Toward Self-Learning Model-Based Evolutionary Algorithms

by

Erik Andreas Meulman

to obtain the degree of Master of Science

at the Delft University of Technology,

to be defended publicly on Wednesday April 17, 2019 at 01:00 PM.

Student number:	4277457	
Project duration:	February 28, 2018 – April 17, 2019	
Thesis committee:	Prof. dr. P. A. N. Bosman,	CWI, TU Delft, daily supervisor
	Prof. dr. K. I. Aardal,	TU Delft, supervisor
	Dr. J. Bierkens,	TU Delft

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 (RL) 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.

Preface

These pages contain the culmination of the largest research project I have undertaken to date. They describe my attempt at combining machine learning techniques with model-based evolutionary algorithms (MBEAs) in order to create self-learning model-based evolutionary algorithms. During the research it became evident that the combination of these two techniques is not at all trivial, especially given the black-box setting MBEAs normally operate in. This black-box setting forced me to use reinforcement learning (RL) as "learning algorithm". Luckily, I already acquired some experience with RL during an internship I did prior to this work, which proved invaluable during my research. Apart from this thesis I also wrote a paper for GECCO, which is currently under review as a workshop paper. The research was done in the context of a final thesis for the master program Applied Mathematics at Delft University of Technology, and was performed in close collaboration with Centrum Wiskunde & Informatica in Amsterdam from February 2018 until April of 2019.

I would like to thank everybody who helped me in anyway through this project either directly or indirectly. In particular, I thank Peter Bosman for his help in crystallizing the topic of my research, taking the time and energy to supervise the project, sharing his insight in evolutionary algorithms and co-authoring the paper that resulted from this work.

Secondly, I would like to thank Karen Aardal for referring me to Peter Bosman when I came to her with my idea for a master thesis and her dedication in supervising this thesis despite the sometimes less than ideal circumstances.

Furthermore, I would like Peter Bosman, Karen Aardal and Joris Bierkens for taking the time to take part in my assessment committee.

I also thank all my fellow students who showed interest in the research and asked often enlightening question. In particular, I would like to thank Arjan Cornelissen for his support during late nights in the library, insights in reinforcement learning and the useful discussions relating to this thesis.

Finally, I would like to thank my parents and sister for their undying love, support and patience over the course of this work and in general.

Erik Andreas Meulman
Delft, March 2019

Contents

1 Introduction

In optimization, knowledge about the structure of a problem is often exploited to develop algorithms with high performance. This can, for example, be observed by the extensive use of the gradient in continuous optimization [8]. However, for some problems the structure is not a priori known and cannot directly be obtained. Examples of such problems include optimization of proprietary computational models and numerically approximated mathematical models with high complexity [8]. To enable optimization of these kinds of problems, the field of black-box optimization studies methods to (approximately) solve optimization problems where the objective and/or constraint functions are given by so-called black boxes. A black box, in this context, is any process that, when provided with an input, returns an output, but the inner workings of the process are not analytically available [2]. In mathematical optimization these objects are sometimes called oracles.

Though the structure of a black box in itself is unknown, practical applications frequently require repeatedly optimizing different problem instances that share some underlying structure, we will call such sets of problems problem classes. For example, when optimizing over a proprietary computational model, the settings of the model might change from instance to instance, but the model itself is static throughout all instances. It stands to reason that the construction of more specialized algorithms for such an application, rather than using a general purpose black-box optimization algorithm, 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.

To limit the scope of this thesis we will only consider global continuous black-box optimization, which is also referred to as derivative free optimization [8]. In particular, we will consider a setting where we are handed a black box objective function, also called the fitness function, $f : \mathbb{R}^d \to \mathbb{R}$, and we are tasked with finding $x^* \in \mathbb{R}^d$ such that

$$f(x^*) \approx \sup_{x \in \mathbb{R}^d} f(x), \tag{1.1}$$

where we will assume that the supremum is finite. Note that we are not looking to exactly solve a maximization problem, but only to find a "good" solution with a high fitness value. We will throughout the thesis however often use the word "solve" in reference to finding such a good solution.

The literature on continuous black-box optimization can roughly be subdivided into three subfields: direct-search methods, model-based search methods and evolutionary algorithms. Direct-search methods sample the objective function at a finite number of points at every iteration, and, according to a predefined strategy, decide which actions to take next solely based on those function values [8]. Generally, direct-search methods are computationally inexpensive, however since the next action is based solely on the current iteration, the methods have a low sample efficiency. That makes direct-search methods especially suitable for problems with cheap objective function evaluations. Model-based search methods, on the other hand, maintain a surrogate model of the objective function to guide the choice of trail solutions. That is done by iteratively optimizing over the surrogate model, evaluating the objective value of the resulting solution, and updating the surrogate model based on the newly obtained information [20]. Due to the computational overhead of updating and optimizing over the surrogate model, model-based search methods generally have a high sample efficiency but also a high computational complexity. Hence these methods work best for problems with expensive objective function evaluations. Evolutionary algorithms are algorithms that maintain and iteratively update a set of solutions, called a population, in order to construct a population of solutions with high fitness values [28]. The population

is updated by selecting promising solutions in the population, and, based on the selected solutions, sample new solutions which are added to the population. Due to their stochastic and population-based nature, evolutionary algorithms are generally robust against noisy and multi-modal objective functions [28]. Due to their robustness and broad applicability to real-world problems, and to limit the scope of this work, this thesis will only consider to evolutionary algorithms.

In this thesis we particularly focus on the class of model-based evolutionary algorithms (MBEAs), a subclass of evolutionary algorithms. MBEAs explicitly build a model that is intended to bias the generation of new solutions to regions of above average fitness (as compared to the current population) [6]. The model is based on the information about the objective function gained from earlier populations. MBEAs lend themselves exceptionally well to an automated specialization approach since the algorithms already contain an explicit model, which allows us to specialize the algorithm by adapting the operator that governs how this model is constructed.

Traditionally, specialization of existing MBEAs to a specific application is pursued using parameter tuning [27]. 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 optimized for the problem class.

One way to automate the optimization of algorithm behavior is to use machine learning techniques. Both Andrychowicz et al. [1] and Li et al. [16] showed promising results by applying supervised learning and reinforcement learning, respectively, to gradient descent to improve performance on specific (non-black-box) problem classes. In their paper Chen et al. [5] propose an idea similar to Andrychowicz et al. for point-based black-box optimization problems. They do however need the gradient information of the objective function in order to train their algorithm, which makes it incompatible with black-box scenarios. In this thesis, we address the question of whether a machine learning approach could be used to automate the design of MBEAs to achieve improved performance as well.

Concretely, we will address the following research questions.

Research Q1. Is it possible to develop a self-learning model-based evolutionary algorithm, in the sense that the algorithm can improve its ability to optimize different instances of a problem class, based on data collected during optimization of instances of that class, without the need for a priori problem expertise or manually-engineered and heuristic-driven update rules of the model?

- (a) Which machine learning techniques can be used to develop a self-learning model-based evolutionary algorithm?
- (b) How can these techniques be applied to develop a self-learning model-based evolutionary algorithm?

Research Q2. How does the performance of a self-learning model-based evolutionary algorithm compare to the performance of existing manually-engineered model-based evolutionary algorithms?

In Chapter 2 we properly define the concepts of black-box optimization and give an introduction to evolutionary-, model-based evolutionary-, and estimation of normal distribution algorithms. In Chapter 3 we review the three paradigms of machine learning and their applicability to MBEAs. Chapter 4 contains an introduction in the fundamentals of continuous reinforcement learning and will continue by introducing the Proximal Policy Optimization algorithm. Chapter 5 uses the findings of the preceding three chapters to introduce a self-learning estimation of normal distribution algorithm that uses reinforcement learning as its learning mechanism and ends with an empirical comparison between the proposed algorithm and existing manually-engineered MBEAs. The last chapter presents the conclusions of the thesis and ends with a discussion of the thesis as a whole.

2
Evolutionary algorithms

2.1 Black-box optimization

Black-box optimization, sometimes also called derivative-free optimization, is concerned with the optimization of objective functions for which the structure is not (or insufficiently) known. In particular, no gradient information about the objective is a priori available nor can it be requested directly. To formalize this sort of objective we define the concept of a black-box function.

Definition 2.1: Black-box function

Given a dimensionality $d \in \mathbb{N}$, a black-box function is a function $f : \mathbb{R}^d \to \mathbb{R}$ such that, for any $x \in \mathbb{R}^d$

- $f(x) \in \mathbb{R}$ can be calculated (or evaluated);
- no gradient information is known about f in x;
- no assumptions can be made about the analytic form of f.

A black-box function is named as such since it is often approached as an actual black-box that takes an input and returns an output without any knowledge about the internal workings, this is sometimes also called an oracle in mathematical optimization. Black-box functions are encountered in optimization in a myriad of professional fields such as engineering, medicine, economics and operations research [31]. These functions often arise due to the complexity of the underlying processes they describe, which are either unknown or too costly to model adequately. Examples of black-boxes include proprietary simulation software, and numerically approximated mathematical models [8]. The task of, given a black-box function, finding a solution in its domain that has a high function value, is called a black-box optimization problem.

Definition 2.2: Black-box optimization problem

Given a black-box function $f : \mathbb{R}^d \to \mathbb{R}$, the corresponding black-box optimization problem is finding $x^* \in \mathbb{R}^d$ such that

$$f(x^*) \approx \sup_{x \in \mathbb{R}^d} f(x), \tag{2.1}$$

where we assume that the supremum of f on $\mathbb{R}^d$ is finite. In the context of the black-box optimization problem, f is called the objective function.
Notation: we will often address a black-box optimization problem simply as "optimization problem" or "problem".

Practical applications will often require repeated optimization of problems that are closely related. For example, in the case of proprietary simulation software, a particular application can require to optimize the parameters of a simulation given different settings. In this case the underlying model is constant throughout all optimizations and only the settings change. We call sets of problems that are related in this way black-box optimization problem classes.

Definition 2.3: Black-box optimization problem classes

Given a dimensionality $d \in \mathbb{N}$, a black-box optimization problem class is defined by its function set

$$\mathcal{F} = \{(f : \mathbb{R}^d \to \mathbb{R}) : f \text{ is a black-box function}\}, \tag{2.2}$$

in the sense that any black-box problem in the black-box problem class has its objective function in the function set.
Throughout this work we will assume that there exists a certain distribution over the function set, sampling a function according to this distribution is denoted by $f \sim \mathcal{F}$.
Notation: We will often abbreviate "black-box optimization problem class" by "optimization problem class" or "problem class."

Note that if a problem class results from a practical application, the distribution over the function class is induced by the natural occurrence of functions according to the application. In the case of artificially generated problem classes, the distribution will have to be specified.

2.2 Evolutionary algorithms

If we consider the natural world, and the Darwinian process of evolution in particular, it can be formulated as a black-box optimization problem over all possible organisms. The objective of that problem will encode something along the lines of "The ability of an organism to survive and multiply in its environment", which is often called the fitness of the organism [28]. In order to optimize this objective, natural selection maintains a population of organisms. As time progresses, some individuals in the population will pass away due to their inability to survive in their environment (i.e. their low fitness). The remaining organisms will reproduce, creating new individuals that share the traits of their parents. During reproduction there is a slight chance that some of the traits of a new individual are mutated, possibly developing new traits that were, up until that point, not present in the population. By repeating this process over multiple generations, traits that lead to high fitness survive in the population, while other traits go extinct. This increases the overall fitness of the population, and thereby approximates an optimal solution of the underlying optimization problem, insofar such a solution exists.

Evolutionary algorithms (EAs) are approximate optimization algorithms that are inspired by this concept of natural evolution. EAs are generally population-based, meaning that, just like natural evolution, they maintain a population of feasible solutions of the problem at hand [28]. At the start of the algorithm a population is generated, often by randomly sampling solutions according to an initial distribution. Consequently, the value of the objective function, also called the fitness, of each solution, also called individual, in the population is evaluated. After generating the initial population and evaluating its fitness, an EA will generally consist of five processes it will execute iteratively until a predefined stopping criterion is met. These processes, in order, are:

Selection A subset of individuals in the population is selected, which will often be referred to as parents. The selection is based on the current population and the fitness of the individuals in it. The selected individuals will usually, but not necessarily, be the individuals with highest fitness in the population. The process that selects the individuals is called the selection operator. The selection operator is analogous to the passing away of organisms due to their inability to survive in their environment in natural evolution.

Recombination The parents are used as a basis to form new feasible solutions to the optimization problem at hand. This is done by combining properties of different parents, and thereby creating new individuals, in order to exploit the preferable traits of the parents. The process that recombines the selected individuals into new individuals is called the recombination operator. The recombination operator is analogous to the reproducing of the surviving organisms in natural evolution. The sharing of properties with the selected individuals is inspired by the genetic code being passed from the surviving organisms to their offspring. The new solutions created by the recombination operator will often be referred to as offspring.

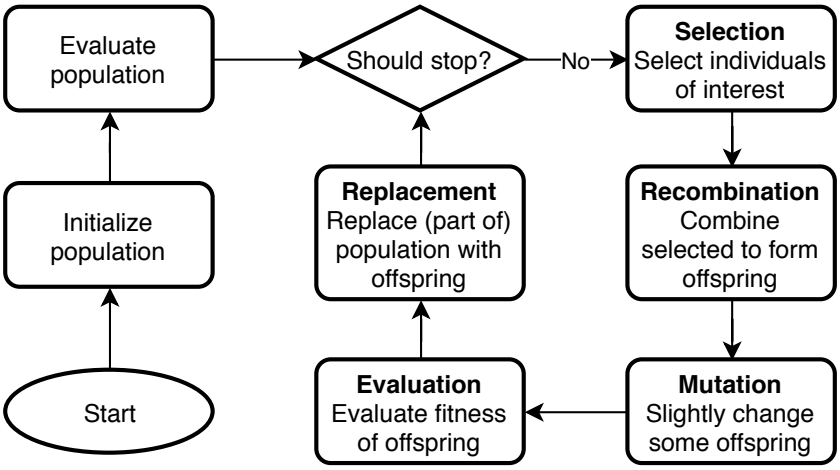


Figure 2.1: Schematic summary of the general EA framework.

Mutation After recombination, a fraction of the offspring is randomly altered by a mutation operator. By mutating a fraction of the offspring we enable the exploration of properties that were not previously present in the population which helps prevent premature convergence to local optima. The mutation operator is analogous to the mutation that happens randomly in nature.

Evaluation After mutation, the fitness of the offspring individuals is evaluated.

Replacement After evaluation, the offspring are ready to enter the population. This is done by the replacement operator. Common replacement operators will either replace the whole existing population with the offspring, or use another selection operator to select individuals from the existing population and the offspring to form the new population. A popular and relatively easy method to ensure that the maximum fitness of the population is monotonically increasing, copies the highest fitness individual from the existing population to the new population. An EA that uses this kind of strategy is said to be elitist.

The general framework of EAs introduced here is schematically summarized in Figure 2.1. In modern EAs, the operators are not always individually distinguishable since they are sometimes combined in a single operator or step of the algorithm. This is especially true for the recombination and mutation operators. The processes as outlined above will however almost always be conceptually present [28].

2.3 Estimation of distribution algorithms

Traditionally, EAs use manually-engineered operators based on heuristics developed after considerable analysis of the problem at hand. These heuristics are typically static with respect to the information collected during optimization. Therefore, the operators are not able to react to the locally encountered structures in the fitness landscape (the unknown graph of the fitness function), which can be detrimental for optimization problems that express fundamentally different local structures at different locations of the fitness landscape[6]. An example of such an objective function is the 2-dimensional Rosenbrock function,

$$f(x_1, x_2) = (a - x_1)^2 + b\left(x_2 - x_1^2\right)^2 \tag{2.3}$$

which, for $b \gg 1$, consists of a relatively easy structure outside of the $x_2 = x_1^2$ valley, while the valley itself is generally considered much more difficult due to its steep walls and relatively small gradient in the valley. This often results in ill-equipped optimizers easily finding the valley, but once there, continuously zig-zagging from one wall to the other instead of following the gradient of the valley.

In order to address these issues, more recent works have been dedicated to so-called Model-based evolutionary algorithms (MBEAs). These MBEAs replace the static heuristics of traditional EAs with

machine-learning models. The models are then trained on the data that is collected throughout optimization in the form of populations and the fitness of the individuals in them. By learning from observations about the current fitness landscape, MBEAs can adapt their behavior while optimizing, and therefore be more effective in multifaceted objective functions like the Rosenbrock function [6].

A subclass of MBEAs is the class of Estimation of Distribution Algorithms (EDAs). These algorithms build a model of the distribution of high fitness individuals. Typically, EDAs maintain a population distribution over the solution space, $\mathcal{D} : \mathbb{R}^d \to \mathbb{R}_{\geq 0}$, which is used to sample the population from. After evaluating the fitness of the individuals in the population, the population distribution, $\mathcal{D}$, is adjusted to incorporate the newly gained information about the fitness landscape. How the distribution is adjusted based on the new information is governed by an update rule that is (generally) static throughout the optimization process [6]. Such update rules are often based on maximum likelihood estimates and heuristics resulting from extensive analysis of the distribution [4, 12, 31]. The general pseudocode of an EDA is presented in Algorithm 1, where f is the objective function, N_{pop} is the population size, $\mathcal{D}_0$ is the initial population distribution, and UPDATEMODEL is the aforementioned update rule for the distribution. Note that the selection, recombination and mutation operators from the EA framework are combined in the UPDATEMODEL and population sample step. Examples of well-known and widely used EDAs include CMA-ES [12], AMaLGaM [4] and NES [31], which constitute the current state-of-the-art real-valued evolutionary algorithms.

Algorithm 1: Estimation of distribution algorithm

Input: $f, N_{\text{pop}}, \mathcal{D}_0$, UPDATEMODEL

$\mathcal{D} \leftarrow \mathcal{D}_0$; `// init the initial distribution`

while *stopping criterion not met* **do**

  $P \leftarrow (x_i \sim \mathcal{D})_{i=1}^{N_{\text{pop}}}$; `// sample population`

  $F \leftarrow (f(x_i))_{i=1}^{N_{\text{pop}}}$; `// evaluate fitness`

  $D \leftarrow$UPDATEMODEL$(P, F, \mathcal{D})$; `// Update population distribution`

return P;

2.4 Estimation of normal distribution algorithm

A class of EDAs that has shown to be especially popular for real-value black-box optimization is the class of Estimation of Normal Distribution Algorithms (ENDAs). As the name suggests ENDAs are EDAs that use the multivariate normal distribution as their population distribution. Arguments often provided to use the normal distribution as population distribution include:

Maximum entropy The normal distribution is the highest entropy distribution with finite variance over the Euclidean space. That means that, given a particular mean and covariance matrix, the normal distribution makes the least assumptions about the shape of the distribution as possible. It therefore ensures that we do not neglect a particular set of solutions while optimizing [12].

Stability The normal distribution is the only stable distribution with finite variance [10], i.e. the sum of two normally distributed random variables is again normally distributed, which generally simplifies the analysis of the distribution and the algorithms that use it.

Efficient approximation To estimate the parameters of a d-dimensional normal distribution with full covariance matrix, we have to estimate the mean vector, which contains d parameters, and the covariance matrix, which contains $\frac{1}{2}d^2 + \frac{1}{2}d$ parameters since it is a symmetric matrix. The total number of parameters to estimate is therefore $\frac{3}{2}d + \frac{1}{2}d^2$, which grows quadratically, and not exponentially, with the dimension [3]. Hence distribution has efficient space-complexity. Additionally, the maximum-likelihood estimators of the mean vector and covariance matrix, given a dataset, are well-studied and known to have a computational complexity quadratic in the dimension and linear in the size of the dataset.

Intuitive parameters The parameters of the normal distribution are intuitive. The mean directly encodes the location of the distribution, in the sense that the mean is equal to the mode of the distribution, and the distribution is symmetric around the mean. The covariance matrix directly encodes the shape of the distribution, in the sense that the shape and orientation of the iso-density ellipsoid is uniquely determined by the covariance matrix. This interpretability of the parameters significantly increases the ability to develop heuristics for the update rules of ENDAs.

To replace the distribution $\mathcal{D}$ in Algorithm 1 with a normal distribution, we use the standard parameterization of the normal distribution, the mean and covariance matrix, and alter the parameters based on the information about the fitness landscape that is collected during the execution of the algorithm. Concretely, an ENDA maintains a mean vector, $\mu \in \mathbb{R}^d$, and a covariance matrix, $\Sigma \in \mathbb{R}^{d \times d}$. At every time step (or generation) of the ENDA, a new population of $N_{\text{pop}} \in \mathbb{N}$ individuals is sampled from the normal distribution with mean, μ, and covariance matrix, Σ,

$$P = \{x_i \sim \mathcal{N}(\mu, \Sigma)\}_{i=1}^{N_{\text{pop}}} . \tag{2.4}$$

After the population has been sampled, the fitness of the individuals in the population is calculated and stored in a fitness vector,

$$F = (f(x_i))_{i=1}^{N_{\text{pop}}} \tag{2.5}$$

To enable the algorithm to make decisions based on all information the algorithm has collected, the algorithm maintains a history and adds the population, fitness vector, mean vector and covariance matrix to it in every time-step,

$$H = H || \, (\mu, \Sigma, P, F) \in \mathcal{H} \tag{2.6}$$

where $\mathcal{H}$ is the set of all possible histories and $||$ denotes the concatenation of two sequences. The history is then passed on to the mean function, $\hat{\mu} : \mathcal{H} \to \mathbb{R}^d$, which calculates a new mean. The new mean together with the history is then passed to the covariance function, $\hat{\Sigma} : \mathcal{H} \times \mathbb{R}^d \to \mathbb{R}^{d \times d}$, which calculates a new covariance matrix. The new mean and covariance matrix are then used to sample a new population and this cycle continues until a predefined stopping criterion is met. The pseudo-code for a general ENDA is presented in Algorithm 2.

Contribution

The behavior of an ENDA is, to a great extent, defined by the mean and covariance function. Just like the update rule of EDAs, the traditional ENDAs generally use carefully manually-engineered mean and covariance functions based on maximum-likelihood estimates and heuristics found after extensive analysis of ENDAs and the normal distribution. Much like the step from traditional EAs to MBEAs, our work proposes to take the next step from MBEAs to self-learning MBEAs. In particular, we propose the use of machine learning techniques to learn update rules for ENDAs that are specialized to a particular class of black-box optimization problems. In this way we aim to develop self-learning ENDAs that, by optimizing different problems in the same problem class, learn which structures are prevalent in the problem class and how to navigate those structures efficiently.

Algorithm 2: Estimation of normal distribution algorithm

Input: $f, n, \hat{\mu}, \hat{\Sigma}, \mu^{(0)}, \Sigma^{(0)}$

$\mu \leftarrow \mu^{(0)}$; // init mean

$\Sigma \leftarrow \Sigma^{(0)}$; // init covariance

$H \leftarrow ()$; // init history

while *stopping criterion 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

  $H \leftarrow H||(\mu, \Sigma, P, F)$; // extend history

  $\mu \leftarrow \hat{\mu}(H)$; // calculate new mean

  $\Sigma \leftarrow \hat{\Sigma}(H, \mu)$; // calculate new covariance

return P;

3
Machine learning

As the name suggests, machine learning is the field of research that studies systems (or machines), often called agents, that "learn" [21]. Learning in this context means improving the performance of the agent based on observations it has made about its environment. More concretely, consider an agent that receives a sequence of inputs, which could be scalars or vectors, but also images or text files. These inputs yield some information about the environment of the agent, since they originated from that environment. The process of distilling those inputs down to the essence of information it contains, and using that to improve the performance of the agent on its predefined task, is the subject of study in machine learning. This is done by combining ideas from statistics, computer science, cognitive science, mathematical optimization and many other fields in science and mathematics [11].

3.1 Self-learning ENDA

The goal of this thesis is to use machine learning techniques to develop a self-learning ENDA, which we define to be an ENDA that improves its ability to quickly find high fitness solutions by altering its behavior based on previous optimizations of functions in the problem class at hand. As remarked in Section 2.4, the behavior of an ENDA is almost completely governed by the mean and covariance function. Hence, to develop a self-learning ENDA it is sufficient to create a self-learning mean and covariance function. Therefore the goal of developing self-learning ENDAs can be reformulated to finding a mean and covariance function that maximizes the ability to construct high fitness solutions based on the previous optimization of objective functions. This work will disregard the covariance function in order to focus on the application of machine learning techniques without the inherent difficulties of the covariance function, which include:

Governs exploration The covariance matrix controls the shape of the normal distribution [12], therefore the covariance function controls the exploration vs exploitation behavior of the algorithm. Since it is not a priori known what a good exploration strategy is, it is not directly evident what a desired behavior constitutes or what a good performance measure is to base learning on.

Positive definiteness The covariance matrix of a (non-degenerate) normal distribution should be positive definite [10]. Therefore the covariance function should map the history to the space of positive definite matrices. This makes optimization over the space of covariance functions especially difficult.

Non-intuitive metrics Lastly, the standard Euclidean metric of positive definite matrices does not correspond well to the intuitive metric on covariance matrices with respect to their resultant normal distributions. More concretely, given two covariance matrices

$$\Sigma_1 = \begin{bmatrix} 0.01 & 0 \\ 0 & 1 \end{bmatrix} \qquad \text{and} \qquad \Sigma_2 = \begin{bmatrix} 1 & \sqrt{0.99} \\ \sqrt{0.99} & 1 \end{bmatrix}, \tag{3.1}$$

then Σ_1 and Σ_2 are equidistant to the identity matrix with respect to the Euclidean metric. However the iso-density ellipsoids of the corresponding normal distributions around 0, depicted in Figure 3.1, show that Σ_2 is intuitively closer to I than Σ_1, when we consider the resultant distributions. Therefore, to properly optimize over the set of covariance functions, a proper metric should first be decided upon, which we leave for further research.

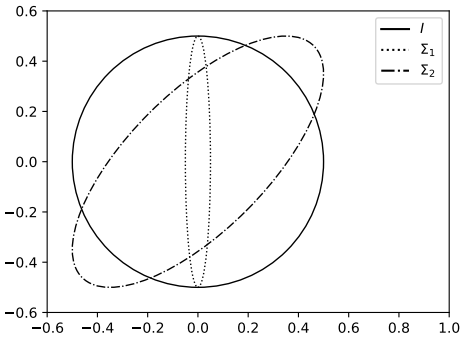


Figure 3.1: Iso-density ellipsoids of normal distributions with mean 0 and different covariance matrices indicated by the legend and (3.1).

3.2 Paradigms of machine learning

Machine learning can be roughly subdivided in three paradigms: unsupervised learning, supervised learning and reinforcement learning. The main component that separates these paradigms is the type of feedback the agent receives in addition to the input sequence [21].

In unsupervised learning, the agent receives no feedback in addition to the inputs. It is therefore mostly concerned with learning patterns in the input sequence as provided. Typical tasks in unsupervised learning include clustering and dimensionality reduction. Also, distribution estimation, the central part of EDAs, is considered to be a form of unsupervised learning [11].

On the other end of the spectrum we have supervised learning. In supervised learning the agent is given a sequence of desired outputs corresponding to the input sequence. The agent is then tasked with learning a mapping that, according to some performance measure, generalizes the given data set of input and desired output pairs best. Typical tasks for supervised learning include classification and regression [21].

In reinforcement learning the agent learns by interacting with its environment. After an input from the environment, often reflecting the state of the environment, the agent responds with an action. Based on the action, the state of the environment changes and the environment reacts with a scalar feedback, called a reward. The task of the agent is to find a behavior - a mapping from states to actions - such that the future reward is maximized [21]. Reinforcement learning is most well-known for its recent advances in the game of Go [26] and its applications in robotics [15].

Since we are looking to create a self-learning mean function, which is a mapping from the history space to d-dimension Euclidean space, unsupervised learning is not evidently applicable in this case. Additionally, we have a natural feedback in the form of the population fitness, which makes either supervised learning or reinforcement learning more suitable due to their ability to incorporate feedback in learning. The rest of this chapter will introduce the basic concepts of these paradigms and how they can be applied to the mean function.

3.3 Supervised learning

As stated earlier, supervised learning considers the task of learning a mapping that generalizes a given dataset of input-output pairs, also known as regression in mathematics. In particular, given some dataset of input-output pairs $(x_i, y_i)_{i=1}^N \subseteq (X \times Y)^N$ where X and Y are sets and y_i was generated by an unknown function $y = g(x)$, the task is to find a function $\hat{g} : X \to Y$ that approximates g. Generally[1], the method used to solve this starts out by considering a parameterized function $\hat{g}^{(0)}$:

[1]We only give a quick introduction here, for a more complete overview the reader is advised to read chapter 18 of "Artificial Intelligence: A Modern Approach" by Stuart Russel and Peter Norvig.

$\mathbb{R}^{D^{(0)}} \times X \to Y$, which we will assume to be a neural network (see Appendix A for an introduction in neural networks). We also subdivide the data in a training set, $(x_i, y_i)_{i=1}^K$, and a test set, $(x_i, y_i)_{i=K+1}^N$, where $K \in \{1, \ldots, N-1\}$. Then, by solving

$$\theta^{(t)} = \underset{\theta \in \mathbb{R}^{D^{(t)}}}{\arg\min} \frac{1}{K} \sum_{i=1}^{K} \mathcal{L}(\hat{g}^{(t)}(\theta, x_i), y_i), \tag{3.2}$$

for $t = 0$ and with a loss function $\mathcal{L} : Y \times Y \to \mathbb{R}$ that is appropriate for the underlying problem, we can find the parameter vector $\theta^{(0)}$ that gives the best fit of $\hat{g}^{(0)}$ to the training set. By now considering the loss on the test set, $\frac{1}{N-K} \sum_{i=K+1}^{N} \mathcal{L}(\hat{g}^{(0)}(\theta^{(0)}, x_i), y_i)$, we can find out if we are overfitting or underfitting. Namely, if the test set loss is significantly higher than the training set loss, we are overfitting to the training data. If the test set loss is approximately equal or lower than the training set loss we are possibly underfitting. Another sign of underfitting is a relatively large training loss. By then altering the parameterization to a new parameterization $g^{(1)} : \mathbb{R}^{D^{(1)}} \times X \to Y$ we can counter the under- or overfitting behavior. In particular, when overfitting the number of parameters should generally be decreased, when underfitting, the number of parameters should generally be increased. After changing the parameterization, (3.2) is solved again for the new parameterization to get the best fit parameters. By repeating this process, we can find a parameterization that is neither overfitting nor underfitting the training data, hence it generalizes as best as possible given the type of parameterization used, such as a neural network [21].

3.3.1 Direct supervised learning

To apply this methodology to the mean function, suppose we have some dataset of "good" examples mapping histories to means, $(H_i, \mu_i^*)_{i=1}^N$. Then, by the algorithm described above we can find a mean function $\hat{\mu}^* : \mathcal{H} \to \mathbb{R}^d$, such that $\hat{\mu}^*$ generalizes the data with respect to, for example, the Euclidean distance loss

$$\mathcal{L} : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R},\ (x, y) \mapsto ||x - y||_2^2. \tag{3.3}$$

Let us assume we can find such a $\hat{\mu}^*$. A major drawback from this approach is that the resultant function will never have a significantly better performance than the mechanism with which the dataset is created. Since "good" behavior of a mean function is not a priori known, it is furthermore difficult (if not outright impossible) to devise a good dataset. We can record the optimization history of existing algorithms that show good performance, but then the resulting mean function will never be better than its source and the algorithm can only specialize on a specific optimization problem class insofar the source algorithm is specialized on that problem class. Hence, supervised learning in its purest form does not allow us to create self-learning mean functions. It could however be used to find good starting parameters for a parameterized mean function, by cloning an existing algorithm, in order to speed up another self-learning mechanism.

3.3.2 Indirect supervised learning

Another methodology that is closely related to supervised learning is an idea proposed by Andrychowicz et al. in order to learn optimization steps in gradient descent algorithms [1]. The paper considers the problem of finding $x^* = \arg\min f(x)$ where $f : \mathbb{R}^d \to \mathbb{R}$ and the gradient $\nabla f : \mathbb{R}^d \to \mathbb{R}^d$ is known. Standard gradient descent uses

$$x^{(t+1)} = x^{(t)} - \alpha \cdot \nabla f(x^{(t)}) \tag{3.4}$$

to find the minimizer, where $\alpha > 0$ is the step-size (or learning rate in machine learning). The paper proposes

$$x^{(t+1)} = x^{(t)} + g(\nabla f(x^{(t)}), \theta) \tag{3.5}$$

where $g : \mathbb{R}^d \times \mathbb{R}^n \to \mathbb{R}^d$ is modeled as a recurrent neural network (RNN) and $\theta \in \mathbb{R}^n$ is the parameter vector of the network. The parameter vector θ is adjusted so as to minimize

$$\mathcal{L}(\theta) = \mathbb{E}_f \left[\sum_{t=1}^{T} f(x^{(t)}) \right] \tag{3.6}$$

where the objective function f is sampled from a predefined problem class and $T \in \mathbb{N}$ is the horizon.

Then $\mathcal{L}(\theta)$ is minimized by noting that $f(x^{(t)}) = f(x^{(0)} + \sum_{\tau=0}^{t} g(\nabla f(x^{(\tau)}), \theta))$ and hence by the chain rule

$$\nabla_\theta f(x^{(t)}) = \nabla_x f(x)|_{x=x^{(t)}} \cdot \sum_{\tau=0}^{t-1} \nabla_\theta g \left(\nabla f(x^{(\tau)}), \theta \right). \tag{3.7}$$

This enables us to use a stochastic gradient descent type approach. The expected value is approximated by repeatedly sampling an objective function from the problem class and applying (3.5) until a predefined stopping criterion is met. Based on the realized trajectory, (3.6) is minimized using gradient descent and (3.7).

Andrychowicz et al. show that this approach outperforms existing manually-engineered stochastic gradient descent algorithms on quadratic functions and machine learning tasks. However, there is no evident way to adjust this approach to work with the ENDA framework since, per definition of black-box problems, the gradient of the objective function is unknown, which is heavily relied on in both calculating the optimization step as the learning step used to update the optimizer itself.

3.4 Reinforcement learning

Reinforcement learning 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 that 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 future reward. 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, we can consider the rest of the algorithm, including the current fitness function, to be the part of the environment. The state of the environment then becomes the history of the algorithm while the action is the new mean. As a result, the reinforcement learning agent directly encodes the mean function, and an interaction is a single time-step in the ENDA framework. To make the policy adjustable and stochastic, we can use a parameterized distribution based on a neural network, which is often used in reinforcement learning [18]. Lastly, to specify to the agent that we want to maximize the fitness function, we will have to design an appropriate reward signal. This reward signal however finds its natural analog in the average population fitness of the current population, since the population fitness is exactly what we want to optimize.

This approach has its downsides. For example, by the already significant complexity of a reinforcement learning agent, the resulting ENDA will become even more complex. Also, due to the inherent stochasticity of the policy, the resulting mean function will be stochastic, which will almost certainly lead to suboptimal optimization paths. However, relative to unsupervised and supervised learning, reinforcement learning is the best fit as a machine learning paradigm to develop a self-learning mean function, because it directly learns a mapping. Taking the reward function to be the average population reward ensures that the agent maximizes the population fitness. And lastly, since the paradigm is based on trial-and-error learning, there is no need for a gradient of the objective function in the learning mechanism, which enables us to train the resulting ENDA on black-box optimization problem classes.

4

Reinforcement learning in continuous spaces

In order to develop a self-learning mean-function based on reinforcement learning (RL), as suggested in Chapter 3, we first consider the basics of RL. This chapter introduces the basics of RL and goes on to highlight the results of the field of particular interest to this work. This chapter will not be a complete overview of RL. For a more complete introduction the interested reader is advised to read "Reinforcement learning: An introduction" by Richard S. Sutton and Andrew G. Barto [29], which large parts of this chapter are based on.

Reinforcement learning (RL) is based on the concept of learning by interaction. The basic idea is easily explained by considering training a dog to fetch a ball. Dogs are, to the best of our knowledge, born with little prior knowledge about how to fetch a ball. Furthermore, our ability as humans to communicate complex concepts to a dog is fairly limited. Hence, we are in a situation that we want the dog to learn to fetch the ball, but we cannot simply explain the dog our intentions. We do however know that dogs like certain treats. A solution is to throw the ball and when the dog fetches the ball, give it a treat. That way the "good" behavior of fetching the ball when thrown is reinforced as the dog is rewarded with a treat since, we assume, the dog wants to maximize the amount of treats it gets. This way we can let the dog solve a certain problem, fetching the ball, without us having to explicitly explain step by step how it should do that, which would be near impossible. Reinforcement learning is the paradigm of machine learning that develops a computational approach to this concept of learning by doing.

4.1 Mathematical model

To approach learning by interaction from a computational point of view, reinforcement learning considers a formalism called the agent-environment interface. Based on that interface the dynamics of the problem is modelled using Markov decision processes which enable structured analysis of the problem. Both the agent-environment interface and the concept of Markov decision processes are introduced in this section.

4.1.1 Agent-environment interface

In RL the problem of learning by interaction, such as the dog example sketched above, can be captured by the agent-environment interface. In this interface the entity that learns and makes decisions is called the agent. The things that influence the agent or that the agent can interact with constitute the environment. In the case of our dog, we can consider the dog to be the agent and everything else, including us ourselves, to be the environment.

The agent can interact with the environment via an action. Actions can be anything, for example we could describe some of the actions of our dog as walk, jump, turn x degrees right/left, pick-up and drop. Now the idea is that the agent will, at every time-step, choose an appropriate action to achieve some goal. However, without any input, the agent has no idea what the effects of its actions are. Nature solved this by giving the dogs eyes and ears to observe the state of its environment. To drive the agent toward some desired goal, the environment presents a scalar reward to the agent after every interaction. The agent is programmed to maximize the expected future reward by changing its behavior accordingly.

In the case of our dog the reward is a treat that it will get after correctly achieving the desired goal, fetching the ball. The agent will interact with the environment for a preset number of interactions, or until some criterion for termination is met. One period of consecutive interactions is called an episode. In the case of our dog we can define an episode to be a single stretch of time in which we train the dog consisting of multiple ball throws. We can however also define a single throw to be an episode. This illustrates that the definition of an episode in some cases can be quite ambiguous. In general however an episode should contain multiple interactions. In RL the distinction between episodes is generally made in the idea that every episode is a new instance of the environment, which generally means having different initial conditions. The agent-environment interface is summarized in Figure 4.1.

It is important to realize that this interface merely formalizes the interaction between the agent and the environment. To do anything useful with it we still have to model the dynamics of the problem, i.e. how the environment processes the actions it receives from the agent to update its state, and how the agent decides which action to take when it observes a certain state.

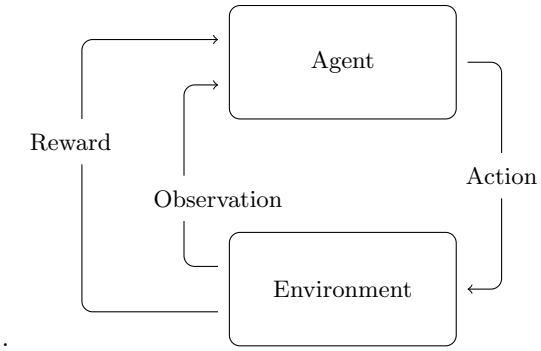


Figure 4.1: A schematic representation of the agent-environment interface.

4.1.2 Markov decision processes

The agent-environment interface as described in Section 4.1.1 formalizes the interaction between the agent and the environment, but does not describe the dynamics of either. To model the dynamics of both, we use a formalization of stochastic control processes closely related to Markov chains called Markov decision processes (MDPs). However, before we introduce MDPs we first have to do some mathematical groundwork in the form of definitions.

We start with the formal definitions of the states and actions introduced in the previous subsection, and the spaces they live in. With an eye on our application, optimization in finite-dimension Euclidean space, we have to ensure that our model can handle such spaces.

Definition 4.1: State space

A state space is a finite-dimensional Euclidean space $\mathcal{S}$. We call an element $\xi \in \mathcal{S}$ a state.

Definition 4.2: Action space

An action space is a finite-dimensional Euclidean space $\mathcal{A}$. We call an element $\alpha \in \mathcal{A}$ an action.

Using the state and action spaces we define the reward as a function of (state, action, state)-tuples. To see that all three arguments are necessary, consider someone selling a car. She starts in a state `has car`, after the action `try to sell car`, she either sold the car, landing her in a state `has no car`, or she did not, keeping her in `has car`. If she sells the car she gets a reward, money. If she does not sell the car she gets no reward. In this case the resulting state is essential to calculate the reward. The action cannot be left out either since the action `crash car` can leave her with the same initial and resulting

state while never giving her a reward. Leaving the initial state out would also not work since, if she is in the state `has no car` the action `try to sell car` will never result in a reward.

Definition 4.3: Reward function

Given a state space, $\mathcal{S}$, and an action space, $\mathcal{A}$, a reward function is a function $R : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$.

Now that we properly defined the structures introduced in Section 4.1.1, we can start to introduce the dynamics, starting with the dynamics of the environment. Since the exact dynamics of an environment are assumed to be either unknown or too complex to approach deterministically, we assume the dynamics to be stochastic. The stochasticity allows us to use statistical information about underlying processes to model them without the need of fully understanding them in the fine detail necessary for a deterministic model.

At the start of an episode, the environment is in an initial state, this state does not however have to be constant over all episodes of the environment. We will model this non-constant dynamic using an initial state distribution.

Definition 4.4: State distribution

Given a state space, $\mathcal{S}$, a state distribution is defined by a probability density function $\rho : \mathcal{S} \to \mathbb{R}_{\geq 0}$.

To complete the dynamics of the environment we should define how the state of the environment changes after the agent performs an action. Since the new state is dependent on both the previous state and the action performed, the new state is sampled from a probability distribution that changes depending on the current state and action, we call such a function a state-action transition kernel.

Definition 4.5: State-action transition kernel

Given a state space, $\mathcal{S}$, and an action space, $\mathcal{A}$, a state-action transition kernel is a function $P : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}_{\geq 0}$, such that for all $\xi, \alpha \in \mathcal{S} \times \mathcal{A}$, the map $(\xi') \mapsto P(\xi, \alpha, \xi')$ is a probability density function.

Using all previous definitions in this subsection we are now able to define a mathematical object that completely defines the environment, the Markov decision process.

Definition 4.6: Markov decision process (MDP)

Given a state space, $\mathcal{S}$, an action space, $\mathcal{A}$, an initial state distribution, ρ_0, a state-action transition kernel, P, and a reward function, R, the 5-tuple $(\mathcal{S}, \mathcal{A}, \rho_0, P, R)$ is called a Markov decision process.

Note that by itself this definition does not achieve much, it is merely a collection of mathematical objects. That is to be expected however since the MDP only encodes the environment, which still needs an agent to interact with. Specifically, we still need to model how the agent chooses an action based on an observation. This is formalized by a policy, a distribution over the action space that changes depending on the current state. The policy is stochastic to make the coming analysis possible, intuitively the stochasticity allows the agent to explore different behaviors and through that exploration converge to the best one.

Definition 4.7: Policy

Given an MDP, $(\mathcal{S}, \mathcal{A}, \rho_0, P, R)$, a policy is a function $\pi : \mathcal{S} \times \mathcal{A} \to \mathbb{R}_{\geq 0}$, such that for all $\xi \in \mathcal{S}$, the map $(\alpha) \mapsto \pi(\xi, \alpha)$ is a probability density function.
Notation: Given an MDP, let Π denote the set of all possible policies.

By combining the model of the environment, the MDP, and the model of the agent, the policy, we can complete the underlying interaction model and fully define the dynamics of our model. Combining these two gives rise to a stochastic sequence of environment states and agent actions, governed by the MDP and the policy. Additionally, the reward function of the MDP then, based on the sequence of states and actions, defines a stochastic sequence of rewards that scores the performance of the agent at every time-step.

Definition 4.8: Equipped MDP

Given an MDP, $(\mathcal{S}, \mathcal{A}, \rho_0, P, R)$, and a policy, π, we can equip the MDP with the policy π, giving rise to two stochastic processes, the state process $\{x^{(t)}\}_{t=0}^{\infty}$ and the action process $\{a^{(t)}\}_{t=0}^{\infty}$ governed by

$$x^{(0)} \sim \rho_0(\bullet), \tag{4.1}$$

$$\forall t \in \mathbb{Z}_{\geq 0}, \qquad a^{(t)} \sim \pi(x^{(t)}, \bullet), \tag{4.2}$$

$$\forall t \in \mathbb{N}, \qquad x^{(t)} \sim P(x^{(t-1)}, a^{(t-1)}, \bullet). \tag{4.3}$$

These processes in itself give rise to the reward process, $\{r^{(t)}\}_{t=0}^{\infty}$, governed by

$$\forall t \in \mathbb{Z}_{\geq 0}, \qquad r^{(t)} = R(x^{(t)}, a^{(t)}, x^{(t+1)}). \tag{4.4}$$

Notation: We will denote the expected value operator under the assumption that the MDP is equipped with policy π by $\mathbb{E}_\pi$. Also, the term "state" is overloaded by the element of a state space $\xi \in \mathcal{S}$ and a random variable that is a state $x \in \mathcal{S}$. The same holds for the term "action" where $\alpha \in \mathcal{A}$ is an element and $a \in \mathcal{A}$ is a random variable.

For notational convenience we introduce the concept of a episodic trajectory, which is the sequence of stochastic states and actions due to an equipped MDP truncated after a certain time-step.

Definition 4.9: Trajectory

Given an MDP, $(\mathcal{S}, \mathcal{A}, \rho_0, P, R)$, equipped with a policy, π, and some trajectory length $t \in \mathbb{N}$, the tuple of random variables

$$z_t = (x^{(0)}, a^{(0)}, x^{(1)}, \ldots, x^{(t)}) \in (\mathcal{S} \times \mathcal{A})^t \times \mathcal{S} = \mathcal{Z}_t \tag{4.5}$$

is called a trajectory of length t and $\mathcal{Z}_t$ is the trajectory space. Here $\{x^{(t)}\}_{t=0}^{\infty}$ and $\{a^{(t)}\}_{t=0}^{\infty}$ are the state and action processes, respectively, due to the equipped MDP.

Now that we fully defined the dynamics of our interaction model we can start to look at the goal of our analysis. Definition 4.8 defines the dynamics of our model for a given policy, but as we said earlier the goal of the agent is to find a policy that maximizes the reward obtained in an episode. Reinforcement learning normally considers the discounted return as the quantity to maximize.

Definition 4.10: Discounted return

Given a discount factor $\gamma \in [0,1]$, an episode length $T \in \mathbb{N}$, an MDP, $(\mathcal{S}, \mathcal{A}, \rho_0, P, R)$, equipped with a policy, π. Let $\mathcal{Z}_T$ be the trajectory space of the trajectory resulting from the episode, then the discounted return is the function

$$G_T : \mathcal{Z}_T \to \mathbb{R}, (\xi_0, \alpha_0, \ldots, \xi_T) \mapsto \sum_{t=0}^{T-1} \gamma^t R(\xi_t, \alpha_t, \xi_{t+1}). \tag{4.6}$$

Given an MDP and an episode length T we can now define the goal of an agent as finding a policy $\pi^* \in \Pi$ such that

$$\forall \pi \in \Pi,\ J(\pi) \leq J(\pi^*) \quad \text{where} \quad J : \Pi \to \mathbb{R},\ (\pi) \mapsto \mathbb{E}_\pi \left[G_T(z_T) \right]. \tag{4.7}$$

where z_T is the trajectory resulting from an episode.

4.2 Actor-Critic methods

Actor-critic methods are the basis of most algorithms for continuous reinforcement learning. An actor-critic algorithm generally consists of an actor, that is the policy that decides which actions to take in a particular state, and a critic, that tells the agent how good the chosen action was. Then the actor is changed based on the critic using a fundamental theorem in reinforcement learning called the policy gradient theorem.

4.2.1 Policy gradient theorem

In order for the agent to find a policy that maximizes the discounted return, it is convenient to parameterize the optimization domain by using a parameterized policy. Parameterization of the policy space allows the agent to use continuous optimization to approximate optimization over the allowed policies.

Definition 4.11: Parameterized policy

Given an MDP, $(\mathcal{S}, \mathcal{A}, \rho_0, P, R)$, and a parameter dimension $D \in \mathbb{N}$, a parameterized policy is a function $\pi : \mathbb{R}^D \times \mathcal{S} \times \mathcal{A} \to \mathbb{R}_{\geq 0}$ such that:

1. for all $\theta \in \mathbb{R}^D$, the map $(\xi, \alpha) \mapsto \pi(\theta, \xi, \alpha)$ is a policy;
2. for all $\xi, \alpha \in \mathcal{S} \times \mathcal{A}$, the map $(\theta) \mapsto \pi(\theta, \xi, \alpha)$ is continuous.

Notation: We will often denote a parameterized policy with parameter vector $\theta \in \mathbb{R}^D$ as $\pi_\theta : \mathcal{S} \times \mathcal{A} \to \mathbb{R}_{\geq 0}, (\xi, \alpha) \mapsto \pi(\theta, \xi, \alpha)$.

Using a parameterized policy, we can approximate (4.7) by finding $\theta^* \in \mathbb{R}^d$ such that

$$\theta^* = \arg\max_{\theta \in \mathbb{R}^D} J(\pi_\theta). \tag{4.8}$$

Note however that by optimizing over the parameter space instead of the policies themselves, the agent generally optimizes over a subset of policies, this can potentially limit the performance of the resultant policy.

To find θ^* in (4.8), one of the many flavors of gradient ascent[1] is often used, which, in its simplest form, is described by the iterative formula,

$$\theta \leftarrow \theta + \beta \cdot \nabla_\theta J(\pi_\theta) \tag{4.9}$$

[1] Readers interested in which optimizers are often used in ML how they work are advised to read `http://ruder.io/optimizing-gradient-descent/index.html`

where $\beta > 0$ is often called the learning rate or step size. To apply gradient ascent we need to be able to calculate $\nabla_\theta J(\pi_\theta)$ for arbitrary values of $\theta \in \mathbb{R}^D$. This is difficult because $J(\pi_\theta)$ is an expected value dependent on probabilities which in itself are again dependent on θ. Therefore, to calculate $\nabla_\theta J(\pi_\theta)$ we need the policy gradient theorem proposed by Sutton et al. [30].

Theorem 4.12: Policy gradient theorem

Given a discount factor $\gamma \in [0, 1)$, an MDP, $(\mathcal{S}, \mathcal{A}, \rho_0, P, R)$, equipped with a parameterized policy, π, and an episode length $T \in \mathbb{N}$,

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\sum_{\tau=0}^{T-1} \nabla_\theta \log\left(\pi\left(\theta, x^{(\tau)}, a^{(\tau)}\right)\right) \sum_{t=\tau}^{T-1} \gamma^t R\left(x^{(t)}, a^{(t)}, x^{(t+1)}\right)\right] \tag{4.10}$$

Sketch of proof. (based on [23]) First off, for any $t \in [T]$ define $Z_t : \mathbb{R}^D \times \mathcal{Z}_t \to \mathbb{R}_{\geq 0}$ to be the probability density function of the trajectory of the first t time-steps. Now by linearity of the expected value operator and the gradient operator,

$$\begin{aligned}
\nabla_\theta J(\theta) &= \nabla_\theta \mathbb{E}_{\pi_\theta}[G_T(z_T)], \\
&= \nabla_\theta \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{T-1} \gamma^t R\left(x^{(t)}, a^{(t)}, x^{(t+1)}\right)\right], \\
&= \sum_{t=0}^{T-1} \gamma^t \nabla_\theta \mathbb{E}_{\pi_\theta}\left[R\left(x^{(t)}, a^{(t)}, x^{(t+1)}\right)\right].
\end{aligned}$$

Take any $t \in [T-1]$, note that since $R\left(x^{(t)}, a^{(t)}, x^{(t+1)}\right)$ is only dependent on the trajectory up until $x^{(t+1)}$ we have

$$\mathbb{E}_{\pi_\theta}\left[R\left(x^{(t)}, a^{(t)}, x^{(t+1)}\right)\right] = \int_{\zeta=(\xi_0,\alpha_0,\ldots,\xi_{t+1})\in\mathcal{Z}_{t+1}} R(\xi_t, \alpha_t, \xi_{t+1}) Z_{t+1}(\theta, \zeta)\, \mathrm{d}\zeta$$

Taking the gradient on both sides yields

$$\begin{aligned}
\nabla_\theta \mathbb{E}_{\pi_\theta}\left[R\left(x^{(t)}, a^{(t)}, x^{(t+1)}\right)\right] &= \nabla_\theta \int_{\zeta\in\mathcal{Z}_{t+1}} R(\xi_t, \alpha_t, \xi_{t+1}) Z_{t+1}(\theta, \zeta)\, \mathrm{d}\zeta, \\
&= \int_{\zeta\in\mathcal{Z}_{t+1}} R(\xi_t, \alpha_t, \xi_{t+1}) \frac{\nabla_\theta Z_{t+1}(\theta, \zeta)}{Z_{t+1}(\theta, \zeta)} Z_{t+1}(\theta, \zeta)\, \mathrm{d}\zeta, \\
&= \int_{\zeta\in\mathcal{Z}_{t+1}} R(\xi_t, \alpha_t, \xi_{t+1}) (\nabla_\theta \log\left(Z_{t+1}(\theta, \zeta)\right) Z_{t+1}(\theta, \zeta)\, \mathrm{d}\zeta, \\
&= \mathbb{E}_{\pi_\theta}\left[R\left(x^{(t)}, a^{(t)}, x^{(t+1)}\right) \nabla_\theta \log\left(Z_{t+1}\left(\theta, z^{(t+1)}\right)\right)\right].
\end{aligned}$$

Per definition of the equipped MDP we have

$$Z_{t+1} : (\theta, (\xi^{(0)}, \alpha^{(0)}, \xi^{(1)}, \ldots, \xi^{(t+1)})) \mapsto \rho + 0(\xi^{(0)}) \pi(\theta, \xi_0, \alpha_0) P(\xi_0, \alpha_0, \xi_1) \ldots P(\xi_t, \alpha_t, \xi_{t+1}) \tag{4.11}$$

hence

$$\log(Z_{t+1}(\theta, z^{(t+1)})) = \log\left(\rho_0\left(x^{(0)}\right)\right) + \sum_{\tau=0}^{t}\left(\log\left(\pi\left(\theta, x^{(\tau)}, a^{(\tau)}\right)\right) + \log\left(P\left(x^{(\tau)}, a^{(\tau)}, x^{(\tau+1)}\right)\right)\right) \tag{4.12}$$

again taking the gradient on both sides and using the fact that the gradient is a linear operator we get

$$\nabla_\theta \log\left(Z_{t+1}\left(\theta, z^{(t+1)}\right)\right) = \sum_{\tau=0}^{t} \nabla_\theta \log\left(\pi\left(\theta, x^{(\tau)}, a^{(\tau)}\right)\right). \tag{4.13}$$

Finally resulting in

$$\begin{aligned}\nabla_\theta J(\theta) &= \sum_{t=0}^{T-1} \gamma^t \mathbb{E}_{\pi_\theta}\left[R\left(x^{(t)}, a^{(t)}, x^{(t+1)}\right) \sum_{\tau=0}^{t} \nabla_\theta \log\left(\pi\left(\theta, x^{(\tau)}, a^{(\tau)}\right)\right)\right] \\ &= \mathbb{E}_{\pi_\theta}\left[\sum_{\tau=0}^{T-1} \nabla_\theta \log\left(\pi\left(\theta, x^{(\tau)}, a^{(\tau)}\right)\right) \sum_{t=\tau}^{T-1} \gamma^t R\left(x^{(t)}, a^{(t)}, x^{(t+1)}\right)\right],\end{aligned}$$

where the last equality is due to swapping the sums and changing the limits accordingly.

The most beautiful thing about this theorem is that the policy gradient turns out to be independent from the dynamics of the environment. This is convenient because the dynamics of the environment are unknown to the agent. Additionally, the resulting gradient is quite intuitive. To see that, note that the logarithm is monotone increasing. So if the log of the probability increases, the probability itself increases. The factor $\sum_{t=\tau}^{T-1} \gamma^t R\left(x^{(t)}, a^{(t)}, x^{(t+1)}\right)$ is the discounted return received after choosing action $a^{(t)}$, hence it is an estimation of the "goodness" of picking action $a^{(t)}$ in state $x^{(t)}$. Since the gradient log probability of choosing $a^{(t)}$ in state $x^{(t)}$ is multiplied by the "goodness" of that choice, a high "goodness" will increase the probability of picking that action while a negative "goodness" will decrease that probability.

Approximation

Since we do not know the exact dynamics of the environment, we cannot calculate the expected value of (4.10) exactly. Hence, we will have to approximate the gradient using sampling. To that end, suppose the agent interacted with the environment for $N \in \mathbb{N}$ episodes, resulting in N realized trajectories

$$\forall k \in [N]: \qquad \hat{z}_T^{(k)} = \left(\hat{x}^{(0,k)}, \hat{a}^{(0,k)}, \ldots, \hat{x}^{(T,k)}\right) \tag{4.14}$$

where the hat denotes a realization of the random variable and the double superscript, for example in $\hat{x}^{(t,k)}$, denotes that the quantity is from interaction t in episode k. Then by taking the average of (4.9) over the N realized trajectories we get

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{k=1}^{N} \left(\sum_{\tau=0}^{T-1} \nabla_\theta \log\left(\pi\left(\theta, \hat{x}^{(\tau,k)}, \hat{a}^{(\tau,k)}\right)\right) \sum_{t=\tau}^{T-1} \gamma^t R\left(\hat{x}^{(t,k)}, \hat{a}^{(t,k)}, \hat{x}^{(t+1,k)}\right)\right) \tag{4.15}$$

Note that this estimate becomes increasingly accurate as $N \to \infty$.

4.2.2 Advantage estimation

Though the estimator in (4.15) theoretically works we can use a so-called baseline to reduce the variance of the estimate.

Definition 4.13: Baseline

Given a state space, $\mathcal{S}$, a baseline is a function $b : \mathcal{S} \to \mathbb{R}$. Where it is important that the baseline is only dependent on the states.

The variance of the gradient estimate can then be reduced by subtracting an appropriate baseline, b, from the discounted return received after choosing action $a^{(t)}$, resulting in

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\sum_{\tau=0}^{T-1} \nabla_\theta \log\left(\pi\left(\theta, x^{(\tau)}, a^{(\tau)}\right)\right) \left(\sum_{t=\tau}^{T-1} \gamma^t R\left(x^{(t)}, a^{(t)}, x^{(t+1)}\right) - b\left(x^{(\tau)}\right)\right)\right]. \tag{4.16}$$

Using the linearity of the expected value operator in combination with Theorem 4.12 and Theorem 4.14, it can trivially be shown that (4.16) holds.

Theorem 4.14: Baseline bias

Given an MDP, $(\mathcal{S}, \mathcal{A}, \rho_0, P, R)$, equipped with a parameterized policy, π_θ, a baseline, b, and an episode length $T \in \mathbb{N}$, the following holds

$$\forall \tau \in \{0, \dots, T-1\}: \qquad \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log\left(\pi\left(\theta, x^{(\tau)}, a^{(\tau)}\right)\right) b\left(x^{(\tau)}\right)\right] = 0 \tag{4.17}$$

Sketch of proof. Take $\tau \in \{0, \dots, T-1\}$ arbitrarily, then denoting the trajectory with length τ by z_τ yields

$$\begin{aligned}\mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log\left(\pi\left(\theta, x^{(\tau)}, a^{(\tau)}\right)\right) b\left(x^{(\tau)}\right)\right] &= \mathbb{E}_{\pi_\theta}\left[\mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log\left(\pi\left(\theta, x^{(\tau)}, a^{(\tau)}\right)\right) b\left(x^{(\tau)}\right)\middle| z_\tau\right]\right], \\ &= \mathbb{E}_{\pi_\theta}\left[b\left(x^{(\tau)}\right)\mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log\left(\pi\left(\theta, x^{(\tau)}, a^{(\tau)}\right)\right)\middle| z_\tau\right]\right],\end{aligned}$$

where the last equality holds since under the condition that z_τ is known, $x^{(\tau)}$ is also known so it can be taken outside of the conditional expectation. Now focussing on the inner expectation, and since z_τ is assumed known we can say that there exists some $\xi \in \mathcal{S}$ such that $x^{(\tau)} = \xi$, we get

$$\begin{aligned}\mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log\left(\pi\left(\theta, x^{(\tau)}, a^{(\tau)}\right)\right)\middle| z_\tau\right] &= \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log\left(\pi\left(\theta, x^{(\tau)}, a^{(\tau)}\right)\right)\middle| x^{(\tau)} = \xi\right], \\ &= \int_{\alpha \in \mathcal{A}} \nabla_\theta \log\left(\pi\left(\theta, \xi, \alpha\right)\right) \pi(\theta, \xi, \alpha)\, \mathrm{d}\alpha, \\ &= \int_{\alpha \in \mathcal{A}} \nabla_\theta \pi(\theta, \xi, \alpha)\, \mathrm{d}\alpha, \\ &= \nabla_\theta 1 = 0.\end{aligned}$$

Altogether, we get

$$\mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log\left(\pi\left(\theta, x^{(\tau)}, a^{(\tau)}\right)\right) b\left(x^{(\tau)}\right)\right] = \mathbb{E}_{\pi_\theta}\left[b\left(x^{(\tau)}\right) \cdot 0\right] = 0 \tag{4.18}$$

The next step is to find a baseline that maximally reduces the variance of the policy gradient estimate. A near optimal baseline is the state-value function [23].

Definition 4.15: State-value function

Given a discount factor $\gamma \in [0, 1)$, an MDP, $(\mathcal{S}, \mathcal{A}, \rho_0, P, R)$, equipped with a policy, π, and an episode length T, the state-value function for policy π at time step $\tau \in \{0, \dots, T\}$ is defined as

$$V_\pi^{(\tau)} : \mathcal{S} \to \mathbb{R}, (\xi) \mapsto \mathbb{E}_\pi\left[\sum_{t=\tau}^{T-1} \gamma^t R\left(x^{(t)}, a^{(t)}, x^{(t+1)}\right)\middle| x^{(t)} = \xi\right]. \tag{4.19}$$

An interpretation of the state-value function, $V_\pi^{(\tau)}(\xi)$, is the expected discounted return after encountering some state $\xi \in \mathcal{S}$ at time step τ following policy π.

To get an intuition why using the value function as baseline works so well, suppose that we know $V_{\pi_\theta}^{(\tau)}$ and consider the quantity

$$A^{(\tau)} = \sum_{t=\tau}^{T-1} \gamma^t R\left(\hat{x}^{(t)}, \hat{a}^{(t)}, \hat{x}^{(t+1)}\right) - V_{\pi_\theta}^{(\tau)}(\hat{x}^{(\tau)}) \tag{4.20}$$

called the advantage, for some realized trajectory $\hat{z}_T = (\hat{x}^{(0)}, \hat{a}^{(0)}, \ldots, \hat{x}^{(T)})$. The first term in (4.20) is the realized discounted return achieved by the agent after choosing action $\hat{a}^{(\tau)}$ at time step τ, an estimate of how good the agent actually performed after choosing that action. The second term is the discounted return that the agent is expected to achieve after being in state $\hat{x}^{(\tau)}$ at time step τ. Hence, the advantage is an estimate of the difference in discounted return (i.e. performance), achieved after $\hat{x}^{(\tau)}$ at time step τ, due to choosing the action $a^{(\tau)}$. Or simply put, the advantage tells us whether choosing $a^{(\tau)}$ yields a better (or worse) than expected discounted return.

In general however $V_{\pi_\theta}^{(\tau)}$ is not known. Therefore, to calculate the advantage, the value function is often approximated by continually fitting a parameterized function $\hat{V} : \mathbb{R}^{D_V} \times \mathcal{S} \to \mathbb{R}$ to the trajectories already collected for the policy gradient update.

It is common practice to, given a new set of realized trajectories, first update the policy based on that data and afterwards update the approximate value function. Such an approach ensures that the approximate value function is not correlated to the current data, which could bias the gradient estimate.

Given a realized trajectory $z_T = (x^{(0)}, a^{(0)}, \ldots, x^{(T)})$, the analysis above results in the policy gradient estimator

$$\hat{g} = \sum_{\tau=0}^{T-1} \hat{A}^{(\tau)} \nabla_\theta \log \left(\pi \left(\theta, x^{(\tau)}, a^{(\tau)} \right) \right), \tag{4.21}$$

where

$$\hat{A}^{(\tau)} = \sum_{t=\tau}^{T-1} \gamma^t R \left(\hat{x}^{(t)}, \hat{a}^{(t)}, \hat{x}^{(t+1)} \right) - \hat{V}(\phi, \hat{x}^{(\tau)}) \tag{4.22}$$

and $\phi \in \mathbb{R}^{D_V}$ is the current state-value function parameter vector.

4.3 Proximal policy optimization (PPO)

Policy gradient methods, as defined above, have two inherent difficulties [25]. First, since the policy parameters change after a single gradient ascent step and the sampled data is only valid for the current policy parameters, we can only do a single gradient step per sample. This significantly limits the amount of information that can be extracted from the collected data, which leads to a bad sample efficiency. To extract more information from the collected data we should consider that, since we have the gradient of the objective at the current policy, we have a first-order approximation of the objective which is valid in some neighborhood of the current policy. Using this first-order approximation to optimize over a subset of policies that are in some sense "close" to the current policy, we can do multiple optimization steps based on the collected data, significantly improving sample efficiency.

The second difficulty concerns the use of the gradient itself. Since we take the gradient of the objective with respect to the policy parameters, we implicitly use the Euclidean norm of the parameter space in that gradient, as it is the standard norm of the Euclidean space. So in parameterizing the policy we implicitly changed the "natural metric" of the optimization domain. As an effect, much like the non-intuitive metrics of the covariance matrix in Section 3.1, two policies that are intuitively close do not have to be close in parameter space. This discrepancy between policy space and parameter space can lead to a relatively small gradient step having radical effects on the policy. To avoid this we have to ensure that the policy resulting from an optimization step is in some way intuitively "close" to the previous policy.

To resolve these issues we first consider Trust region policy optimization (TRPO) that optimizes a surrogate-model of the discounted cumulative reward in a predefined "trusted"-region to optimize the policy with respect to J, which allows us to do multiple optimization steps while ensuring that the new policy remains in a neighborhood of the previous policy. The new objective is then used instead of the policy gradient described in (4.21) to obtain what is known as the Proximal policy optimization (PPO) algorithm as introduced by Schulman et al. [25].

4.3.1 Trust region policy optimization (TRPO)

Trust Region Policy Optimization (TRPO) is the most well-known trust-region method in RL. As the name suggests, TRPO maximizes J in a "trusted region" around the current policy $\pi_{\theta_{\text{old}}}$, where $\theta_{\text{old}} \in \mathbb{R}^D$ is the current policy parameter vector. Before we get into what a "trusted region" is, we have to consider which objective to optimize in the trusted region, since we cannot directly optimize J as we cannot calculate the expected value directly. For the policy gradient method we derived that we could use gradient ascent in combination with the estimated policy gradient in (4.21), in order to maximize $J(\pi_\theta)$. Since we collected the data used to calculate $\hat{A}^{(\tau)}$ under policy $\pi_{\theta_{\text{old}}}$ the gradient in (4.21) also has to be calculated in θ_{old}. Now by noticing that

$$\nabla_\theta \log\left(\pi\left(\theta, \hat{x}^{(\tau)}, \hat{a}^{(\tau)}\right)\right)\Big|_{\theta_{\text{old}}} = \frac{\nabla_\theta \pi\left(\theta, \hat{x}^{(\tau)}, \hat{a}^{(\tau)}\right)\big|_{\theta_{\text{old}}}}{\pi\left(\theta_{\text{old}}, \hat{x}^{(\tau)}, \hat{a}^{(\tau)}\right)}, \tag{4.23}$$

we can rewrite the policy gradient approximator in (4.21) as

$$\hat{g}_{\theta_{\text{old}}} = \sum_{\tau=0}^{T-1} \hat{A}^{(\tau)} \frac{\nabla_\theta \pi\left(\theta, \hat{x}^{(\tau)}, \hat{a}^{(\tau)}\right)\big|_{\theta_{\text{old}}}}{\pi\left(\theta_{\text{old}}, \hat{x}^{(\tau)}, \hat{a}^{(\tau)}\right)}. \tag{4.24}$$

Since $\hat{A}^{(\tau)}$ is constant with respect to θ and the gradient operator is linear, we can rewrite (4.24) as

$$\hat{g}_{\theta_{\text{old}}} = \nabla_\theta \sum_{\tau=0}^{T-1} \hat{A}^{(\tau)} \frac{\pi\left(\theta, \hat{x}^{(\tau)}, \hat{a}^{(\tau)}\right)}{\pi\left(\theta_{\text{old}}, \hat{x}^{(\tau)}, \hat{a}^{(\tau)}\right)}\Bigg|_{\theta_{\text{old}}} \tag{4.25}$$

By disregarding the gradient we obtain the objective function

$$L^{\text{TRPO}}_{\theta_{\text{old}}}(\theta) = \sum_{\tau=0}^{T-1} \hat{A}^{(\tau)} \frac{\pi\left(\theta, \hat{x}^{(\tau)}, \hat{a}^{(\tau)}\right)}{\pi\left(\theta_{\text{old}}, \hat{x}^{(\tau)}, \hat{a}^{(\tau)}\right)}, \tag{4.26}$$

which is an importance sampling estimator of $J(\pi_\theta)$ with respect to the sampling policy $\pi_{\theta_{\text{old}}}$. As a result, we can optimize (4.26) as long as we stay in a neighborhood of $\pi_{\theta_{\text{old}}}$.

To define a neighborhood of $\pi_{\theta_{\text{old}}}$ let us define the Kullback-Leibler divergence (KL-divergence).

Definition 4.16: Kullback-Leibler divergence

Let p, q be two probability densities over the same continuous random variable, then the Kullback-Leibler divergence is defined as

$$\text{KL}[p||q] = \int_x p(x) \log\left(\frac{p(x)}{q(x)}\right) \mathrm{d}x \tag{4.27}$$

To get a thorough intuitive understanding of what the Kullback-Leibler divergence is, the reader is advised to read the Medium post by Marko Cotra on this subject[2], which will be summarized here. First of all, it is important to note that the KL-divergence is not a distance metric since it is not commutative, i.e. generally for distributions p and q, $KL[p||q] \neq KL[q||p]$. However intuitively, given two distributions p and q the KL-divergence from p to q, $KL[p||q]$, is a measure for how well p can be distinguished from q when you sample from p. So the better we can distinguish p from q, the "further away" p is from q. In particular, if $p = q$ then and only then $KL[p||q] = 0$, which is a very useful property when comparing two distributions.

In TRPO the KL-divergence is used to define the trusted region around $\pi_{\theta_{\text{old}}}$. Introducing a constraint on the KL-divergence for every encountered state in a realized trajectory yields an optimization problem

[2]The post can be found at `https://medium.com/@cotra.marko/making-sense-of-the-kullback-leibler-kl-divergence-b0d57ee10e0a`

with many constraints, which is technically difficult to solve. Therefore TRPO constraints the allowed average KL-divergence over states encountered in a realized trajectory, $z_T = (x^{(0)}, a^{(0)}, \ldots, x^{(T)})$,

$$\frac{1}{T}\sum_{\tau=0}^{T-1} \mathrm{KL}\left[\pi_{\theta_{\text{old}}}\left(\hat{x}^{(\tau)}, \bullet\right) \middle|\middle| \pi_\theta\left(\hat{x}^{(\tau)}, \bullet\right)\right]. \tag{4.28}$$

Putting it all together, TRPO optimizes the policy based on a trajectory $\hat{z}_T = (\hat{x}^{(0)}, \hat{a}^{(0)}, \ldots, \hat{x}^{(T)})$ sampled under policy $\pi_{\theta_{\text{old}}}$, solving the constrained optimization problem

$$\begin{aligned} &\max_{\theta\in\mathbb{R}^D} && \sum_{\tau=0}^{T-1} \hat{A}^{(\tau)} \frac{\pi\left(\theta, \hat{x}^{(\tau)}, \hat{a}^{(\tau)}\right)}{\pi\left(\theta_{\text{old}}, \hat{x}^{(\tau)}, \hat{a}^{(\tau)}\right)} \\ &\text{s.t.} && \frac{1}{T}\sum_{\tau=0}^{T-1} \mathrm{KL}\left[\pi_{\theta_{\text{old}}}\left(\hat{x}^{(\tau)}, \bullet\right) \middle|\middle| \pi_\theta\left(\hat{x}^{(\tau)}, \bullet\right)\right] \leq \delta_{\text{KL}} \end{aligned} \tag{4.29}$$

where $\delta_{\text{KL}} > 0$ is a hyper parameter that specifies the "size" of the neighborhood of feasible solutions. This problem can efficiently be approximately solved using the conjugate gradient method, after making linear approximation to the objective and quadratic approximation to the constraint [25]. After solving (4.29), the value function approximator used to calculate the advantages is updated based on the trajectory and a new trajectory is sampled under the new policy. Again, (4.29) is solved with new trajectory and this process continues until some termination criterion is met.

TRPO is known as a very data efficient and robust method. Schulman et al. show that TRPO tends to give monotonic improvement on a wide variety of RL-tasks [24]. However, the resulting algorithm is relatively complicated since it has to solve (4.29) at every policy optimization step. Also, the agent cannot use parameter sharing between the approximate value function and the policy, which is often convenient to speed up training. In order to overcome these difficulties, Schulman et al. introduced Proximal Policy Optimization, an algorithm, inspired by TRPO and based on gradient ascent, that is simpler to implement, allows for parameter sharing, and empirically shows better sample efficiency.

4.3.2 Clipped objective

PPO uses the trust-region idea from TRPO and encodes it in a new objective in such a way that it can be optimized using first-order optimization techniques, such as gradient ascent. Just like with TRPO, suppose we have a current parameterized policy $\pi_{\theta_{\text{old}}}$ under which we sampled a trajectory $\hat{z}_T = (\hat{x}^{(0)}, \hat{a}^{(0)}, \ldots, \hat{x}^{(T)})$, and let $\tau \in \{0, \ldots, T-1\}$, then we would like a way to ensure a new policy π_θ does not differ too much from $\pi_{\theta_{\text{old}}}$, just like with TRPO. To that end we introduce the probability ratio at time step τ,

$$\psi_\tau : \mathbb{R}^D \to \mathbb{R}, (\theta) \mapsto \frac{\pi_\theta\left(\hat{x}^{(\tau)}, \hat{a}^{(\tau)}\right)}{\pi_{\theta_{\text{old}}}\left(\hat{x}^{(\tau)}, \hat{a}^{(\tau)}\right)}, \tag{4.30}$$

which is an indication of how much π_θ differs from $\pi_{\theta_{\text{old}}}$ in the point $(\hat{x}^{(\tau)}, \hat{a}^{(\tau)})$. In particular, if $\psi_\tau(\theta) \approx 1$ then π_θ is approximately equal to $\pi_{\theta_{\text{old}}}$ in $(\hat{x}^{(\tau)}, \hat{a}^{(\tau)})$. Therefore, to keep π_θ from differing too much we are looking to penalize policies that move $\psi_\tau(\theta)$ far away from 1.

Looking at the objective in (4.29) we see

$$L^{\text{TRPO}}(\theta) = \sum_{\tau=0}^{T-1} \hat{A}^{(\tau)} \frac{\pi\left(\theta, \hat{x}^{(\tau)}, \hat{a}^{(\tau)}\right)}{\pi\left(\theta_{\text{old}}, \hat{x}^{(\tau)}, \hat{a}^{(\tau)}\right)} = \sum_{\tau=0}^{T-1} \hat{A}^{(\tau)} \psi_\tau(\theta). \tag{4.31}$$

If we focus on a single term, $L_\tau^{\text{TRPO}}(\theta) = \hat{A}^{(\tau)}\psi_\tau(\theta)$, we observe that the objective encourages to increase $\psi_\tau(\theta)$ when $\hat{A}^{(\tau)} > 0$, and decrease $\psi_\tau(\theta)$ when $\hat{A}^{(\tau)} < 0$. Just like with TRPO we would like to optimize this objective without changing the policy too much, or without moving the probability ratio, $\psi_\tau(\theta)$, too far from 1. To that end, consider the clipped objective

$$L_\tau^{\text{CLIP}}(\theta) = \hat{A}^{(\tau)} \text{clip}(\psi_\tau(\theta), 1-\varepsilon, 1+\varepsilon), \tag{4.32}$$

where $\varepsilon > 0$ is a hyper parameter. The clipped objective is a modified version of the TRPO objective where the probability ratio is clipped. By clipping the probability ratio we remove the incentive to

move $\psi_\tau(\theta)$ outside of $[1-\varepsilon, 1+\varepsilon]$. In particular, if during optimization $\psi_\tau(\theta)$ is not in $[1-\varepsilon, 1+\varepsilon]$ we have $\nabla_\theta L_\tau^{\text{CLIP}}(\theta) = 0$ and, since we use first order optimization techniques, the optimization will stop. Now in order to make the PPO objective a lower bound of the (unclipped) TRPO objective we take the minimum of the clipped objective and the TRPO objective,

$$L_\tau^{\text{PPO}}(\theta) = \min\left(\hat{A}^{(\tau)}\psi_\tau(\theta), \hat{A}^{(\tau)}\text{clip}(\psi_\tau(\theta), 1-\varepsilon, 1+\varepsilon)\right) \leq L_\tau^{\text{TRPO}}(\theta). \tag{4.33}$$

Note that locally around θ_{old} we have $L_\tau^{\text{PPO}}(\theta) \approx L_\tau^{\text{TRPO}}(\theta)$, which holds with equality if $\psi_\tau(\theta) \in [1-\varepsilon, 1+\varepsilon]$. However, when θ moves away from θ_{old} to increase $L_\tau^{\text{TRPO}}(\theta)$, $L_\tau^{\text{PPO}}(\theta)$ will (at some point) become constant, no longer incentivizing θ to move away from θ_{old}, keeping π_θ in the neighborhood of $\pi_{\theta_{\text{old}}}$.

By taking the sum over all time steps in the trajectory we get the total PPO objective,

$$L^{\text{PPO}}(\theta) = \sum_{\tau=0}^{T-1} L_\tau^{\text{PPO}}(\theta) = \sum_{\tau=0}^{T-1} \min\left(\hat{A}^{(\tau)}\psi_\tau(\theta), \hat{A}^{(\tau)}\text{clip}(\psi_\tau(\theta), 1-\varepsilon, 1+\varepsilon)\right) \tag{4.34}$$

which is the central component of PPO itself.

4.3.3 Algorithm

Combining the PPO objective in (4.34) with a slightly altered version of the policy gradient algorithms, we get the PPO algorithm, which is outlined in pseudocode in Algorithm 3. Here we use the fact that the PPO objective does not incentivize the optimizer to move θ far away from θ_{old} to justify taking multiple, $N_{\text{epoch}} \in \mathbb{N}$, gradient ascent steps on the same sampled trajectory, with little risk of θ leaving the neighborhood of θ_{old}. That theoretically allows us to extract more information from the trajectory than vanilla policy gradient methods. Note that PPO has no hard constraints like TRPO, therefore it is possible for θ to leave the neighborhood of θ_{old}. However the relaxation of constraints introduced in PPO enables us to use simple optimization techniques such as gradient ascent, significantly decreasing the complexity of the algorithm as a whole with relative to TRPO.

Algorithm 3: Proximal Policy Optimization

Initialize θ, ϕ ; `// init policy and value function parameters`
while *stopping criterion not met* **do**
  $\theta_{\text{old}} \leftarrow \theta$;
  $\hat{z}_T \leftarrow$ sample in new instance of MDP under π_θ ; `// Sample trajectory`
  $\hat{A}^{(0)}, \ldots, \hat{A}^{(T-1)} \leftarrow$ calculate advantage according to (4.22) ; `// calculate advantages`
  for $i = 1, \ldots, N_{epoch}$ **do**
    $\theta \leftarrow \theta + \beta \cdot \nabla_\theta L^{\text{PPO}}(\theta)$; `// Optimize policy`
  $\phi \leftarrow$ update value function according to ϕ ; `// Update value function`

5
Learning the mean function

To apply the formalism of reinforcement learning 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, $H^{(t)} \in \mathcal{H}$, is the state, the new mean, $\mu^{(t+1)} \in \mathbb{R}^d$, is the action, and an interaction is a single time-step in the Estimation of Normal Distribution Algorithm (ENDA) framework, as defined in Section 2.4. To make the policy adjustable and stochastic, we will use a parameterized distribution based on a neural network, which is further detailed in Section 5.1. 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, $\hat{\xi} : \mathcal{H} \to \mathbb{R}^{m_{in}}$, 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 post processor, $\hat{\rho} : \mathcal{H} \times \mathbb{R}^{m_{out}} \to \mathbb{R}^d$, to obtain the new mean, $\mu^{(t+1)}$. 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 Section 5.2. 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, $\hat{r} : \mathcal{H} \to \mathbb{R}$. 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 Section 5.3. The self-learning mean function as outlined here is schematically summarized in Figure 5.1.

To maximize the expected discounted cumulative reward we use Proximal Policy Optimization (PPO), as introduced in Section 4.3 [25]. In this chapter we will approach PPO as a black box, altering the policy parameters so as to maximize the expected future reward. This makes the resulting framework easily adaptable to advances in the field of RL that could yield better performing algorithms.

To specialize the resulting mean function to a particular problem class, $\mathcal{F}$, as defined in Definition 2.3, 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

[1] We disregard the use of RNN encoders as described in [14]

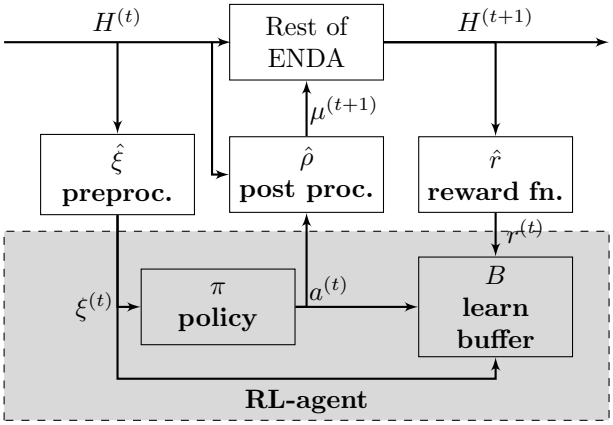


Figure 5.1: A schematic representation of the mean function.

turn. During execution of the ENDA, the RL-Agent will accumulate the 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 based on 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 4 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 4 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.

Algorithm 4: RL-based self-learning ENDA

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

1 $\theta \leftarrow$ randomInit(); // init policy parameters
2 $B \leftarrow$Queue(max_len $= M$); // init buffer
3 $t_{RL} \leftarrow 0$;
4 $t_{learn} \leftarrow 0$;
5 **while** $t_{RL} < T_{\max}$ **do**
6 $f \sim \mathcal{F}$; // sample function from class
7 $\mu \leftarrow \mu^{(0)}$;
8 $\Sigma \leftarrow \Sigma^{(0)}$;
9 $H \leftarrow ()$;
10 $t_{EA} \leftarrow 0$;
11 **while** *EA stopping criterion is not met* **do**
12 $P \leftarrow (x_i \sim \mathcal{N}(\mu, \Sigma))_{i=1}^{n}$; // sample population
13 $F \leftarrow (f(x_i))_{i=1}^{n}$; // evaluate fitness
14 $H \leftarrow H||(\mu, \Sigma, F, P)$; // append history
15 **if** $t_{EA} > 0$ **then**
16 $r \leftarrow \hat{r}(H, a)$; // calculate reward
17 $B \leftarrow B$.put(ξ, a, r); // add interaction to buffer
18 $t_{learn} = t_{learn} + 1$;
19 **if** $t_{learn} == N$ **then**
20 $\theta \leftarrow$ learn(π, θ, B); // update params.
21 $t_{learn} \leftarrow 0$
22 $\xi \leftarrow \hat{\xi}(H)$; // preproc. history
23 $a \sim \pi(\theta, \xi)$; // sample action from policy
24 $\mu \leftarrow \hat{\rho}(H, a)$; // postproc. action
25 $\Sigma \leftarrow \hat{\Sigma}(H)$; // calculate new covariance
26 $t_{EA} \leftarrow t + 1$;
27 $t_{RL} \leftarrow t_{RL} + 1$;
28 **return** θ;

5.1 The policy

The policy is the parametrized distribution from which an action is sampled based on the preprocessed history. Since the optimization task 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 Section 5.2, this effect can become negligible. In accordance with existing literature, the function approximator is a neural network [18]. This allows approximation of non-linear continuous functions while having an analytic gradient through backpropagation, which is necessary to apply gradient ascent in the PPO algorithm. 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 representation of the policy and general network topology is given in Figure 5.2. 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.

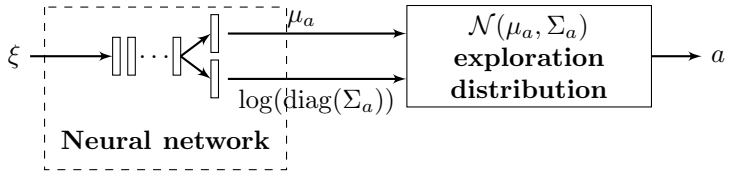


Figure 5.2: A schematic representation of the policy of SL-ENDA.

5.2 The pre- and post processor

The preprocessor, $\hat{\xi} : \mathcal{H} \to \mathbb{R}^{m_{\text{in}}}$, embeds the history space in m_{in}-dimensional Euclidean space. The main reason to use a preprocessor instead of feeding the history directly to the RL-agent, is to keep the dimensionality of the policy domain constant. This is necessary because the neural network used in the policy can only handle a domain of constant dimensionality. As an additional benefit of using such an embedding, it allows the agent to operate in a different space than the solution space. We will call this space the agent 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 [17], as well as to force invariances on the resulting mean function, which extends the validity region of the algorithm [12], 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. Throughout we will call a preprocessed history an observation.

The constant dimensionality of the observation is realized by selecting the $k \in \mathbb{N}$ most recent populations and fitness vectors in the history and basing the observation on those. We will generally assume $k \geq 2$ in order to preserve a temporal component in the observation. The selected populations and fitness vectors are then individually transformed to the agent space using a solution transformation $\nu_{\text{sol}} : \mathcal{H} \times \mathbb{R}^d \to \mathbb{R}^d$ and a fitness transformation $\nu_{\text{fit}} : \mathcal{H} \times \mathbb{R} \to \mathbb{R}$, respectively, which we will define later. The final preprocessor is then defined as

$$\hat{\xi}\left(H^{(t)}\right) = \text{Flatten}\left(\left(\left(\nu_{\text{sol}}\left(H^{(t)}, x_i\right)\right)_{x_i \in P^{(\tau)}}, \left(\nu_{\text{fit}}\left(H^{(t)}, F_i^{(\tau)}\right)\right)_{i=1}^{N_{\text{pop}}}\right)_{\tau=\max(t-k,0)}^{t}\right), \tag{5.1}$$

where we assume $t \geq k$ otherwise the observation is padded with zeros in order to achieve the necessary dimensionality of $m_{\text{in}} = (N_{\text{pop}} + 1) \cdot d \cdot k$ and Flatten is the operator that deterministically turns the mathematical structure given to it in a vector of all its values.

The post processor, $\hat{\rho}$, is used to transform the action of the agent back from the agent space to the solution space. This can be realized by taking the inverse of the solution transformation, ν_{sol}, with respect to the second argument. However, in order to do this we have to ensure that such an inverse is properly defined while constructing the solution transformation.

The rest of this section will define the fitness transformation in the first subsection and define two solution transformations in the second subsection.

5.2.1 Fitness transformation

Since the latest fitness vector generally contains the most information about the current situation of the ENDA, we will use that fitness vector as basis for normalization. Thereby ensuring that the latest fitness vector is properly normalized in the observation. By virtue of the black-box setting we can assume nothing about the objective function, apart from the assumption that it attains a maximum on $\mathbb{R}^d$ and is defined on the whole of $\mathbb{R}^d$, we therefore use a standard normalization technique for scalar data and define the fitness transformation as

$$\forall H^{(t)} \in \mathcal{H}, y \in \mathbb{R}, \qquad \nu_{\text{fit}}(H^{(t)}, y) = \left(\frac{y - \overline{F}^{(t)}}{\text{std}\left(F^{(t)}\right)} \right)_{i=1}^{N_{\text{pop}}}, \tag{5.2}$$

where $\overline{F}^{(t)}$ is the average of $F^{(t)}$ and $\text{std}(F^{(t)})$ is the standard deviation of $F^{(t)}$.

5.2.2 Solution transformation

Just like the fitness transformation we use the latest population as a basis for normalization. Before defining the solution transformations let us first introduce the desirable properties of these transformations:

Translation invariance In order for the agent to generalize well, we would like it to act the same on an objective function that is translated in the solution space as on its original. So given an objective function $f : \mathbb{R}^d \to \mathbb{R}$ and a vector $a \in \mathbb{R}^d$ we would like the agent to act the same on f as on $(x) \mapsto f(x + a)$. We can impose this using the solution transformation by interpreting the translation of the objective function as a translation of the solution space. Hence we call a solution transformation ν_{sol} translation invariant if for any $a \in \mathbb{R}^d$, $H^{(t)} \in \mathcal{H}$,and $x \in \mathbb{R}^d$,

$$\nu_{\text{sol}}\left(\left(\mu^{(t')} + a, \Sigma^{(t')}, P^{(t')} + a, F^{(t')}\right)_{t'=0}^{t}, x + a\right) = \nu_{\text{sol}}\left(H^{(t)}, x\right) \tag{5.3}$$

holds, where $P^{(t')} + a = (x_i + a)_{x_i \in P^{(t')}}$.

Scale invariance In order for the agent to generalize well we would like it to act the same on an objective function that is scaled in the solution space as on its original. So given an objective function $f : \mathbb{R}^d \to \mathbb{R}$ and a scalar $\alpha > 0$ we would like the agent to act the same on f as on $(x) \mapsto f(\alpha \cdot x)$. Again, we can impose this using the solution transformation by interpreting the scaling of the objective function as a scaling of the solution space. Hence we call a solution transformation ν_{sol} scale invariant if for any $\alpha > 0$, $H^{(t)} \in \mathcal{H}$ and $x \in \mathbb{R}^d$,

$$\nu_{\text{sol}}\left(\left(\alpha\mu^{(t')}, \alpha^2\Sigma^{(t')}, \alpha P^{(t')}, F^{(t')}\right)_{t'=0}^{t}, \alpha x\right) = \nu_{\text{sol}}\left(H^{(t)}, x\right) \tag{5.4}$$

holds, where $\alpha P^{(t')} = (\alpha x_i)_{x_i \in P^{(t')}}$.

Rotation invariance In order for the agent to generalize well we would like it to act the same on an objective function that is rotated in the solution space as on its original. So given an objective function $f : \mathbb{R}^d \to \mathbb{R}$ and a rotation matrix $A \in \text{SO}(d)$, where $\text{SO}(d)$ is the d-dimensional special orthogonal group, we would like the agent to act the same on f as on $(x) \mapsto f(Ax)$. Again, we can impose this using the solution transformation by interpreting the rotation of the objective function as a rotation of the solution space. Hence we call a solution transformation ν_{sol} rotation invariant if for any $A \in \text{SO}(d)$, $H^{(t)} \in \mathcal{H}$ and $x \in \mathbb{R}^d$,

$$\nu_{\text{sol}}\left(\left(A\mu^{(t')}, A\Sigma^{(t')}A^T, AP^{(t')}, F^{(t')}\right)_{t'=0}^{t}, Ax\right) = \nu_{\text{sol}}\left(H^{(t)}, x\right) \tag{5.5}$$

holds, where $AP^{(t')} = (Ax_i)_{x_i \in P^{(t')}}$.

Invertability As stated earlier, the inverse of the solution transformation in the second argument must exist to transform the action from the agent to the solution space. To that end, we call a solution transformation, ν_{sol}, invertible if there exists a mapping $\nu_{\text{sol}}^{-1} : \mathcal{H} \times \mathbb{R}^d \to \mathbb{R}^d$ such that for any $x \in \mathbb{R}^d$ and $H \in \mathcal{H}$,

$$\nu_{\text{sol}}^{-1}\left(H, \nu_{\text{sol}}\left(H, x\right)\right) = x. \tag{5.6}$$

We will consider two transformations in this work, the first is the translation and scale invariant (TSI) transformation and the second is the translation, rotation and scale invariant (TRSI) transformation.

Translation and scale invariant transformation (TSI)

The translation and scale invariant transformation is defined as

$$\forall x \in \mathbb{R}^d, t \in \mathbb{N}, H^{(t)} \in \mathcal{H} \qquad \nu_{\text{sol}}^{\text{TSI}}\left(H^{(t)}, x\right) = \frac{x - \mu^{(t)}}{\sqrt{\lambda_{\max}^{(t)}}}, \tag{5.7}$$

where $\mu^{(t)}$ is the mean at time-step t and $\lambda_{\max}^{(t)}$ is the maximal eigenvalue of the covariance matrix at time-step t.

To show that this transformation is indeed translation invariant we simply take arbitrary $a, x \in \mathbb{R}^d$, $H^{(t)} \in \mathcal{H}$ and show,

$$\begin{aligned}
\nu_{\text{sol}}^{\text{TSI}}\left(\left(\mu^{(t')} + a, \Sigma^{(t')}, P^{(t')} + a, F^{(t')}\right)_{t'=0}^{t}, x + a\right) &= \frac{x + a - (\mu^{(t)} + a)}{\sqrt{\lambda_{\max}^{(t)}}}, \\
&= \frac{x - (\mu^{(t)})}{\sqrt{\lambda_{\max}^{(t)}}}, \\
&= \nu_{\text{sol}}^{\text{TSI}}\left(H^{(t)}, x\right),
\end{aligned}$$

where we used the fact that the covariance matrices are not influenced by translation and hence the maximum eigenvalue of the covariance function isn't either.

Showing scale invariance can also be done by simply following the definition. Take $\alpha > 0$, $x \in \mathbb{R}^d$ and $H^{(t)} \in \mathcal{H}$ then for any matrix $M \in \mathbb{R}^{d \times d}$ with maximum eigenvalue $\lambda_{\max}$ we have $\alpha^2 \lambda_{\max}$ is the maximum eigenvalue of $\alpha^2 M$, which is a well-known result of linear algebra. Using this result we can show that (5.4) holds,

$$\begin{aligned}
\nu_{\text{sol}}^{\text{TSI}}\left(\left(\alpha\mu^{(t')}, \alpha^2\Sigma^{(t')}, \alpha P^{(t')}, F^{(t')}\right)_{t'=0}^{t}, \alpha x\right) &= \frac{\alpha x - \alpha\mu^{(t)}}{\sqrt{\alpha^2 \lambda_{\max}^{(t)}}} \\
&= \frac{\alpha}{\alpha} \frac{x - \mu^{(t)}}{\sqrt{\lambda_{\max}^{(t)}}} \\
&= \nu_{\text{sol}}^{\text{TSI}}\left(H^{(t)}, x\right).
\end{aligned}$$

Lastly, let us define

$$\forall x \in \mathbb{R}^d, t \in \mathbb{N}, H^{(t)} \in \mathcal{H}, \qquad \left(\nu_{\text{sol}}^{\text{TSI}}\right)^{-1}\left(H^{(t)}, x\right) = \sqrt{\lambda_{\max}^{(t)}} \cdot x + \mu^{(t)}, \tag{5.8}$$

and take $x \in \mathbb{R}^d$ arbitrarily, then

$$\left(\nu_{\text{sol}}^{\text{TSI}}\right)^{-1}\left(H^{(t)}, \nu_{\text{sol}}^{\text{TSI}}\left(H^{(t)}, x\right)\right) = \frac{\sqrt{\lambda_{\max}^{(t)}} \cdot x + \mu^{(t)} - \mu^{(t)}}{\sqrt{\lambda_{\max}^{(t)}}} = \frac{\sqrt{\lambda_{\max}^{(t)}} \cdot x}{\sqrt{\lambda_{\max}^{(t)}}} = x \tag{5.9}$$

which proves that $\nu_{\text{sol}}^{\text{TSI}}$ is invertible with inverse $\left(\nu_{\text{sol}}^{\text{TSI}}\right)^{-1}$. All in all, this leads us to conclude that $\nu_{\text{sol}}^{\text{TSI}}$ is a translation and scale invariant solution transformation.

Translation, rotation and scale invariant transformation (TRSI)

The translation, rotation and scale invariant transformation uses the last mean, $\mu^{(t)}$ and the Cholesky decomposition of the covariance matrix, $L^{(t)}\left(L^{(t)}\right)^T = \Sigma^{(t)}$, to transform the individuals in the population back to the space where the last population is standard normally distributed,

$$\forall x \in \mathbb{R}^d, t \in \mathbb{N}, H^{(t)} \in \mathcal{H}, \qquad \nu_{\text{sol}}^{\text{TRSI}}(H^{(t)}, x) = \left(L^{(t)}\right)^{-1}\left(x - \mu^{(t)}\right). \tag{5.10}$$

Note that the proof for translation invariance is completely analogous to the TSI-transformation, hence we will not repeat it here. To proof rotation invariance take $A \in \mathrm{SO}(d)$, $x \in \mathbb{R}^d$ and $H^{(t)} \in \mathcal{H}$ arbitrarily, then note that we can write $A\Sigma^{(t')}A^T = AL^{(t')}(AL^{(t')})^T$ which defines a Cholesky decomposition of $A\Sigma^{(t')}A^T$. Hence we find

$$\begin{aligned}
\nu_{\text{sol}}^{\text{TRSI}}\left(\left(A\mu^{(t')}, A\Sigma^{(t')}A^T, AP^{(t')}, F^{(t')}\right)_{t'=0}^{t}, Ax\right) &= \left(AL^{(t')}\right)^{-1}\left(Ax - A\mu^{(t)}\right) \\
&= \left(\left(L^{(t)}\right)^{-1}A^{-1}\right)\left(A\left(x - \mu^{(t)}\right)\right) \\
&= \left(L^{(t)}\right)^{-1}\left(x - \mu^{(t)}\right) \\
&= \nu_{\text{sol}}^{\text{TRSI}}\left(H^{(t)}, x\right)
\end{aligned}$$

which proves that $\nu_{\text{sol}}^{\text{TRSI}}$ is rotation invariant.

For scale invariance note that for any $\alpha > 0$ we have $\alpha^2\Sigma^{(t')} = \alpha L^{(t')}(\alpha L^{(t')})^T$ and $(\alpha L^{(t')})^{-1} = \alpha^{-1}(\alpha L^{(t')})^{-1}$, then actually proving (5.4) goes analogous to the TSI-transformation.

Now define

$$\forall x \in \mathbb{R}^d, \qquad \left(\nu_{\text{sol}}^{\text{TRSI}}\right)^{-1}(H^{(t)}, x) = L^{(t)}x + \mu^{(t)}, \tag{5.11}$$

and take $x \in \mathbb{R}^d$ arbitrarily, then

$$\left(\nu_{\text{sol}}^{\text{TRSI}}\right)^{-1}\left(H^{(t)}, \nu_{\text{sol}}^{\text{TRSI}}\left(H^{(t)}, x\right)\right) = L^{(t)}\left(L^{(t)}\right)^{-1}\left(x - \mu^{(t)}\right) + \mu^{(t)} = x - \mu^{(t)} + \mu^{(t)} = x \tag{5.12}$$

which proves that $\nu_{\text{sol}}^{\text{TRSI}}$ is invertible with inverse $\left(\nu_{\text{sol}}^{\text{TRSI}}\right)^{-1}$. Which, all in all, leads us to conclude that $\nu_{\text{sol}}^{\text{TRSI}}$ is a translation, rotation and scale invariant solution transformation.

5.3 The reward function

In order for the agent to learn something, it needs to known when something good has happened and when something bad has happened [21]. This kind of feedback is called a reward and is the signal that drives the agent to prefer certain policies over other policies. In other words, a reward signal defines the goal of a reinforcement learning problem [29]. Additionally, the reward signal can be used to guide the agent to its goal. For example, in a game of chess the most natural reward signal would be received at the end of match and look something like $+1$ when the opponent is checkmated, 0 on a draw and -1 when the opponent checkmates the agent. Such a reward signal would only inform the agent about its goal, forcing it to play a whole match before knowing whether the actions it took were good. In contrast, professional chess engines usually use an evaluation function that maps board positions to an estimate of the likelihood of that position leading to a win. Using a good evaluation function as the reward function guides the agent toward a winning board position, by rewarding the agent when it is approaching a winning position, not only when that position is reached. Adjusting the reward function in order to guide the agent to its goal is often called reward shaping in the RL literature and is very important for creating environments that can be "solved" efficiently [19].

To align the goals of the agent with the goals of the ENDA we should analyze what those goals are. The primary goal of an ENDA, *"Find the highest fitness solution in the solution space of the optimization*

problem at hand", follows directly from the underlying optimization problem. A secondary goal is more grounded in the realm of practicality and the fact that we do not have infinite resources to optimize the objective, which leads us to introduce the secondary goal *"Spend the least amount of evaluations of the objective functions in order to find the highest fitness solution"*. We make the common assumption in black-box optimization that objective function evaluations are a more limited resource than the computational resources needed to execute a step in the ENDA.

Reformulating the goals of the ENDA in context of the mean function, which is necessary to devise a reward function for the agent, yields *"Find a mean that, given a history, maximizes the expected maximum fitness of the individuals in the resulting population"*. Note that this goal does not directly encode the primary goal of an ENDA. However given the fact that the mean function has no direct influence over the covariance matrix, it uses all degrees of freedom it has to find a population that contains the highest fitness individual it can. Also, since the history contains all data collected about the optimization problem at hand and this reformulation tries to find the highest fitness individual at every time step, this goal encourages the mean function to find the highest fitness solution as quickly, and therefore[2] in as few objective function evaluations as possible. To simplify the coming analysis, we replace the expected maximum fitness of the population with the expected fitness of an individual in the population. The goal can now mathematically be formulated as, given a particular history $H \in \mathcal{H}$, find $\mu^* \in \mathbb{R}^d$ such that

$$\mu^* = \arg\max_{\mu \in \mathbb{R}^d} J\left(\mu, \hat{\Sigma}(H, \mu)\right), \qquad \text{where} \qquad J(\mu, \Sigma) = \mathbb{E}\left[f(x) | x \sim \mathcal{N}(\mu, \Sigma)\right] \tag{5.13}$$

and $\hat{\Sigma}$ is the (predefined) covariance function of the ENDA, which is unknown to the agent. Hence we will want to find a reward function $r : \mathcal{H} \to \mathbb{R}$ such that finding a mean function $\hat{\mu}^* : \mathcal{H} \to \mathbb{R}^d$ that maximizes the expected reward maximizes $J(\mu, \hat{\Sigma}(H, \mu)$.

Now that we examined the role of a reward function and the goal that is to be achieved by the agent, we can propose reward functions that achieve this goal. The rest of this section will propose three different reward functions and analyze their theoretical characteristics. We will assume we are currently at time step $t \in \mathbb{N}$ in the execution of the ENDA, and we therefore have access to $H^{(t)} = (\mu^{(\tau)}, \Sigma^{(\tau)}, P^{(\tau)}, F^{(\tau)})_{\tau \in [t]}$.

5.3.1 Fitness reward

Since the average population fitness, $\overline{F}^{(t)}$, is the maximum likelihood estimate of $J(\mu, \Sigma)$, it is natural to define the reward function as

$$r_{\text{fitness}}(H^{(t)}) = \overline{F}^{(t)} := \frac{1}{N_{\text{pop}}} \sum_{i \in [N_{\text{pop}}]} F_i^{(t)}. \tag{5.14}$$

We call this reward the fitness reward.

A property of r_{fitness} is that the reward is of the same scale as the fitness. That, however, is not preferable since a problem class can contain scaled versions of problems. A scale variant reward could theoretically skew the policy to be more performant on large scale objective functions, since a large scale objective function has a larger potential gain of reward. Another property of r_{fitness} is translation variance with respect to the objective function. Hence an objective function that is translated by a constant positive scalar will result in a systemically higher reward than the original objective function. This is undesired since it makes the agent more likely to reinforce any behavior it displays on positively translated objective functions and disregard any behavior it displays on negatively translated objective functions. To accommodate for scale and translation invariance, the fitness reward should be normalized.

[2]We assume that the population size is a constant throughout any run of the ENDA

5.3.2 Normalized fitness reward

The normalized fitness reward is a normalized version of the fitness reward, designed to address the scale and translation variance of the fitness reward. In the ideal case where $f_{\min} = \min_{x \in \mathbb{R}^d} f(x)$ and $f_{\max} = \max_{x \in \mathbb{R}^d} f(x)$ exist, are known and are finite, the reward could be normalized in the standard way,

$$r_{\text{norm. fitness}}(H) = \frac{r_{\text{fitness}}(H) - f_{\min}}{f_{\max} - f_{\min}}. \tag{5.15}$$

Note that the transformation is merely a translation and a scaling of the range with respect to r_{fitness}, not changing the position of the maximum. Therefore we would expect r_{fitness} and $r_{\text{norm. fitness}}$ to result in the same policy, since the systemic reward scale should theoretically not influence the RL algorithm.

Sadly, we do not in general know $f_{\max}$, and $f_{\min}$ often does not even exist. One way to solve these difficulties is to calculate the rewards in retrospect. Say the algorithm was used to optimize a particular objective function f resulting in a total history

$$H^{(T)} = \left(\mu^{(t)}, \Sigma^{(t)}, P^{(t)}, F^{(t)}\right)_{t \in [T]}, \tag{5.16}$$

where $T \in \mathbb{N}$ is the total number of iterations in EA time. Then for any $t \in [T]$ we can define the reward

$$r^{(t)}_{\text{retrospect}}(H^{(T)}) = \frac{r_{\text{fitness}}\left(H^{(t)}\right) - \min_{\tau \in [T]} \min_{i \in [N_{\text{pop}}]} F_i^{(\tau)}}{\max_{\tau \in [T]} \max_{i \in [N_{\text{pop}}]} F_i^{(\tau)} - \min_{\tau \in [T]} \min_{i \in [N_{\text{pop}}]} F_i^{(\tau)}}, \tag{5.17}$$

leaving the reward a scalar between 0 and 1. Note however that this breaks the requirement that the reward at time-step t is a function of $H^{(t)}$ as it now needs all fitness vectors in $H^{(T)}$. Hence, using this reward theoretically breaks the foundational assumptions of the RL theory as described in Chapter 4. We will therefore disregard it and leave the effects of using this reward for further research.

In order to adhere to the predefined framework, we will consider the estimated maximum and minimum to be a functions of $H^{(t)}$. In particular we will estimate the maximum with the maximum fitness value encountered until the previous time step,

$$\hat{f}_{\max,\text{delay}}\left(H^{(t)}\right) = \max_{\tau \in [t-1]} \max_{i \in [N_{\text{pop}}]} F_i^{(\tau)}. \tag{5.18}$$

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. If the current fitness vector was used in the approximation of the maximum, any improvement of the maximum would result in a reward close to 1 but never higher.

The minimum is estimated by the maximum of three terms:

Initial minimum The initial minimum uses the minimal fitness of the initial population, or

$$\hat{f}_{\min,\text{init}}\left(H^{(t)}\right) = \min_{i \in [N_{\text{pop}}]} F_i^{(0)}. \tag{5.19}$$

This approach introduces a constant baseline that cannot be influenced by the agent, thereby assuring that it cannot be exploited. Taking the overall minimum fitness up until time-step t would allow the agent to make one move to a low fitness area resulting in a low minimum, significantly improving the rest of the received rewards by normalization.

Decaying minimum The decaying minimum interpolates exponentially between the current overall minimum and current overall maximum, or mathematically

$$\hat{f}_{\min,\text{decay}}\left(H^{(t)}\right) = \beta^t \cdot \min_{\tau \in [t-1]} \min_{i \in [N_{\text{pop}}]} F_i^{(\tau)} + (1 - \beta^t) \cdot \max_{\tau \in [t-1]} \max_{i \in [N_{\text{pop}}]} F_i^{(\tau)}, \tag{5.20}$$

where $\beta \in [0, 1]$ is the decaying factor. Again the current fitness vector is disregarded for reasons analog to the maximum estimator. The exponential decay encourages the agent to stay increasingly closer to the current maximum, allowing early exploration and punishing late stage divergence.

Window minimum The window minimum uses the minimum fitness of the last $k \in [t-1]$ populations,

$$\hat{f}_{\text{min,window}}\left(H^{(t)}\right) = \min_{\tau \in \{t-k,\dots,t-1\}} \min_{i \in [N_{\text{pop}}]} F_i^{(\tau)}. \tag{5.21}$$

Using a window allows the minimum to stay relatively close to the fitness of the current population, ensuring that the change in fitness stays significant with respect to the normalization.

The estimated minimum value can then be written as

$$\hat{f}_{\text{min}}\left(H^{(t)}\right) = \max\left(\hat{f}_{\text{min,init}}\left(H^{(t)}\right), \hat{f}_{\text{min,decay}}\left(H^{(t)}\right), \hat{f}_{\text{min,window}}\left(H^{(t)}\right)\right), \tag{5.22}$$

letting us define the normalized fitness reward as

$$r_{\text{norm. fitness}}(H) = \frac{r_{\text{fitness}}(H) - \hat{f}_{\text{min}}\left(H^{(t)}\right)}{\hat{f}_{\text{max,delay}}\left(H^{(t)}\right) - \hat{f}_{\text{min}}\left(H^{(t)}\right)}. \tag{5.23}$$

As we set out, the normalized fitness reward trivially is translation and scale invariant with respect to the objective function. However, due to the lack of knowledge about the structure of the black-box objective functions, the normalization constants have to be estimated using data collected during optimization. This makes for seemingly ad-hoc minimum and maximum estimates without much rigorous reasoning. Especially the three minimum terms are a product of much trial-and-error based empirical research. This raises the question whether, by looking at ratios of steps in fitness value throughout optimization, we can derive a reward signal that normalizes itself without the need for ad-hoc estimators.

5.3.3 Differential reward

The differential reward, defined as

$$r_{\text{diff}}(H^{(t)}) = \frac{\overline{F}^{(t)} - \overline{F}^{(t-1)}}{f_{\max} - \overline{F}^{(t)}}. \tag{5.24}$$

looks at the improvement in average population fitness since the previous time-step (numerator), relative to an estimation of the current error (denominator). To derive the differential reward let us consider the mean function goal, (5.13), and define the (theoretically) maximum achievable goal,

$$J_{\max} := \sup_{\mu \in \mathbb{R}^d, \Sigma \in \text{PD}^d} J(\mu, \Sigma) = \max_{x \in \mathbb{R}^d} f(x) =: f_{\max} \tag{5.25}$$

where the supremum is necessary since we cannot achieve a deterministic distribution with positive definite matrices, so the maximum does not have to exist, and the second equality holds when f is continuous. We can now rewrite (5.13) as a minimization of the error, the distance from the current value to its maximum,

$$\mu^* = \arg\min_{\mu \in \mathbb{R}^d} J_{\max} - J\left(\mu, \hat{\Sigma}(\mu)\right), \tag{5.26}$$

which per definition of $J_{\max}$ has optimal value at least 0. For notational convenience let us now introduce a short hand notation

$$\forall t \in [T], \qquad J^{(t)} = J\left(\mu^{(t)}, \hat{\Sigma}\left(\mu^{(t)}\right)\right) \approx \overline{F}^{(t)} \tag{5.27}$$

where T is the number of time-steps the ENDA is ran, $\mu^{(t)}$ is the mean of the population of the ENDA at time-step t and $\hat{\Sigma}$ is the covariance function of the ENDA. Note that the approximation goes to equality as $N_{\text{pop}} \to \infty$. Since we are minimizing in (5.26) we would like

$$J_{\max} - J^{(t-1)} > J_{\max} - J^{(t)} \tag{5.28}$$

to hold for every time-step t, and in particular we would like to maximize

$$\psi\left(H^{(t)}\right) = \frac{J_{\max} - J^{(t-1)}}{J_{\max} - J^{(t)}}. \tag{5.29}$$

We can interpret $\psi(H^{(t)})$ as the relative loss of error in the last time-step, in the sense that the numerator denotes the error of the previous population and the denominator denotes the error of the current population. Another intuition can be gained by considering the final error of the ENDA,

$$J_{\max} - J^{(T)} = \frac{J_{\max} - J^{(T-1)}}{\psi(H^{(T)})} = \frac{J_{\max} - J^{(T-2)}}{\psi(H^{(T-1)}) \cdot \psi(H^{(T)})} = \cdots = \frac{J_{\max} - J^{(0)}}{\prod_{t=1}^{T} \psi\left(H^{(t)}\right)}, \tag{5.30}$$

hence the final error of the ENDA can be minimized by maximizing $\psi(H^{(t)})$ for each $t \in [T]$.

At this point an attentive reader could note that (given that we can estimate $J_{\max}$) ψ defines a reward function that is different from the differential reward and would be correct. To find the differential reward we need the observation that ψ has the bias that the reward is always positive, which given a good value function (see Section 4.2.2) should be countered but is still better to avoid. To that end we use the simple trick of adding $0 = J^{(t)} - J^{(t)}$ to the numerator yielding

$$\psi\left(H^{(t)}\right) = \frac{J_{\max} - J^{(t-1)}}{J_{\max} - J^{(t)}} = \frac{J_{\max} + J^{(t)} - J^{(t)} - J^{(t-1)}}{J_{\max} - J^{(t)}} = 1 + \frac{J^{(t)} - J^{(t-1)}}{J_{\max} - J^{(t)}} \approx 1 + r_{\text{diff}}(H^{(t)}). \tag{5.31}$$

We can therefore conclude that maximizing ψ is equivalent to maximizing r_{diff} since they differ by a constant.

Since $f_{\max}$ is not generally known beforehand, we need to substitute it with a known quantity before we can use (5.24) as a reward function. Contrary to the estimate used in Section 5.3.2, we need to make sure the approximation of $f_{\max}$ is always bigger than $\overline{F}^{(t)}$. This is necessary to ensure the denominator is positive, which is an assumption we used to get from (5.28) to (5.29).

The most natural choice for $f_{\max}$ is the highest fitness encountered. Assuming f does not contain a plateau, is sufficiently smooth and $n \geq 2$ we find

$$\forall \mu \in \mathbb{R}^d, \Sigma \succ 0 : \qquad \mathbb{P}\left[\max_{i \in [N_{\text{pop}}]} f(x_i) > \frac{1}{N_{\text{pop}}} \sum_{i=1}^{n} f(x_i) \mid x_1, \ldots, x_{N_{\text{pop}}} \sim \mathcal{N}(\mu, \Sigma)\right] = 1. \tag{5.32}$$

Hence we know that

$$\hat{f}_{\max,\text{diff}}(H^{(t)}) = \max_{\tau \in t} \max_{i \in [N_{\text{pop}}]} F_i^{(\tau)} > \overline{F}^{(t)}, \tag{5.33}$$

which qualifies $\hat{f}_{\max,\text{diff}}(H^{(t)})$ as a substitute for $f_{\max}$, and we will use it as such for the differential reward.

5.4 Experiments

In this section we empirically compare the performance of the introduced self-learning MBEA (SL-ENDA) to that of AMaLGaM [4] and CMA-ES [12]. 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 5.3 and Section 5.2, respectively. We start however with a description of the experimental set-up.

5.4.1 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 $N_{\text{pop}} = 20 + 10 \cdot d$, 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 agent [25]. 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 [7], 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.

Problem classes

To train the agent, function classes have to be defined to sample objective functions from. All function classes 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 [13]. The sphere function is used as a baseline to compare optimal convergence rates. The ellipsoidal function tests performance on ill-conditioned and (when rotated) non-separable functions. Lastly, the Rosenbrock function tests performance on non-convex unimodal functions with considerable local structure. All base functions are translated such that their optimum is located in $\vec{0} \in \mathbb{R}^d$.

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

$$\mathcal{F}_b = \left\{ (x) \mapsto -f_b(Rx + a) : R \in SO(d), a \in B(\vec{0}, 100) \right\}, \tag{5.34}$$

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 $B(\vec{0}, 100) \subset \mathbb{R}^d$ is the ball of radius 100 around the origin. Functions are sampled from $\mathcal{F}_b$ by uniformly sampling a and R from $B(\vec{0}, 100)$ and $SO(d)$, respectively.

Evaluation

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 4.

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 $\vec{0} \in \mathbb{R}^d$ 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 4.

The implementation in Python 3.6 used to produce the results in this section is publicly available.[3] PPO was implemented using the Tensorflow library and based on the OpenAI baselines library [9]. All experiments were performed on a 64-core (4 x 16-core AMD Opteron(tm) Processor 6386 SE) server running Fedora 28.

5.4.2 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 5.3 shows the performance of SL-ENDA equipped with the differential, fitness and normalized fitness reward, as introduced in Section 5.3, 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 (number

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

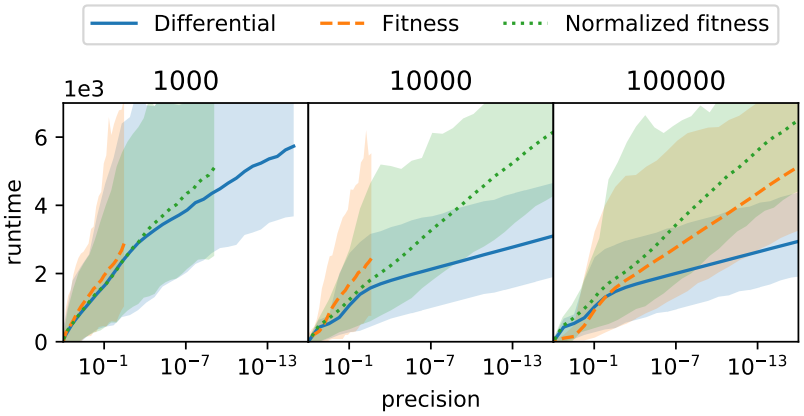


Figure 5.3: Runtime until precision is achieved by SL-ENDA equipped with differential, fitness and normalized fitness reward on the 2-dimensional Rosenbrock function class. The vertical axis indicates runtime (number of fitness function evaluations) until the precision on the horizontal axis is first 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 precision that was achieved for at least 10% of the sampled functions.

of fitness function evaluations) 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 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.

5.4.3 Agent space analysis

To find the most performant agent space we compare the use of TRSI- and TSI-space, as introduced in Section 5.2. 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 5.4 shows that the TRSI-space results in a highest 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 5.4 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 thesis.

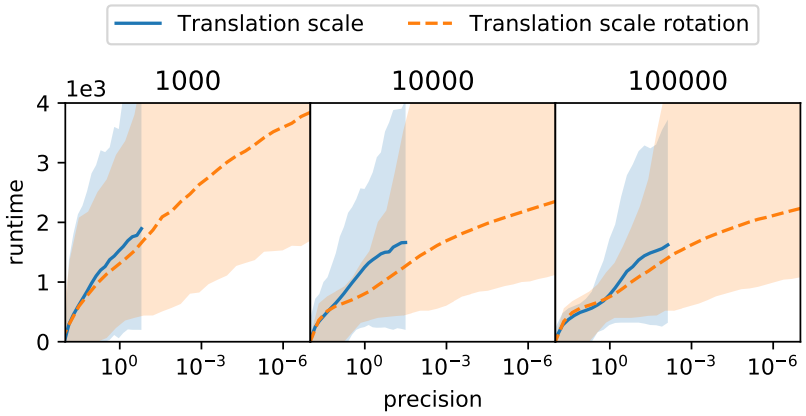
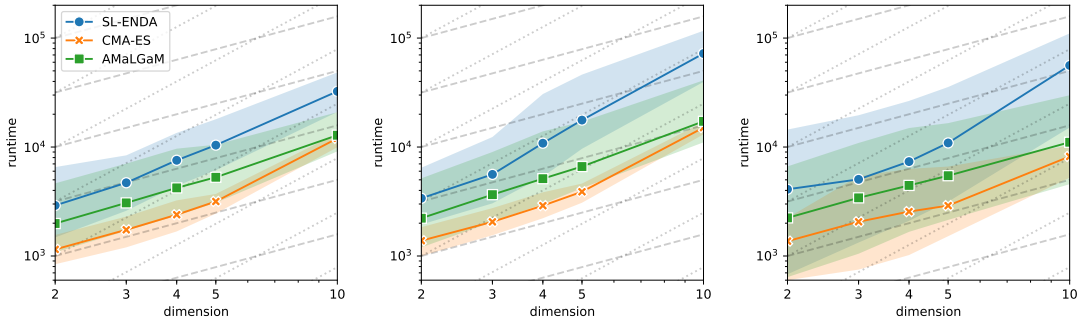


Figure 5.4: Comparison of runtime vs. precision of the TSI- and TRSI-space on the 2-dimensional Rosenbrock function class after learning for 10^3, 10^4 and 10^5 functions. The vertical axis indicates runtime (number of fitness function evaluations) until the precision on the horizontal axis is first 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 precision that was achieved for at least 10% of the sampled functions.



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

Figure 5.5: 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 (number of fitness function evaluations) 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.

5.4.4 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. [4] was followed as closely as possible. The only major deviation from the source 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[4] were used as source and could directly be implemented in the ENDA framework without alterations.

Figure 5.5 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

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

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 the sphere function 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.

6
Discussion and conclusions

6.1 Discussion

The results in this thesis 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. Though the comparison with AMaLGaM and CMA-ES shows that the resulting algorithm has performance and scalability that comes close (same order of magnitude) to the existing manually-engineered algorithms, 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. Additionally, it is important to thoroughly investigate the impact of parameters such as population size, reinforcement learning algorithm and neural network topology on the learning time and final performance of learned algorithms, to better understand the potential of this new technique.

One could argue that the RL-agent in SL-ENDA could simply learn to calculate a weighted average of the individuals in the current population, which would in principle be sufficient. For example, AMaLGaM does this by taking the average over selected solutions. Though the neural network is technically able to approximate such a weighted average, the results suggest that this does not happen since that would imply that SL-ENDA would share or transcend the performance of AMaLGaM, which it does not. The exact reason for this result is not known at time of writing, however the author suspects it is an effect of the precision based convergence criterion. In particular, when the algorithm achieves a predefined precision the episode is terminated, which means the agent stops receiving a reward. Since the agent tries to maximize the expected cumulative discounted reward, the agent will try to stretch the episode for as long as it keeps receiving positive reward. This could lead to suboptimal final policies that stretch the episode length by not improving the precision as much as technically possible at every step. If this is the case this could be prevented by awarding a reward, larger than the expected reward that is obtained by stretching the episode, when the convergence criterion is met. When using such a reward strategy, extra care should be taken that the agent cannot force premature convergence to obtain the large reward. Further research is necessary to support this theory.

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 and agent spaces on such paths. Additionally, both quantitative and qualitative analyses should be used to investigate 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 work shows that self-learning EAs can be designed and, given the observed ultimate performance even in the restricted setting of this first thesis 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 the key components are and the importance of their proper design (e.g. TSI-space vs. TRSI-space and differential reward vs. normalized 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 Euclidean distance of positive definite matrices and the "natural" distance measure of covariance matrices with respect to probability, as described in [31], 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 investigate the application of self-learning embeddings, such as for example a recurrent neural network embedding [14], to embed the optimization history for algorithms such as SL-ENDA. Such embeddings also open the door to population-size and dimension-agnostic algorithms. It is the belief of the author that the introduction of recurrent neural network modules in the policy neural network can significantly improve the performance of SL-ENDA, since it introduces a (short-term) memory to the system which further enables the agent to use information from earlier populations. This belief is strengthened by the successful use of RNN modules in [1, 5].

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 thesis 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. [26].

Alternatively, the need for an agent space due to the neural network policy can also be circumvented altogether by using a more straightforward parameterized policy. For example, a linear combination of the individuals in the current population where the weights are learned parameters. In addition to avoiding the need for an agent space, such a simple parameterization can assist in understanding what the algorithm "learns" and what the effects of different reward functions are, since it is easily inspectable due to the limited number and interpretability of parameters. Note however that by simplifying the parametrization, the subset of policies that the agent is able optimize over is shrunk, potentially decreasing the performance of the resulting algorithm.

As an alternative to SL-ENDA, we could parameterize the mean function and, using a black-box optimization method, maximize a performance measure, like the reward function, over a set of functions sampled from the problem 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 [22]. This approach was found to be highly parallelizable, but it needed at least three times 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 thesis, due to its natural fit, reinforcement learning was used as main machine learning paradigm. However, there are ways, unexplored in this thesis, to apply other paradigms to MBEAs. A good example is the indirect supervised learning approach introduced by Chen et al. [5]. 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 direct-search optimization 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.

6.2 Conclusions

In this thesis, we set out to develop a self-learning model-based evolutionary algorithm that can improve its ability to optimize different problems in a class of optimization problems based on previous

optimization attempts of problems in that class. A review of the three paradigms of machine learning: unsupervised learning, supervised learning and reinforcement learning, showed that reinforcement learning has the most theoretical potential to be used in a self-learning model-based evolutionary algorithm due to the natural duality between average population fitness, in EAs, and reward signal, in RL. Unsupervised learning was altogether found to be unsuitable due to its inability to directly learn a mapping. Supervised learning can theoretically be applied to an ENDA but is either only able to clone the behavior of an existing algorithm, or is dependent on the gradient of the objective function for its learning. We were successful in developing a proof-of-principle framework that uses reinforcement learning to learn to adapt the mean of an estimation of normal distribution algorithm to achieve efficient optimization. (Research Q1)

We have shown empirically that both the choice of agent space and the choice of reward function can have a significant impact on the performance of the algorithm. In case of the agent space, the results show that the addition of imposed invariances on the mean function can lead to significant decreases in reinforcement learning training time. The results for the reward function indicate that a proper choice of reward function can significantly increase the optimization convergence speed, as well as decrease the training time necessary to obtain a good performing estimation of normal distribution algorithm.

We furthermore empirically found that self-learning model-based evolutionary algorithms can yield algorithms that, on unimodal noiseless functions, have performance and scalability that comes close to that of existing, broadly-used and carefully manually-engineered algorithms, such as AMaLGaM and CMA-ES. (Research Q2)

Bibliography

[1] M. Andrychowicz, M. Denil, S. Gomez, M. W. Hoffman, D. Pfau, T. Schaul, B. Shillingford, and N. De Freitas. Learning to learn by gradient descent by gradient descent. *Advances in Neural Information Processing Systems*, pages 3981–3989, 2016.

[2] C. Audet and W. Hare. *Derivative-Free and Blackbox Optimization (Springer Series in Operations Research and Financial Engineering)*. Springer, 2017.

[3] P. A. N. Bosman. Design and application of iterated density-estimation evolutionary algorithms. *Ph.D. Dissertation, Utrecht University*, 2003.

[4] P. A. N. Bosman, J. Grahl, and D. Thierens. Benchmarking parameter-free amalgam on functions with and without noise. *Evolutionary Computation*, 21:445–469, 2013.

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

[6] R. Cheng, C. He, Y. Jin, and X. Yao. Model-based evolutionary algorithms: a short survey. *Complex & Intelligent Systems*, 4(4):283–292, aug 2018.

[7] D. Clevert, T. Unterthiner, and S. Hochreiter. Fast and accurate deep network learning by exponential linear units (elus). *CoRR*, abs/1511.07289, 2015.

[8] A. R. Conn, K. Scheinberg, and L. N. Vicente. *Introduction to Derivative-Free Optimization (MPS-SIAM Series on Optimization)*. Society for Industrial and Applied Mathematics, 2009.

[9] P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, Y. Wu, and P. Zhokhov. Openai baselines. `https://github.com/openai/baselines`, 2017.

[10] R. Durrett. *Probability: Theory and Examples (Cambridge Series in Statistical and Probabilistic Mathematics)*. Cambridge University Press, 2010.

[11] Z. Ghahramani. *Unsupervised Learning*, pages 72–112. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[12] N. Hansen. The cma evolution strategy: A tutorial. *arXiv preprint arXiv:1604.00772*, 2016.

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

[14] Y. Keneshloo, T. Shi, C. K. Reddy, and N. Ramakrishnan. Deep reinforcement learning for sequence to sequence models. *arXiv preprint arXiv:1805.09461*, 2018.

[15] J. Kober, J. A. Bagnell, and J. Peters. Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*, 32(11):1238–1274, 2013.

[16] K. Li and J. Malik. Learning to Optimize. *arXiv preprint arXiv:1606.01885*, Jun 2016.

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

[18] 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. Human-level control through deep reinforcement learning. *Nature*, 518:529–533, 2015.

[19] A. Y. Ng, D. Harada, and S. J. Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *Proceedings of the Sixteenth International Conference on Machine Learning*, ICML '99, pages 278–287, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[20] L. M. Rios and N. V. Sahinidis. Derivative-free optimization: a review of algorithms and comparison of software implementations. *Journal of Global Optimization*, 56(3):1247–1293, jul 2012.

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

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

[23] J. Schulman. *Optimizing expectations: From deep reinforcement learning to stochastic computation graphs*. PhD thesis, Ph.D. Dissertation, UC Berkeley, 2016.

[24] J. Schulman, S. Levine, P. Moritz, M. I. Jordan, and P. Abbeel. Trust Region Policy Optimization. *arXiv e-prints*, page arXiv:1502.05477, Feb 2015.

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

[26] 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. Mastering the game of Go without human knowledge. *Nature*, 550:354–359, 2017.

[27] S. Smit and A. Eiben. Comparing parameter tuning methods for evolutionary algorithms. In *2009 IEEE Congress on Evolutionary Computation*. IEEE, may 2009.

[28] J. Smith and A. E. Eiben. *Introduction to Evolutionary Computing (Natural Computing Series)*. Springer, 2013.

[29] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018.

[30] R. S. Sutton, D. McAllester, S. Singh, and Y. Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Proceedings of the 12th International Conference on Neural Information Processing Systems*, NIPS'99, pages 1057–1063, Cambridge, MA, USA, 1999. MIT Press.

[31] D. Wierstra, T. Schaul, T. Glasmachers, Y. Sun, J. Peters, and J. Schmidhuber. Natural evolution strategies. *Journal of Machine Learning Research*, 15:949–980, 2014.

Index

A
Artificial neural networks

Throughout this work the term "neural network" is used to denote, what is more formally known as, an artificial neural network. This appendix presents a short introduction into artificial neural networks loosely based on Chapter 6 of "Deep Learning" by Goodfellow et al.. Since we only use so-called feedforward networks in this thesis, this appendix will be limited to such networks and we will use the terms "neural network" and "feedforward network" interchangeably.

A neural network is a function $f : \mathbb{R}^{d_{\text{in}}} \to \mathbb{R}^{d_{\text{out}}}$, where $D \in \mathbb{N}$ is the number of parameters, $d_{\text{in}} \in \mathbb{N}$ is the domain dimensionality and $d_{\text{out}} \in \mathbb{N}$ is the dimensionality of the range. These functions are called networks because they are often represented by a composition of multiple parameterized functions, the order of composition is generally described by a directed graph. When this graph is acyclic we speak of a feedforward network. For example, we can have three parameterized functions,

$$i \in [3]: \qquad h^{(i)} : \mathbb{R}^{D^{(i)}} \times \mathbb{R}^{d^{(i)}} \to \mathbb{R}^{d^{(i+1)}} \qquad D^{(i)}, d^{(i)}, d^{(4)} \in \mathbb{N}, \tag{A.1}$$

connected in a chain such that

$$f : \mathbb{R}^{D^{(1)}} \times \mathbb{R}^{D^{(2)}} \times \mathbb{R}^{D^{(3)}} \times \mathbb{R}^{d^{(1)}} \to \mathbb{R}^{d^{(4)}}, \quad (\theta^{(1)}, \theta^{(2)}, \theta^{(3)}, x) \mapsto h^{(3)}(\theta^{(3)}, h^{(2)}(\theta^{(2)}, h^{(1)}(\theta^{(1)}, x))). \tag{A.2}$$

We call $h^{(1)}$ the first layer, $h^{(2)}$ the second layer, and so forth. The last layer ($h^{(3)}$ in this case) is also called the output layer and all non-output layers ($h^{(1)}$ and $h^{(2)}$ in this case) are often called hidden layers.

These functions are called neural networks because the idea to use a composition of relative functions to estimate a complex function is loosely inspired on neuroscience. In this interpretation each layer consists of multiple units, each outputting a scalar value, that act in parallel on the same data. Each unit then acts as a neuron that takes many inputs from the previous layer and returns a scalar value representing some information about the input. For example, in image recognition the each unit in the first layer might indicate whether there exists an edge in a particular orientation at a particular position in the input image. The units in the second layer can then, based on the edge information, recognize lines with particular positions and orientations. Based on these lines the third layer can then, for example, recognize which particular digit is written in the image. These intermediate representations of the input data are called features. Since the features resulting of each layer are based on the features of the preceding layer, features of later layers are able to describe more complex properties of the input[1].

In the image recognition example sketched above, the features are interpretable by human spectators. This has the advantage that we, as researchers, can interpret what the function "calculates". However, it severely limits the resulting function in the sense that it can never use relations present in the input that are unknown to humans. To avoid this limitation the individual layers are parameterized, allowing them to represent different features based on the chosen parameters. The resulting total function f can then be approached as a "normal" parameterized function, where the parameters of the function are the parameters of all layers combined. In particular, if the gradient of the individual layers is known we can repeatedly use the chain rule to calculate the analytical gradient of f with respect to any of its parameters, this process is often referred to as backpropagation.

[1] This idea is very intuitively visualized in the Tensorflow playground (`http://playground.tensorflow.org`)

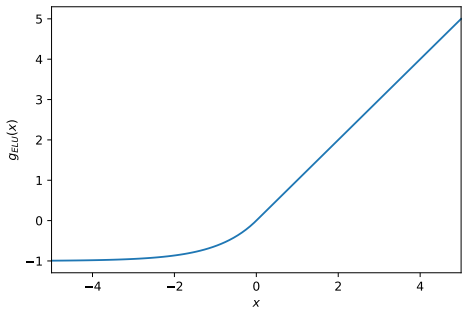


Figure A.1: Plot of ELU-function as defined in (A.6) for $x \in [-5, 5]$.

A.1 Layer types

Having explained the layered nature of a neural network and its interpretation, we are left with defining the layers themselves. We will do this by considering the limitations of linear layers and extending them to so-called fully-connected layers.

A.1.1 Linear layer

Given that a layer is a parameterized function,

$$h^{(i)} : \mathbb{R}^{D^{(i)}} \times \mathbb{R}^{d^{(i)}} \to \mathbb{R}^{d^{(i+1)}}, \tag{A.3}$$

it is natural to start with a linear function,

$$h^{(i)} : \left(\mathbb{R}^{d^{(i)} \times \mathrm{d}^{(i+1)}} \times \mathbb{R}^{d^{(i+1)}}\right) \times \mathbb{R}^{d^{(i)}} \to \mathbb{R}^{d^{(i+1)}}, \qquad ((A^{(i)}, b^{(i)}), x) \mapsto A^{(i)}x + b^{(i)}. \tag{A.4}$$

In this case every feature is a affine combination of the features of the previous resulting from the previous layer. Note however that the composition of two linear functions is again a linear function. Hence, using linear layers would result in a linear final function, which would totally nullify the layered nature of a neural network.

A.2 Fully-connected layer

In order to avoid the nullification of the layeredness that resulted from the linear layer, we have to introduce non-linearity in the layer. Most neural networks do this using an affine transformation, as with the linear layer, followed by a fixed nonlinear element-wise function $g^{(i)} : \mathbb{R}^{d^{(i)}} \to \mathbb{R}^{d^{(i)}}$, which is called an activation function. Concretely, the new layer, called a fully-connected layer, is defined as

$$h^{(i)} : \left(\mathbb{R}^{d^{(i)} \times \mathrm{d}^{(i+1)}} \times \mathbb{R}^{d^{(i+1)}}\right) \times \mathbb{R}^{d^{(i)}} \to \mathbb{R}^{d^{(i+1)}}, \qquad \left(\left(A^{(i)}, b^{(i)}\right), x\right) \mapsto g^{(i)}\left(A^{(i)}x + b^{(i)}\right). \tag{A.5}$$

As activation function we use the so-called Exponential Linear Units (ELUs) in this thesis. The ELU function is a scalar function defined as

$$g_{\mathrm{ELU}} : \mathbb{R} \to \mathbb{R}, \quad (x) \mapsto \begin{cases} x & \text{if } x > 0, \\ \alpha\left(\exp(x) - 1\right) & \text{if } x \leq 0 \end{cases}, \tag{A.6}$$

where $\alpha > 0$ is a hyper parameter which is normally chosen to be 1, in this case the ELU function is continuously differentiable [7]. The ELU-function is plotted in Figure A.1. To use the ELU-function as activation function, as in (A.5), it is applied element-wise to the result of the affine transformation.