Toward Self-Learning Model-Based EAs

Erik A. Meulman
Centrum Wiskunde & Informatica
Amsterdam, The Netherlands
Delft University of Technology
Delft, The Netherlands
E.A.Meulman@student.tudelft.nl

Peter A.N. Bosman
Centrum Wiskunde & Informatica
Amsterdam, The Netherlands
Delft University of Technology
Delft, The Netherlands
Peter.Bosman@cwi.nl

ABSTRACT

Model-based evolutionary algorithms (MBEAs) are praised for their broad applicability to black-box optimization problems. In practical applications however, they are mostly used to repeatedly optimize different instances of a single problem class, a setting in which specialized algorithms generally perform better. In this paper, we introduce the concept of a new type of MBEA that can automatically specialize its behavior to a given problem class using tabula rasa self-learning. For this, reinforcement learning is a naturally fitting paradigm. A proof-of-principle framework, called SL-ENDA, based on estimation of normal distribution algorithms in combination with reinforcement learning is defined. SL-ENDA uses an RL-agent to decide upon the next population mean while approaching the rest of the algorithm as the environment. A comparison of SL-ENDA to AMaLGaM and CMA-ES on unimodal noiseless functions shows mostly comparable performance and scalability to the broadly used and carefully manually crafted algorithms. This result, in combination with the inherent potential of self-learning model-based evolutionary algorithms with regard to specialization, opens the door to a new research direction with great potential impact on the field of model-based evolutionary algorithms.

CCS CONCEPTS

• **Theory of computation** → **Stochastic control and optimization**; • **Computing methodologies** → *Reinforcement learning*;

KEYWORDS

Estimation of distribution algorithms, machine learning, reinforcement learning, black-box optimization

ACM Reference Format:
Erik A. Meulman and Peter A.N. Bosman. 2019. Toward Self-Learning Model-Based EAs. In *Genetic and Evolutionary Computation Conference Companion (GECCO '19 Companion), July 13–17, 2019, Prague, Czech Republic.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3319619.3326819

GECCO '19 Companion, July 13–17, 2019, Prague, Czech Republic

ACM ISBN 978-1-4503-6748-6/19/07…$15.00
https://doi.org/10.1145/3319619.3326819

1 INTRODUCTION

Model-based evolutionary algorithms (MBEAs) are often applauded for their broad applicability to usually difficult optimization problems, especially in a black-box setting. In practical applications however, MBEAs are frequently used to repeatedly solve instances of the same problem class. It stands to reason that the construction of a more specialized algorithm for such an application can (significantly) improve performance. The development of specialized algorithms however is a laborious and expensive endeavor that requires expertise of both the algorithms and the application itself. A method to automatically generate specialized versions of algorithms for specific applications, without the need for application-specific expertise, could therefore be very promising. This holds especially when applied to MBEAs, since these algorithms already have an inherent potential to deal with many real-world issues, such as noise, lack of gradients, and multiple objectives.

Traditionally, specialization of existing MBEAs to a specific application is pursued using parameter tuning. This approach uses optimization techniques to find appropriate values for algorithm parameters, such as population size, threshold values and smoothing factors, to achieve better performance on a given problem class. Although parameter tuning can lead to better performing parameters, the parameters themselves can often only influence the optimization on a global level, such as managing robustness to local optima and premature convergence. They however cannot exploit local geometric structures specific to the problem class. To exploit such geometric structures, the behavior of the algorithm itself has to be adapted to the problem class.

One way to automate the adaptation of algorithm behavior is to use machine learning techniques. Andrychowicz et al.[1] showed promising results by applying supervised learning to gradient descent to improve performance on specific fitness function classes. A closely related self-learning direct-search approach for continuous black-box problems was presented by Chen et al.[3], which showed comparable performance to Bayesian optimization. In this paper, we address the question of whether such a machine learning approach could be used for automated improvement of MBEAs as well.

The paper focuses on single-objective estimation of normal distribution algorithms (ENDAs). The ENDA framework concerns the maximization of a black-box objective function $f : \mathbb{R}^d \rightarrow \mathbb{R}$, also called the fitness, where $d \in \mathbb{N}$ is the function dimensionality. A generational time-step starts by sampling a population from a normal distribution

$$P^{(t)} = \left(x_i \sim \mathcal{N}\left(\mu^{(t)}, \Sigma^{(t)}\right)\right)_{i \in [n]} \quad (1)$$

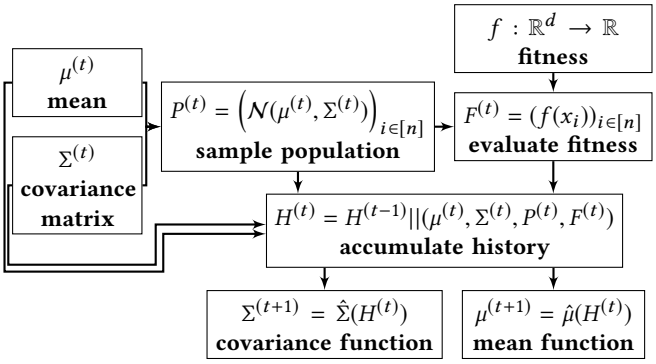


Figure 1: A schematic representation of a single step in the ENDA framework.

with mean vector $\mu^{(t)} \in \mathbb{R}^d$, positive definite covariance matrix $\Sigma^{(t)} \in \mathbb{R}^{d\times d}$, population size n, and where $[n]$ denotes $\{1, \ldots, n\}$. The fitness of the population is evaluated to a fitness vector

$$F^{(t)} = (f(x_i))_{i\in[n]}. \tag{2}$$

The mean vector, covariance matrix, population and fitness vector are then accumulated with the same data of earlier steps to form a history,

$$H^{(t)} = H^{(t-1)}|| \left(\mu^{(t)}, \Sigma^{(t)}, P^{(t)}, F^{(t)}\right). \tag{3}$$

The mean and covariance functions, denoted by $\hat{\mu}$ and $\hat{\Sigma}$ respectively, then take the history to calculate a new mean vector and covariance matrix with the goal to maximize the expected fitness of an individual in the following populations. In traditional ENDAs, the mean and covariance functions would include selection of fitter individuals and subsequent maximum likelihood estimation based on, among others, the selected individuals. The ENDA framework is schematically summarized in Figure 1. Well-known MBEAs that also use the normal distribution as a model are AMaLGaM[2] and CMA-ES[5].

In this paper, we explore the applicability of machine learning techniques to ENDAs for the purpose of developing algorithms that learn to (optimally) adapt themselves instead of relying on a priori designed expertise-driven existing algorithms. We proceed by introducing a proof-of-principle method to incorporate reinforcement learning in the existing ENDA framework to automate algorithm optimization. To study the impact and potential of the proposed approach, we analyze the resulting algorithm and compare its performance with existing ENDAs.

The remainder of this paper is organized as follows. In section 2 we analyze the three paradigms of machine learning and substantiate the choice of reinforcement learning as paradigm for the rest of the paper. Section 3 introduces a basic framework for reinforcement learning and proceeds to apply this framework to the mean function of an ENDA resulting in a proof-of-principle self-learning ENDA, SL-ENDA. In Section 4 we empirically explore the impact of two key components of SL-ENDA to its performance and present an empirical performance comparison between SL-ENDA and two well-known MBEAs, AMalGaM and CMA-ES. The design decisions, results and possibilities for further research are discussed in Section 5. Section 6 concludes with the most notable results and concepts presented in this paper.

2 MACHINE LEARNING TECHNIQUES

To optimize the behavior of ENDAs, as defined above, with respect to an objective function class, only the set of mean and covariance functions have to be considered, since those two functions fully define the behavior. We mostly disregard the covariance function so that we can focus on the analysis of machine learning techniques without the inherent difficulties resulting from the constrained nature of a covariance matrix, such as its positive definiteness. We believe however that this analysis could be extended to provide for automated learning of covariance functions, since we use little to no properties of the mean function itself.

2.1 Three paradigms of machine learning

Machine learning can roughly be subdivided in three paradigms: unsupervised learning, supervised learning and reinforcement learning [10]. Unsupervised learning studies the task of finding patterns in data without any feedback[10]. Well-known unsupervised learning tasks include clustering and anomaly detection. Since we do have a feedback signal, namely the population fitness of the various populations, and unsupervised learning is not directly able to form a mapping from histories to means as required, unsupervised learning is not evidently applicable to learning a mean function.

Supervised learning concerns learning a function that maps an input to an output based on example input-output pairs[10]. In our case to obtain the input-output pairs, we could use existing algorithms and record their optimization paths. However, this would ultimately at best yield a clone of the source algorithm. In other words, it is not a priori known what mean should be returned based on a supplied history. Hence, we do not have the required input-output pairs. As a result, supervised learning is not a natural fit to learn the mean function and we will therefore, for now, disregard it. It is noteworthy that the approach of Andrychowicz et al.[1] to improve performance of gradient descent did use supervised learning. However, their learning mechanism heavily relies on the gradient of the objective function, which is generally unknown in black-box optimization.

Reinforcement learning (RL) is learning what to do - how to map states to actions - so as to maximize a scalar reward signal. RL is not based on input-output pairs but rather on learning which actions yield the most reward by trying[14]. To model the mean function in the RL context we let the history be the state, the mean be the action and the average population fitness be a key part of the scalar reward signal. The resulting RL agent will maximize the average population fitness by learning how to map histories to a new mean, which is exactly what we want to achieve. Reinforcement learning thus seems a natural fit to our goal and we will therefore use it in our proof of principle.

3 LEARNING THE MEAN WITH RL

3.1 Placing the ENDA in an RL framework

Formally, RL considers an agent repeatedly interacting with an environment over the course of several episodes. At the start of every interaction the environment is in some state which is passed

to the agent. The agent responds with an action according to its policy, which is roughly a mapping from states to actions. The state is changed by the action in an, to the agent, a priori unknown way. Based on the "goodness" of the new state, the environment then returns a scalar reward to the agent. By interacting repeatedly with the environment in this way the agent can change its policy so as to maximize its expected discounted cumulative reward,

$$\mathbb{E}\left[\sum_{t=0}^{T} \gamma^t r^{(t)}\right], \tag{4}$$

where $T < \infty$ is the number of interactions for a given episode, $\gamma \in [0, 1]$ is a discount factor and $r^{(t)}$ is the reward at interaction t. To allow the agent to optimize its behavior, it has to explore different behaviors, therefore the policy generally is stochastic.

To apply this formalism to the mean function, $\hat{\mu}$, we will consider the rest of the algorithm, including the current fitness function f, to be the environment. The history, $H^{(t)} \in \mathbb{H}$, is the state, the new mean, $\mu^{(t+1)}$, is the action, and an interaction is a single time-step in the ENDA framework. To make the policy adjustable and stochastic, we will use a parameterized distribution based on a neural network, which is further detailed in subsection 3.2. However, since neural networks cannot have domains with variable dimensionality[1], we cannot pass the history directly to the policy. Therefore the history will be transformed to a constant dimensionality space by a preprocessor, $\hat{\xi} : \mathbb{H} \to \mathbb{R}^{m_{in}}$, before being passed to the policy. Consequently, the action, $a \in \mathbb{R}^{m_{out}}$, sampled from the policy will have to be passed through a postprocessor, $\hat{\rho} : \mathbb{H} \times \mathbb{R}^{m_{out}} \to \mathbb{R}^d$, to obtain the new mean, $\mu^{(t+1)} \in \mathbb{R}^d$. The pre- and postprocessor allow the policy to operate in a space that is different from the solution space, which we call the agent space. This enables us to introduce invariances to the agent space, potentially leading to better generalization and faster learning. This notion and the design of the agent space will be further explored in subsection 3.3. As a consequence of the pre- and post processor, the policy will be a parametrized distribution over the preprocessed histories and the actions, $\pi : \mathbb{R}^D \times \mathbb{R}^{m_{in}} \times \mathbb{R}^{m_{out}} \to \mathbb{R}$, where D is the number of parameters of the policy. Lastly, to specify to the agent that we want to maximize the fitness function, we will have to design an appropriate reward function, $\hat{r} : \mathbb{H} \to \mathbb{R}$. In addition to specifying the goal, a properly designed reward function can also guide the agent to that goal. The details of the reward function design are presented in subsection 3.4. The mean function is schematically summarized in Figure 2.

To maximize the expected discounted cumulative reward (4) we use a well-known RL algorithm called Proximal Policy Optimization (PPO) [12]. We will approach PPO as a black box, altering the policy parameters so as to maximize (4). This makes the resulting framework easily adaptable to advances in the field of RL.

To specialize the resulting mean function to a particular problem class, we define a problem class, $\mathcal{F}$, to consist of a set of functions and a predefined distribution over those functions. We start with a random policy parameter vector $\theta \in \mathbb{R}^D$ and repeatedly sample a function, $f \sim \mathcal{F}$. We run the ENDA with the mean function, as specified above, on each sampled fitness function in turn. During execution of the ENDA, the RL-Agent will accumulate the

[1]We disregard the use of RNN encoders as described in [7]

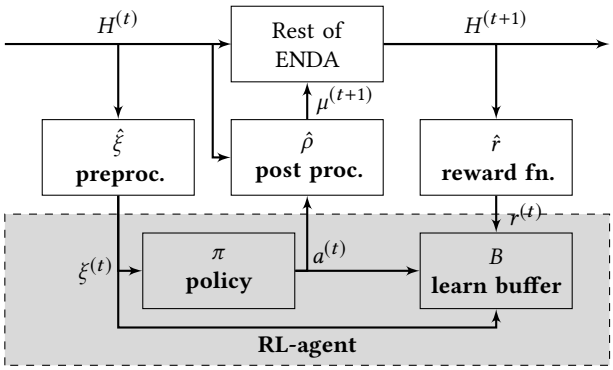


Figure 2: A schematic representation of the mean function.

preprocessed states, actions and rewards in a queue like buffer, B, of predefined size $M \in \mathbb{N}$. Every $N \in \mathbb{N}$ interactions, the agent updates the parameter vector θ by maximizing (4), using gradient ascent, with respect to all interactions in the buffer. Afterwards, the execution of the ENDA is continued with the updated policy parameters. When the buffer is full the oldest interaction in the buffer will be replaced by the new interaction.

Algorithm 1 presents the pseudocode of an RL-based self-learning ENDA with the reinforcement learning mean function (light grey shading) and the training loop (dark grey shading) as introduced above. Without the dark shaded region and by replacing the light shaded region with an arbitrary mean function, algorithm 1 represents the standard ENDA framework. To accommodate real-world applications, the function sampling at line 6 can be replaced by another process that provides objective functions. To avoid overfitting, the objective functions should be a good representation of the function class under investigation. The rest of the section will detail the implementation of the policy, pre- and post processor and reward function.

3.2 The policy

The policy is the parametrized distribution from which an action is sampled based on the preprocessed history. Since the domain of the fitness function is unbounded and continuous in nature, we use a Gaussian policy. This entails that the policy distribution is a multivariate normal with its mean and covariance matrix calculated by a function approximator. To avoid the difficulties of learning a valid (i.e. positive definite) full covariance matrix, we will assume a diagonal covariance matrix for the policy distribution. As a consequence the policy cannot efficiently explore correlations in the agent space. However, if the agent space is properly normalized, which will be discussed in subsection 3.3, this effect can become negligible. In accordance with existing literature the function approximator is a neural network[9]. This allows approximation of non-linear continuous functions while having an analytic gradient through backpropagation, which is necessary to apply gradient ascent to optimize (4). The network topology in this work consists of a series of fully connected hidden layers, leading into two parallel fully connected layers of size m_{out} of which the outputs are the mean and log variances of the exploration distribution. A schematic

Algorithm 1: RL-based self-learning ENDA

Input: $n, \mu^{(0)}, \Sigma^{(0)}, \hat{\Sigma}, \mathcal{F}, \pi, \hat{\xi}, \hat{\rho}, \hat{r}, M, N, T_{\max}$

$\theta \leftarrow$ RANDOMINIT(); // init policy parameters
$B \leftarrow$ QUEUE(max_len = M); // init buffer
$t_{RL} \leftarrow 0$;
$t_{learn} \leftarrow 0$;
while $t_{RL} < T_{\max}$ **do**
  $f \sim \mathcal{F}$; // sample function from class
  $\mu \leftarrow \mu^{(0)}$;
  $\Sigma \leftarrow \Sigma^{(0)}$;
  $H \leftarrow ()$;
  $t_{EA} \leftarrow 0$;
  while *EA stopping criterion is not met* **do**
    $P \leftarrow (x_i \sim \mathcal{N}(\mu, \Sigma))_{i=1}^{n}$; // sample population
    $F \leftarrow (f(x_i))_{i=1}^{n}$; // evaluate fitness
    $H \leftarrow H||(\mu, \Sigma, F, P)$; // append history
    if $t_{EA} > 0$ **then**
      $r \leftarrow \hat{r}(H, a)$; // calculate reward
      $B \leftarrow B$.PUT(ξ, a, r); // add interaction to buffer
      $t_{learn} = t_{learn} + 1$;
      if $t_{learn} == N$ **then**
        $\theta \leftarrow$ LEARN(π, θ, B); // update params.
        $t_{learn} \leftarrow 0$
    $\xi \leftarrow \hat{\xi}(H)$; // preproc. history
    $a \sim \pi(\theta, \xi)$; // sample action from policy
    $\mu \leftarrow \hat{\rho}(H, a)$; // postproc. action
    $\Sigma \leftarrow \hat{\Sigma}(H)$; // calculate new covariance
    $t_{EA} \leftarrow t + 1$;
  $t_{RL} \leftarrow t_{RL} + 1$;
return θ;

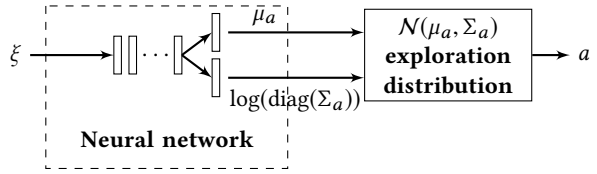


Figure 3: A schematic representation of the policy of SL-ENDA.

representation of the policy and general network topology is given in Figure 3. It is crucial to understand that the exploration distribution is part of the mean function and therefore different from the population distribution of the main ENDA framework.

3.3 The pre- and post processor

The main goal of the preprocessor is to keep the dimensionality of the policy domain constant. This is necessary because the neural network can only handle a domain of constant dimensionality. Additionally, the preprocessor allows the agent to operate in a different space than the solution space. Such a transformation to an agent space can be used for normalization, which can increase the numerical stability of the learning process of the RL-agent[8], as well as to force invariances on the resulting mean function, which extends the validity region of the algorithm [5], and lastly, to reflect prior knowledge about the function class it will be used on, for example by inverting a known rotation or elongated axis.

To ensure constant dimensionality, we use a truncated version of the optimization history consisting of the last $k \in \mathbb{N}$ populations and fitness vectors. We let $k \geq 2$ to preserve a temporal component in the preprocessed history. The transformation is then performed on a per ENDA time-step basis. To ensure the time-steps are relatable to each other, each population and fitness vector is transformed with respect to the most recent population and fitness vector. For example, suppose we are currently at time-step $t \in \mathbb{N}$ and we want to transform time-step $\tau \leq t$. Since we assume nothing about the fitness function, we use a standard normalization technique for scalar data,

$$\tilde{F}^{(\tau)} = \left(\frac{F_i^{(\tau)} - \overline{F}^{(t)}}{\text{std}(F^{(t)})} \right)_{i=1}^{n}, \tag{5}$$

where $\overline{F}^{(t)}$ is the average and $\text{std}(F^{(t)})$ is the standard deviation of $F^{(t)}$. For the populations we consider two transformations in this work:

TSI The translation and scale invariant (TSI) transformation uses the mean, $\mu^{(t)}$, and the biggest eigenvalue of the population covariance matrix, $\lambda_{\max}^{t} = ||\Sigma^{(t)}||_{\infty}$, to transform the individuals in the population,

$$\tilde{P}_{\text{TSI}}^{(\tau)} = \left(\frac{x_i^{(\tau)} - \mu^{(t)}}{\sqrt{\lambda_{\max}^{(t)}}} \right)_{i=1}^{n}. \tag{6}$$

TRSI The translation, rotation and scale invariant (TRSI) transformation uses the mean, $\mu^{(t)}$ and the square root of the inverse of the population covariance matrix, $(\Sigma^{(t)})^{-\frac{1}{2}}$, to transform the individuals in the population back to the space where the current population is standard normally distributed,

$$\tilde{P}_{\text{TRSI}}^{(\tau)} = \left(\left(\Sigma^{(t)}\right)^{-\frac{1}{2}} \left(x_i^{(\tau)} - \mu^{(t)}\right) \right)_{i=1}^{n}. \tag{7}$$

Note that both transformations transform the populations to a space where the current population is normalized, in the sense that in any direction the variance is at most 1. The transformations, however, do not impose any normalization guarantees on the previous populations. This is an effect of the need for relatability between populations in the agent space. Also, the transformations do not encode any prior knowledge about any function class and can therefore be used to train on any function class.

Altogether, a preprocessed history consists of the last k populations and fitness vectors transformed using TSI or TRSI. For extra structure, the individuals in a population and the fitness vectors are sorted with respect to solution fitness. Then, all transformed and sorted populations and fitness vectors are flattened to form one real-valued vector of dimensionality $n \cdot (d+1) \cdot k$, where if $t < k$ the vector is padded with zeros.

Lastly, the post processor inverts the transformation applied to the populations by the preprocessor to transform the action from the agent space to the solution space. The action will therefore be a d-dimensional real-valued vector. In the case of the TSI space the post processor is

$$\hat{\rho}_{\text{TSI}}\left(H^{(t)}, a\right) = \sqrt{\lambda_{\max}^{(t)}} \cdot a + \mu^{(t)}, \tag{8}$$

where $\lambda_{\max}^{(t)}$ is the maximum eigenvalue of the population covariance matrix of time-step t. The post processor for the TRSI space is

$$\hat{\rho}_{TRSI}\left(H^{(t)}, a\right) = \left(\Sigma^{(t)}\right)^{\frac{1}{2}} a + \mu^{(t)}. \tag{9}$$

3.4 The reward function

The reward function allows the environment to specify a goal for the agent. In the case of ENDAs we want to construct populations with high expected fitness. We consider maximization of fitness (instead of minimization) mostly to simplify the expressions of the introduced reward functions, which is maximized by convention. Hence, we want to find $\mu^* \in \mathbb{R}^d$ and $\Sigma^* \in \mathbb{R}^{d\times d}$ such that

$$\mu^*, \Sigma^* = \underset{\mu,\Sigma\in\mathbb{R}^d\times\mathbb{R}^{d\times d}}{\arg\max} \mathbb{E}\left[f(x)|x \sim \mathcal{N}(\mu, \Sigma)\right], \tag{10}$$

where $\mathbb{E}$ is the expected value operator. Additionally, we would like to find these μ^* and Σ^* in as little time-steps as possible.

Since the RL-agent tries to maximize the cumulative discounted reward, as defined in (4), we have to define the reward function in such a way that the expected fitness is maximized when the cumulative discounted reward is maximized. In this work we consider three reward functions:

Fitness reward Since the average population fitness, $\overline{F}^{(t)}$, is the maximum likelihood estimate of the expected value in (10), it is natural to define the reward function as

$$\hat{r}_{\text{fit.}}(H^{(t)}) = \overline{F}^{(t)} = \frac{1}{n}\sum_{i=1}^{n} F_i^{(t)}. \tag{11}$$

This reward function has the property that it is neither scale- nor translation invariant. This could lead to difficulties with function classes containing scaled or translated fitness functions.

Normalized fitness reward Alternatively, the average population fitness can be normalized,

$$\hat{r}_{\text{norm.fit.}}(H^{(t)}) = \frac{\overline{F}^{(t)} - \hat{f}_{\min}(H^{(t)})}{\hat{f}_{\max}(H^{(t)}) - \hat{f}_{\min}(H^{(t)})}. \tag{12}$$

Since the maximum fitness value is not a priori known and the minimum often doesn't even exist, they have to be estimated. The maximum is estimated by the maximum fitness value encountered until the previous time-step,

$$\hat{f}_{\max}(H^{(t)}) = \max_{\substack{\tau\in[t-1]\\ i\in[n]}} F_i^{(\tau)}. \tag{13}$$

The current fitness vector is not included in the maximum to ensure that improving the current maximum is encouraged over remaining at the current maximum. The minimum is estimated by the maximum of three terms,

$$\hat{f}_{\text{initial}}(H^{(t)}) = \min_{i\in[n]} F_i^{(0)}, \tag{14}$$

$$\hat{f}_{\text{decayed}}(H^{(t)}, \beta) = \beta^t \min_{\substack{\tau\in[t-1]\\ i\in[n]}} F_i^{(\tau)} + (1-\beta^t)\hat{f}_{\max}(H^{(t)}), \tag{15}$$

$$\hat{f}_{\text{window}}(H^{(t)}, w) = \min_{\substack{\tau\in\{t-w,\ldots,t-1\}\\ i\in[n]}} F_i^{(\tau)}, \tag{16}$$

where $\beta \in (0, 1)$ and $w \in [t-1]$ are parameters. The initial term, (14), provides a constant baseline that cannot be influenced by the agent. The decayed term, (15), encourages the agent to stay increasingly closer to the current maximum, allowing early exploration and punishing late stage divergence. The window term, (16), allows the minimum to stay relatively close to the current population fitness, ensuring that a change in fitness stays significant.

Differential reward The differential reward, defined as

$$\hat{r}_{\text{diff.}}(H^{(t)}) = \frac{\overline{F}^{(t)} - \overline{F}^{(t-1)}}{\max_{\substack{\tau\in[t-1]\\ i\in[n]}} F_i^{(\tau)} - \overline{F}^{(t)}}, \tag{17}$$

looks at the improvement in average population fitness since the previous time-step (numerator), relative to an estimation of the current precision (denominator). The reward can mathematically be derived from the assumption $\overline{F}^{(t)} > \overline{F}^{(t-1)}$, which could lead to problems with multi-modal functions.

4 EXPERIMENTS

In this section we empirically compare the performance of SL-ENDA to that of AMaLGaM [2] and CMA-ES [5]. However, to get a good grasp on the potential of self-learning MBEAs we first empirically select the most performant reward function and agent space from the functions and spaces proposed in section 3. We start however with a description of the experimental set-up.

4.1 Experimental set-up

During experimentation a covariance function based on AMaLGaM is used, where the anticipated mean shift is replaced by an additional mean-shift term in the covariance function, much like the rank-one update found in CMA-ES, to make it fit in the ENDA framework. The population size for all tested algorithms is set to $n = 20 + 10 \cdot d$, which is larger than the recommended population size, but since we are establishing feasibility, we leave parameter tuning for further work.

The Proximal Policy Optimization (PPO) algorithm is used for the reinforcement learning agent since it is a well-known algorithm with good performance for continuous state/action agents[12]. The policy is a multivariate normal distribution with a diagonal covariance matrix. The parameters for the policy distribution are parameterized using a neural network consisting of 2 fully connected layers, each with 128 units and ELU activation, leading into 2 parallel layers of d units resulting in the mean and log standard

deviation vectors for the normal distribution of the policy. In preliminary experiments this network was found to work, but thorough exploration of other network topologies is encouraged for further research.

To train the agent, function classes have to be defined to sample objective functions from. All function classes tested here have a base function $f_b : \mathbb{R}^d \to \mathbb{R}$, that define the underlying properties of the class. The base functions tested in this paper are the sphere (f_1), ellipsoidal (f_2) and Rosenbrock (f_8) function as specified in the BBOB 2010 noiseless function definition[6]. All base functions are translated such that their optimum is located in $\mathbf{0} \in \mathbb{R}^d$.

For any $b \in \{1, 2, 8\}$ the function class based on f_b is defined as

$$\mathcal{F}_b = \{(x) \mapsto -f_b(Rx + a) : R \in SO(d), a \in B(\mathbf{0}, 100)\}, \quad (18)$$

where the minus is added to account for maximization with ENDAs and minimization in BBOB, $SO(d)$ is the d-dimensional special orthogonal group, also called the rotation group, and $B(\mathbf{0}, 100) \subset \mathbb{R}^d$ is the ball of radius 100 around the origin. Functions are sampled from $\mathcal{F}_b$ by uniformly sampling a and R from $B(\mathbf{0}, 100)$ and $SO(d)$, respectively.

After training on the function class for a, per experiment specified, number of sampled functions, the resulting ENDA is evaluated on a predefined set of 1000 functions sampled from the class. To keep the ENDA static during evaluation learning is disabled by ignoring lines 1-6, 15-21 and 27-28 in algorithm 1.

We define the runtime as the number of objective function evaluations to reach an average population fitness, $\overline{F}$, such that the precision, $(\max_{x \in \mathbb{R}^d} f(x)) - \overline{F}$, is smaller than some $\varepsilon > 0$ as central measure of performance and call it the runtime. The algorithms are initialized with mean $\mathbf{0} \in \mathbb{R}^d$ and the identity matrix as covariance matrix. Optimization of an objective function, i.e. an EA run, is terminated when a threshold precision $\varepsilon_{\max} \ll \varepsilon$ is reached or after 1000 generations, i.e. this describes the stopping criterion on line 11 of algorithm 1.

The implementation in Python 3.6 used to produce the results in this section is publicly available.[2] PPO was implemented using the Tensorflow library and based on the OpenAI baselines library [4]. All experiments were performed on a 64-core (4 x 16-core AMD Opteron(tm) Processor 6386 SE) server running Fedora 28.

4.2 Reward function analysis

As stated earlier, the definition of the reward function is one of the most crucial parts of an environment. Not only does it specify to the agent what its goal is, a good reward also guides the agent towards that goal, significantly improving sample efficiency and the stability of the learning process.

Figure 4 shows the performance of SL-ENDA equipped with the differential, fitness and normalized fitness reward, as introduced in subsection 3.4, after training on 10^3, 10^4 and 10^5 functions (left to right), sampled from the 2-dimensional Rosenbrock function class. The vertical axis indicates runtime until the precision on the horizontal axis is first achieved. The lines mark an average over 1000 sampled functions and the shaded area is the corresponding 99% confidence interval. The lines are terminated on the highest

[2]https://github.com/realtwister/LearnedEvolution

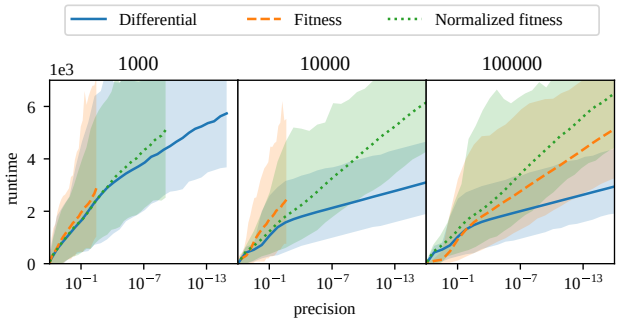


Figure 4: Runtime until precision is achieved by SL-ENDA equipped with differential, fitness and normalized fitness reward on the 2-dimensional Rosenbrock function class.

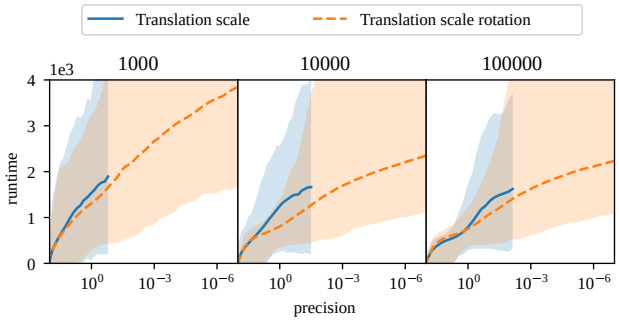


Figure 5: Comparison of runtime vs. precision of the TSI- and TRSI-space on the 2-dimensional Rosenbrock function class after learning for 10^3, 10^4 and 10^5 functions.

precision that was achieved for at least 10% of the sampled functions. The TRSI agent space is used throughout the experiment.

After 1000 functions, although all three rewards show comparable convergence speeds, the differential reward on average achieves a higher precision. This indicates that all three reward functions specify a similar goal early on, achieving high precision. However, the differential reward more efficiently guides the agent to such high precision achieving behavior. This can be observed by the fitness reward only achieving a precision of 10^{-2} after training on 10^4 functions. Superior guidance can also be concluded from the fact that the differential reward is mostly converged after 10^4 functions, which can be observed from the nearly identical curves at the 10^4 and 10^5 training functions mark. Additionally, the behavior learned under the differential reward shows a higher convergence speed than both the fitness and the normalized fitness reward. We can therefore conclude that the differential reward outperforms both the fitness and normalized fitness reward.

4.3 Agent space analysis

To find the most performant agent space we compare the use of TRSI- and TSI-space, as introduced in subsection 3.3. The experimental procedure is comparable to the reward function comparison

in the previous section. The algorithm is equipped with the differential reward function throughout this experiment.

Figure 5 shows that the TRSI-space results in a precision of at least 10^{-7} after training on only 10^3 functions while the highest achieved precision with TSI-space is 10^{-2}, which is achieved after training on 10^5 functions. The faster convergence in RL-time confirms that the addition of invariances, such as the rotation invariance in the TRSI-space, can significantly decrease the number of functions needed to learn a policy that achieves high precision. Additionally, Figure 5 does not indicate any penalty with regards to EA-time convergence speed as a result of the introduced rotational invariance. Based on these results we conclude that the use of TRSI-space leads to the best performance and we will therefore use it throughout the rest of the paper.

4.4 Performance comparison with existing algorithms

Following the BBOB standard we consider the runtime complexity with respect to the dimensionality of the problem as performance measure to compare SL-ENDA to AMaLGaM and CMA-ES. To keep the comparison as transparent as possible, both AMaLGaM and CMA-ES were implemented in the ENDA framework. For the implementation of AMaLGaM, Bosman et al.[2] was followed as closely as possible. The only major deviation from the pseudocode in the paper is the replacement of the anticipated mean-shift, which cannot be implemented in the ENDA framework, by an additional mean-shift term in the covariance function, much like the rank-one update found in CMA-ES. For the implementation of CMA-ES the GECCO 2013 CMA-ES tutorial slides[3] were used as source and could directly be implemented in the ENDA framework without alterations.

Figure 6 shows the runtime until precision 10^{-4} is reached by the three algorithms on the sphere (a), ellipsoid (b) and Rosenbrock (c) function class, as described above, for problem dimensionality $d \in \{2, 3, 4, 5, 10\}$. The figures show the runtime averaged over 1000 sampled functions. The 99% confidence interval is given by the shaded area. SL-ENDA was trained on $2{\cdot}10^5$ functions uniformly sampled from the function class it is evaluated on.

On all three function classes and for all tested dimensionalities SL-ENDA is able to find the optimum and achieve at least 10^{-4} precision. On average the implementations of both AMaLGaM and CMA-ES have a lower runtime. This result however, is not statistically significant for AMaLGaM in the tested dimensionalities. The fact that for all three function classes SL-ENDA achieves comparable, same order of magnitude, runtime as AMaLGaM and in some cases CMA-ES, shows that self-learning MBEAs can, tabula rasa, learn optimization behavior that comes near that of existing, broadly used algorithms.

SL-ENDAs perceived scalability on both sphere and ellipsoid is polynomial, between linear and quadratic. This is slightly worse than the perceived linear scalability of AMaLGaM. It is important to note that the population size, $n = 20 + 10 \cdot d$, is not the advised population size for either AMaLGaM or CMA-ES. This can explain the unexpected scalability behavior of CMA-ES with respect to AMaLGaM. In additional experiments, not shown here, with the recommended population size of CMA-ES, $n = 4 + \lfloor 3 \cdot \ln(d) \rfloor$, its expected linear scalability on sphere was observed. Considering Rosenbrock, the perceived scalability of SL-ENDA is non-polynomial. A possible explanation for this behavior is the relatively high constant term in the population size, which could result in a relatively high runtime on low-dimensional functions. Disregarding the result for the 2-dimensional case would yield an approximate quadratic scalability. Altogether, SL-ENDA is truly outperformed here, scalability-wise, by both CMA-ES and AMaLGaM.

[3] http://www.cmap.polytechnique.fr/~nikolaus.hansen/gecco2013-CMA-ES-tutorial.pdf

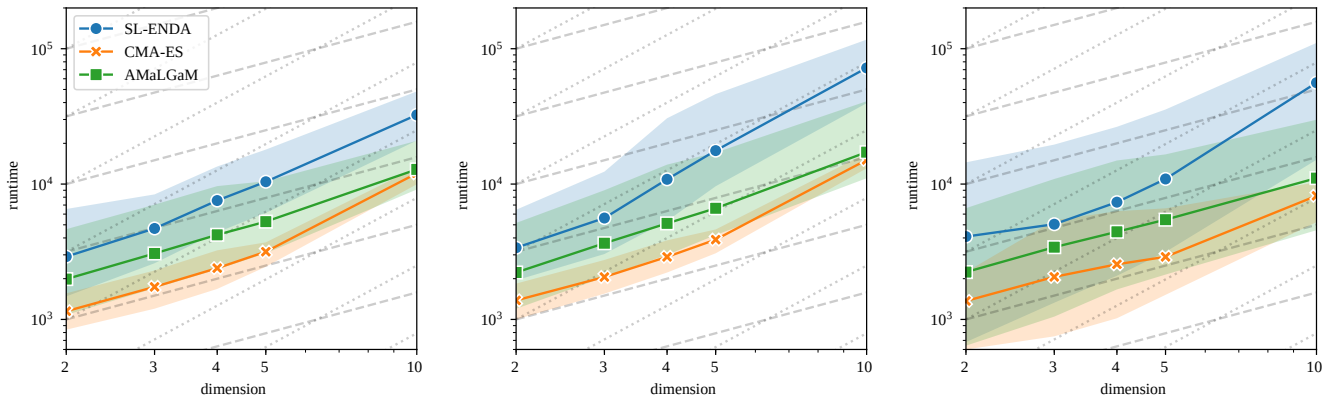
5 DISCUSSION

The results in this paper show that the proposed algorithm, SL-ENDA, is able to improve its optimization behavior on a problem class based on earlier optimization of problems in that class. However, this was only shown for the problem classes induced by the sphere, ellipsoidal and Rosenbrock functions. To make truly quantitative statements about SL-ENDA, benchmarking, on for example BBOB, is advised. This will also enable more complete comparisons with existing algorithms. Furthermore, the performance of SL-ENDA on multi-modal and noisy functions is, at the time of writing, an open question, which could be answered by the aforementioned benchmarking.

One could argue that the RL-agent in SL-ENDA could simply learn to calculate a weighted average of the current population, which would in principle be sufficient, for example AMaLGaM does this by taking the average over selected solutions. However, since the agent has access to multiple consecutive populations, it is theoretically able to exhibit much more advanced behavior by recognizing the local structure of the objective function and changing its direction and step-size on a per generation basis.

It would be very interesting to look at the learned optimization behavior of self-learning MBEAs, both to understand their inner workings and to uncover potential new insights in black-box optimization itself. Such analyses could for example entail the qualitative comparison of the optimization paths of SL-ENDA and existing algorithms on the same objective function, as well as the effect of different reward functions on such paths. Additionally, both quantitative and qualitative analysis should be used to research the generalization of self-learned specialized algorithms to objective function classes they were not trained on. This could lead to insights in shared underlying structures of different function classes and the degree to which the developed self-learning algorithms are able to specialize.

This paper shows that self-learning MBEAs can be designed and, given the observed learning ability even in the restricted setting of this first paper on this topic, are a potentially powerful new technique. It should however be clear that the work reported here is a proof-of-principle, showing what key components are and the importance of their proper design (e.g. TSI-space vs. TRSI-space and differential reward vs. average fitness reward). An important next step in this space is the development of a reinforcement-learning-based covariance function. To develop such a function, the highly constrained and high-dimensional space of positive definite matrices has to be explored. Additionally, the difference between the



(a) Sphere (f_1) **(b) Ellipsoid (f_2)** **(c) Rosenbrock (f_8)**

Figure 6: Comparison of runtime scalability w.r.t. problem dimension of SL-ENDA, CMA-ES and AMaLGaM for function classes Sphere (a), Ellipsoid (b) and Rosenbrock (c). Plots show runtime until $f_{max} - \overline{F} < 10^{-4}$ versus the problem dimensionality in log-log scaling. Markers and shaded interval denote the mean and 99% confidence interval over 1000 sampled functions per dimensionality, respectively. Grid lines show linear (dashed) and quadratic (dotted) scaling.

Euclidean distance of positive definite matrices and the "natural" distance measure of covariance matrices with respect to probability, as described in [15], have to be taken into account. Lastly, we have to take into account that the reward function is possibly much less trivial for the covariance function, since the main goal of the covariance matrix is to manage the exploration vs. exploitation trade off, which is not easily captured in a scalar feedback signal.

As seen in the comparison of the agent spaces, a particular choice of space can significantly impact the learning time of the agent. It is therefore promising to research the application of self-learning embeddings, such as for example a recurrent neural network embedding[7], to embed the optimization history for algorithms such as SL-ENDA. Such embeddings also open the door to population-size and dimension-agnostic algorithms. Another area in which this work can be extended upon, is pretraining the policy of the agent by supervised learning on optimization paths generated by existing algorithms. Preliminary testing that we did in the context of this paper showed that this technique can lead to significant learning time reduction. It should however be noted that pretraining has the potential to reduce final performance, as shown by silver et al. [13].

As an alternative to SL-ENDA, we could parameterize the mean function with a neural network and, using a black-box optimization method, maximize a performance measure, like the reward function, over a set of functions sampled from the function class. In theory this would be a more "pure" approach to optimizing the behavior of the ENDA, since we can then directly optimize the performance measure under consideration. Such an approach does, however, not allow behavior optimization to proceed on a per ENDA-generation basis, which makes it far less sample efficient but potentially more robust to multi-modal functions. Salimans et al. showed that using classic evolution strategies on general RL benchmarks can match the performance of conventional RL-algorithms [11]. This approach was found to be highly parallelizable, but it needed at least 3x as much data to achieve matching performance. Applying the ideas of Salimans et al. to self-learning ENDAs is left for further research.

Finally, we note that throughout this paper, due to its natural fit, reinforcement learning was used as main machine learning paradigm. However, there are ways, unexplored in this paper, to apply other paradigms to MBEAs. A good example is the indirect supervised learning approach introduced by Andrychowicz et al. [1]. The approach uses the chain-rule in combination with the gradient of the objective function and back propagation, to calculate a parameter update of a neural network that encodes the optimization step of a gradient descent scheme. The approach could be adapted, by for example approximating the gradient numerically, to train the mean function of self-learning MBEAs. This example shows that there are many unexplored ways to apply machine learning to MBEAs, adding to its attractiveness as a new topic for research.

6 CONCLUSION

We have introduced a framework that uses reinforcement learning to, tabula rasa, learn to adapt the model-governing parameters of a model-based evolutionary algorithm to achieve efficient optimization. The results show that self-learning model-based evolutionary algorithms can yield algorithms that, on unimodal noiseless functions, have performance and scalability that comes close (same order of magnitude) to that of existing, broadly-used and carefully manually-engineered algorithms. This conclusion, together with the inherent potential of self-learning evolutionary algorithms with regard to specialization, supports the idea that self-learning model based evolutionary algorithms offer a promising new direction for further research that potentially has great impact on the field of (model-based) evolutionary algorithms.

REFERENCES

[1] M. Andrychowicz, M. Denil, S. Gomez, M. W. Hoffman, D. Pfau, T. Schaul, B. Shillingford, and N. De Freitas. 2016. Learning to learn by gradient descent

by gradient descent. *Advances in Neural Information Processing Systems* (2016), 3981–3989.

[2] P. A. N. Bosman, J. Grahl, and D. Thierens. 2013. Benchmarking Parameter-Free AMaLGaM on Functions With and Without Noise. *Evolutionary Computation* 21 (2013), 445–469.

[3] Y. Chen, M. W. Hoffman, S. G. Colmenarejo, M. Denil, T. P. Lillicrap, M. Botvinick, and N. de Freitas. 2017. Learning to learn without gradient descent by gradient descent. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. 748–756.

[4] P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, Y. Wu, and P. Zhokhov. 2017. OpenAI Baselines. https://github.com/openai/baselines. (2017).

[5] N. Hansen. 2016. The CMA evolution strategy: A tutorial. *arXiv preprint arXiv:1604.00772* (2016).

[6] N. Hansen, S. Finck, R. Ros, and A. Auger. 2010. *Real-parameter black-box optimization benchmarking 2010: Presentation of the noiseless functions.* Technical Report. INRIA.

[7] Y. Keneshloo, T. Shi, C. K. Reddy, and N. Ramakrishnan. 2018. Deep Reinforcement Learning For Sequence to Sequence Models. *arXiv preprint arXiv:1805.09461* (2018).

[8] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. 2015. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971* (2015).

[9] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. 2015. Human-level control through deep reinforcement learning. *Nature* 518 (2015), 529–533.

[10] S. Russell and P. Norvig. 2009. *Artificial Intelligence: A Modern Approach* (3rd ed.). Prentice Hall Press, Upper Saddle River, NJ, USA.

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

[12] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. 2017. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347* (2017).

[13] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis. 2017. Mastering the game of Go without human knowledge. *Nature* 550 (2017), 354–359.

[14] R. S. Sutton and A. G. Barto. 2018. *Reinforcement Learning: An Introduction* (second ed.). The MIT Press. http://incompleteideas.net/book/the-book-2nd.html

[15] D. Wierstra, T. Schaul, T. Glasmachers, Y. Sun, J. Peters, and J. Schmidhuber. 2014. Natural Evolution Strategies. *Journal of Machine Learning Research* 15 (2014), 949–980.