children inherit characteristics from their parents, whereas mutation is adding some randomness to keep the next generation diverse.

All three steps are crucial for proper functioning of an evolutionary algorithm. This because using selection alone will tend to fill the population with copies of the best individual from the population and therefore does not lead to improvement of the population. Furthermore, using selection and crossover operators only will tend to cause the algorithms to converge on a good but sub-optimal solution instead of converging to a global optimum. Using mutation alone induces a random walk through the search space. And finally, Using only selection and mutation creates a parallel, noise-tolerant, hill climbing algorithm.

At this point, we see some similarities between Bayesian Optimization algorithms and Evolutionary Algorithms. In subsection 2.1 we stated for Bayesian optimizers that there is a trade-off between exploration and exploitation. In the case of using Evolutionary Algorithms we have the exact same problem. Evolutionary Algorithms have a trade-off between novelty and quality. On one hand, we want to increase the diversity of the population by the variation operators recombination and mutation. These operators push the population towards novelty. On the other hand, the diversity of the population is decreased by selection of parents, which pushes the population to quality.

Finally, it can be stated that there are different classes of Evolutionary Algorithms, which can be seen as a flavour on the foregoing of this subsection. Historically different flavours of Evolutionary Algorithms have been associated with different data types to represent solutions. For example, Genetic Algorithms use binary strings to represent solutions, while Evolution Strategies use real-valued vectors. One Evolutionary Strategy is Covariance Matrix Adaption - Evolutionary Strategy (CMA-ES). This optimizer is described in the next chapter.

3 Covariance Matrix Adaption - Evolutionary Strategy (CMA-ES)

The CMA-ES algorithm consists of five steps:

1. Sampling a population, using a multivariate Normal distribution,
2. Evaluating the sampled population on the fitness function,
3. Selection of a portion of the population, which are the best fit solutions,
4. Updating the current distribution using the information of the best fit solutions,
5. Repeating the foregoing steps until a stopping criterion is met.

These steps are explained in more detail in the rest of this chapter. Hereby, intensive use is made of a paper of Hansen [3], including his notation.

3.1 Population sampling

At each iteration, a population is sampled from the multi-variate Normal distribution. A population consist of λ new search points. Each search point x_i is sampled according Equation (2).

$$x_i^{(g+1)} \sim \mathcal{N}\left(\mathbf{m}^{(g)}, \left(\sigma^{(g)}\right)^2 \mathbf{C}^{(g)}\right), \quad for\ i = 1, ..., \lambda \tag{2}$$

In this equation $x_i^{(g+1)}$ is the i'th search point from generation $g+1$. The vector $\mathbf{m}$ is the mean vector of the multi-variate normal distribution, which is initially set to a random point in the domain or if one knows how the objective function looks like, a point near the possible optimum. The matrix $\mathbf{C}$ is the covariance matrix and is initially set to the identity matrix $\mathbf{I}$, whereas $\left(\sigma^{(g)}\right)^2$ is the step size at generation g.

3.2 Population evaluation

The sampled population is evaluated by retrieving the value of the fitness function for each $x_i^{(g+1)}$.

3.3 Selection of best solutions

In case of minimization of the fitness function, μ of the λ $x_i^{(g+1)}$'s are selected, such that $f\left(x_{1:\lambda}^{(g+1)}\right) \leq f\left(x_{2:\lambda}^{(g+1)}\right) \leq ... \leq f\left(x_{\mu:\lambda}^{(g+1)}\right)$.

3.4 Updating distribution parameters

From Equation (2) it can be seen that the used Normal distribution has three parameters:

1. A mean vector $\mathbf{m}$,
2. A covariance matrix $\mathbf{C}$ and
3. A step size control parameter σ^2.

All three parameters are updated at each iteration.

3.4.1 Updating the mean vector

At each iteration, the mean vector $\mathbf{m}$ is updated according to Equation (3).

$$\mathbf{m}^{(g+1)} = c_m \sum_{i=1}^{\mu} w_i \mathbf{x}_{i:\lambda}^{(g+1)} \equiv \mathbf{m}^{(g)} + c_m \sum_{i=1}^{\mu} w_i (\mathbf{x}_{i:\lambda}^{(g+1)} - \mathbf{m}^{(g)}) \tag{3}$$

For the weights w_i the following holds:

$$c_m \sum_{i=1}^{\mu} w_i = 1, \quad w_1 \geq w_2 \geq ... \geq w_\mu > 0 \tag{4}$$

If all w_i is set to $1/\mu$, then $\mathbf{m}^{(g+1)}$ is the equally weighted mean of the μ best solutions. The constant $c_m \leq 1$ is a learning rate, which is usually set to 1. If $c_m < 1$ and the step size $\sigma = 1/c_m$, then the $x_i^{(g+1)}$ are sampled from a wider range. This can be advantageous on noisy functions with (a lot of) local optima.

3.4.2 Updating the covariance matrix

It is known that the sample variance of a Gaussian distribution can be estimated with the unbiased estimator s^2. The bias of an estimator is the difference between an estimator's expected value and the true value of the estimator being estimated. If an estimator has zero bias, the estimator is called unbiased. The estimator s^2 is computed according to Equation (5).

$$s^2 = \frac{n}{n-1}\hat{\sigma}^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \mathbf{m})^2 \tag{5}$$

Therefore, the sample or empirical covariance matrix $\mathbf{C}$ of a sampled generation in case of the CMA-ES is:

$$\mathbf{C}_{\lambda}^{(g+1)} = \frac{1}{\lambda-1}\sum_{i=1}^{\lambda}\left(\mathbf{x}_i^{(g+1)} - \mathbf{m}^{(g)}\right)\left(\mathbf{x}_i^{(g+1)} - \mathbf{m}^{(g)}\right)^T \tag{6}$$

To obtain a better covariance matrix, that is, a covariance matrix which results in better $x_i's$ in the next generation, only the μ best x_i values can be used. By also using the weighted selection mechanism from Equation (4), Equation (7) is obtained.

$$\mathbf{C}_{\mu}^{(g+1)} = \sum_{i=1}^{\mu} w_i \left(\mathbf{x}_{i:\lambda}^{(g+1)} - \mathbf{m}^{(g)}\right)\left(\mathbf{x}_{i:\lambda}^{(g+1)} - \mathbf{m}^{(g)}\right)^T \tag{7}$$

Sampling from $\mathbf{C}_{\mu}^{(g+1)}$ tends to produce better fitness function values, since the sampled x_i's are closer to the optimum. To achieve fast search, the population size λ must be small. However, for small populations this estimation can be influenced by outliers and becomes more unreliable. Hence, the covariance matrix adaptation must be adjusted, which is usually done with the so-called rank-μ-update.

Rank-μ update
To obtain a reliable estimate of the covariance matrix for small population sizes, the weighted mean of the estimated covariance matrices from all generations can be used.

$$\mathbf{C}^{(g+1)} = \frac{1}{g+1}\sum_{i=1}^{g}\frac{1}{\sigma^{(i)^2}}\mathbf{C}_{\mu}^{(i+1)} \tag{8}$$

In Equation (8) all covariance matrices have the same weight. In order to assign higher weights to more recent generations, exponential smoothing can be used.

$$\mathbf{C}^{(g+1)} = (1 - c_\mu)\,\mathbf{C}^{(g)} + c_\mu \frac{1}{\sigma^{(g)^2}} \mathbf{C}_\mu^{(g+1)} \tag{9}$$

Using Equation (7), this can be written as:

$$\mathbf{C}^{(g+1)} = (1 - c_\mu)\,\mathbf{C}^{(g)} + c_\mu \sum_{i=1}^{\mu} w_i \left(\frac{\mathbf{x}_{i:\lambda}^{(g+1)} - \mathbf{m}^{(g)}}{\sigma^{(g)}} \right) \left(\frac{\mathbf{x}_{i:\lambda}^{(g+1)} - \mathbf{m}^{(g)}}{\sigma^{(g)}} \right)^T \tag{10}$$

In Equation (9) and (10) $c_\mu \leq 1$ is the learning rate for updating the covariance matrix. If $c_\mu = 0$, no learning takes place and $\mathbf{C}^{(g+1)} = \mathbf{C}^{(0)}$. If $c_\mu = 1$, no prior information is used. Hansen et al [2] examined that approximately 37% of the information in $\mathbf{C}^{(g+1)}$ is older than $1/c_\mu$ generations. Therefore, the choice of c_μ is very important. If c_μ is low, the learning is very slow whereas if c_μ is large the covariance matrix $\mathbf{C}$ degenerates.

Rank-One Update

In the limit case, only a single point can be used for updating the covariance matrix at each generation and the covariance matrix is updated according to Equation (11).

$$\mathbf{C}^{(g+1)} = (1 - c_\mu)\,\mathbf{C}^{(g)} + c_\mu \left(\frac{\mathbf{x}_{1:\lambda}^{(g+1)} - \mathbf{m}^{(g)}}{\sigma^{(g)}} \right) \left(\frac{\mathbf{x}_{1:\lambda}^{(g+1)} - \mathbf{m}^{(g)}}{\sigma^{(g)}} \right)^T \tag{11}$$

Since in Equation (11) the maximum likelihood is added to the covariance matrix, the probability to generate the maximum likelihood in the next generation increases. If in Equation (11) the term $\mathbf{x}_{i:\lambda}^{(g+1)} - \mathbf{m}^{(g)}$ is negative, the negative sign vanished. Therefore, in practice this term is replaced by the evolution path p_c resulting in Equation (12).

$$\mathbf{C}^{(g+1)} = (1 - c_\mu)\,\mathbf{C}^{(g)} + c_\mu \mathbf{p}_c^{(g+1)} \mathbf{p}_c^{(g+1)^T} \tag{12}$$

An evolution path $\mathbf{p}_c$ can be expressed by a sum of consecutive steps. This summation is referred to as cumulation. To construct an evolution path, the step-size σ is disregarded. In the CMA-ES algorithm, the evolution path $\mathbf{p}_c$ is defined as in Equation (13).

$$\mathbf{p}_c = (1 - c_c)\,\mathbf{p}_c^{(g)} + \sqrt{c_c\,(2 - c_c)\,\mu_{eff}} \frac{\mathbf{m}^{(g+1)} - \mathbf{m}^{(g)}}{\sigma^{(g)}} \tag{13}$$

As in Equation (9) and (10) the constant $c_c \leq 1$ in Equation (13) is the learning rate, $1/c_c$ is the backward time horizon of the evolution path $\mathbf{p}_c$ that contains roughly 63% of the overall weight. A time horizon between $\sqrt{n}$ and n is effective. Finally, the term $\sqrt{c_c\left(2-c_c\right)\mu_{eff}}$ is a normalization constant.

Practically every CMA-ES implementation puts the above together, leading to Equation (14).

$$\mathbf{C}^{(g+1)} = \left(1 - c_1 - c_\mu\right)\mathbf{C}^{(g)} + c_1 \mathbf{p}_c^{(g+1)} \mathbf{p}_c^{(g+1)^T} + c_\mu \sum_{i=1}^{\mu} w_i \left(\frac{\mathbf{x}_{i:\lambda}^{(g+1)} - \mathbf{m}^{(g)}}{\sigma^{(g)}}\right)\left(\frac{\mathbf{x}_{i:\lambda}^{(g+1)} - \mathbf{m}^{(g)}}{\sigma^{(g)}}\right)^T \tag{14}$$

3.4.3 Updating the step size

Finally, the step size σ is updated according Equation (15).

$$\sigma^{(g+1)} = \sigma^{(g)} exp\left(\frac{d_\sigma}{c_\sigma}\left(\frac{\parallel \mathbf{p}_\sigma^{(g+1)} \parallel}{E \parallel \mathcal{N}\left(\mathbf{0}, \mathbf{I}\right) \parallel} - 1\right)\right) \tag{15}$$

There are a lot of parameters involved in the CMA-ES algorithm. However, three parameters have the biggest influence on the efficiency of this algorithm. These parameters are given in Table 2. Only these parameters will be taken into account when testing this algorithm on the test suite.

Table 2: Most relevant parameters of CMA-ES

Parameter	**Function in model**
λ	Population size: effects the diversity of the population.
μ	Number of parents: effects the quality of the population.
σ	Step size: the degree of mutation.

4 Experimental set-up

To test the effectiveness and efficiency of the CMA-ES optimizer and compare it to an Bayesian optimizer, both type of optimizers must be tested on the

same objective functions. Therefore, a test suite has been composed which is described in section 4.1. The goal is to determine the effectiveness and efficiency of the CMA-ES on the test suite. The statistics used are described in section 4.2. Finally, 4.3 describes how the algorithms are implemented in Python.

4.1 Test Suite

The optimizers will be evaluated with the COCO (Comparing Continuous Optimizers) benchmark functions, as presented by Flinck et al (2009) [4][5], which has been used for the Black-Box-Optimization-Benchmarking (BBOB) workshops during GECCO conferences in several years, as well as for the Congress on Evolutionary Computation (CEC) in 2015. Nine of these benchmark functions have been selected and form the test suite. These functions are given in Table 3. In choosing these nine functions we mainly aimed to construct a diverse test suite, meaning that the functions have different complexities and varying shapes. The test suite consists of five noiseless functions and four noisy functions. The noisy functions are noisy counterparts of the noiseless functions. This, in order to analyze the effect of the noise on the performance of the optimizers.

Table 3: Test Suite Functions

Noiseless functions		**Noisy functions**	
Number	**Name**	**Number**	**Name**
f_1	Sphere	f_{102}	Sphere with moderate uniform noise
f_8	Rosenbrock	f_{110}	Rosenbrock with Gaussian noise
f_{17}	Schaffers F7	f_{124}	Schaffers F7 with seldom Cauchy noise
f_{21}	Gallagher's Gaussian 101-me Peaks	f_{128}	Gallagher's Gaussian peaks, 101-me with Gaussian noise
f_{23}	Katsuura		

The numbering of the functions (e.g. f_1) corresponds to the BBOB numbering. Each test function has several instances. Each instance is randomly

shifted in both the x-space and the f-space. By optimizing various instances it is easy to determine how the optimizer performs on the type of function in question without changing any starting points. Finally, the f-axes are upside down and all function are to be minimized, so the optima are peaks.

Since a robot usually has several dozens of dimensions and to figure out the effect of adding more dimensions to the problem, the optimizers will be tested on real-valued vectors of 2, 5, 10, 20 and 40 dimensions.

4.2 Statistics

The goal of an optimizer is, given a starting point, moving as fast as possible to the optimum. Therefore, we are interested in both effectiveness (solution quality) and efficiency (speed).

4.2.1 Effectiveness

In Evolutionary Computing, the Mean Best Fitness (MBF) is a common measure to determine the extent to which the algorithm is able to find the optimum [1]. The MBF can be determined by recording the fitness of the best individual at termination for each run. That is, for each of the 50 instances per test function the best fitness at termination is recorded. Then, the MBF of the test suite is the average of these values over all test functions and instances.

Since each instance is randomly shifted in both the x-space and the f-space, the best fitness value at termination is difficult to interpret. Because of the random shifts, the optimum also shifts and hence differs per instance. Therefore, to measure the effectiveness, the distance from the starting point to the optimum in the f-space is used. Furthermore, the distance traveled from the starting point to the optimum is measured in percentages. This is because not every starting point is just as far away from the optimum. So if a value of 0.2 is measured at iteration t, this means that the distance to the optimum has been reduced by 80 percent relative to the starting point.

4.2.2 Efficiency

The number of function evaluations has been used to assess the speed of the CMA-ES algorithm. In case of the CMA-ES the number of function

evaluations depends on the population size at each iteration. In case of BO, at each iteration one new point is evaluated.

4.3 Python implementation

The optimizers are tested in Python on the test suite. For the CMA-ES algorithm, the library *cma* is used. The code below shows how this package was used to optimize the first instance of test function f_{128} in 10 dimensions.

```
import cma
from cma import bbobbenchmarks as bb

dim = 10
initial_sigma = 200
popsize = (4 + int(3*np.log(dim)))
CMA_mu = int(popsize / 2)

opts = cma.CMAOptions({'popsize': popsize, 'CMA_mu': CMA_mu})
objective = bb.F128(1)

es = cma.CMAEvolutionStrategy(dim * [mean], initial_sigma, opts)
number_of_iterations = 100
for k in range(number_of_iterations):
   x_values = es.ask()
   fitness_values = [objective.evaluate(x) for x in x_values]
   es.tell(x_values, fitness_values)
   es.logger.add()
```

The BO algorithm is implemented with the package *BayesianOptimization* from the library *bayes_opt*. The code below shows that the *BayesianOptimization* package was used to optimize the first instance of test function f_{128} in 2 dimensions.

```
from bayes_opt import BayesianOptimization

objective = bb.F128(1)
def target(x, x2):
    return -objective.evaluate([x, x2])

bo = BayesianOptimization(target, {'x': (MIN, MAX), 'x2': (MIN,
```

```
    MAX)})
kernel = gp.kernels.Matern()
opts = {'kernel': kernel, 'alpha': 1e-5}

bo.maximize(init_points=2, n_iter=0, acq='ei', **opts, kappa=10)
for i in range(number_of_iterations):
   bo.maximize(init_points=0, n_iter=1, acq='ei', **gp_params,
      kappa=10)
   best_fitness = bo.res['max']['max_val']
```

The parameter *kappa* determines whether the focus is on exploration or exploitation. The lower *kappa*, the more the emphasis is placed on exploitation. Furthermore, the Expected Improvement (EI) utility function was used. This utility function is defined as:

$$EI(x) = \underset{max}{\mathbb{E}}\left[f(\mathbf{x}) - f(\mathbf{x}^+), 0)\right],$$

where $f(\mathbf{x}^+)$ is the value of the best sampled $\mathbf{x}_i$ so far.

5 Results

This section describes the effectiveness and efficiency of both the CMA-ES and the BO optimizer on the test suite.

But before both the effectiveness and the efficiency are identified, the initial σ of the CMA-ES algorithm will be determined. This is done because this parameter has a major influence on the number of iterations needed. In terms of optimization: the initial σ can be seen as an exploration parameter. If the initial σ is too low, then few new areas of the objective function are discovered in a few iterations and hence the speed of convergence is low. On the other hand, if the initial σ is too high, then you can quickly get close to the optimum, but it is difficult to actually reach the optimum, because a wide range is being discovered for candidate solutions. That is why it is important to first find a good value for the initial σ.

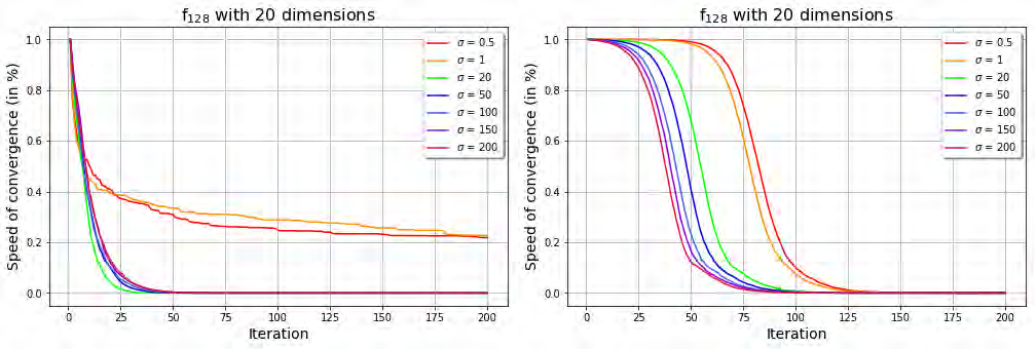
5.1 Effect of initial σ of the CMA-ES algorithm

Figure 1 shows the effect of the initial σ on the speed of convergence. The y-axis shows the distance to the optimum relative to the starting point per iteration in percentages.

The plots are obtained by testing the CMA-ES on the first 50 instances of function f_{128} from the test suite. This function is used, since it is one of the most noisy functions in the test suite. This prevents a value from being selected whereby the algorithm gets stuck in a local optimum.

From Figure 1, it can be seen that for low σ values is much slower than for higher σ values. Furthermore, there is a clear effect on the speed of convergence with regard to the average distance from the starting point to the optimum. In Figure 1a an initial mean vector of zeros is used, which is relatively closer to the optimum than a vector of 100,000's, which is used in Figure 1b. If the starting point is further away from the optimum, there is a kind of a warm up period perceptible. Once the CMA-ES has found the right direction, it goes to the optimum just as quickly as with a higher sigma.

Based on the plots in Figure 1, a value of 200 is used as an initial σ, when testing the functions of the test suite. Regardless of the starting point, this value performs very reasonably in terms of speed of convergence.



(a) (b)

Figure 1: Effect of initial σ on convergence with initial mean the $\mathbf{0}^T$-vector (a) and the $\mathbf{100{,}000}^T$-vector (b).

A fairly high initial σ is also logical, since the algorithm in the first iteration does not know anything about the form of the fitness function yet. The emphasis should then be on exploration instead of exploitation.

5.2 Effectiveness

Figure 2 shows the effectiveness of both optimizers on the five noiseless test functions of the test suite. The effectiveness is measured as the distance from

the optimum, relative to the starting point, at termination. Figure 2 shows the effectiveness of the optimizers for different termination conditions. As termination condition, a specified number of iterations has been used.

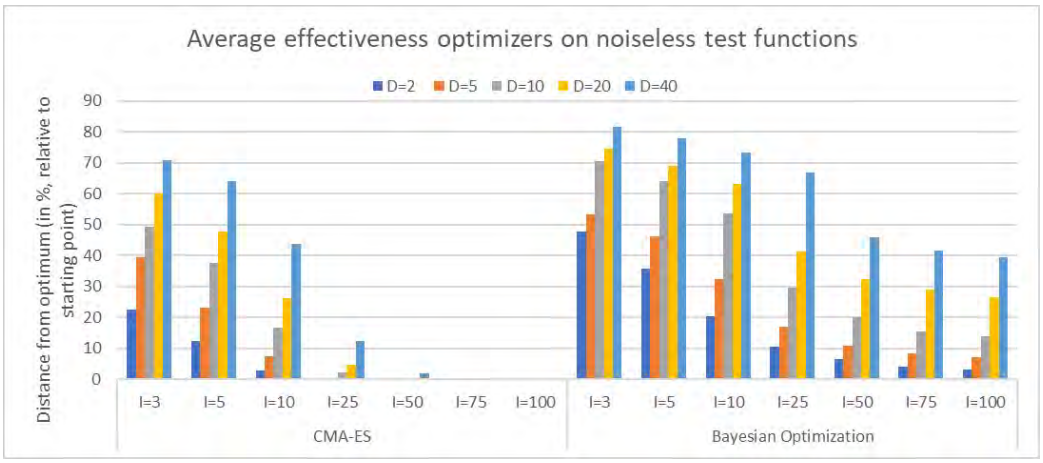


Figure 2: Effectiveness of CMA-ES and BO on noiseless test functions, with D the number of dimensions and I the number of iterations (stop condition).

Figure 2 shows that regardless of the termination condition, the effectiveness of the CMA-ES is better. That is, at termination the CMA-ES approaches the optimum more than the BO algorithm. Furthermore, in case of BO the improvement of the solution quality is relatively limited after 50 iterations. One cause is the *kappa* parameter in the BO algorithm. This parameter determined the degree of exploration and exploitation and is fixed. Based on Figure 2, this parameter should actually be adjusted after 50 iterations so that the focus is more on exploitation. This, because after 50 iterations the BO algorithm do not find new promising areas, taking into account the limited increase of the solution quality.

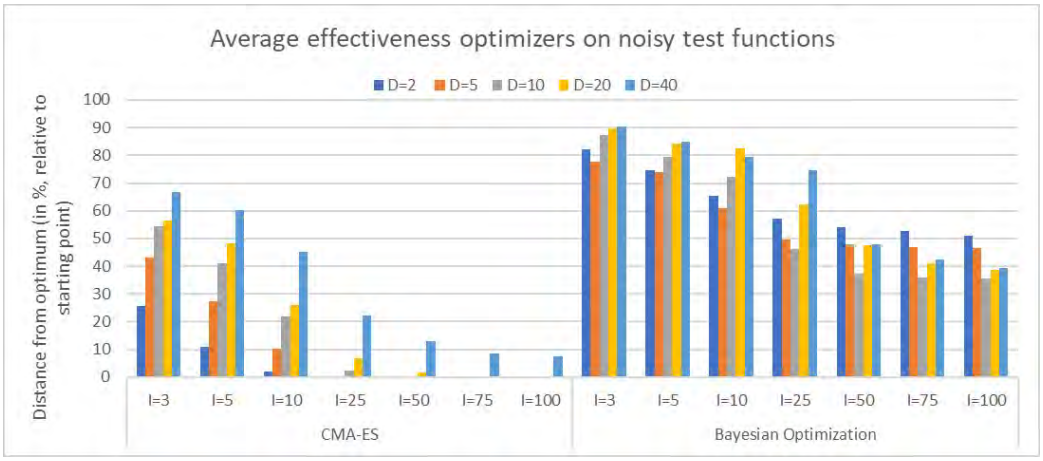


Figure 3: Effectiveness of CMA-ES and BO on noisy test functions, with D the number of dimensions and I the number of iterations (stop condition).

Figure 3 shows the effectiveness of the optimizers on the four noisy test functions. Figure 3 shows a similar picture as with the noiseless test functions. The solution quality of the CMA-ES is in all cases better. However, there are some differences with the noiseless functions. To begin with, the solution quality on the noisy test functions is less good than with the noiseless functions. Furthermore, given the stop condition, the results of the different dimensions are relatively close to each other with BO. And finally, it is striking that BO does not perform better for 2-dimensional instances than for 5-dimensional instances. And 5-dimensional instances do not perform better than 10-dimensional instances. This is counterintuitive. A possible explanation is that the used BO parameters work differently at different dimensions on noisy functions.

5.3 Efficiency

The efficiency of the CMA-ES is measured by the number of iterations and the number of evaluations it takes to reach the optimum. In Table 4, 95%-Confidence Intervals (CI's) for both the number of iterations and the number of evaluations are given for several dimensions. From the table it can be seen that as the number of dimensions increases the number of iterations and evaluations also increases.

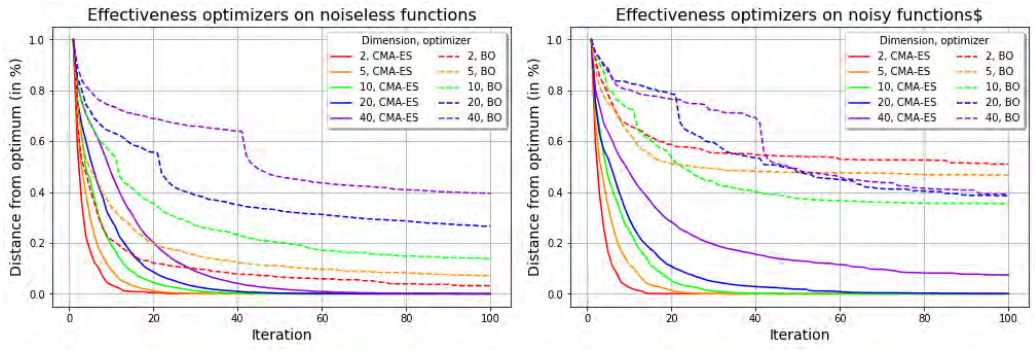
Table 4: 95%-Confidence Intervals Efficiency Test Suite

	Number of dimensions			
	2	**5**	**10**	**20**
Evaluations				
Mean	1465.69	3446.60	8298.73	21014.87
Standard deviation	1070.53	2144.12	8401.22	30570.75
CI upper bound	1535.63	3586.69	8847.61	23012.16
CI lower bound	1395.75	3306.52	7749.85	19017.58
Iterations				
Mean	244.28	430.83	829.87	1751.24
Std dev.	178.42	268.01	840.12	2547.56
CI upper bound	255.94	448.34	884.76	1917.68
CI lower bound	232.62	413.32	774.99	1584.80

The CI's are computed from testing the first 50 instances of each of the nine test functions twice. In this way, 900 values have been generated for each dimension. The CI is than computed as:

$$CI = \bar{X} \pm 1.96 \frac{\sigma}{\sqrt{n}}$$

Figure 4 shows for both optimizers how the optimum is approached for the first 100 iterations.

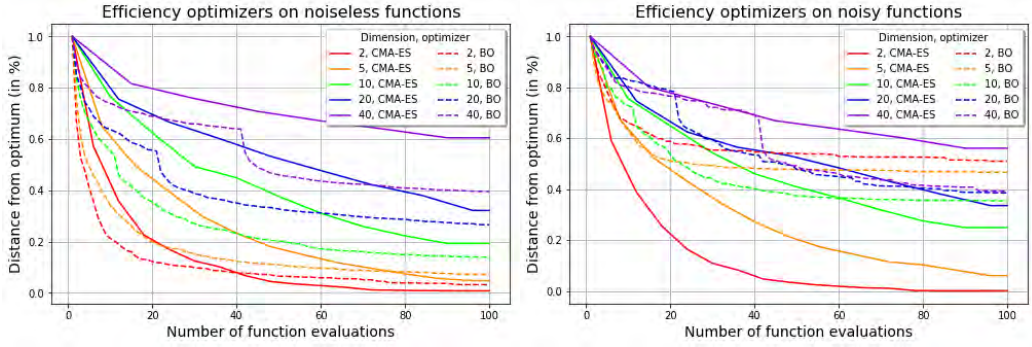


(a) (b)

Figure 4: Effectiveness of CMA-ES and BO on noiseless test functions (a) and Effectiveness CMA-ES on noisy test functions (b).

However, this does not give a good picture of the efficiency because the number of function evaluations per iteration differs strongly per optimizer. BO uses one function evaluation per iteration. However, the CMA-ES uses a population size of 4 + 3*ln(number of dimensions) per iteration. That means that for 2-dimensional instances there are 6 evaluations per iteration, while for 40-dimensional instances the population size of 15 is used. Therefore the results of the CMA-ES must be adjusted so that a comparison can take place based on the number of function evaluations.

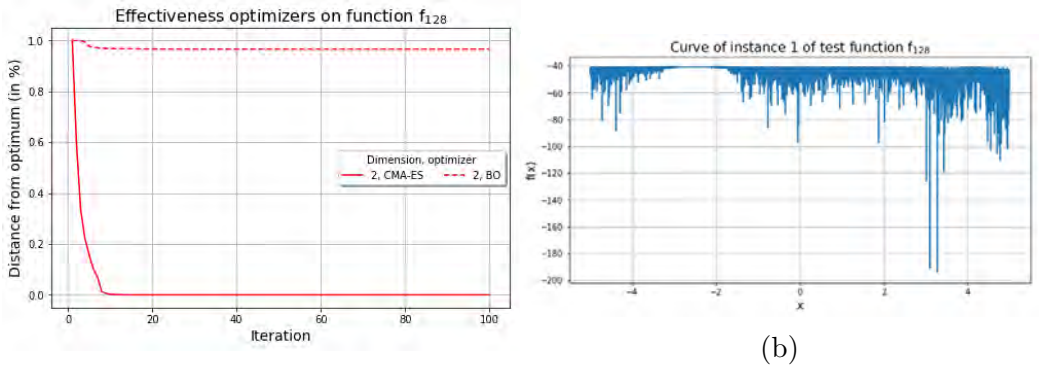
Figure 5 shows the speed of convergence per function evaluation. In the noiseless case, the BO algorithm in general performs better than the CMA-ES. In case of noisy functions the CMA-ES performs better. However, within the first 100 function evaluations BO performs better in the 40-dimensional case. This is due to the fact that the CMA-ES uses 15 function evaluations per iteration in the 40-dimensional case. Therefore, one iteration with the CMA-ES algorithms corresponds to 15 iterations with BO.



(a) (b)

Figure 5: Efficiency of CMA-ES and BO on noiseless (a) and noisy (b) test functions

Striking during testing was that the CMA-ES performs very consistently across the different dimensional instances and functions. BO shows the opposite. Figure 6 shows the great difference in performance between the two optimizers. For an important part, the disappointing result of BO is due to the fact that the objective function is very rugged. BO fits a smooth GP which aims to approach the objective function. Because the objective function is far from smooth, the BO algorithm fails.



(a) (b)

Figure 6: Performance of the optimizers on test function f_{128}

6 Conclusion

The goal of this research paper is to compare two sample-efficient optimizers based on their effectiveness and efficiency on a test suite. We are especially interested in speed of convergence to the global optimum. Based on the results, some conclusions can be drawn.

In the first place, from Evolutionary Strategies it is known that it perform fairly constantly on a wide range of function types. The CMA-ES algorithm also shows this on the test suite. In all cases the optimizer managed to reach the optimum, rounded to two decimal places. This is in contrast to the BO optimizer, which in many cases did not reach the optimum within a fair amount of time. The BO algorithms tends to get stuck in local optima more often. Especially for highly irregular function with a lot of peaks, it is difficult for a BO algorithm to estimate the true function curve.

Regarding the effectiveness, in all cases the CMA-ES approaches the optimum better at termination then the BO algorithm. The more noisy the objective function is, the better is the effectiveness of the CMA-ES compared to BO.

Regarding the efficiency, BO is very efficient and can compete with the CMA-ES. It should be noted here that this only applies when the maximum number of function evaluations is relatively small. As more function evaluation is ac-

ceptable, CMA-ES becomes an increasingly better choice.

Furthermore, based on the results, the CMA-ES algorithm always seems to converge. This in contrast to the BO algorithm. This makes it safer to choose the CMA-ES algorithm in case the objective function is completely unknown.

Finally, using the BO algorithm it is important to limit the x-domain within which the optimum is sought. When the domain is defined on the whole real line and no boundaries in the x-domain are set, the variance of the Gaussian Process goes to infinity and hence the areas around minus infinity and plus infinity will be discovered. Hence, for a successful application of the BO algorithm some knowledge of the objective function is more or less required.

Which algorithm can ultimately be best used in evolutionary robotics can not be determined unambiguously based on the results. On one hand, BO is an attractive choice because with a few function evaluations a good result can be achieved. And few function evaluations within evolutionary robotics means that a few runs of robots to locomote is needed, which is desirable. On the other hand, CMA-ES is very robust in its performance. CMA-ES practically always reaches the global optimum very close, while BO in common cases get stuck if the objective function is very rugged. Therefore, for rugged objective functions, the number of function evaluations increases very fast in case of BO without moving to the optimum. Furthermore, CMA-ES constantly converges over the iteration, whereas BO converges less constantly. All in all, this makes the CMA-ES a safer choice.

7 Discussion

The results in this research paper show that the CMA-ES algorithm is more robust than the BO algorithm. Nevertheless, it can not by definition be concluded that CMA-ES performs better. BO can achieve a reasonable result with few function evaluations. The results with the BO algorithm depend to a large extent on the used utility function. For the results in this paper, the Matèrn kernel function is used. It is worth testing other kernel functions. Furthermore, hyper parameters of the BO algorithm can be optimized, which can lead to better and more robust results.

Ultimately, optimization of black-box functions is about the right level of

exploration and exploitation. The BO algorithm can be in entirely different areas of the x-domain per iteration. In the CMA-ES algorithm, exploration and exploitation are linked together in an elegant way. Exploitation is done by adapting the mean vector and exploration is done by adapting the covariance matrix for the direction and the σ for the range. This research paper shows that this leads to good results with a limited amount of effort.

References

[1] J.E. Smith A.E. Eiben. *Introduction to Evolutionary Computing*. Springer, Amsterdam, The Netherlands, 2nd edition, 2015.

[2] Koumoutsakos P Hansen N, Müller SD. Reducing the time complexity of the derandomized evolution strategy with covariance matrix adaptation (cma-es). *Evolutionary Computation*, page 11(1):1–18, 2003.

[3] Hansen N. The cma evolution strategy: A tutorial. *Saclay–Ile-de-France*, 2016.

[4] R. Ros S. Finck, N. Hansen and A.Auger. Real-parameter black-box optimization benchmarking 2009: Presentation of the noiseless functions. *Research Center PPE*, Technical Report 2009/20, 2009.

[5] R. Ros S. Finck, N. Hansen and A.Auger. Real-parameter black-box optimization benchmarking 2009: Presentation of the noisy functions. *Research Center PPE*, Technical Report 2009/21, 2009.