Univerity of St Andrews

Postgraduate Thesis

Deep Reinforcement Learning for Population Size Control in CMA-ES

Author:
Jack Smith

Supervisor:
Nguyen Dang,
Diederik Vermetten,
Carola Doerr

August 15, 2023

Abstract

Covariance matrix adaptation evolution strategy (CMA-ES) is a state-of-the-art algorithm for black-box continuous optimisation. The population size is the most critical parameter for the performance of CMA-ES and requires careful tuning for optimal results. However, this can be prohibitively expensive in the black-box optimisation scenario. This project proposes a novel strategy to adapt the population size in a data-driven fashion using deep reinforcement learning. We start by benchmarking CMA-ES on a number of fixed population sizes and simple adaptive population size approaches, along with reimplementing the current leading strategy for population size adaptation, PSA-CMA-ES. We show that this novel reinforcement learning strategy can often produce performance comparable to PSA-CMA-ES on a selection of benchmark functions.

Declaration of Authorship

I declare that the material submitted for assessment is my own work except where credit is explicitly given to others by citation or acknowledgement. This work was performed during the current academic year except where otherwise stated. The main text of this project report is 13,634 words long, including project specification and plan. In submitting this project report to the University of St Andrews, I give permission for it to be made available for use in accordance with the regulations of the University Library. I also give permission for the title and abstract to be published and for copies of the report to be made and supplied at cost to any bona fide library or research worker, and to be made available on the World Wide Web. I retain the copyright in this work.

Contents

Chapter 1

Introduction

Optimisation is a common problem in mathematics, and also in the real world, where we want to find the minimum point in the range of a function. Black-box optimisation is a subset of this, where neither the function nor its gradients are explicitly known, so we can only gather information about the function by evaluating it at different points. This rules out commonly known optimisation approaches such as gradient descent, and we must therefore use different approaches. Black-box functions are often very expensive to evaluate and thus reducing the number of total function evaluations over the course of an optimisation procedure is key for minimising time and computational expense.

CMA-ES is a numerical optimisation algorithm which belongs to the class of evolutionary algorithms, which means it is broadly based on the concept of biological evolution. It is a stochastic, gradient-free method and only relies on function evaluations so can be effectively utilised in black-box optimisation scenarios. One significant benefit of CMA-ES compared to other evolutionary algorithms is that most of the parameters of the algorithm depend only on the dimensionality of the problem, and require little tuning by hand. However, one parameter which can benefit greatly from tuning is the population size. Well-crafted population size schemes can reduce the total number of function evaluations as well as improve the quality of the solution. Often, tuning this parameter requires significant knowledge of the algorithm itself, as well as the problem domain, and so can be very expensive in time and computation.

Hyperparameter optimisation is a powerful approach to achieving good results for any algorithm featuring parameters affecting its performance. There are many approaches to statically optimise these parameters but often the need for these parameters to vary over the algorithm's runtime is ignored. Dynamic algorithm configuration is a field concerning the adaption of an algorithm's strategy parameters over multiple time steps. This can often be very beneficial to the performance of an algorithm.

The aim of this project is to consider population size control in CMA-ES as a dynamic algorithm configuration problem. The main contribution of this work will be to implement a novel method to control the population size using deep reinforcement learning. This is a technique within machine learning which allows an AI-based system to learn from trial and error based on feedback from its actions.

We will first benchmark traditional constant population size approaches to see how fixed sizes perform. We will then implement simple variable population size schemes to see how the performance differs. These are comparable to traditional hyperparameter optimisation techniques as these schemes perform the same actions regardless of the problem at hand.

Then we will re-implement the current state-of-the-art dynamic algorithm configuration method for population size control, PSA-CMA-ES, which adapts the population size using the internal state of CMA-ES. Here, the population size will be adapted in a way specific to the problem being solved.

Finally, we will consider population size control in the context of a reinforcement learning problem. While reinforcement learning is a well-studied area with a wide range of applications, it has never been used for population size control in CMA-ES. We will use the internal state of the algorithm to train a reinforcement learning agent to be able to adapt the population size in a way comparable to, or improve upon PSA-CMA-ES.

Chapter 2

Background

In this chapter, we will cover the theoretical foundations and key concepts central to this project, which are necessary to understand the methodology and results.

We will start by exploring the inner workings of CMA-ES, the optimisation algorithm of primary interest. Next, we will provide background information on this research's benchmarking and evaluation environment, IOHprofiler. This environment plays a critical role in assessing the effect on performance from the population size within CMA-ES on a standardised set of benchmark functions. Additionally, we will introduce the dynamic algorithm configuration method employed in this project: reinforcement learning. We will cover the mathematical foundations of this, along with introducing the algorithm used.

Overall, this chapter aims to equip the reader with a comprehensive understanding of the fundamental theories underpinning the CMA-ES optimisation algorithm, the evaluation framework, and reinforcement learning.

2.1 CMA-ES

Covariance matrix adaptation evolution strategy (CMA-ES) is a stochastic, gradient-free method for black-box function optimisation. It is particularly good on weakly structured, multimodal, or high-dimensional functions, and belongs to the class of evolutionary algorithms [11].

One important feature is that the default settings are based only on the dimensionality of the search space [24]. This means that most parameters require little to no tuning for reasonable results. However, population size can have a large impact on performance, so adapting this parameter depending on the problem of interest is crucial [1, 20, 24].

The general strategy for the algorithm starts by sampling candidate solutions from a multivariate normal distribution. Their fitness is evaluated according to an objective function f, and using this, some of these individuals are selected to become parents for the next generation. These values are used to update the distribution's mean vector and covariance matrix. This procedure repeats until specified termination criteria are met. The steps for CMA-ES are outlined below [11].

1. A random sample of candidate solutions is generated from a multivariate normal distribution. These are ranked on the quality of their objective function values.
2. A subset of these candidate solutions are recombined by finding their weighted average. This value is used as the new mean of the sampling distribution.

3. The evolution path $\mathbf{p}_c$ is updated. This is used to update the covariance matrix of the sampling distribution.
4. Likewise, the conjugate evolution path $\mathbf{p}_c$ is updated to adapt the step size.
5. This procedure is repeated until certain stopping criteria are met. These can consist of the quality of the solution, properties of the covariance matrix, or the total number of function evaluations.

2.1.1 Algorithm Details

We will now go into more detail about each step within CMA-ES as described above.

Sampling

The basic equation for sampling the search points for generation t is as follows:

$$\boldsymbol{x}_k^{(t+1)} \sim \boldsymbol{m}^{(t)} + \sigma^{(t)} \mathcal{N}\left(\mathbf{0}, \boldsymbol{C}^{(t)}\right), \tag{2.1}$$

for $k = 1, .., \lambda$.

- $\mathcal{N}\left(\mathbf{0}, \boldsymbol{C}^{(t)}\right)$ is a multivariate normal distribution with zero mean and covariance matrix $\boldsymbol{C}^{(t)}$.
- $\boldsymbol{x}_k^{(t+1)} \in \mathbb{R}^d$ is the k-th candidate solution in generation $t+1$.
- $\boldsymbol{m}^{(t)} \in \mathbb{R}^d$ is the mean value of the distribution at generation t.
- $\sigma^{(t)} \in \mathbb{R}_+$ is the standard deviation (which dictates the step size) at generation t.
- $\boldsymbol{C}^{(t)} \in \mathbb{R}^{n^2}$ is the covariance matrix at generation t.
- $\lambda \in \mathbb{N}$ is the population size. This is the number of candidate solutions sampled at each generation. The choice of λ can greatly impact performance.

To define an iteration step we must calculate $\boldsymbol{m}^{(t+1)}$, $\boldsymbol{C}^{(t+1)}$, and $\sigma^{(t+1)}$, to update the distribution for generation $t+1$.

Updating the mean

The new mean is calculated as a weighted average of $\mu \leq \lambda$ selected points from the candidate solutions of the previous generation $\boldsymbol{x}_1^{(t+1)}, ..., \boldsymbol{x}_\lambda^{(t+1)}$. The parameter μ is known as the parent population size, which is the number of candidate solutions used to recalculate the mean and covariance of the distribution. We usually set $\mu = \lfloor \lambda/2 \rfloor$. This is why we set a lower bound for λ at 4, so we have more than one sample to recombine.

We must first ensure that the candidate solutions are ranked such that

$$f(\boldsymbol{x}_{1:\lambda}^{(t+1)}) \leq ... \leq f(\boldsymbol{x}_{\lambda:\lambda}^{(t+1)}), \tag{2.2}$$

where f is the objective function to be minimised. Thus, we can see the new mean is calculated from the best μ candidate solutions. Then, the formula for recombination is as follows:

$$\boldsymbol{m}^{(t+1)} = \sum_{i=1}^{\mu} w_i \boldsymbol{x}_{i:\lambda}^{(t+1)}. \tag{2.3}$$

Here, w_i are the recombination weights, which give the weightings of each candidate solution in the new mean. There are several schemes to select these recombination weights, as long as they satisfy

$$\sum_{i=1}^{\mu} w_i = 1, \tag{2.4}$$

$$w_1 \geq ... \geq w_\mu > 0. \tag{2.5}$$

We can also calculate an effective sample size from the weights. This is given by

$$\mu_{eff} = \left(\frac{||\mathbf{w}||_1}{||\mathbf{w}||_2}\right) = \frac{1}{\sum_{i=1}^{\mu} w_i^2}. \tag{2.6}$$

Using this, we can rewrite equation (2.3) as

$$\mathbf{m}^{(t+1)} = \mathbf{m}^{(t)} + c_m \sum_{i=1}^{\mu} w_i (\mathbf{x}_{i:\lambda}^{(t+1)} - \mathbf{m}^{(t)}), \tag{2.7}$$

where c_m is a learning rate, usually set to 1.

Updating the covariance matrix

The most straightforward way to estimate the new covariance matrix is to calculate the updated empirical covariance matrix [10]. We calculate this with

$$\boldsymbol{C}_{\text{emp}}^{(t+1)} = \frac{1}{\lambda - 1} \sum_{i=1}^{\lambda} \left(\boldsymbol{x}_i^{(t+1)} - \frac{1}{\lambda}\sum_{j=1}^{\lambda}\right)\left(\boldsymbol{x}_i^{(t+1)} - \frac{1}{\lambda}\sum_{j=1}^{\lambda}\right)^T. \tag{2.8}$$

Assuming the previously generated candidate solutions are random variables, we have

$$\mathbb{E}[\boldsymbol{C}_{\text{emp}}^{(t+1)} | \boldsymbol{C}^{(t)}] = \boldsymbol{C}^{(t)} \tag{2.9}$$

Typically, $\mu_{eff} \approx \lambda/4$ is small and therefore it is not possible to get a reliable estimator for the covariance matrix using the method described above. Thus, information from previous generations has to be used additionally. For example, after a sufficient number of generations, the weighted mean of the estimated covariance matrices from all generations,

$$\mathbf{C}^{t+1} = \frac{1}{t+1} \sum_{i=0}^{t} \frac{1}{\sigma^{(i)^2}} \mathbf{C}_\mu^{(i+1)}, \tag{2.10}$$

becomes a reliable estimator. To assign recent generations greater importance in the calculation, we use exponential smoothing. Choosing $C^{(0)} = \mathbf{I}$ and a learning rate $0 < c_\mu < 1$, we get

$$\begin{aligned} \mathbf{C}^{(t+1)} &= (1 - c_\mu)\mathbf{C}^{(t)} + c_\mu \frac{1}{\sigma^{(t)^2}} \mathbf{C}_\mu^{(t+1)} \\ &= (1 - c_\mu)\mathbf{C}^{(t)} + c_\mu \sum_{i=1}^{\mu} w_i \mathbf{y}_{i:\lambda}^{(t+1)} \mathbf{y}_{i:\lambda}^{(t+1)^\intercal} \end{aligned} \tag{2.11}$$

Here, $\mathbf{y}_{i:\lambda}^{(t+1)} = (\mathbf{x}_{i:\lambda}^{(t+1)} - \mathbf{m}^{(t)})/\sigma^{(t)}$. We also choose $c_\mu = \min(1, \mu_{eff}/4)$.

This scheme is called rank-μ update because the sum of the outer product is of rank $\min(\mu, n)$. We can generalise this to λ weight values which neither need to sum to 1, nor be non-negative.

$$\begin{aligned} \mathbf{C}^{(t+1)} &= (1 - c_\mu \sum_i w_i)\mathbf{C}^{(t+1)} + c_\mu \sum_{i=1}^{\lambda} w_i \mathbf{y}_{i:\lambda}^{(t+1)} \mathbf{y}_{i:\lambda}^{(t+1)^\intercal} \\ &= \mathbf{C}^{(t)^{1/2}} \left(\mathbf{I} + c_\mu \sum_{i=1}^{\lambda} w_i \left(\mathbf{z}_{i:\lambda}^{(t+1)} \mathbf{z}_{i:\lambda}^{(t+1)^\intercal} - \mathbf{I} \right) \right) \mathbf{C}^{(t)^{1/2}}. \end{aligned} \tag{2.12}$$

Here $\mathbf{z}_{i:\lambda}^{(t+1)} = \mathbf{C}^{(t)^{-1/2}} \mathbf{y}_{i:\lambda}^{(t+1)}$. This is the mutation vector expressed in the coordinate system where the sampling is isotropic and the respective coordinate system transformation does not rotate the original principal axes of the distribution.

The above equation for the covariance matrix expresses the update in the natural coordinate system. This uses λ weights, about half of which are negative. If the weights are chosen such that the sum of the weights is zero, the changes are only made along the axis in which the samples are realised.

The value $1/c_\mu$ is the backwards time horizon, which contributes roughly 63% of the overall information. The choice of this value is crucial: too small and the learning will be very slow, and a value which is too large will lead to failure because the covariance matrix becomes degenerate. A good choice for this is independent of the function we are trying to optimise, and can be reasonably approximated to first-order as $c_\mu \approx \mu_{eff}/n^2$, meaning the characteristic time horizon will be given by n^2/μ_{eff}.

In addition to this, we also perform the rank-one update for the covariance matrix. Here we repeatedly update the covariance matrix in the generation sequence using a single selected step only.

$$\mathbf{C}^{(t+1)} = (1 - c_1)\mathbf{C}^{(t)} + c_1 \mathbf{y}^{(t+1)} \mathbf{y}^{(t+1)^\intercal}. \tag{2.13}$$

We define the evolution path $\mathbf{p}_c \in \mathbb{R}^n$, which is the sequence of successive steps taken over a number of generation, as

$$\mathbf{p}_c^{(t+1)} = (1 - c_c)\mathbf{p}_c^{(t)} + \sqrt{c_c(2 - c_c)\mu_{eff}} \frac{\mathbf{m}^{(t+1)} - \mathbf{m}^{(t)}}{c_m \sigma^{(t)}}. \tag{2.14}$$

The factor $\sqrt{c_c(2-c_c)\mu_{eff}}$ is chosen such that

$$\mathbf{p}_c^{(t+1)} \sim \mathcal{N}(\mathbf{0}, \mathbf{C}). \tag{2.15}$$

The rank-one update of the covariance matrix via the evolution path is therefore

$$\mathbf{C}^{(t+1)} = (1-c_1)\mathbf{C}^{(t)} + c_1 \mathbf{p}_c^{(t+1)} \mathbf{p}_c^{(g+1)^\intercal}. \tag{2.16}$$

We can combine these two update schemes into a full procedure for updating the covariance matrix.

$$\mathbf{C}^{(t+1)} = (1 - c_1 - c_\mu \sum_i w_i)\mathbf{C}^{(t)} + c_1 \mathbf{p}_c^{(t+1)} \mathbf{p}_c^{(t+1)^\intercal} + c_\mu \sum_{i=1}^{\lambda} w_i \mathbf{y}_{i:\lambda}^{(t+1)} \left(\mathbf{y}_{i:\lambda}^{(g+1)}\right)^\intercal. \tag{2.17}$$

Updating the step size

Similarly to equation (2.14), we construct a conjugate evolution path $\mathbf{p}_\sigma \in \mathbb{R}^n$. This is calculated as follows:

$$\mathbf{p}_\sigma^{(t+1)} = (1-c_\sigma)\mathbf{p}_\sigma^{(t)} + \sqrt{c_\sigma(2-c_\sigma)\mu_{eff}}\mathbf{C}^{(t)^{-1/2}} \frac{\mathbf{m}^{(t+1)} - \mathbf{m}^{(t)}}{c_m \sigma^{(t)}}. \tag{2.18}$$

As before, c_σ is a backward time horizon of the evolution path. From this, we can deduce that

$$\sigma^{(t+1)} = \sigma^{(t)} \exp\left(\frac{c_\sigma}{d_\sigma}\left(\frac{||\mathbf{p}_\sigma^{(t+1)}||}{\mathbb{E}||\mathcal{N}(\mathbf{0}, \mathbf{I})||} - 1\right)\right). \tag{2.19}$$

We generally use the approximation

$$\mathbb{E}||\mathcal{N}(\mathbf{0}, \mathbf{I})|| \approx \frac{\sqrt{d}}{2}\left(1 - \frac{1}{4d} + \frac{1}{21d^2}\right). \tag{2.20}$$

We call this value χ_n. This step-size adaptation method is called CSA [11].

2.1.2 Population Size

The budget of a CMA-ES run is the total number of function evaluations, and the used budget is considered the metric for evaluating the runtime of the algorithm. The population size determines how this budget is 'spent' - a large population size will use more function evaluations and result in fewer generations, and vice versa for small population sizes. The population size is the most critical parameter for CMA-ES [1, 24, 20]. We generally choose the default value to be $\lambda = 4 + \lfloor 3\log(d) \rfloor$ [11].

The benefit of a large population size is that it will be able to explore more of the function landscape as more function evaluations will be made in each generation. Likewise, small populations will tend to sample in a smaller region but will be able to exploit this region more efficiently. Thus, tuning the population is a trade-off between exploration and exploitation.

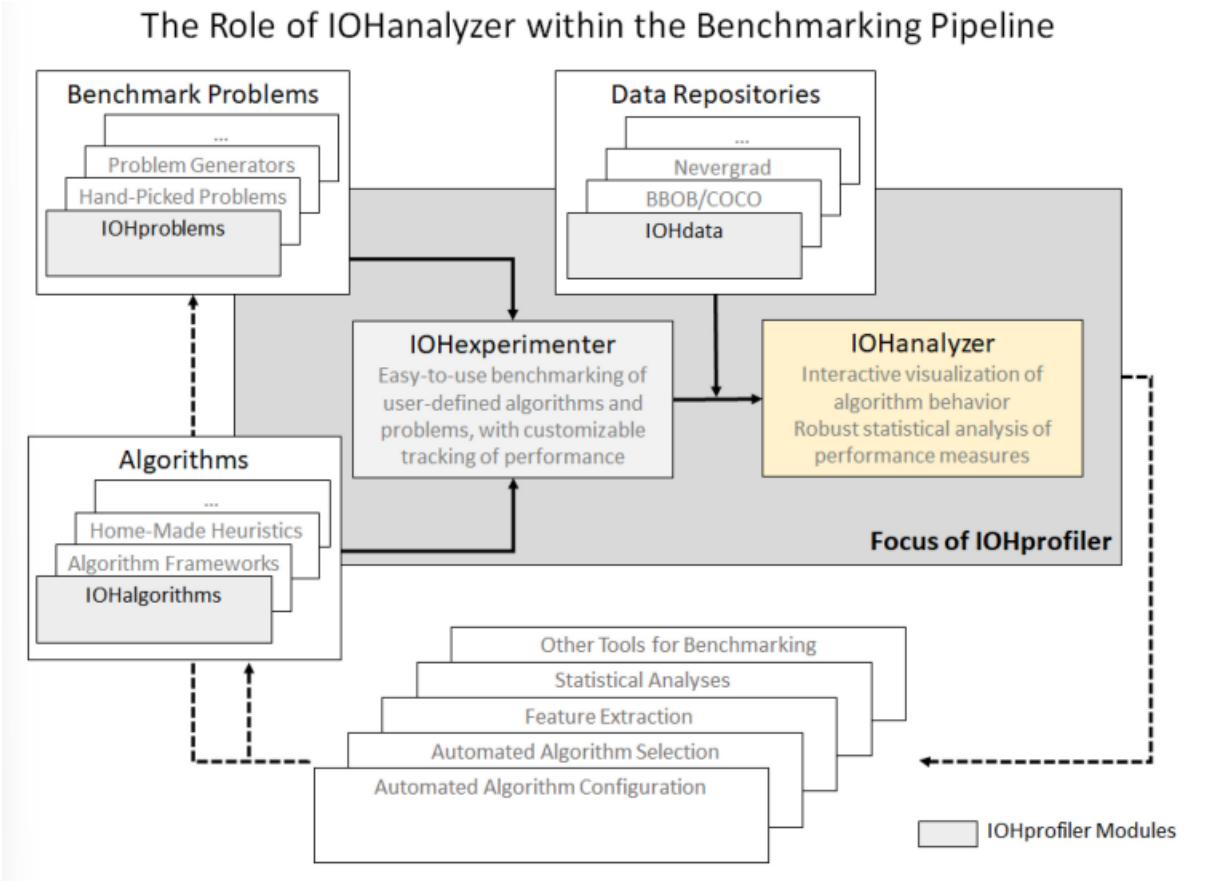


FIGURE 2.1: Diagram showing the roles of IOHprofiler and its different modules within the benchmarking pipeline.

Large population sizes can be expecially beneficial when the function is multimodal, weakly structured, or noisy, as they tend not to get stuck in local minima [26, 24]. However, keeping the population size large may lead to the distribution converging to a non-optimal point as it fails to exploit promising areas of the search space or the budget is used up too quickly, stopping detailed searches from occurring.

This trade-off motivates the necessity for adapting the population size during the course of the optimisation procedure. Large population sizes may be needed at the start, but they may benefit from being reduced when all candidate solutions are sampled from a small area.

This trade-off can, to a certain degree, be mitigated by choosing a sensible middle ground for the population size, but we will show significant performance gains can be realised by adapting this parameter on-the-fly.

2.2 IOHprofiler

IOHprofiler is a benchmarking environment for evaluating iterative optimisation heuristics (IOH). These are algorithms which are entirely sampling-based and so lend themselves well to black-box problems. This environment integrates elements from the entire benchmarking pipeline [6]. There are two primary modules to this environment.

2.2.1 IOHexperimenter

To reliably and reproducibly compare and benchmark different approaches we need a standardised testing framework to implement and perform these experiments.

IOHexperimenter is an interface between optimisation problems and solvers while allowing for robust and flexible logging of the optimisation process [25]. A selection of problem suites such as PBO [7], W-Model [34], and BBOB problems [13] are provided.

It is often useful to test algorithms on different versions of the same problem. This can be used to test whether an algorithm is invariant to certain transformations of the problem at hand. Thus, each problem in the portfolio can be transformed continuously to specify a problem instance, defined as $F = T_y \circ f \circ T_x$, where $f : X \to \mathbb{R}$ is a benchmark problem. T_x and T_y are automorphisms on the domain and range of f representing transformations of the function such as translations or rotations. To generate a problem instance, a tuple (f, i, n) must be specified, where $i \in \mathbb{N}_{>0}$ is the problem instance, and n is the dimensionality of the problem [25].

IOHexperimenter also provides logging functionality. Loggers can be coupled to a problem to store relevant information. Triggers can be used to define when the logger records data. These can be events such as every specified number of function evaluations, or a decrease in objective function value. The logger saves the data in a format able to be read by IOHanalyzer (see 2.2.2).

2.2.2 IOHanalyzer

Benchmarking and performance analysis play a crucial role in understanding the behaviour of any optimisation algorithm. This generally will involve manual setup, execution and analysis which can be time-consuming and error-prone. IOHanalyzer is a generic tool for the analysis, comparison and visualisation of iterative optimisation heuristics [33].

As input, data generated from IOHexperimenter, or preloaded data sets, can be used. It provides two primary perspectives for analysing this data. Fixed-target running time focuses on the distribution of the number of function evaluations at various target precisions. The fixed-budget analysis looks at the best function value at various budget target values.

IOHanalyzer provides functionality to generate graphs and perform statistical tests to thoroughly analyze the performance of different algorithms over a benchmark function portfolio.

2.3 Reinforcement Learning

Reinforcement learning (RL) is the study of agents and how they learn through trial and error. It is based on the idea of punishing or rewarding an agent for its behaviour making it more likely to perform beneficial actions in the future.

The way the agent behaves is dictated by its policy. In deep reinforcement learning, this policy is a deep neural network allowing the agent to comprehend complex environments, similar to the way deep neural networks in supervised learning can perform more complex tasks than linear regression.

The structure and content of this section are based on Spinning Up in Deep RL from OpenAI [27].

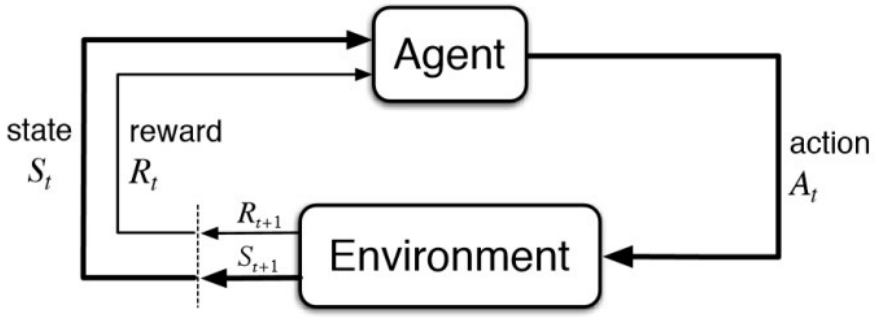


FIGURE 2.2: The interaction between an agent and its environment [2].

2.3.1 Key Concepts

The key players of reinforcement learning are the agent and the environment. The environment is the world with which the agent interacts. At each timestep, the agent gets an observation from the environment, which doesn't necessarily contain all the environment's information.

The agent then chooses an action to take which will in turn modify the environment. Furthermore, the agent also receives a reward, which signifies how good or bad the action was. The agent aims to maximise its cumulative reward, and we aim to teach the agent to do this in a data-driven fashion.

A key assumption for reinforcement learning is that it is a Markov decision process. This means that the future only depends on the current state of the system; it has no knowledge of the past [32]. In the language of reinforcement learning, this says that the next action only depends upon the current state of the environment, not any other past states or actions. These terms will be explained in more detail below.

States, Observations & Actions

A state s is a complete description of the state of the world. An observation o is a partial observation of a state which may omit some information. When the agent has access to the whole environment, we say the environment is fully observed. Likewise, if the agent only has a partial worldview, we say it is partially observed.

The set of all valid actions for a given environment is called the action space. This can be discrete or continuous. The distinction is very important - many algorithms can only work with one kind of action space. For example, TD3 can only work with continuous action spaces. while Deep Q-networks will only work with discrete action spaces. On the other hand, A2C will work with both [28].

A simple diagram showing the interaction between agent and environment is shown in Fig. 2.2.

Policies

A policy is a rule used by an agent to decide what action to take at each time step based on the current observation. If it is deterministic, it is denoted by

$$a_t = \mu(s_t). \tag{2.21}$$

If stochastic, we denote it to be

$$a_t \sim \pi(\cdot|s_t), \tag{2.22}$$

where π is a distribution of all the possible actions which the agent can make. The terms policy and agent will often be used interchangeably, as the policy is essentially the decision maker for the agent.

In deep reinforcement learning we deal with parameterised policies, where the outputs are computable functions that depend on a set of parameters (the weights and biases of a neural network) which we can adjust through optimisation algorithms such as gradient descent, to change the behaviour.

We denote the policies by θ or ϕ and write this as a subscript on the policy symbol.

There are two main kinds of stochastic policies: categorical policies can be used in discrete action spaces, and Gaussian policies can be used in continuous action spaces.

Trajectories

A trajectory τ is a sequence of states and actions.

$$\tau = (s_0, a_0, s_1, a_1, ...) \tag{2.23}$$

The very first state of this world, s_0 is randomly sampled from the start state distribution. It is often denoted $\rho(\cdot)$. State transitions are governed by the laws of the environment, and depend only on the most recent action. This is due to the fact that we treat reinforcement learning as a Markov decision process. They can be deterministic, which is denoted as

$$s_{t+1} = f(s_t, a_t). \tag{2.24}$$

Otherwise, they are stochastic, denoted by

$$s_{t+1} \sim P(\cdot|s_t, a_t). \tag{2.25}$$

An agent acts depending on decisions chosen by its policy. When a terminal state has been reached, the trajectory is complete. We call this an episode (this is similar to the notion of an epoch in supervised learning).

Reward Functions

The reward function is a critically important part of RL. It depends on the current state of the world, the action just taken, and the next state of the world. The general form of a reward function is

$$r_t = R(s_t, a_t, s_{t+1}). \tag{2.26}$$

The goal of the agent is to maximise some cumulative reward over a trajectory. Reward functions have two primary characteristics: discount and horizon.

Discount is the weight placed on each reward depending on how far in the past it was received by the agent. Discounted rewards place less weight on rewards that happened further in the past. Reward functions can be discounted or undiscounted.

Horizon is the window of steps in the past from which the reward is calculated. Finite-horizon rewards only consider the reward from a limited number of previous steps, while infinite-horizon rewards consider all rewards since the start of the episode.

The finite-horizon undiscounted reward is shown in eq. 2.27, and the infinite-horizon discounted reward is shown in eq 2.28. These are the two most common general forms for the reward function. Here, $\gamma \in (0, 1)$ is called the discount factor and dictates how quickly the impact of each time step decreases over time. It also ensures that the infinite sum will always converge.

$$R(\tau) = \sum_{t=0}^{T} r_t. \tag{2.27}$$

$$R(\tau) = \sum_{t=0}^{\infty} \gamma^t r_t \tag{2.28}$$

The RL Problem

The problem, in essence, is to select a policy which maximises the expected reward when the agent acts according to it. To talk about expected returns, we have to define a probability distribution over a trajectory. Suppose the environment transitions and the policy are stochastic.

Then the probability of a trajectory of length T with policy π is

$$P(\tau|\pi) = \rho_0(s_0) \prod_{t=0}^{T-1} P(s_{t+1}|s_t, a_t)\pi(a_t|s_t). \tag{2.29}$$

We can therefore calculate the expected return by integrating this distribution over the trajectory with the reward function.

$$J(\pi) = \int_{\tau} P(\tau|\pi)R(\tau) = \mathop{\mathbb{E}}_{s_0 \sim p(s_0)}[R(\tau)]. \tag{2.30}$$

The problem can then be easily expressed as

$$\pi^* = \arg\max_{\pi} J(\pi), \tag{2.31}$$

so π^* is the policy with the highest expected return. It is often useful to know the value of a state-action pair, which is the expected return from starting in the given state and acting according to the policy forever. There are four main functions expressing this idea.

- On-policy value function: expected return from starting in state s and acting according to π until a terminal state is reached.

$$V^{\pi}(s) = \mathop{\mathbb{E}}_{\tau \sim \pi}[R(\tau)|s_0 = s] \tag{2.32}$$

- On-policy action-value function: similar to the previous function but the first action a_0 is specified.

$$Q^{\pi}(s) = \mathop{\mathbb{E}}_{\tau \sim \pi}[R(\tau)|s_0 = s, a_0 = a] \tag{2.33}$$

- Optimal value function: expected return from starting in state s and always acting according to the optimal policy in the environment.

$$V^{*}(s) = \max_{\pi} \mathop{\mathbb{E}}_{\tau \sim \pi}[R(\tau)|s_0 = s] \tag{2.34}$$

- Optimal action-value function: similar to the previous function but the first action a_0 is specified.

$$Q^{*}(s) = \max_{\pi} \mathop{\mathbb{E}}_{\tau \sim \pi}[R(\tau)|s_0 = s, a_0 = a] \tag{2.35}$$

There is an important connection between the optimal action-value function and the action selected by the optimal policy. The optimal policy will select the action a which maximises the expected return starting from state s. As a result, if we have Q^* we can directly obtain the optimal action via

$$a^{*}(s) = \arg\max_{a} Q^{*}(s, a) \tag{2.36}$$

The Bellman Equations

All of the four equations above obey self-consistency relations called the Bellman equations. These encode the intuition that the value of the starting point is the reward you expect from being there, plus the value of where you land next. The equations are shown in eq. 2.37 and eq. 2.38.

$$V^{\pi}(s) = \mathop{\mathbb{E}}_{a \sim \pi}[r(s, a, s') + \gamma V^{\pi}(s')]. \tag{2.37}$$

$$Q^{\pi}(s, a) = \mathop{\mathbb{E}}_{s' \sim P(s,a)}\left[r(s, a, s') + \gamma \mathop{\mathbb{E}}_{a' \sim \pi}[Q^{\pi}(s', a')]\right]. \tag{2.38}$$

Sometimes it is easier to describe how good an action is relative to the average rather than in an absolute sense. We can encode this with the advantage function, which describes how much better it is to take action a in state s compared to taking a random action according to policy π.

It is given by

$$A^{\pi}(s, a) = Q^{\pi}(s, a) - V^{\pi}(s). \tag{2.39}$$

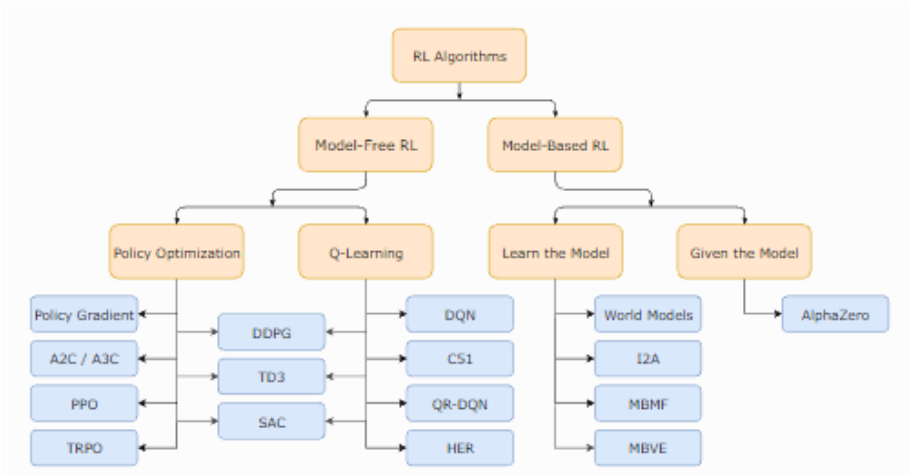


FIGURE 2.3: A taxonomy of a selection of modern reinforcement learning algorithms [27].

2.3.2 Model-Based vs Model-Free Reinforcement Learning

An important branching point for RL algorithms is whether the agent has access to a model of the environment. A model is a function which can predict state transitions and rewards.

The main upside to having a model is it allows the agent to plan by thinking ahead and evaluating a range of choices. These results can then be distilled into a learned policy. However, a ground truth model of the environment is not usually available to the agent. If an agent wants to use a model, it has to learn the model purely from experiencing its environment. This poses a number of challenges, primarily that bias in the model can be exploited by the agent, resulting in an agent which performs well with respect to the learned model, but terribly in the real world.

The alternative to model-based RL is model-free approaches. These tend to be easier to implement and tune so have therefore become more popular. There are two main approaches to representing and training agents with model-free RL:

Policy optimisation - methods in this family represent a policy explicitly as $\pi_\theta(a|s)$. These methods optimise θ either directly by gradient ascent on the performance objective J, or indirectly by maximising local approximations of J. This is usually performed on-policy, which means each update only uses data collected while acting according to the most recent version of the policy. Policy optimisation also usually involves learning $V_\phi(s)$ for the on-policy value function.

Q-learning - here we learn an approximator $Q_\theta(s,a)$ for the optimal action-value function Q^*. They use an objective function based on the Bellman equation. This optimisation is usually performed off-policy, which means the policy and value networks' updates can be done on data that was not produced by the newest policies, regardless of how the agent was chosen to explore the environment when the data was obtained. The actions taken by the Q-learning agent are given by

$$a(s) = \arg\max_a Q_\theta(s,a). \tag{2.40}$$

Policy optimisation methods are principled, meaning they directly optimise for the quantity in question. This tends to make them stable and reliable. On the other hand, Q-learning

indirectly optimises agent performance, so there are many more failure modes. However, it has the advantage of being substantially more sample efficient when they do work, because they can reuse data more effectively.

2.3.3 Policy Optimisation

Here we will derive the simplest policy gradient. Consider the case of a stochastic, parameterised policy π_θ. We want to maximise the expected return $J(\pi_\theta)$ with the finite-horizon undiscounted return. To optimise the policy we will use gradient ascent.

$$\theta_{k+1} = \theta_k + \alpha \nabla_\theta J(\pi_\theta)|_{\theta_k}. \tag{2.41}$$

We need an expression for the actual policy gradient for numerical computation. To do this, we need to analytically derive the gradient of policy performance, and form a sample estimate which can be computed from a finite number of steps.

First, we can rewrite the gradient of the trajectory as

$$P(\tau|\theta) = P(\tau|\theta) \nabla_\theta \log P(\tau|\theta). \tag{2.42}$$

Additionally, we can calculate the log-probability of a trajectory to be

$$\log P(\tau|\theta) = \log \rho_0(s_0) + \sum_{t=0}^{T} \left(\log P(s_{t+1}|s_t, a_t) + \log \pi_\theta(a_t|s_t) \right). \tag{2.43}$$

The gradients of ρ_0, $P(s_{t+1}|s_t, a_t)$ and $R(\tau)$ have no dependence on θ so are all zero. Thus we can combine all these facts to get a formula for the gradient of the log-probability (grad-log-prob) of a trajectory.

$$\nabla_\theta \log P(\tau|\theta) = \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t). \tag{2.44}$$

Finally, we can use these facts to derive an analytical expression for the policy gradient.

$$\nabla_\theta J(\pi_\theta) = \mathop{\mathbb{E}}_{\tau \sim \pi_\theta} [\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) R(\tau)]. \tag{2.45}$$

This expectation lets us calculate a numerical value of the policy gradient. By letting the agent act according to policy π, we can collect a set of trajectories, $\mathcal{D} = \{\tau_i\}_{i=1,\ldots,N}$ and calculate a sample mean. Thus, the policy gradient can be estimated using

$$\hat{g} = \frac{1}{|\mathcal{D}|} \sum_{\tau \in \mathcal{D}} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) R(\tau). \tag{2.46}$$

Loss Functions

We also need a loss function. This is not a loss function in the sense of supervised learning. The gradient of this loss is equal to the policy gradient when a set of (state, action, weight) is plugged in which has been collected according to the current policy. This is because the data distribution depends on the parameters. A loss function is usually defined on a fixed data distribution which is independent of the parameters we are trying to optimise.

Furthermore, it doesn't measure performance. A loss function usually evaluates the performance metric we care about, but in our case in RL, the loss function doesn't approximate this at all. The loss function with data generated by the current parameters is the negative gradient of performance. After that first step of gradient descent, there is no more connection to performance. Thus, minimising this loss function has no guarantee of improving the expected return.

A useful intermediate result which will be used repeatedly throughout policy optimisation is the expected Grad-Log-Prob (EGLP) lemma. Suppose P_θ is a parameterised probability distribution over a random variable x. Then,

$$\mathop{\mathbb{E}}_{x\sim P_\theta}[\nabla_\theta \log P_\theta(x)] = 0 \tag{2.47}$$

Reward-to-go

Let us examine the expression for policy gradient:

$$\nabla_\theta J(\pi_\theta) = \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) R(\tau)] \tag{2.48}$$

If we take a step with this gradient, we push the log-probabilities of each action in proportion to $R(\tau)$, the sum of all rewards obtained. This does not make much sense as agents should only reinforce their actions based on the consequences. Rewards obtained before taking an action have no bearing on how good that action was; only rewards afterwards matter.

We can show that this intuition is already contained in the expression for the policy gradient. We can rewrite it as

$$\nabla_\theta J(\pi_\theta) = \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) \sum_{t'=t}^{T} R(s_{t'}, a_{t'}, s_{t'+1})]. \tag{2.49}$$

Here it is explicit that actions are only reinforced based on the rewards obtained after they are taken. This form is called the reward-to-go policy gradient because the sum of rewards after a point in a trajectory is called the reward-to-go from that point. Mathematically,

$$\hat{R}_t = \sum_{t'=t}^{T} R(s_{t'}, a_{t'}, s_{t'+1}). \tag{2.50}$$

Baselines

An immediate consequence of the EGLP lemma is that for any function b depending only on the state s_t,

$$\mathop{\mathbb{E}}_{a_t \sim \pi_\theta}[\nabla_\theta \log \pi_\theta(a_t|s_t) b(s_t)] = 0. \tag{2.51}$$

Therefore, any number of terms can be added or subtracted from our policy gradient without changing its expectation.

$$\nabla_\theta J(\pi_\theta) = \mathop{\mathbb{E}}_{\tau \sim \pi_\theta}[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)(\sum_{t'=t}^{T} R(s_{t'}, a_{t'}, s_{t'+1}) - b(s_t))] \tag{2.52}$$

Any function b used like this is called a baseline. The most common choice of the baseline is the on-policy value function $V^\pi(s_t)$. This is the average return an agent gets if it starts in state s_t and then acts according to π for the rest of its life. This choice also has the effect of reducing variance in the sample estimate for its policy gradient. This results in faster and more stable policy learning. This encodes the intuition that if an agent gets what it expects when it should feel neutral about it.

We have seen so far that the policy gradient has the general form

$$\nabla_\theta J(\pi_\theta) = E_{\tau \sim \pi_\theta}[\nabla_\theta \log \pi_\theta(a_t|s_t) \Phi_t] = 0 \tag{2.53}$$

There are also a few other choices for Φ_t:

The on-policy action-value function,

$$\Phi_t = Q^{\pi_\theta}(s_t, a_t), \tag{2.54}$$

can be used as a baseline. Alternatively, the advantage function, (see eq. 2.39) can be used. This describes how much better or worse it is than other actions on average relative to the current policy.

$$\Phi_t = A^{\pi_\theta}(s_t, a_t). \tag{2.55}$$

2.3.4 Deep Deterministic Policy Gradient

Deep Deterministic Policy Gradient (DDPG) is an algorithm which learns both a Q-function as described in the previous section and a policy at the same time. It does this by using off-policy data and the Bellman equation to learn the Q-function and uses this Q-function to learn the policy. As described previously, if you know the optimal action-value function $Q^*(s, a)$, then in any given state, the optimal action $a^*(s)$ can be found by solving

$$a^*(s) = \arg\max_a Q^*(s, a). \tag{2.56}$$

DDPG learns an approximator to $Q^*(s,a)$ and an approximator to $a^*(s)$ in a way which is specific to continuous action spaces. Because the action space is continuous, the function $Q^*(s,a)$ is assumed to be differentiable with respect to the action. This allows efficient usage of a gradient-based learning procedure for a policy $\mu(s)$. We can therefore substitute the action in our Q-function for $\mu(s)$, giving

$$\max_a Q(s,a) \approx Q(s,\mu(s)). \tag{2.57}$$

Equation 2.38 is the starting point for learning an approximator to $Q^*(s,a)$. Suppose the approximator is a neural network $Q_\phi(s,a)$ with parameters ϕ and we have also collected a set of transitions, $\mathcal{D}$. We can define a mean-squared Bellman error (MSBE) function to tell us how close our approximation is to the true value.

$$L(\phi,\mathcal{D}) = \mathop{\mathbb{E}}_{(s,a,r,s',d)\sim\mathcal{D}}\left[\left(Q_\phi(s,a) - \left(r+\gamma(1-d)\max_{a'} Q_\phi(s',a')\right)\right)^2\right]. \tag{2.58}$$

Here, d is interpreted as a boolean value of the state being terminated (i.e. 1 = true, 0 = false). The algorithm also makes use of a replay buffer, the set of previous experiences, $\mathcal{D}$. For stable behaviour, this buffer should be sufficiently large to contain a wide range of experiences, but not so large to hinder performance.

The function

$$r+\gamma(1-d)\max_{a'} Q_\phi(s',a'), \tag{2.59}$$

is called the target as this is what we are trying to match with the Q-function. However, this also depends on ϕ, the same parameters we are trying to train, making MSBE minimisation unstable. To solve this, we use a second network, ϕ_{target} which lags the first by a certain number of time steps.

This target network is updated each time the main network is updated. This is done via polyak averaging.

$$\phi_{target} \to \rho\phi_{target} + (1-\rho)\phi. \tag{2.60}$$

The hyperparameter $\rho \in (0,1)$ is called the polyak coefficient, and is usually kept close to 1.

The target policy network is calculated in the same way as the target Q-function - the policy parameters are polyak averaged over the course of training. Thus, Q-learning in DDPG is performed by minimising

$$L(\phi,\mathcal{D}) = \mathop{\mathbb{E}}_{(s,a,r,s',d)\sim\mathcal{D}}\left[\left(Q_\phi(s,a) - (r+\gamma(1-d)Q_{\phi_{target}}(s',\mu_{\theta_{target}}(s')))\right)^2\right], \tag{2.61}$$

by gradient descent. Here $\mu_{\theta_{target}}$ is the target policy.

Policy learning is fairly simple. We want a deterministic policy $\mu_\theta(s)$ which gives the continuous action that maximises $Q_\phi(s, a)$. We assume this is differentiable with respect to the action, so we can just perform gradient descent with respect to the policy parameters only in order to solve

$$\max_\theta \mathop{\mathbb{E}}_{s\sim\mathcal{D}}[Q_\phi(s, \mu_\theta(s))]. \tag{2.62}$$

2.3.5 Twin Delayed DDPG

While DDPG can often obtain good results, it can be brittle with respect to its hyperparameters and requires careful tuning. This is due to the agent overestimating Q-values due to noise. Twin Delayed DDPG (TD3) is a refinement of DDPG which uses a number of tricks to combat this overestimation, leading to much more robust and consistent performance.

Overestimation bias is a property of Q-learning in which the maximisation of a noisy value estimate induces a consistent overestimation. This is unavoidable given the imprecision of the estimator and is further exaggerated by the nature of temporal difference learning in which an estimate of the value function is updated using the estimate of a subsequent state. The accumulated error can cause arbitrarily bad states to be estimated as high values, resulting in suboptimal policy updates [9].

In Q-learning, the value function can be learned using temporal difference learning, based on the Bellman equation.

$$Q^\pi(s, a) = r + \gamma E_{s',a'}[Q^\pi(s', a')], \quad a' \sim \pi(s'). \tag{2.63}$$

For a large state space, this value can be estimated with a differentiable function approximator Q^θ. In deep Q-learning the network θ is updated with temporal difference learning with a secondary frozen target network $Q_{\theta'}(s, a)$ to maintain a fixed objective y over multiple updates:

$$y = r + \gamma Q_{\theta'}(s', a'), a' \sim \pi_{\phi'}(s'), \tag{2.64}$$

where the actions are selected from a target actor network $\pi_{\phi'}$. The weights of a target network are either updated periodically to exactly match the weights of the current network or by some proportion τ at each time step $\theta' \to \tau\theta + (1 - \tau)\theta'$. This update can be applied in an off-policy manner.

In Q-learning with discrete actions, the value estimate is updated with a greedy target $y = r+\gamma \max_{a'} Q(s', a')$, however, if the target is susceptible to error this value estimate will generally be greater than the true maximum. For example, in DDPG, this manifests in the algorithm being very brittle with respect to the hyperparameters and other tuning mechanisms. Twin Delayed DDPG (TD3) is a refinement of DDPG which relies on three key tricks.

Clipped Double-Q learning: TD3 learns two Q-functions and uses the smaller of the two values to form the targets in the MSBE.

Delayed policy updates: Rather than updating the policy network every time the Q-function is updated, TD3 does this less frequently. Generally, one policy network update is done every two Q-function updates.

Target policy smoothing: TD3 adds noise to the target action. This stops the policy from exploiting errors in the Q-function by smoothing it with respect to changes in the action. The target actions are therefore

$$a'(s') = \text{clip}(\mu_{\theta_{target}}(s') + \text{clip}(\epsilon, -c, c), a_{low}, a_{high}), \quad \epsilon \sim \mathcal{N}(0, \sigma). \tag{2.65}$$

This smoothing effect stops the policy from exploiting a sharp peak for some actions, which will cause brittle and unpredictable behaviour.

Both Q-functions use a single target, calculated as the smaller of the two Q-functions' target values.

$$y(r, s', d) = r + \gamma(1 - d) \min_{i=1,2} Q_{\phi_{i,target}}(s', a'(s')). \tag{2.66}$$

We then regress to the target

$$L(\phi_1, \mathcal{D}) = \mathop{\mathbb{E}}_{(s,a,r,s,d)\sim\mathcal{D}} \left[\left(Q_{\phi_1}(s, a) - y(r, s', d) \right)^2 \right], \tag{2.67}$$

$$L(\phi_2, \mathcal{D}) = \mathop{\mathbb{E}}_{(s,a,r,s,d)\sim\mathcal{D}} \left[\left(Q_{\phi_2}(s, a) - y(r, s', d) \right)^2 \right]. \tag{2.68}$$

Lastly, the policy is learned by maximising Q_{ϕ_1} in the same way as DDPG:

$$\max_{\theta} \mathop{\mathbb{E}}_{s\sim D}[Q_{\phi_1}(s, \mu_\theta(s))]. \tag{2.69}$$

2.4 Conclusion

In this chapter, we have covered the theory for the crucial concepts of this project. We have discussed the inner workings of CMA-ES and understood how the population can impact performance. Furthermore, the experimental and benchmarking framework, IOHprofiler, has been introduced. Finally, the key concepts and terminology of reinforcement learning were explained, along with an introduction to the specific reinforcement learning algorithm (TD3) used in this project.

Chapter 3

Related Work

In this section, our attention is directed towards exploring the existing literature on CMA-ES, reinforcement learning, and dynamic algorithm configuration. We will delve into population size control schemes, including restart strategies and PSA-CMA-ES. Additionally, we will investigate the application of reinforcement learning for dynamically configuring CMA-ES to adapt strategy parameters on-the-fly.

3.1 Restart Strategies

The simplest approach to population size adaptation is to stop the CMA-ES run and restarting with new parameters. For population size control, this strategy works by stopping one CMA-ES run when certain criteria are met, and starting a new independent CMA-ES run with an updated population size. Only the current best objective function value is carried over to the restart.

An example of this is the IPOP scheme, in which the population size doubles every restart [1]. None of the other strategy parameters are changed. This approach consistently outperformed the default CMA-ES over a portfolio of 25 benchmark problems.

A refinement to this scheme is the BIPOP strategy, in which there are two regimes - a period where IPOP is used, and another local search strategy, where small population sizes and a small variance are employed. This means the algorithm benefits from a large population where the function is well-structured, and a more narrow search with a smaller population size for where the function is more weakly structured and a large population size would be unhelpful [15].

The first phase of the search is a multi-restart strategy, where the population size is doubled every restart, such that

$$\lambda_{large} = 2^{i_{restart}} \lambda_{default}, \tag{3.1}$$

for each restart $i_{restart}$. The second phase uses a small population size which is randomly drawn from

$$\lambda_{small} = \lfloor \lambda_{default} \left(\frac{\lambda_{large}}{2\lambda_{default}} \right)^{\mathcal{U}[0,1]^2} \rfloor. \tag{3.2}$$

Here, $\mathcal{U}[0,1]$ is the uniform distribution on the closed interval $[0,1]$.

However, there are potential disadvantages to this approach. The population size control scheme is entirely specified in advance, rather than reacting to the internal state of the algorithm or the state of the problem. Thus, population size is adapted more frequently than needed so the change is not necessarily advantageous. Moreover, sophisticated stopping mechanisms are required for the efficiency of the restart strategies. These can be very hard to design, especially for noisy and multimodal functions.

Another approach is to adapt the population size during the optimisation procedure, and several strategies have been proposed. pcCMA-ES is a strategy for noisy optimisation and estimates the noise strength on the objective function and adapts the population size accordingly [17]. CMAES-APOP is a similar method but for noiseless functions and the population size control is based on the decrease in objective function value over iterations [22].

3.2 PSA-CMA-ES

PSA-CMA-ES is a method for controlling the population size during the optimisation procedure [24]. In this scheme, the population size is adapted based on the internal state of the algorithm rather than being chosen in advance such as with restart schemes. In particular, the change is based on the norm of a novel evolution path. This is done as follows.

We let the parameter vector of our normal distribution be θ, and its movement vector to be $\Delta\theta^{(t+1)} = (\Delta\boldsymbol{m}^{(t+1)}, \text{vech}(\Delta\Sigma^{(t+1)}))$. vect(A) denotes a vector of the flattened upper triangular elements of A. The components are computed as

$$\Delta\boldsymbol{m}^{(t+1)} = \boldsymbol{m}^{(t+1)} - m^{(t)} \tag{3.3}$$

$$\Delta\Sigma^{(t+1)} = (\sigma^{(t+1)})^2\boldsymbol{C}^{t+1} - (\sigma^{(t)})^2\boldsymbol{C}^{t} \tag{3.4}$$

The evolution path is used to accumulate the movement of $\Delta\theta$. To ensure that the evolution path is stable and independent of strategy parameters, it is defined as

$$\boldsymbol{p}_\theta^{(t+1)} = (1-\beta)p_\theta^{(t)} + \sqrt{\beta(2-\beta)}\frac{\mathcal{I}_{\theta^t}^{1/2}\Delta\theta^{(t+1)}}{\mathbb{E}[||\mathcal{I}_{\theta^t}^{1/2}\Delta\theta^{(t+1)}||^2]^{1/2}} \tag{3.5}$$

Here $\mathcal{I}_{\theta^t}^{1/2}$ is the square root of the Fisher information matrix, which provides invariance against parameterisation of the probability distribution. The normalisation factor of $\mathbb{E}[||\mathcal{I}_{\theta^t}^{1/2}\Delta\theta^{(t+1)}||^2]^{1/2}$ avoids scaling of the parameter movement due to the change in population size. We use the approximated value

$$\mathbb{E}[||\mathcal{I}_{\theta^t}^{1/2}\Delta\theta^{(t+1)}||^2]^{1/2} \approx \frac{nc_m^2}{\mu_w}$$
$$+\frac{2n(n-\chi_n^2)}{\chi_n^2}\left(\frac{c_\sigma}{d_\sigma}\right)+\frac{1}{2}\left[1+8\frac{n-\chi_n^2}{\chi_n^2}\left(\frac{c_\sigma}{d_\sigma}\right)^2\right]\left[\frac{(n^2+n)c_\mu}{\mu_w}+(n^2+n)c_c(2-c_c)c_1c_\mu\mu_w\sum_{i=1}^{\lambda}w_i^2+c_1^2(n^2+n)\right].$$

We can now use $\mathbf{p}_\theta$ to update the population size as follows.

$$\lambda^{(t+1)} = \lambda^{(t)}\exp\left(\beta\left(1-\frac{||\mathbf{p}_\theta||^2}{\alpha}\right)\right). \tag{3.6}$$

The step size will generally increase as the population increases, leading to an artificial tendency for the change in this quantity to be captured by the evolution path. This leads to unstable adaptation of the population size so we must introduce a correction factor. We update the step size after changing the population according to

$$\sigma^{(t+1)} \to \sigma^{(t+1)} \frac{\sigma^*(\lambda^{(t+1)})}{\sigma^*(\lambda^{(t)})}. \tag{3.7}$$

The factor σ^* is calculated with

$$\sigma^*(\lambda^{(t)}) = \frac{c \cdot n \cdot \mu_w}{n - 1 + c^2 \cdot \mu_w}. \tag{3.8}$$

Here c is the weighted average of the expected value of the normal order statistics from the population λ,

$$c = -\sum_{i=1}^{\lambda} w_i \mathbb{E}[\mathcal{N}_{i:\lambda}]. \tag{3.9}$$

This strategy proved to be very efficient with well-structured multimodal functions but struggled with weakly structured problems. Furthermore, the performance relative to the default population on unimodal functions decreased as the dimensions were scaled up. This suggests that the population size was adapted greater than necessary for these unimodal and weakly-structured problems. The overall performance was comparable to the BIPOP-CMA-ES scheme [23, 20].

The advantage of this method is that the population size is adapted on-the-fly during a CMA-ES run, rather than having multiple static runs. Additionally, the adaption of the population size is done using information about the internal state of the algorithm, rather than being specified beforehand. This means the algorithm will generalise well and be simpler to run [23].

3.3 Reinforcement Learning for Step-size Adaptation

The population size isn't the only parameter where the performance of CMA-ES can be improved with tuning. The step size can also benefit from more complex tuning. While the standard method described in Chapter 2 (CSA) works well, it has also been formulated as a reinforcement learning problem [30].

Similar to our problem of controlling the population size, the aim here was to use a reinforcement learning agent to adjust the step size, $\sigma^{(t)}$, based on some state information, s_t, of CMA-ES.

$$\sigma^{(t+1)} \sim \pi(s_t). \tag{3.10}$$

The strategy employed a continuous action space for σ. The state space consisted of

1. The current step size value $\sigma^{(t)}$.
2. The current evolution path $\mathbf{p}_c^{(t)}$.
3. A history of h previous changes in the objective value.

4. A history of h previous step sizes.

The value of $h = 40$ was chosen. The reward used was the negative objective function value, so the agent was rewarded by finding a smaller best objective value, therefore optimising the agent for any-time performance. This rewarded the agent more for getting closer to the optimum over the course of a whole CMA-ES run, rather than by providing a better value only compared to the previous generation.

It has been shown that pure reinforcement learning-based approaches can require a lot of computational resources and are sample inefficient for dynamic algorithm configuration [29, 31]. Thus, guided policy search (GPS) was used to exploit existing heuristics. In this instance, they used the normal step size adaptation strategy as a teacher to learn a policy that either imitates or improves upon this.

They found that these learned policies are capable of outperforming the default step-size strategy on a range of black-box problems. It also showed good generalisation capabilities to new functions and higher dimensions.

Chapter 4

Design

Based on the theory and literature covered previously, this section will focus on the design-specific decisions, along with reasoning for the benchmarks chosen and other general considerations.

The CMA-ES algorithm was implemented based on the pre-existing ModularCMAES code available on GitHub[1].

In this section, we will delve into specific implementation details related to logging, modifications made to the existing ModularCMAES code, and the creation of a reinforcement learning environment. Ensuring reproducibility, a fundamental aspect of credible research, was a key priority, necessitating accurate reimplementation of previous work. As a result, we will also address the challenges encountered during the reimplementation process.

Creating the reinforcement learning environment involved setting up a suitable framework that allowed interactions between the CMA-ES algorithm and the benchmark functions. The environment was implemented using DACBench [2], which is a library for benchmarking dynamic algorithm configuration problems [8].

4.1 Problem Portfolio

The problem set employed consisted of the Black-box Optimisation Benchmark (BBOB) [14] functions from IOHProfiler [13]. These functions comprise a comprehensive collection of 24 continuous optimisation benchmark problems, exhibiting diverse levels of difficulty and optimisation challenges.

The portfolio is divided into five different sections, where each section comprises functions which share similar characteristics and properties.

- $f1 - f5$: Separable functions.
- $f6 - f9$: Functions with low or moderate conditioning.
- $f10 - f14$: Unimodal functions with high conditioning.
- $f15 - f19$: Multimodal functions with adequate global structure.
- $f20 - f24$: Multimodal functions with weak global structure.

[1] ModularCMAES version 0.0.2.8.4. https://github.com/IOHprofiler/ModularCMAES
[2] DACBench version 0.2.0. https://github.com/automl/DACBench

Here, separability refers to a multivariate function which can be rewritten as a product of univariate functions [19]. Conditioning refers to how sensitive the function's output is to a small change in the input [3]. Surface plots of all 24 functions are shown in Appendix 1.

Instance 1 was selected for all of the BBOB problems. Since CMA-ES remains invariant to translations and rotations, using different instances would not significantly impact the algorithm's performance.

4.2 Basic Experimental Setup

All experiments followed a standardised procedure to ensure easy comparison between the different population size adaptation schemes.

A dimensionality of 10 was selected for the experiments, as it offered reasonable computation times while still being representative of general performance and commonly available in the pre-loaded data from other experiments provided by IOHProfiler. This decision facilitated straightforward testing and direct comparisons with the results of the PSA-CMA-ES implementation.

The experiment budget for each run was set at $2500 \cdot d$, where d is the dimensionality of the problem. This served as the sole stopping criterion. Each algorithm was executed 25 times on each problem-dimension pair, and the data was collected using the logging functionality in IOHprofiler.

By following this standardized approach, we ensured consistency and comparability in the experimental setup, enabling us to draw meaningful conclusions about the performance of the CMA-ES algorithm under a variety of population size adaptation strategies.

A subclass of the ModularCMAES class was created to allow for easy tracking of all the parameters and preprocessing of the data before inputting it into IOHAnalyzer. The tracked parameters were as follows:

- Objective function value.
- σ: step size.
- λ: population size.
- $\mathbf{p}_c$: evolution path for the covariance matrix.
- $\mathbf{p}_\sigma$: evolution path for the step-size.

4.3 Constant Population

In this study, we began by establishing baseline performance measurements using constant population sizes. This approach provided valuable insights into which functions were better suited for either large or small population sizes.

These experiments were run using the standard code from ModularCMAES with all other parameters set to their defaults so no other design decisions were required. The outline for the algorithm is shown in Algorithm 1.

Algorithm 1 Basic CMA-ES Procedure

Require: $\mathbf{m} \in \mathbb{R}^n$

Ensure: $c_m = 1$, $\mathbf{C} = \mathbf{I}$, $\mathbf{p}_c = \mathbf{p}_s = \mathbf{0}$.

while not terminate **do**

$\mu \leftarrow \lfloor \lambda/2 \rfloor$

$w_i \leftarrow \frac{\log(\mu+0.5)-\log(i)}{\sum_{i=1}^{\mu}(\log(\mu+0.5)-\log(i))}, \;\; i = 1, .., \mu$ ▷ Setting the weights for recombination

$w_i = 0, \;\; i = \mu + 1, ..., \lambda$

$\mu_{eff} = (\sum_{i=1}^{\lambda} w_i^2)^{-1}$ ▷ Calculating effective population size

$c_\sigma \leftarrow (\mu_{eff} + 2)/(n + \mu_{eff} + 5)$

$d_\sigma \leftarrow 1 + 2\max(0, \sqrt{(\mu_{eff} - 1)/(n + 1)} - 1) + c_\sigma$

$c_c \leftarrow (4 + \mu_{eff}/n)(n + 4 + 2\mu_{eff}/n)$

$c_1 \leftarrow 2/((n + 1.3)^2 + \mu_{eff})$

$\mathbf{x}_i^{(t+1)} \sim \mathbf{m}^{(t)} + \sigma^{(t)}\mathcal{N}(\mathbf{0}, \mathbf{C}^{(t)}), \;\; i = 1, ..., \lambda$ ▷ Sample new candidate solutions

$\mathbf{m}^{(t+1)} = \mathbf{m}^{(t)} + c_m \sum_{i=1}^{\lambda} w_i(\mathbf{x}_{i:\lambda^t} - \mathbf{m}^{(t)})$ ▷ Find the new mean

$\mathbf{p}_\sigma^{(t+1)} = (1 - c_\sigma)\mathbf{p}_\sigma^{(t)} + \sqrt{c_\sigma(2 - c_\sigma)\mu_{eff}}\mathbf{C}^{(t)^{-1/2}} \frac{\mathbf{m}^{(t+1)} - \mathbf{m}^{(t)}}{c_m \sigma^{(t)}}$ ▷ Update the step-size evolution path

$\mathbf{p}_c^{(t+1)} = (1 - c_c)\mathbf{p}_c^{(t)} + \sqrt{c_c(2 - c_c)\mu_{eff}} \frac{\mathbf{m}^{(t+1)} - \mathbf{m}^{(t)}}{c_m \sigma^{(t)}}$ ▷ Update the covariance matrix evolution path

$\sigma^{(t+1)} = \sigma^{(t)} \exp\left(\frac{c_\sigma}{d_\sigma}\left(\frac{||\mathbf{p}_\sigma^{(g+1)}||}{\mathbb{E}||\mathcal{N}(\mathbf{0},\mathbf{I})||} - 1\right)\right)$ ▷ Update the step-size

$\mathbf{y}_{i:\lambda}^{(t+1)} = \mathbf{x}_{i:\lambda}^{(t+1)} - \mathbf{m}^t$

$\mathbf{C}^{(t+1)} = (1 - c_1 - c_\mu \sum_i w_i)\mathbf{C}^{(t)} + c_1 \mathbf{p}_c^{(t+1)} \mathbf{p}_c^{(t+1)^\intercal} + c_\mu \sum_{i=1}^{\lambda} w_i \mathbf{y_i}_{:\lambda}{}^{(t+1)} \left(\mathbf{y_i}_{:\lambda}{}^{(t+1)}\right)^\intercal$ ▷ Update the covariance matrix

end while

4.3.1 Experimental Details

For the experiments involving a small population, we chose a value of $\lambda = 4$, as it represents the smallest sensible value for CMA-ES (see section 2.1.1).

To determine the default population size, we used the formula $\lambda \approx 4 + \lfloor 3 \log(d) \rfloor$, which is the recommended value for CMA-ES [11]. Here, d refers to the dimensionality of the problem. For our specific experiments where $d = 10$, this resulted in a default population size of $\lambda = 10$.

In order to explore a broader range of population sizes and understand their impact on performance, we also conducted experiments using a large population of $\lambda = 128$. By incorporating such a significantly larger population, we ensured that there was a substantial variation in the order of magnitude of λ.

4.3.2 Conclusion

For the first set of experiments, we benchmark the standard CMA-ES configuration with static population sizes. This gives us a set of baseline performance readings to compare to more complex experiments.

4.4 Simple Population Adaptation

In this subsequent phase, we proceeded to compare the performance of the constant population size approach with four different schemes: linearly increasing and decreasing and exponentially increasing and decreasing.

These schemes updated the population size on-the-fly rather than restarting the algorithm each time, as discussed in Chapter 3. The linear schemes updated the population size by adding or subtracting a fixed value from the population every specified number of time steps. Likewise, the exponential schemes multiplied or divided the population by a fixed amount.

4.4.1 Algorithm Details

The updated population size was stored in a temporary variable, $\hat{\lambda}$. This was then updated and bounded to ensure the population stayed within a predefined range. The values $\lambda_{min} = 8$ and $\lambda_{max} = 128$ were chosen to ensure a wide range of sensible values. This was done as

$$\hat{\lambda} \leftarrow \min(\max(\lambda', \lambda_{min}), \lambda_{max}). \tag{4.1}$$

Finally, population size was updated by rounding this temporary value by the rounding scheme specified in the experiment (see section 4.4.2). These values were chosen such that the adaptation happened over the course of the whole optimisation procedure, rather than being concentrated at the beginning or the end.

These schemes aimed to dynamically adjust the population size at regular intervals over time to observe their impact on the algorithm's performance. The algorithm for these methods is shown in Algorithm 2.

The exact details of the population size adjustment are shown in Table 4.1.

Certain BBOB functions would benefit from larger population sizes, as they allowed CMA-ES to explore the search space more extensively and effectively. On the other hand, we expect that

Algorithm 2 Simple Adaptive Population Size CMA-ES

Require: $\mathbf{m} \in \mathbb{R}^n$.
Ensure: $c_m = 1$, $\lambda_{min} = 4$, $\lambda_{max} = 512$, $\mathbf{C} = \mathbf{I}$, $\mathbf{p}_c = \mathbf{p}_s = \mathbf{p}_\theta = \mathbf{0}$, $f = 32$.
1: **while** not terminate **do**
2: Perform one CMA-ES iteration as described in Algorithm 1.
3: **if** t modulo $f = 0$ **then**
4: $\hat{\lambda}^{(t)} = \text{updatePopsize}(\lambda^{(t)})$ ▷ Calculate new population size
5: $\lambda^{(t+1)} = \text{round}(\hat{\lambda}^{(t)})$ ▷ Round new population size and update the algorith
6: **end if**
7: $t \leftarrow t + 1$
8: **end while**

Strategy	Adaptation	Frequency
Linearly Increasing	$\lambda \leftarrow \lambda + 10$	32 steps
Linearly Decreasing	$\lambda \leftarrow \lambda - 10$	32 steps
Exponentially Increasing	$\lambda \leftarrow 1.2 \cdot \lambda$	32 steps
Exponentially Decreasing	$\lambda \leftarrow \lambda / 1.2$	32 steps

TABLE 4.1: Adaption parameters for the simple population size adaption experiments.

some functions exhibit improved performance with smaller population sizes, indicating that a more focused exploration strategy was beneficial for converging towards optimal solutions in these cases.

Furthermore, we wanted to gain insights into the effects of the adaptive schemes and which functions benefited from each approach. Identifying which characteristics of certain BBOB functions require adaptive population sizes would be useful to tailor our attention in future experiments. Identifying the difference in performance between the linear and exponential schemes will also be possible, to see if the specific details of the adaption are important.

Additionally, we aimed to highlight certain BBOB functions that proved challenging for the simple population size adaptation schemes. Identifying these problematic functions allowed us to recognize areas where the conventional approaches might not be sufficient. Consequently, we could direct our attention to developing and applying more advanced, complex optimization methods to tackle these difficult landscapes effectively.

4.4.2 Rounding

With the exponential schemes, the calculated population size was not always an integer, so this value had to be rounded before using it for the next generation. Along with the standard rounding method, a stochastic rounding scheme was also tested. In this method, the floating point part of the number was interpreted as a probability of the number being rounded up. For example, 3.1 would have a 10% chance of being rounded to 4 and a 90% chance of being rounded to 3. Formally,

$$\text{round}(x) = \begin{cases} \lfloor x \rfloor & \text{with probability } 1 - (x - \lfloor x \rfloor) \\ \lfloor x \rfloor + 1 & \text{with probability } x - \lfloor x \rfloor \end{cases} \tag{4.2}$$

4.4.3 Conclusion

In this section, we aim to identify the effect of on-the-fly population size adaption via the implementation of a variety of simple schemes. These heuristics will guide subsequent experiments by showing how adaptive population sizes can be beneficial. Furthermore, we will also be able to identify any significant differences in performance from specific implementation details such as the adaptive scheme or rounding method used.

4.5 PSA-CMA-ES

The previous methods of adapting the population size have no information about the algorithm's internal state, so the population size is only being adjusted based on predefined heuristics. Thus, a significant amount of manual tuning is still required for good performance.

The subsequent experiments were based on the PSA-CMA-ES implementation described previously. This algorithm uses a novel evolution path $\mathbf{p}_\theta$ to inform the algorithm at each generation of the ideal population size. Although the data from the original PSA-CMA-ES paper [24, 23] was available on IOHAnalyzer, we decided to reimplement it within the ModularCMAES code to provide consistency across experiments and to allow easy use of this algorithm for future work.

4.5.1 Algorithm Details

The general procedure for the PSA-CMA-ES algorithm is shown in algorithm 3.

Algorithm 3 PSA-CMA-ES

Require: $\mathbf{m} \in \mathbb{R}^n$, update frequency $f \in \mathbb{N}$

Ensure: $c_m = 1$. $\alpha = 1.4$, $\beta = 0.4$, $\lambda_{min} = 4$, $\lambda_{max} = 512$, $\mathbf{C} = \mathbf{I}$, $\mathbf{p}_c = \mathbf{p}_s = \mathbf{p}_\theta = \mathbf{0}$, $\alpha = 1.4$, $\beta = 0.4$.

1: **while** not terminate **do**

2: Perform one CMA-ES iteration as described in Algorithm 1.

3: $\mathbf{p}_\theta^{(t+1)} = (1-\beta)\mathbf{p}_\theta^{(t)} + \sqrt{\beta(2-\beta)}\frac{\mathcal{I}_{\theta t}^{1/2}\Delta\theta^{(t+1)}}{\mathbb{E}[||\mathcal{I}_{\theta t}^{1/2}\Delta\theta^{(t+1)}||^2]^{1/2}}$ ▷ Update population size evolution path

4: $\hat{\lambda}^{(t+1)} = \lambda^{(t)} \exp\left(\beta\left(1 - \frac{||\mathbf{p}_\theta^{(t+1)}||^2}{\alpha}\right)\right)$ ▷ Calculate new population size

5: $\hat{\lambda}^{(t+1)} = \min(\max(\hat{\lambda}^{(t+1)}, \lambda_{min}), \lambda_{max})$

6: $\lambda^{(t+1)} = \text{round}(\hat{\lambda}^{(t)})$ ▷ Update the algorithm with the new population size

7: $\sigma^{(t+1)} = \sigma^{(t+1)}\frac{\sigma^*(\lambda^{(t+1)})}{\sigma^*(\lambda^{(t)})}$ ▷ Adjust the step-size

8: $t \leftarrow t + 1$

9: **end while**

4.5.2 Challenges

The reimplementation of this algorithm was challenging. There were a variety of complex calculations which needed to be made at the right time to ensure correctness. This implementation was tested using the data from the benchmarking paper [23] available on IOHanalyzer (see Appendix 2).

There were two primary design challenges involved. The first one was calculating the evolution path $\mathbf{p}_\theta$, which involved the calculation of the Fisher information matrix of the distribution with respect to the parameter vector θ. There is no existing Python package for this so the implementation was based on the code from the original paper [24].

Secondly, a step size correction factor must be calculated at each generation. This relied on a weighted average of the normal order statistics of the distribution. This can be expensive to calculate exactly so approximations were used. When $\lambda < 50$, Blom's approximation was used to calculate the expected value as it is simple to compute and sufficiently accurate for lower dimensions [16]. The formula for this is

$$\mathbb{E}(r, \lambda) = -\Phi^{-1}\left(\frac{r - \alpha}{\lambda - 2\alpha + 1}\right), \quad \alpha = 0.375. \tag{4.3}$$

However, when the population size is greater than this value, the Davis-Stephens approximation was used [5]. This method is more computationally expensive but provides greater accuracy for large population sizes.

4.5.3 Conclusion

From the reimplementation, we have a consistent benchmark for current state-of-the-art performance for population size control in CMA-ES. We can use this to compare the performance of the reinforcement learning approach, as well as examine its policy compared to PSA-CMA-ES.

4.6 Reinforcement Learning based Population Size Adaptation

At this stage, a reinforcement learning-based system needed to be designed to control the population size during the optimisation procedure. The design of PSA-CMA-ES is based on expert knowledge of the algorithm itself - the aim of this work is to see if we can rediscover a similar adaptation scheme in a data-driven fashion using reinforcement learning.

4.6.1 Experimental Design

Here, we will explain and justify the experimental design decisions for the reinforcement learning environment and algorithm used.

Observation & Action Spaces

The first design decision was choosing the observation and action spaces. There were two approaches considered. The first one was to use raw data from the CMA-ES algorithm, which is more realistic as we do not always know how to engineer novel and informative features in advance. The observation space would consist of an array of population sizes and an array of objective function values from previous generations, along with the current population size, current objective value, and the total number of function evaluations, similar to the approach described in Chapter 3 [30].

The other approach was to use more refined metrics. The observation space would consist of:

- The population size λ.
- The norm of the evolution path $\mathbf{p}_\theta$

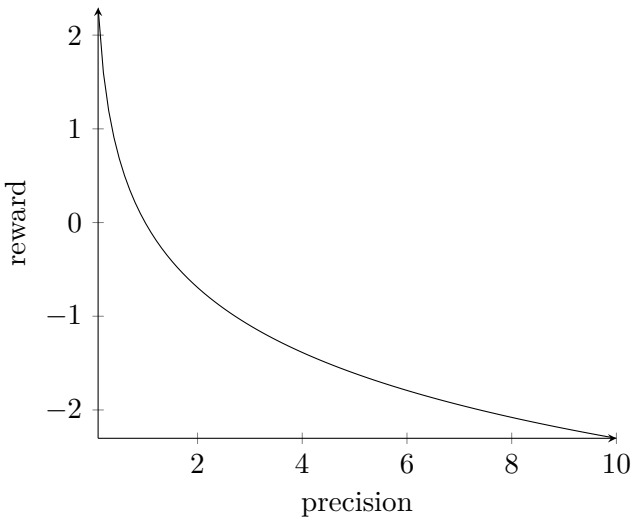


FIGURE 4.1: Reward against precision of best solution found. The precision is the difference between the best-found solution at any given time step and the true global optimum of the function.

- The normalisation factor to ensure the invariance of $\mathbf{p}_\theta$ - $\mathbb{E}[||\mathcal{I}_{\theta^t}^{1/2}\Delta\theta^{(t+1)}||^2]^{1/2}$

The norm of $\mathbf{p}_\theta$ encodes a lot of refined and relevant information about the effect of the population size so it is hoped that this would be beneficial to the reinforcement learning agent and provide enough relevant information to be able to learn. For this reason, these parameters were chosen for the observation space.

There were two choices for how to design the action space: The first was to always keep the population size as an integer and use a discrete action space. The other choice was to treat λ as a continuous variable and use the unrounded value in the intermediate calculations. This value is rounded and the population size is updated for the next generation.

Reward Function

The reward used was based on the precision of the best-found solution at each given time step, i.e. how close this solution is to the global minimum. Let f be the objective function to minimise, f^* be the global minimum of this function, and $f^{(t)}$ be the best so far at step t. The reward was calculated by

$$\text{reward} = -\log(f^* - f^{(t)}), \tag{4.4}$$

letting the algorithm optimise for any-time performance. The negative sign ensures that a larger reward will be given for better precision. A sketch of this reward is shown in Fig. 4.1. This choice of reward had a number of benefits. Firstly, it is invariant to the initial guess and actual function being evaluated, making it more versatile and flexible as the reward will be easily comparable between different functions. Furthermore, having it on a logarithmic scale placed more emphasis on small improvements near the global optimum, so even a small increase in performance will result in a large reward. Likewise, only a small reward is given for an improvement which is far away from the global optimum so little reward is given when the agent's performance is poor.

The RL Algorithm

For the actual algorithm, TD3 was chosen as it is robust with respect to hyperparameters and other kinds of tuning. It also doesn't share the failure modes of other algorithms such as in DDPG or Deep Q-learning, where the learned Q-function can dramatically overestimate its learned Q-function's values, leading to the policy breaking. Thus, we used TD3 as it was more likely to be able to learn and give reasonable and reproducible results without large amounts of fine-tuning.

TD3 is limited to continuous action spaces, so we let the algorithm control the population size as a continuous variable. It was then rounded before the population size was updated in the CMA-ES algorithm. This allowed us to mimic the inner workings of PSA-CMA-ES, where λ is also treated as a continuous variable before finally rounding. As with previous experiments, the population size was bounded between 10 and 512. The lower bound was consistent with the inherent properties of CMA-ES, and the upper bound was large enough to allow largely unrestricted control. However, it prevented the agent from using erroneously large values in the training process, as well as allowing consistent comparison with PSA-CMA-ES.

The TD3 implementation was from Stable Baselines3 [28], which is a set of reliable implementations of reinforcement learning algorithms written in Pytorch. Reproducibility can often be an issue in reinforcement learning research [21] as the algorithms' performance can be very sensitive to the specific implementation details. Using Stable Baselines3 allowed for robust, reproducible, and publicly available implementations to be used. Furthermore, it provided logging functionality and callbacks to evaluate and save models during the training process.

The reinforcement learning environment was implemented using DACBench [8], which is a framework for producing reproducible dynamic algorithm configuration environments. It provided sets of problems for training and testing allowing for consistent and robust experimentation. Furthermore, it provided abstract classes for easy implementation of environments specific to the problem which also included the functionality to read the problem sets. These abstract classes were based on OpenAI's Gym API [4], which is a standard in reinforcement learning and allowed for compatibility with the algorithms provided by Stable Baselines3.

This allowed for very clear and concise benchmarking environments which made it easy to test the effect of any changes made.

Given this information, we hope that the agent will be able to rediscover the population size update stage from the PSA-CMA-ES algorithm.

$$\lambda^{(t+1)} = \lambda^{(t)} \exp\left(\beta\left(1 - \frac{||\mathbf{p}_\theta||^2}{\alpha}\right)\right). \tag{4.5}$$

The goal, therefore, is for the agent to be able to mimic or improve upon equation 4.5.

4.6.2 Training and Testing

We chose to train the agent on individual function-instance-dimension tuples. This would allow the agent to have the greatest chance of success while controlling the population size. As with the previous experiments, we used only $d = 10$ for consistency and ease of comparison, and a budget of $2500 \cdot d$. From this, we could deduce which properties of a function made it difficult to solve even with varying population sizes and whether this was consistent with the limitations of PSA-CMA-ES.

Over the course of the training process, the best-found precision was recorded for each episode, allowing for an interpretable comparison of the quality of the solution during the training process to other non-RL-based approaches. Monitoring this allowed us to understand the change in performance of the agent over the training process. Furthermore, Stable Baselines3 provided callback functionality to monitor training. We used the EvalCallback, which evaluated the model every 100 episodes. The model with the highest cumulative reward over the evaluation procedure was saved.

For evaluating the trained policy we wanted to compare its behaviour to PSA-CMA-ES. IOHexperimenter's logging capabilities were incompatible with the Stable Baselines3 implementation of TD3, meaning a manual approach had to be used.

When evaluating the policy on a BBOB function, the current best objective value, population size, and used budget were recorded. This allowed for easy plotting and analysis of the performance of the agent and its policy. To compare the agent to PSA-CMA-ES, a dummy environment was created. This was exactly the same as the reinforcement learning environment, but rather than the agent controlling the population size at each step, it was simply adapted by PSA-CMA-ES. This allowed for a robust and simple comparison between the two approaches. From this, we will be able to create performance graphs of both methods, similar to the fixed-budget analysis from IOHanalyzer.

4.6.3 Conclusion

Here, we have thoroughly explained and justified all the design decisions for the reinforcement learning environment and algorithm. Furthermore, we have introduced the methods used for monitoring the training process and evaluating the trained model.

Chapter 5

Evaluation

Since we have now covered the necessary theory and experimental design, we can turn to look at the performance of the different approaches. Initially we will get benchmark performance readings on the BBOB function portfolio with the fixed population size experimentes.

Then, we will understand the effect of refining these approaches by naively adapting the population size. This will help us understand where varying population size is most beneficial, and show which features favour increasing or decreasing schemes. Furthermore, we will see what effect the different rounding schemes have on the variations which result in non-integer population sizes.

After this, we can benchmark the current state-of-the-art, PSA-CMA-ES. As this method involves no knowledge of the functions in question, performant results will show good generalisation to real-world problems outside the BBOB problems. These results will also act as a benchmark against the reinforcement learning-based approaches.

We then benchmark these RL-controlled methods to see if they can obtain comparable performance to PSA-CMA-ES. We investigate the change in performance over the training process to understand how long it takes to learn an adequate policy. Finally, we can compare the actual change in population size over the course of the optimisation procedure for the Rl agent compared to PSA-CMA-ES.

5.1 Constant Population

Figure 5.1 shows the performance of the fixed budget approaches on all 24 BBOB functions. The graphs in Fig. 5.1 show a fixed-budget analysis, which tracks the quality of the solution against the number of function evaluations. This helps us understand the best precision we can expect from each population size scheme, as well as investigate convergence speed.

The functions display a variety of behaviour indicating benefits from both large and small population sizes, depending on the function at hand. The default size performs well most of the time and therefore is consistent with $\lambda = 10$ being a sensible choice.

Specifically, functions 15-19 benefit the most from large population sizes for the quality of the final solution compared to other schemes. These are the functions which are multimodal with reasonable global structure. This is expected as the multimodal nature of these functions requires a large population size to adequately explore the whole landscape. A small population is likely to get stuck in a local minimum and try to exploit this region without ensuring it is a global optimum.

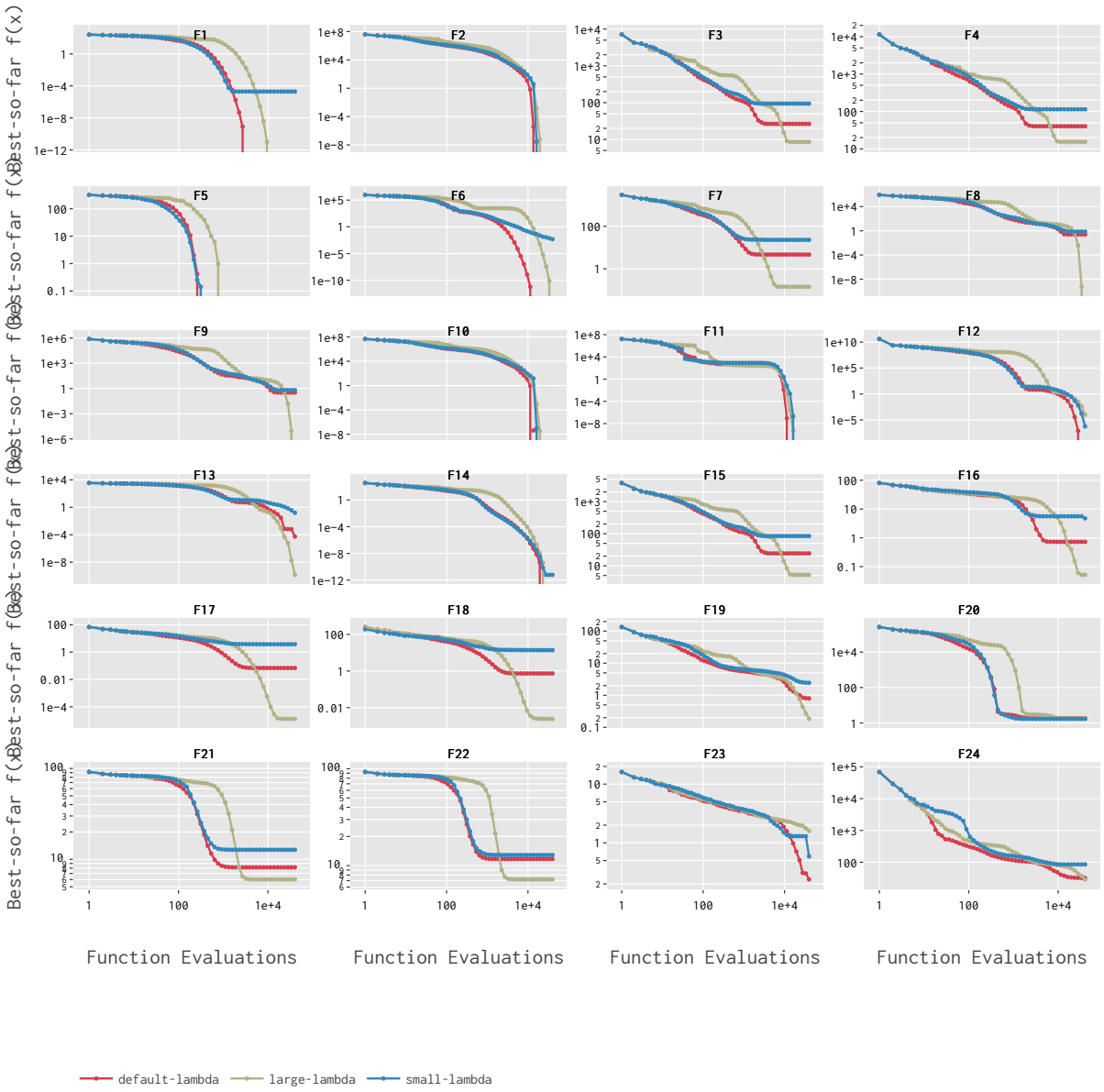


FIGURE 5.1: Fixed budget analysis of the constant population CMA-ES on all 24 BBOB functions.

Small population sizes generally provide poor results, or at best, the same as the default size. This is likely due to inadequate exploration of the search space. They often show results orders of magnitude worse than default or large populations in terms of final precision. We often see these schemes levelling off and providing no improvement after a certain budget has elapsed, suggesting that a local minimum has been found which the small population size cannot escape.

We can also see that CMA-ES performs poorly on functions with weak global structure, regardless of the population size. This suggests that constant population sizes are unsuited for these functions and therefore we expect to see improvements on these functions in particular.

5.2 Simple population size adaptation

We can now turn our attention to the simple population size adaptation strategies. We first compare the relative performance of these schemes. As before, we perform fixed-budget analysis. The results are shown in Figure 5.2.

Both the increasing schemes have very similar performances, regardless of whether the change was additive or multiplicative. Both the rate of convergence and the final precision reached are very similar. The same results can be seen for the decreasing schemes. Therefore, we can conclude that the benefits of adapting the population size come from the variation over the course of the optimisation procedure, rather than the specific method itself.

Furthermore, the decreasing schemes generally perform much better in terms of final precision than the increasing schemes. This is likely due to the large population size at the start being able to explore the whole state space better, before exploiting promising regions more efficiently. This is especially pronounced in multimodal functions, where efficient exploration is necessary to survey the entire function landscape. Likewise, when the population size starts small it can get trapped in a local minimum which can be hard to escape from, even for larger populations at later time steps.

However, when the function is simpler to solve and both methods reach the true global optimum (identified by the precision $< 10^{-12}$), the increasing schemes tend to find this solution with fewer function evaluations. This is likely due to the large number of function evaluations in a small region of space allowing a more fine-grained search.

We now compare the performance of these schemes relative to the default setting. The first thing to note is that the increasing schemes rarely outperform the default. Although they start with a similar population, the increasing schemes will use their budget up too quickly as the optimisation run progresses, leading to insufficient exploration of promising areas.

On the simpler-to-solve functions, for instance, unimodal functions or functions with good global structure, the decreasing schemes and default setting generally perform similarly well. This is likely to be due to the functions being able to be solved with a reasonably efficient budget expenditure. However, the decreasing schemes tend to offer better final precisions on the more difficult functions, especially when the global structure is adequate to poor.

It is on the more difficult functions that we see the greatest increase in performance compared to the default population size. These complex multimodal landscapes require a varying population to adequately explore the whole landscape.

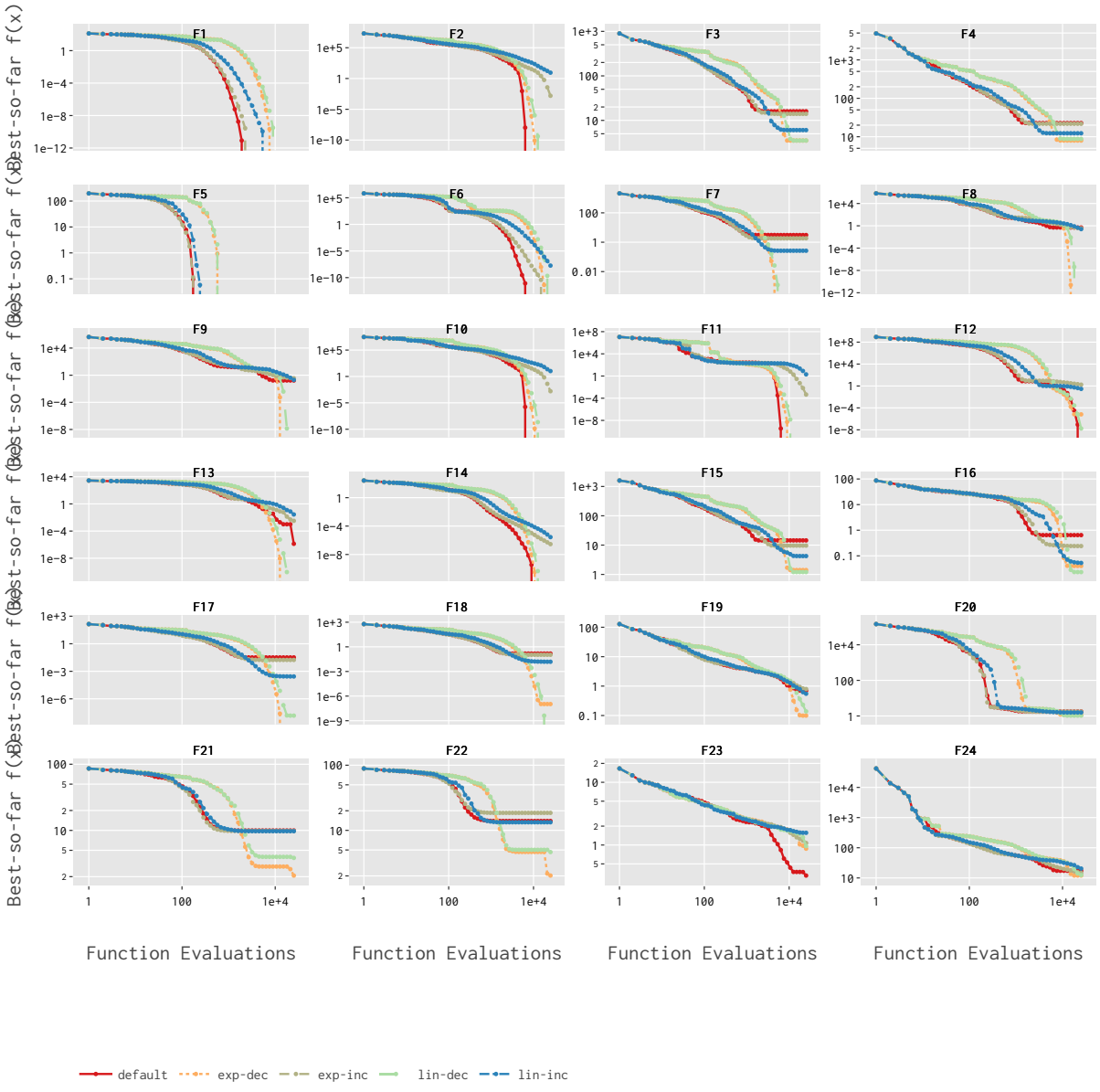


FIGURE 5.2: Fixed budget analysis of the simple population size adaptation schemes on all 24 BBOB functions. The default value of λ is also shown for comparison.

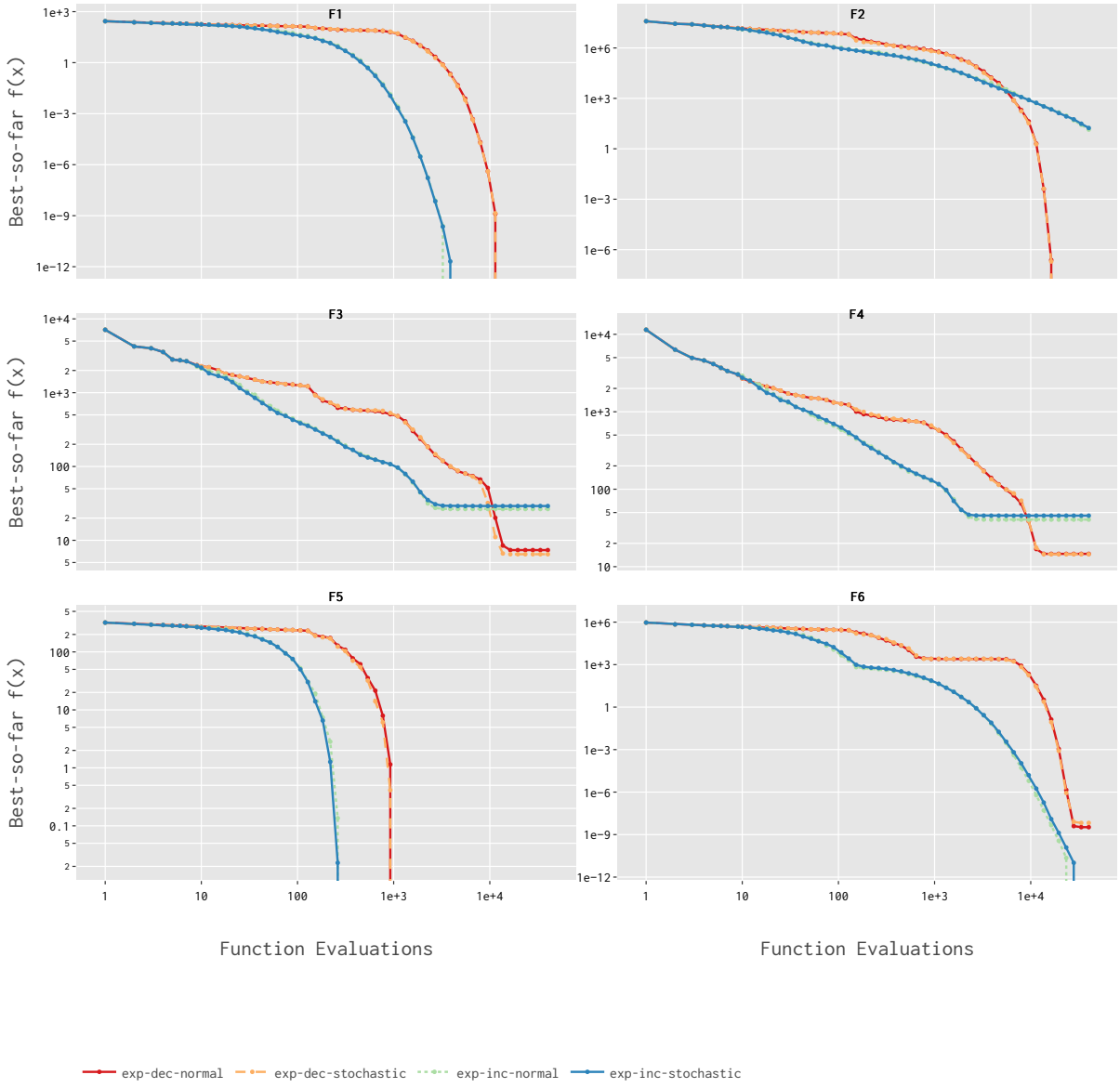


FIGURE 5.3: The effect of the different rounding schemes on the exponential population adaptation methods.

5.2.1 Rounding

As mentioned in the previous chapter, we will now assess the relative performance of normal rounding compared to the stochastic rounding scheme. The experimental setup was the same here as for the previous simple population adaptation runs for the exponential schemes. Thus, there were four different configurations tested - exponential increasing and decreasing, both with normal rounding and stochastic rounding. The results are shown in Fig. 5.3.

We can see clearly that the rounding scheme has no effect on the performance of the algorithm. Both rounding variations have almost identical results for the increasing and decreasing schemes. Thus, we can conclude that the fine-grained nature of the difference between the population size updates has no effect on performance.

To reduce complexity and possible sources of uncertainty, we will choose to only use normal rounding for the subsequent experiments.

5.3 PSA-CMA-ES

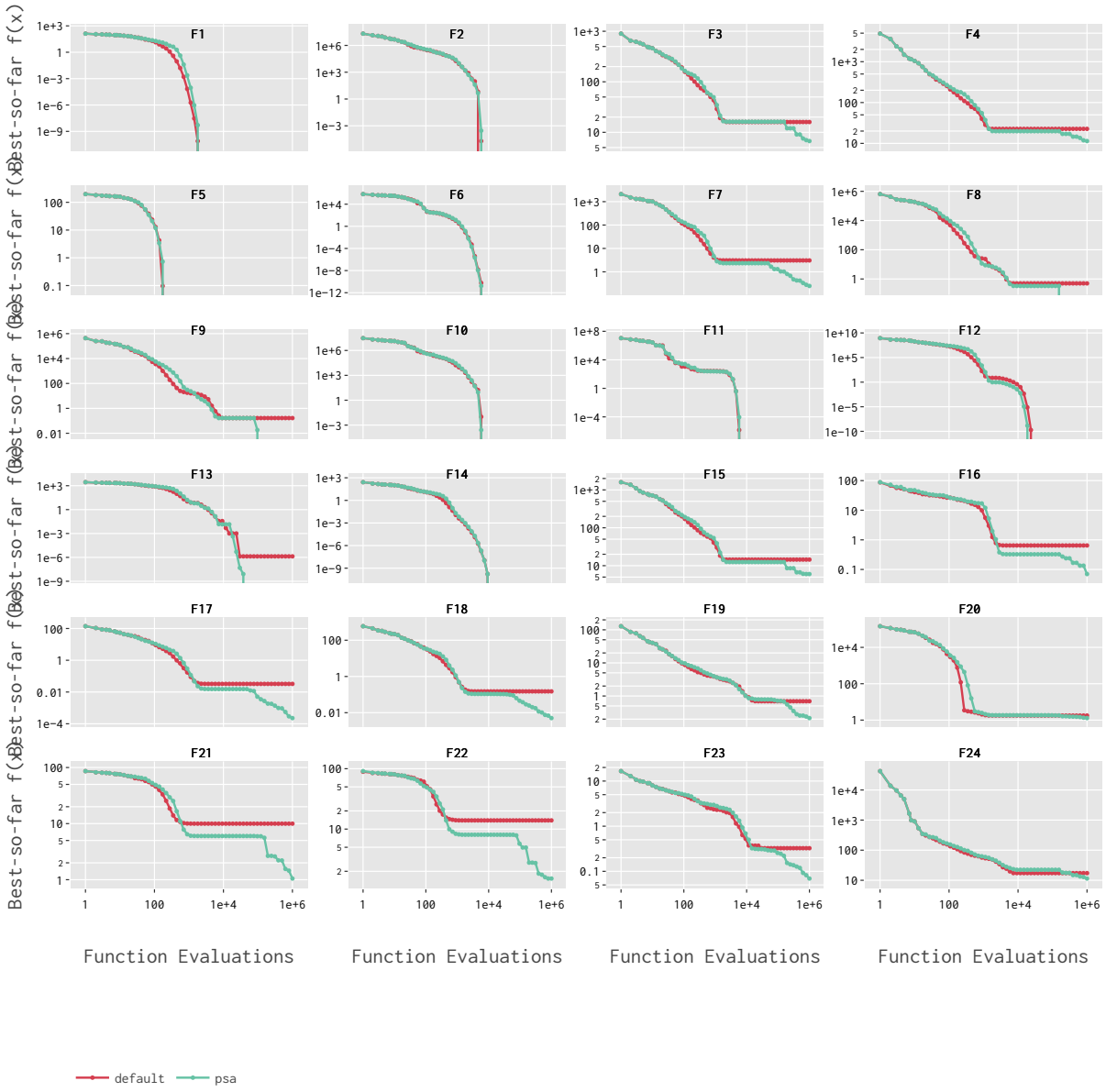


FIGURE 5.4: Fixed-budget analysis of PSA-CMA-ES and default population size on all 24 BBOB functions.

When using PSA-CMA-ES we expect a substantial improvement over the whole portfolio compared to the default population size. While we would like to see an improvement compared to the simple increasing and decreasing schemes, we cannot compare their generalisation capabilities as the simple adaption schemes require extensive domain knowledge to select the optimal method. On the other hand, PSA-CMA-ES retains the quasi-parameter-free nature of CMA-ES as the

same settings can be run on different functions and we hope it will result in good performance without requiring extra tuning.

Before running these experiments, the correctness of the PSA-CMA-ES implementation had to be tested. Data from the benchmarking PSA-CMA-ES paper is available on IOHanalyzer so we can directly compare our own implementation of this algorithm with the original results. A testing summary is shown in Appendix 2. The results for PSA-CMA-ES compared with the default population setting are shown in Figure 5.4.

Importantly, PSA-CMA-ES is never outperformed by the default population size; at worst they reach similar best objective function values. On the simpler functions, primarily separable or unimodal functions, the performance is similar. However, we can start to realise performance gains when the functions are multimodal or weakly structured, and the final precision reached is not arbitrarily small.

For example, on BBOB function 4 which is separable but multimodal, we get increased precision compared to the default CMA-ES. There are often times when we see even more of an increase in performance: functions 9, 13, 16, 17, 21, and 22 all show a benefit of at least an order of magnitude compared to the default static population size. These are comparable to the best naive schemes from before, but we have the benefit of not needing to select and tune the adaption and its associated parameters; it is all done automatically by the algorithm.

5.4 Reinforcement Learning

Finally, we can analyse the performance of our reinforcement learning models. We will look at how the agent performs over the course of the training process, as well as analyse the performance of the best found model and compare its policy to PSA-CMA-ES.

For these results, we chose a small subset of these functions to focus our attention on. We want these to be representative of the whole BBOB problem set while allowing us to perform a more in-depth analysis. The functions chosen are shown below, along with justification for their use [12].

- Function 1: This is the easiest function to solve so can act as a basic point of comparison for subsequent experiments. Furthermore, this will provide the best opportunity for a reinforcement learning agent to be able to learn a similar policy and provide comparable performance.
- Function 5: Simple linear separable function but the optimum is on the domain boundary. This is a simple way of testing how well the algorithm explores the whole function landscape.
- Function 11: Locally irregular and very sensitive to change in only one direction.
- Function 14: Unimodal but can be very sensitive to changes around the optimum. This will test the local search capabilities of the algorithm.
- Function 19: Highly multimodal.
- Function 24: Highly multimodal with weak global structure. This represents the hardest function to solve to good precision in the BBOB portfolio.

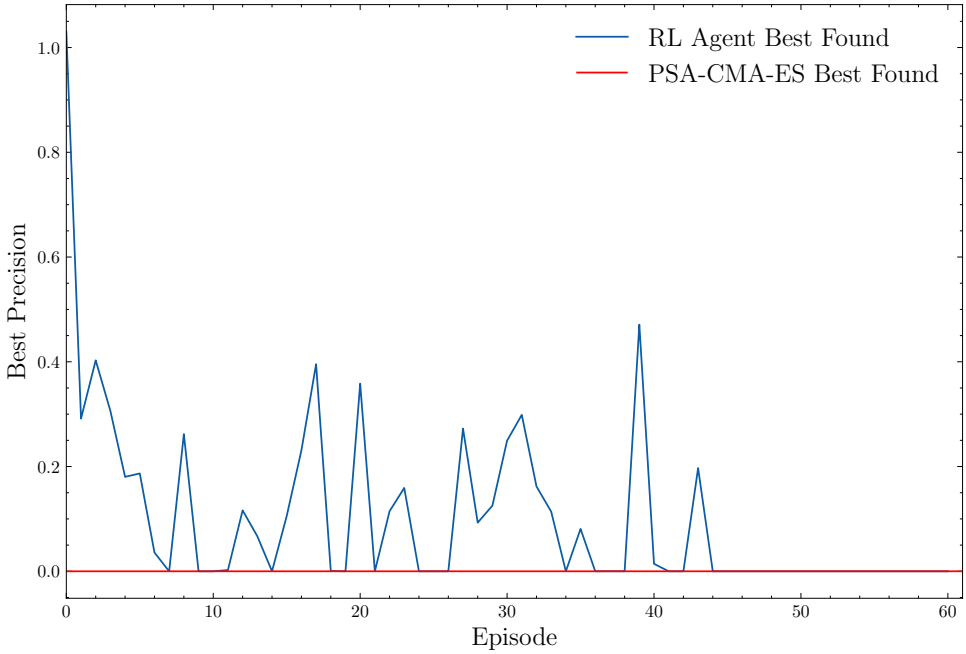


FIGURE 5.5: Learning curve for the RL agent on BBOB function 1. The best precision for each episode is shown. The constant line is the PSA-CMA-ES mean best precision on this function.

5.4.1 Analysis

There are two primary graphs we will use to analyse our reinforcement learning models. The learning curve shows the best-found precision for each episode over the course of the training process. This is compared to the best-found precision for PSA-CMA-ES. This graph will indicate at which point the performance of the model is comparable to PSA-CMA-ES.

The other plot will analyse the behaviour of the best policy found by the reinforcement learning algorithm. Note that this is not necessarily the policy at the end of the training process, but the policy which provides the largest cumulative reward when evaluated. We will show a fixed-budget performance analysis, as well as display the actual variation of the population size over the CMA-ES optimisation procedure for both the RL agent and PSA-CMA-ES.

BBOB Function 1

Figure 5.5 shows the learning curve, and we can see this displays a fairly typical shape. The precision begins comparatively poor, but over time reduces to the point where the policy can consistently reach the same precision as PSA-CMA-ES.

We can also see that the performance of the learned policy is almost identical to PSA-CMA-ES, both in convergence speed and final precision found (see Fig. 5.6). Looking at the variation of the population size over the course of the optimisation procedure, we can see that the policy has rediscovered the actions of PSA-CMA-ES. The population size increases at the start of the optimisation procedure, before dropping to a small value. While the learned policy and PSA-CMA-ES don't behave identically, we can see they share similar characteristics suggesting that the RL agent is able to rediscover some of the behaviour of PSA-CMA-ES.

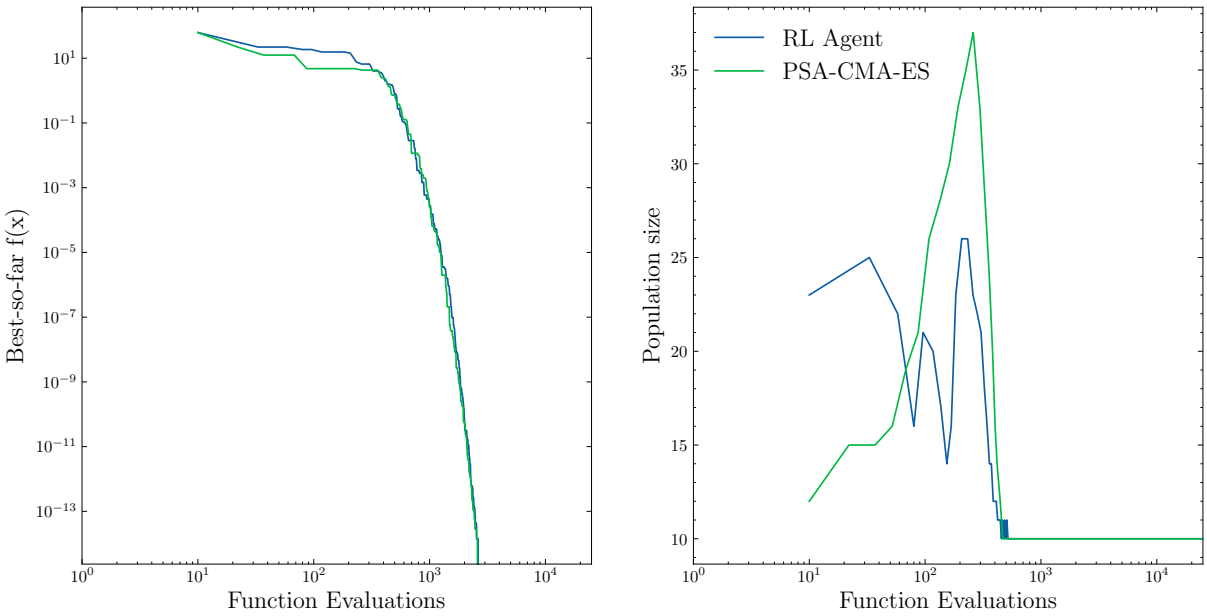


FIGURE 5.6: Reinforcement learning agent policy and PSA-CMA-ES for BBOB Function 1. Left panel: Fixed-budget analysis (best f(x) against function evaluations). Right panel: The population size adaptation for the RL agent and PSA-CMA-ES.

BBOB Function 5

The learning curve for this function is not very enlightening - all of the episodes successfully found the function's optimum regardless of the policy so it was difficult to track the training progress. This can be seen in Fig. 5.7, where the precision of PSA-CMA-ES and the RL agent are both identical.

We can see from the policy evaluation that the best found model did indeed find the global optimum with similar performance to PSA-CMA-ES as shown in Fig. 5.8. However, when looking at the actual population size policy, it is nothing like the behaviour exhibited by PSA-CMA-ES. The best learned policy jumps around erratically, especially throughout the second half of the optimisation procedure, in comparison to the smooth change of PSA-CMA-ES.

BBOB Function 11

The learning curve for this function (Fig. 5.9) displays some very interesting behaviour. The performance at the start of training is poor and slowly gets better over time, as we would expect. However, we see after around 500 episodes, the best precision reached changes very suddenly and the agent is able to solve the function exactly at every episode after this point.

While the agent is able to solve the function to an identical final precision, we can see that the learned policy is again very different to PSA-CMA-ES (Fig. 5.10). The agent keeps the population size constant at the default value over the whole optimisation procedure, while PSA-CMA-ES has an increase in population at the start before dropping to the default. While the two policies are very different, they perform very similarly.

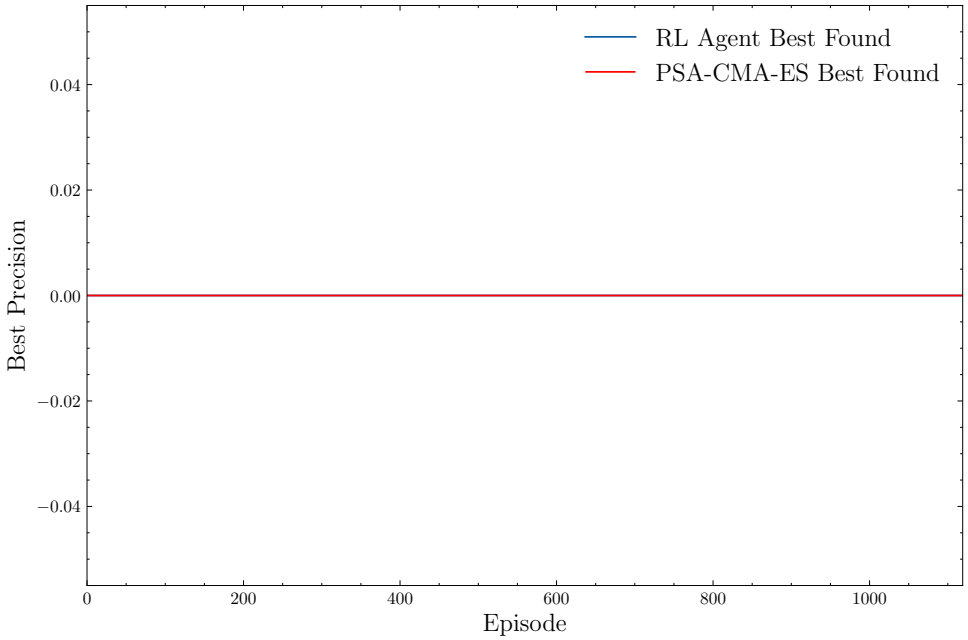


FIGURE 5.7: Learning curve for the RL agent on BBOB function 5. The best precision for each episode is shown. The constant line is the PSA-CMA-ES mean best precision on this function.

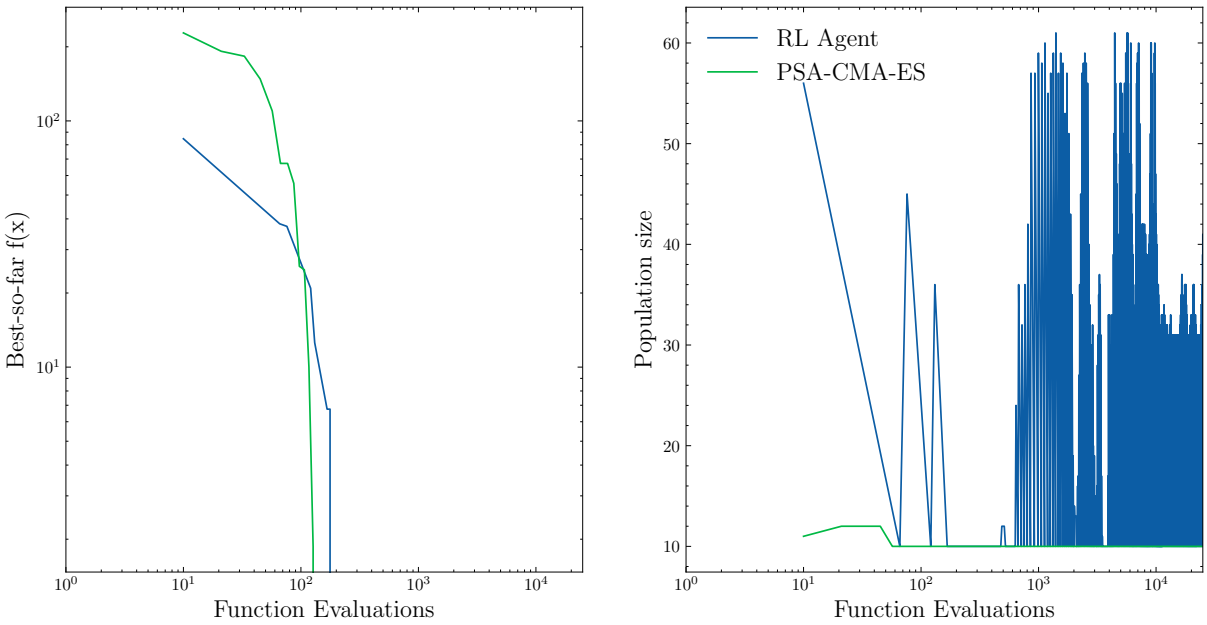


FIGURE 5.8: Reinforcement learning agent policy and PSA-CMA-ES for BBOB Function 5. Left panel: Fixed-budget analysis (best f(x) against function evaluations). Right panel: The population size adaptation for the RL agent and PSA-CMA-ES.

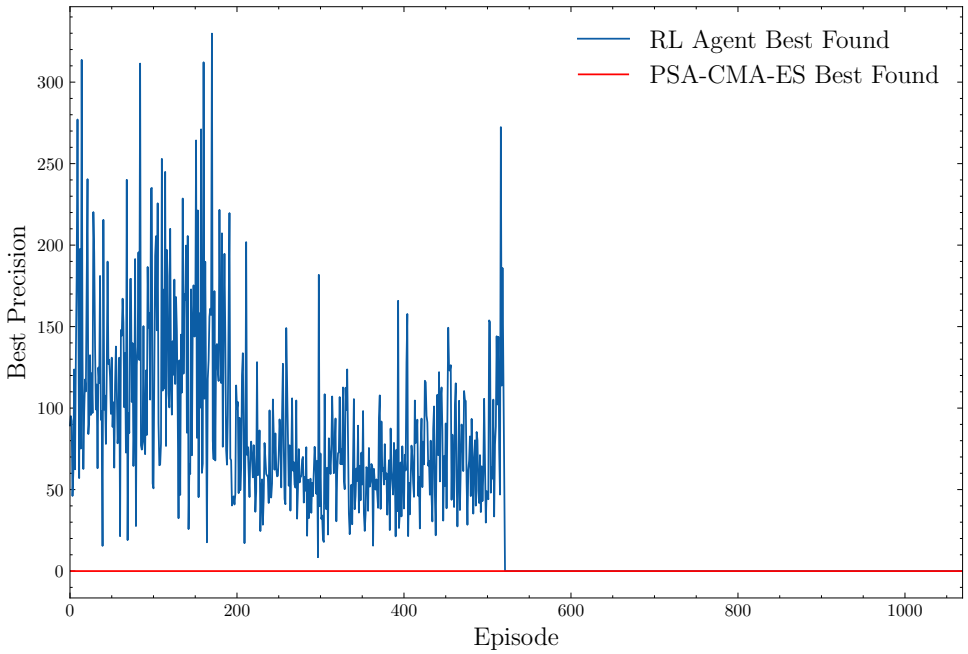


FIGURE 5.9: Learning curve for the RL agent on BBOB function 11. The best precision for each episode is shown. The constant line is the PSA-CMA-ES mean best precision on this function.

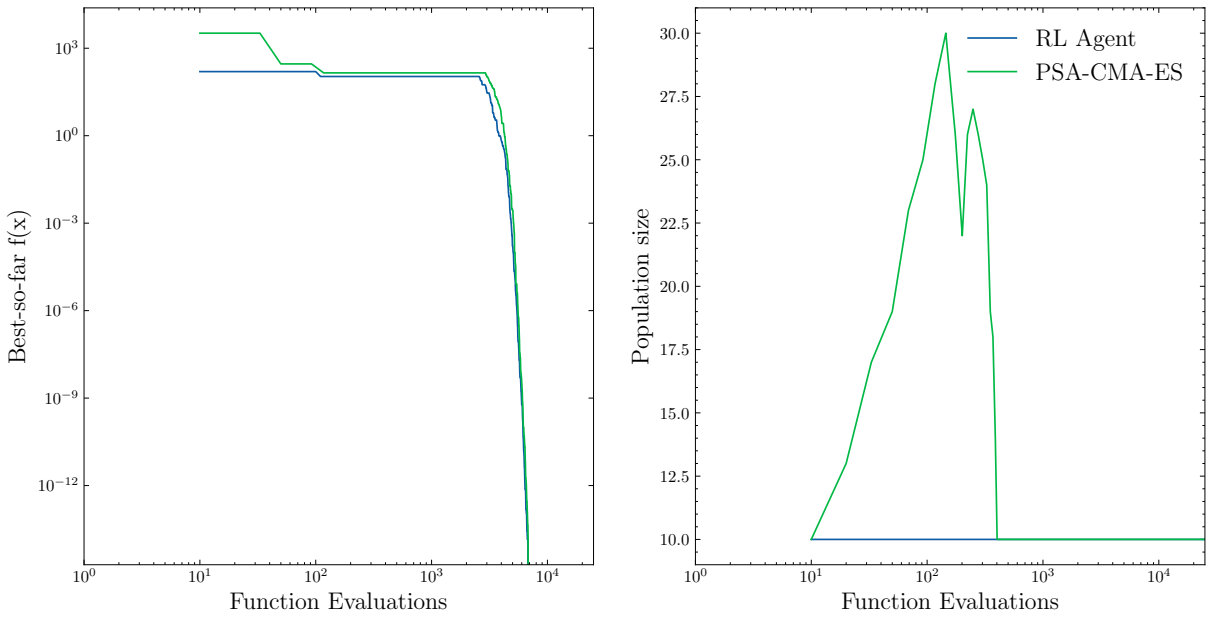


FIGURE 5.10: Reinforcement learning agent policy and PSA-CMA-ES for BBOB Function 11. Left panel: Fixed-budget analysis (best f(x) against function evaluations). Right panel: The population size adaptation for the RL agent and PSA-CMA-ES.

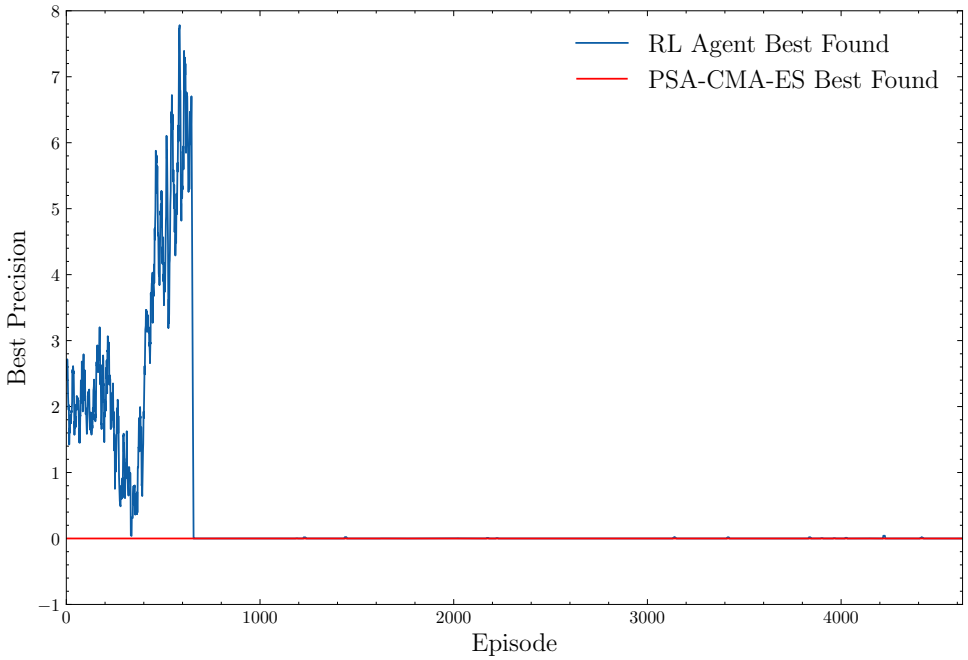


FIGURE 5.11: Learning curve for the RL agent on BBOB function 14. The best precision for each episode is shown. The constant line is the PSA-CMA-ES mean best precision on this function.

BBOB Function 14

Function 14 displays similar learning behaviour compared to the previous function (see Figure 5.11). There is a region of training where the policy struggles to get anywhere near the true optimum, but then suddenly jumps to near perfect solutions every episode after a certain point. It is possible to see slight variation in the solution quality with careful analysis, indicating that is it not entirely consistent after this point.

The best found policy doesn't perform as well: PSA-CMA-ES has a final precision of $\sim 10^{-13}$, while the RL agent only manages a solution quality of $\sim 10^{-5}$, as shown in Fig. 5.12. We can see that the policy learned is erratic and very different to the actions performed by PSA-CMA-ES. As this function is overall more difficult to solve, it is not a surprise that a suboptimal policy cannot solve the function to a similar precision than an expert hand-crafted method.

BBOB Function 19

In Figure 5.13, we can again see a learning curve with two distinct regions. At the start of training the RL policy is significantly worse than PSA-CMA-ES. Then, the policy suddenly gets drastically better to around the same quality as the baseline. Although not every subsequent episode does not have exactly the same precision, we can see that they all lie in a narrow range with a much smaller deviation than in the previous regime.

Seeing this behaviour in the learning curve for multiple different functions suggests that it is not a coincidence. This sharp drop is reminiscent of a phase transition, a concept from statistical physics, but has been studied in the context of deep neural networks [18, 35]. This is where the competition between training error and model complexity can lead to discontinuous changes in the model's performance. Looking into this behaviour would be a route for future work.

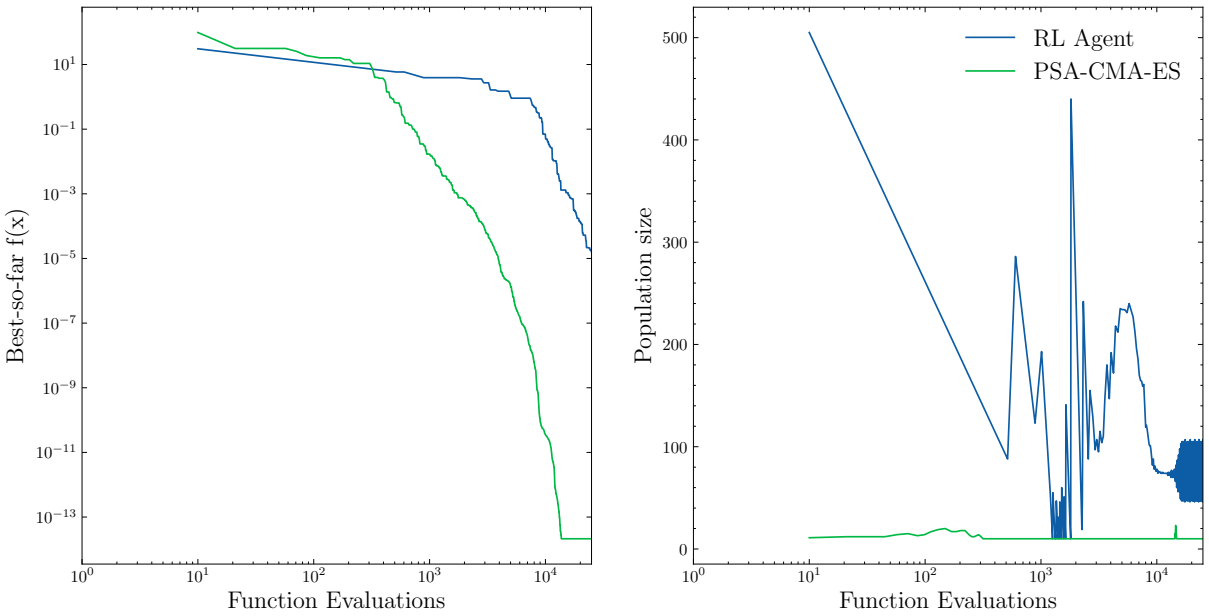


FIGURE 5.12: Reinforcement learning agent policy and PSA-CMA-ES for BBOB Function 14. Left panel: Fixed-budget analysis (best f(x) against function evaluations). Right panel: The population size adaptation for the RL agent and PSA-CMA-ES.

The learned policy for this function performs very differently to PSA-CMA-ES. As we can see in Figure 5.14 The best precision reached is worse than PSA-CMA-ES, but better than default static population size. The population size varies drastically over the course of the optimisation procedure, and does not replicate the behaviour seen in PSA-CMA-ES.

BBOB Function 24

Function 24 is the hardest function to solve, and showed no general trend of learning over the training process. As we can see in Figure 5.15, the best precision essentially varies randomly over the course of training with no real trend towards improving precision. Furthermore, the average precision of the RL agent is significantly higher than the precision found by PSA-CMA-ES. Thus, we can expect the best found policy to essentially behave randomly as the agent does not seem to have learned a beneficial policy.

We can see this to be the case by inspecting Fig. 5.16. Although the final precision is comparable to PSA-CMA-ES, the population size policy is very erratic with no particular structure. It changes from a large population to a much smaller one every generation. This is consistent with the agent struggling to learn a performant policy.

5.4.2 Summary

In this section, we have studied the learning curves and evaluated the best policies of the reinforcement learning agent on a subset of the BBOB function portfolio. On some functions the agent has learned policies with similar performance to PSA-CMA-ES, while others have performed significantly worse. Additionally, we have inspected the actual population size policy of the reinforcement learning agent. While equivalent performance has been attained by the RL

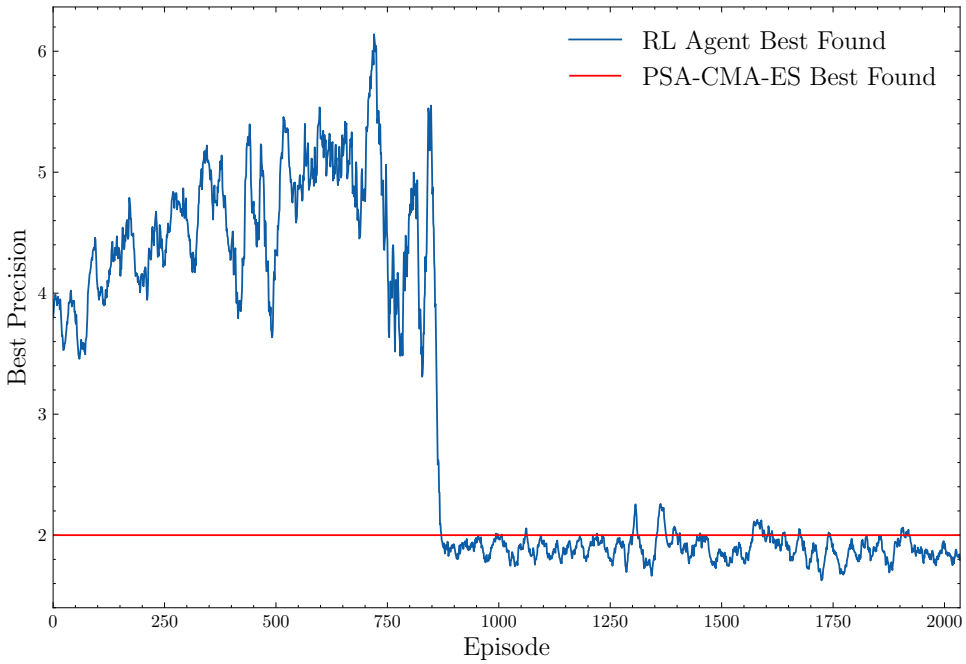


FIGURE 5.13: Learning curve for the RL agent on BBOB function 19. The best precision for each episode is shown. The constant line is the PSA-CMA-ES mean best precision on this function.

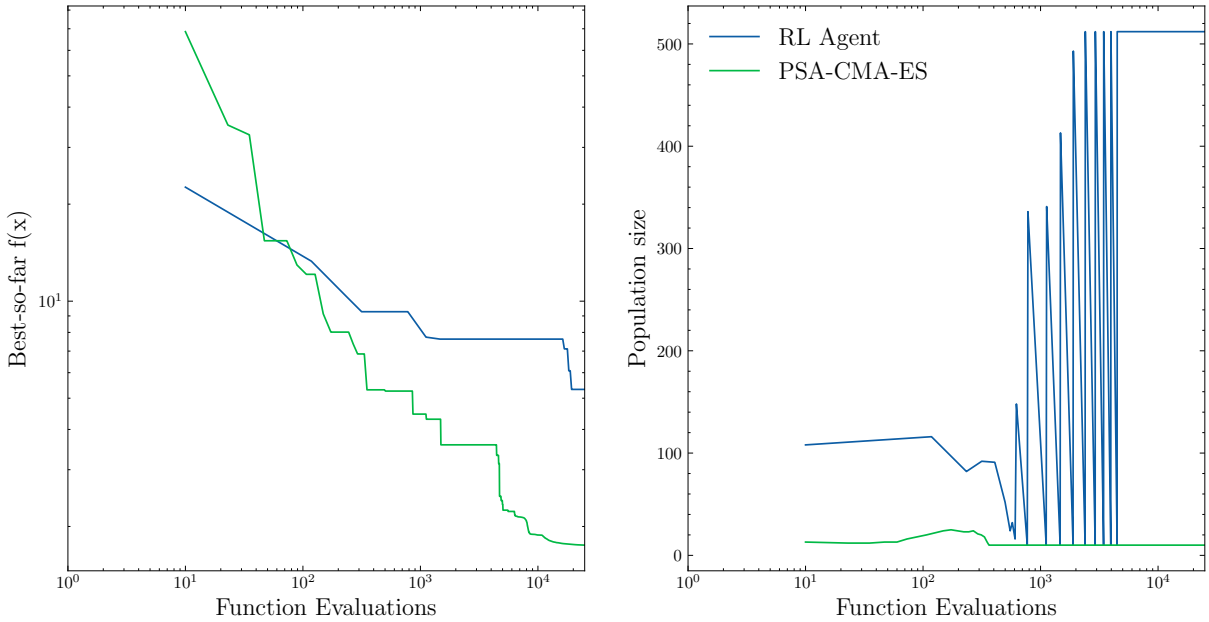


FIGURE 5.14: Reinforcement learning agent policy and PSA-CMA-ES for BBOB Function 19. Left panel: Fixed-budget analysis (best f(x) against function evaluations). Right panel: The population size adaptation for the RL agent and PSA-CMA-ES.

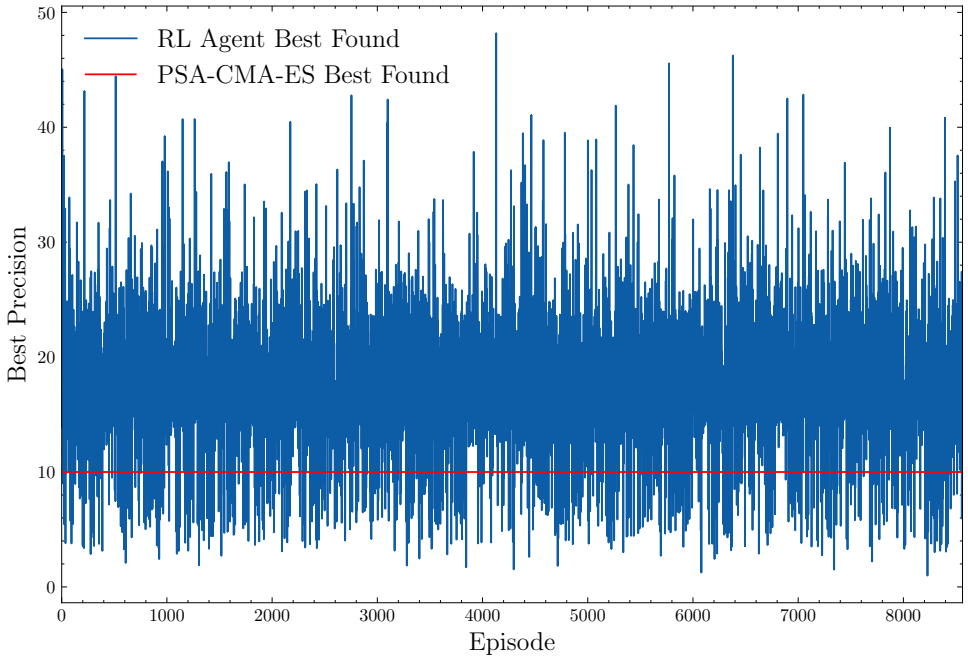


FIGURE 5.15: Learning curve for the RL agent on BBOB function 24. The best precision for each episode is shown. The constant line is the PSA-CMA-ES mean best precision on this function.

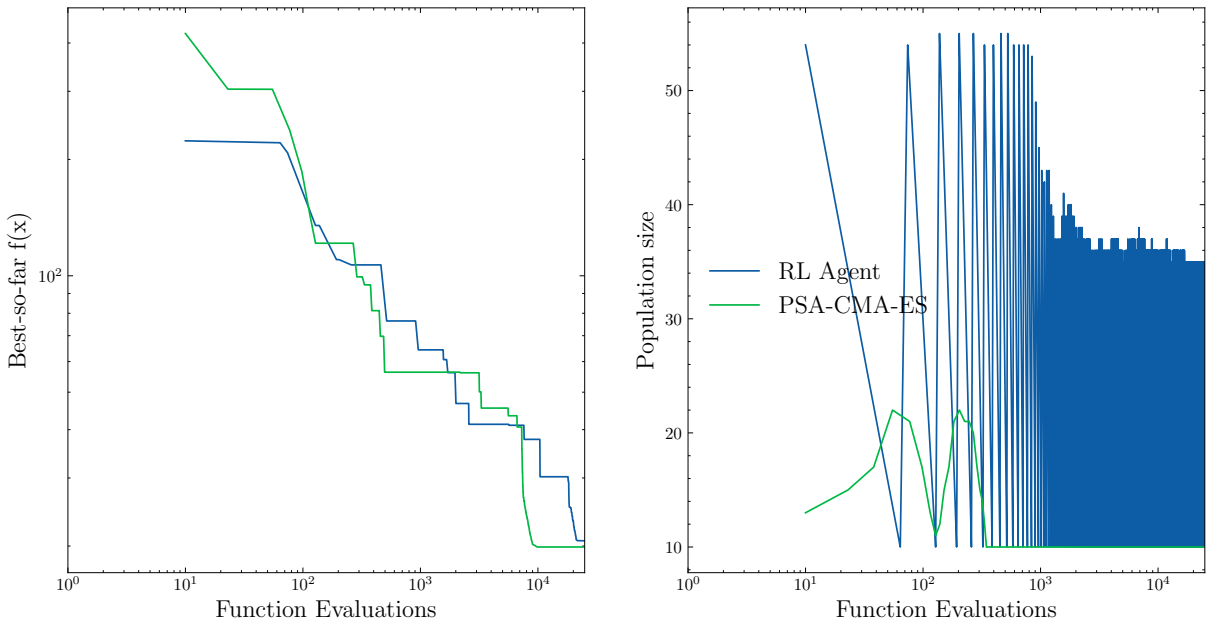


FIGURE 5.16: Reinforcement learning agent policy and PSA-CMA-ES for BBOB Function 24. Left panel: Fixed-budget analysis (best f(x) against function evaluations). Right panel: The population size adaptation for the RL agent and PSA-CMA-ES.

approach, its control of the population size does not seem to replicate the actions of PSA-CMA-ES, with the exception of BBOB function 1.

Furthemore, we have noted some interesting properties of the learning curves of some of the functions. They seem to display two regimes: one where the performance of the agent is poor, and another where the performance is comparable to PSA-CMA-ES. These two regimes are separated by a sharp change in the performance, rather than a gradual learning process.

Chapter 6

Conclusion

In this chapter, we will summarise the project and its scientific contributions. We will also reflect upon the research aims proposed when this project began. Finally, areas for future work will be investigated.

6.1 Summary

This project started with benchmarking the performance of CMA-ES on the BBOB problem set. We used the existing ModularCMAES code and IOHprofiler to do this, building a comprehensive understanding of the population size in CMA-ES.

Following this, simple adaptive population size schemes were integrated into the ModularCMAES code. These were also benchmarked on the same problem set and we found significant performance gains could be realised using on-the-fly population size adaptation. We investigated which schemes are the most beneficial and how sensitive to implementation details the performance is.

The current state-of-the-art for population size adaption, PSA-CMA-ES was also reimplemented into the ModularCMAES environment. This reimplementation was difficult and required thorough testing to ensure the correctness of this implementation. As with previous experiments, we benchmarked this algorithm on the BBOB function portfolio, and we found it can have considerable performance gains compared to previous approaches. This method also has the benefit of retaining the quasi-parameter-free nature of default CMA-ES, so no further tuning is required. This is in contrast to the simple adaptive schemes which require problem and algorithm knowledge to select the best method and strategy parameters.

Finally, a reinforcement learning environment was created to allow an RL agent to control the population size for ModularCMAES. This required careful design of the action space, observation space, and reward function such that the agent will have the greatest chance of successfully learning. We found that a simple reinforcement learning agent can effectively control the population size on the easiest BBOB function (f1) and exhibits a comparable policy to PSA-CMA-ES, suggesting a reinforcement learning agent has the capability to rediscover the behaviour shown by PSA-CMA-ES. On several functions, the performance of the model was comparable to PSA-CMA-ES, but this was not the case in general. Additionally, the population size control policy was not comparable to the actions performed by PSA-CMA-ES.

6.2 Reflection

We can now reflect on the outcome of this work with respect to the original aims. The primary aims of the project were:

1. Implement and analyse the effectiveness of the currently proposed hand-crafted rule on a variety of standard benchmarking problems for continuous black-box optimisation.

   This has been completed successfully as we have benchmarked CMA-ES and its variations, including the current state-of-the-art, PSA-CMA-ES, on the BBOB problem portfolio.

2. Investigate the possibility of learning a population-size control policy for each benchmark in a data-driven fashion via the use of a state-of-the-art off-the-shelf deep reinforcement learning approach. Integrate this into the existing ModularCMAES implementation.

   We have successfully used the ModularCMAES code in a reinforcement learning environment and learned population size control policies for the BBOB functions.

3. Compare performance to hand-crafted population size control schemes to see if it is possible to attain similar or better performance.

   While we have not produced policies with comparable performance to state-of-the-art techniques over the whole portfolio, we have shown that a policy behaving similarly to PSA-CMA-ES can be learned for certain simple individual functions.

Furthermore, there were some secondary objectives identified at the start of the project:

1. Understand how the population size parameter affects the behaviour of the CMA-ES algorithm across different benchmark problems.

   From the variety of experiments performed, we have gained a comprehensive understanding of how population size affects performance in CMA-ES.

2. Improve the quality of the learnt policies by the deep reinforcement learning approach via the use of hyperparameter optimisation and automated reinforcement learning (Auto-RL).

   No hyperparameter tuning approaches were used in this project. This is an area which would benefit greatly from future work.

3. Study the ability of learning across different benchmarks with deep-RL and compare state-of-the-art deep-RL approaches on the DAC problem and analyse their strengths and weaknesses.

   We have only studied one problem set (BBOB) and one algorithm (TD3) in this work. This is again an area for future research.

6.3 Future Work

Whilst we have produced some interesting and useful results in this project, there are multiple avenues to explore for future work.

A straightforward and obvious extension is to apply hyperparameter optimisation techniques so the reinforcement learning algorithm used. There are a wide variety of hyperparameters to be tuned in TD3, so the performance of the learned policy is likely going to be improved by intelligent tuning of these parameters.

Another approach would be to consider different observation spaces, action spaces, and reward functions. Alternatives to the ones used in this project have been discussed but other options could include using discretised population size. Additionally, alternative observation spaces could be considered, such as providing the agent with less processed, or entirely raw data.

An extension to this would be to consider the algorithm choice. While TD3 is a sensible choice for continuous control, discretising the population size would require a different algorithm which might provide better performance. Furthermore, an approach using guided policy search (GPS) such as the method used for controlling the step size [30] could be applied, using PSA-CMA-ES as the teacher.

A very beneficial modification would be to generalise the policy to work for multiple functions. In this paper, we trained each policy on one BBOB function so adapting this approach to multiple functions would be very useful. This would also make it possible to test the generalisation capability of the learned policy to unseen functions - a feature which would provide significant benefit to practical applications of CMA-ES.

Finally, investigating the causes of the sudden change in performance shown in the learning curves could be undertaken. Further experimentation may show which particular characteristics of the function being optimised, or the reinforcement learning environment itself cause this change. Alternatively, a mathematical analysis could be performed. This may help to show any underlying structure in the policy landscape which we hope can help with performance or generalisation capabilities.

Appendix 1 - BBOB function landscapes

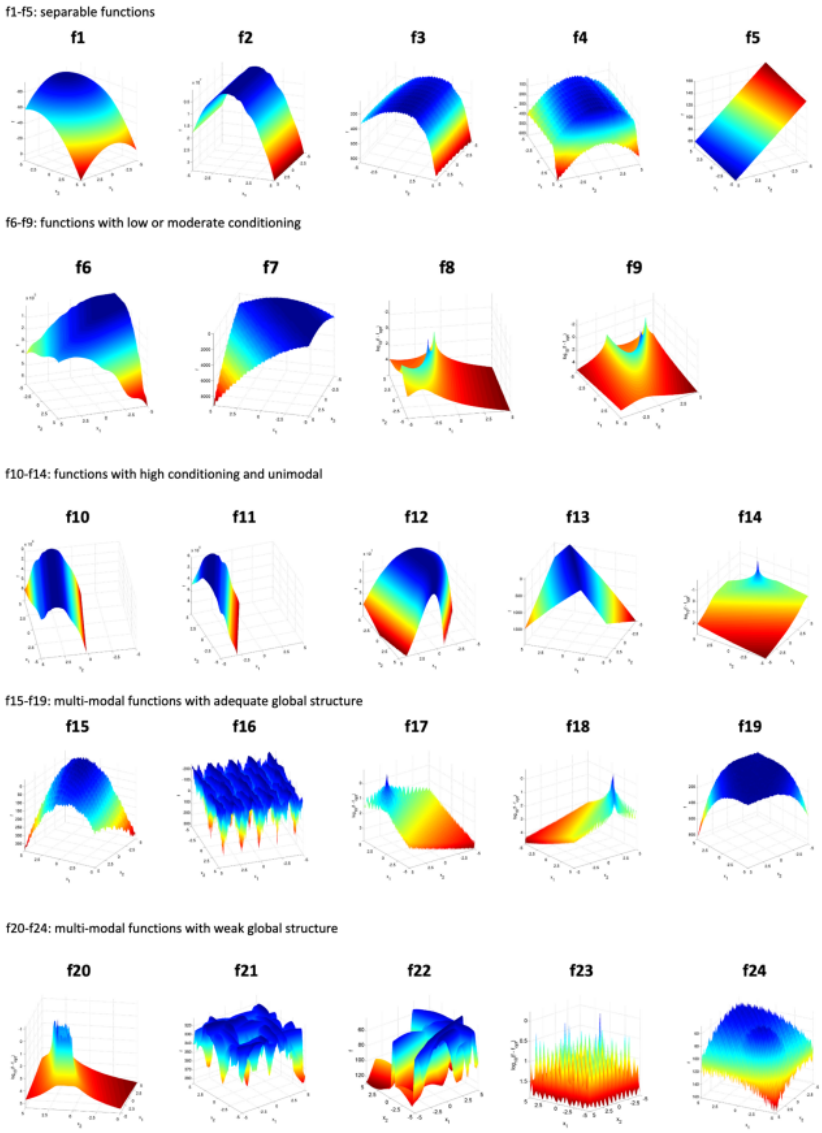
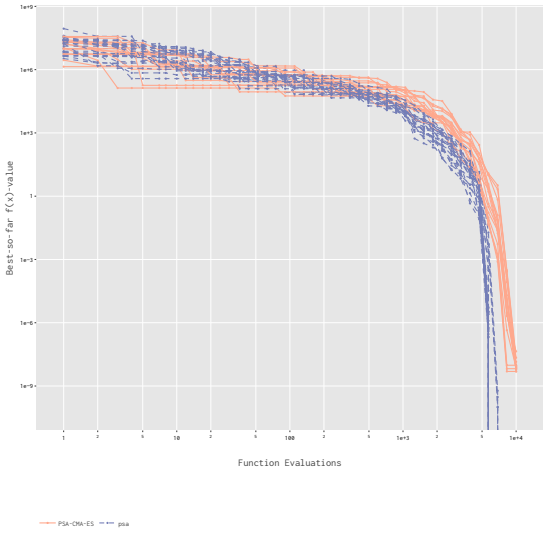


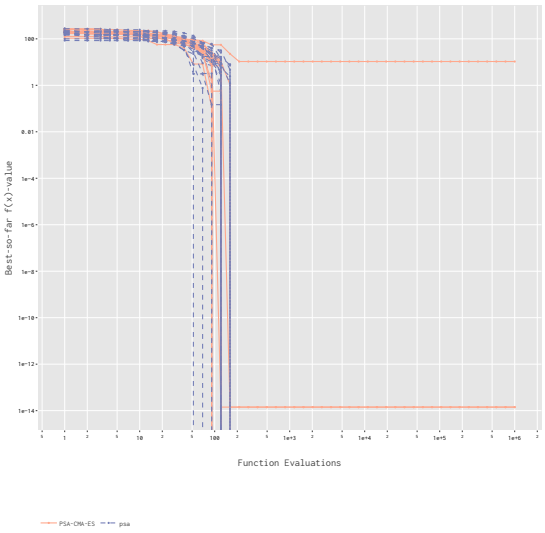
FIGURE 6.1: Solution landscapes for the BBOB function portfolio shown by 3D surface plots with $d = 2$ [12].

Appendix 2 - Testing PSA-CMA-ES Reimplementation

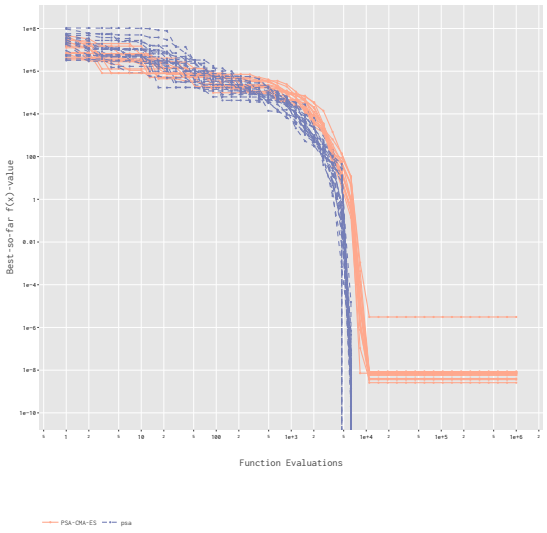
Here we display the comparison of individual runs between the reimplementation of PSA-CMA-ES with the data loaded on IOHanalyzer from the original paper. These are displayed on a subset of the BBOB function portfolio. The differences between the two datasets are likely due to implementation details; the original implementation was not based on the ModularCMAES code. This can account for a slight variation between the two approaches. It is clear that although there are differences between the individual runs, they behave very similarly overall.



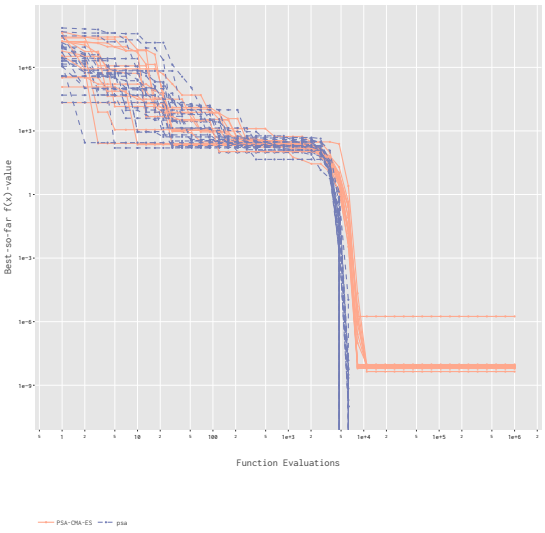
(A) BBOB function 2



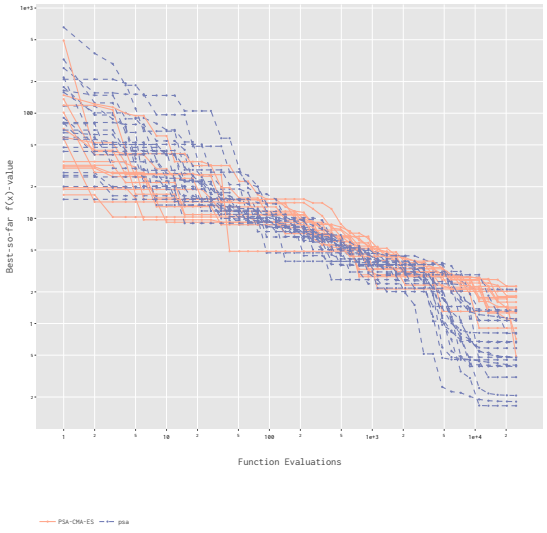
(B) BBOB function 5.



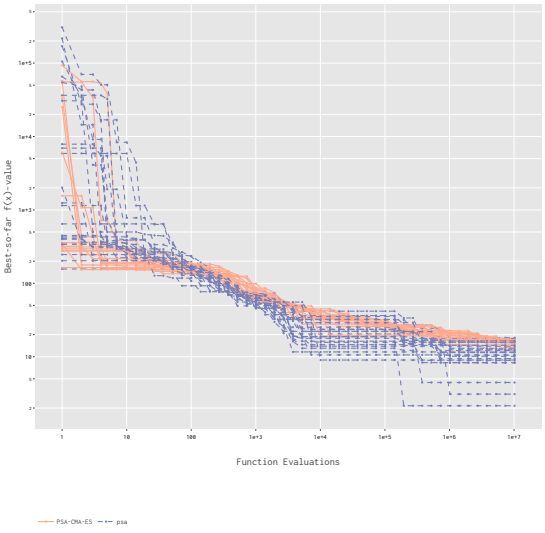
(C) BBOB function 10.



(D) BBOB function 11.



(E) BBOB function 19.



(F) BBOB function 24.

FIGURE 6.2: Fixed-budget analysis of individual runs of data from the original PSA-CMA-ES paper and our implementation. Data labelled 'psa' is our implementation, and data labelled 'PSA-CMA-ES' is the data from IOHanalyzer.

Appendix 3 - Hardware

The experiments concerning the fixed population sizes, simple adaptive population sizes, and PSA-CMA-ES were performed on my laptop with an AMD Ryzen 5 4500U CPU.

The reinforcement learning agents were trained on a St Andrews School of Computer Science GPU client with an NVIDIA GeForce RTX 3060 GPU.

Bibliography

[1] Anne Auger and Nikolaus Hansen. "A restart CMA evolution strategy with increasing population size". In: *2005 IEEE congress on evolutionary computation*. Volume 2. IEEE. 2005, pages 1769–1776.

[2] Abid Ali Awan. *Reinforcement learning for newbies*. May 2022. URL: https://www.kdnuggets.com/2022/05/reinforcement-learning-newbies.html.

[3] Galen Bollinger. "Book Review: Regression Diagnostics: Identifying Influential Data and Sources of Collinearity". In: *Journal of Marketing Research* 18.3 (1981), pages 100–104. DOI: 10.1177/002224378101800318. URL: https://doi.org/10.1177/002224378101800318.

[4] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. *OpenAI Gym*. 2016. eprint: arXiv:1606.01540.

[5] C. S. Davis and M. A. Stephens. "The Covariance Matrix of Normal Order Statisticss". In: *Statist* (1978), pages 206–212. URL: https://purl.stanford.edu/qx112mz6742.

[6] Carola Doerr, Hao Wang, Furong Ye, Sander van Rijn, and Thomas Bäck. *IOHprofiler: A Benchmarking and Profiling Tool for Iterative Optimization Heuristics*. 2018. arXiv: 1810.05281 [cs.NE].

[7] Carola Doerr, Furong Ye, Naama Horesh, Hao Wang, Ofer M. Shir, and Thomas Bäck. "Benchmarking discrete optimization heuristics with IOHprofiler". In: *Appl. Soft Comput.* 88 (2020), page 106027. DOI: 10.1016/j.asoc.2019.106027. URL: https://doi.org/10.1016/j.asoc.2019.106027.

[8] T. Eimer, A. Biedenkapp, M. Reimer, S. Adriaensen, F. Hutter, and M. Lindauer. "DACBench: A Benchmark Library for Dynamic Algorithm Configuration". In: *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence (IJCAI'21)*. ijcai.org, Aug. 2021.

[9] Scott Fujimoto, Herke van Hoof, and David Meger. *Addressing Function Approximation Error in Actor-Critic Methods*. 2018. arXiv: 1802.09477 [cs.AI].

[10] N. Hansen. "The CMA evolution strategy: a comparing review". In: *Towards a new evolutionary computation. Advances on estimation of distribution algorithms*. Edited by J.A. Lozano, P. Larranaga, I. Inza, and E. Bengoetxea. Springer, 2006, pages 75–102.

[11] Nikolaus Hansen. *The CMA Evolution Strategy: A Tutorial*. 2023. arXiv: 1604.00772 [cs.LG].

[12] Nikolaus Hansen, Anne Auger, Raymond Ros, Steffen Finck, and Petr Pok. "Comparing Results of 31 Algorithms from the Black-Box Optimization Benchmarking BBOB-2009". In: *Proceedings of the 12th Annual Conference Companion on Genetic and Evolutionary Computation*. GECCO '10. Portland, Oregon, USA: Association for Computing Machinery, 2010, pages 1689–1696. ISBN: 9781450300735. DOI: 10.1145/1830761.1830790. URL: https://doi.org/10.1145/1830761.1830790.

[13] Nikolaus Hansen, Dimo Brockhoff, Olaf Mersmann, Tea Tusar, Dejan Tusar, Ouassim Ait ElHara, Phillipe R. Sampaio, Asma Atamna, Konstantinos Varelas, Umut Batu, Duc Manh Nguyen, Filip Matzner, and Anne Auger. *COmparing Continuous Optimizers: numbbo/COCO*

on Github. Version v2.3. Mar. 2019. DOI: 10.5281/zenodo.2594848. URL: https://doi.org/10.5281/zenodo.2594848.

[14] Nikolaus Hansen, Steffen Finck, Raymond Ros, and Anne Auger. *Real-Parameter Black-Box Optimization Benchmarking 2009: Noiseless Functions Definitions*. Research Report RR-6829. INRIA, 2009. URL: https://inria.hal.science/inria-00362633.

[15] Nikolaus Hansen and Stefan Kern. "Evaluating the CMA evolution strategy on multimodal test functions". In: *International conference on parallel problem solving from nature*. Springer. 2004, pages 282–291.

[16] H. Leon Harter. "Expected Values of Normal Order Statistics". In: *Biometrika* 48.1/2 (1961), pages 151–165. ISSN: 00063444. URL: http://www.jstor.org/stable/2333139 (visited on 08/12/2023).

[17] Michael Hellwig and Hans-Georg Beyer. "Evolution under strong noise: A self-adaptive evolution strategy can reach the lower performance bound-the pccmsa-es". In: *Parallel Problem Solving from Nature–PPSN XIV: 14th International Conference, Edinburgh, UK, September 17-21, 2016, Proceedings*. Springer. 2016, pages 26–36.

[18] Wolfgang Kinzel. "Phase transitions of neural networks". In: *Philosophical Magazine B* 77.5 (May 1998), pages 1455–1477. DOI: 10.1080/13642819808205038. URL: https://doi.org/10.1080%2F13642819808205038.

[19] Serge Lang. *Algebra*. English. 3. ed. Reading, MA: Addison Wesley, 1993, pages 240–241. ISBN: 0-201-55540-9.

[20] Ilya Loshchilov, Marc Schoenauer, and Michele Sèbag. "Bi-Population CMA-ES Agorithms with Surrogate Models and Line Searches". In: *Proceedings of the 15th Annual Conference Companion on Genetic and Evolutionary Computation*. GECCO '13 Companion. Amsterdam, The Netherlands: Association for Computing Machinery, 2013, pages 1177–1184. ISBN: 9781450319645. DOI: 10.1145/2464576.2482696. URL: https://doi.org/10.1145/2464576.2482696.

[21] Nicolai A. Lynnerup, Laura Nolling, Rasmus Hasle, and John Hallam. *A Survey on Reproducibility by Evaluating Deep Reinforcement Learning Algorithms on Real-World Robots*. 2019. arXiv: 1909.03772 [cs.LG].

[22] Duc Manh Nguyen and Nikolaus Hansen. "Benchmarking CMAES-APOP on the BBOB Noiseless Testbed". In: *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. GECCO '17. Berlin, Germany: Association for Computing Machinery, 2017, pages 1756–1763. ISBN: 9781450349390. DOI: 10.1145/3067695.3084207. URL: https://doi.org/10.1145/3067695.3084207.

[23] Kouhei Nishida and Youhei Akimoto. "Benchmarking the PSA-CMA-ES on the BBOB Noiseless Testbed". In: *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. GECCO '18. Kyoto, Japan: Association for Computing Machinery, 2018, pages 1529–1536. ISBN: 9781450357647. DOI: 10.1145/3205651.3208297. URL: https://doi.org/10.1145/3205651.3208297.

[24] Kouhei Nishida and Youhei Akimoto. "PSA-CMA-ES: CMA-ES with Population Size Adaptation". In: *Proceedings of the Genetic and Evolutionary Computation Conference*. GECCO '18. Kyoto, Japan: Association for Computing Machinery, 2018, pages 865–872. ISBN: 9781450356183. DOI: 10.1145/3205455.3205467. URL: https://doi.org/10.1145/3205455.3205467.

[25] Jacob de Nobel, Furong Ye, Diederick Vermetten, Hao Wang, Carola Doerr, and Thomas Bäck. *IOHexperimenter: Benchmarking Platform for Iterative Optimization Heuristics*. 2022. arXiv: 2111.04077 [cs.NE].

[26] Masahiro Nomura, Youhei Akimoto, and Isao Ono. "CMA-ES with Learning Rate Adaptation: Can CMA-ES with Default Population Size Solve Multimodal and Noisy Problems?" In: *Proceedings of the Genetic and Evolutionary Computation Conference*. GECCO '23. Lisbon, Portugal: Association for Computing Machinery, 2023, pages 839–847. DOI: 10.1145/3583131.3590358. URL: https://doi.org/10.1145/3583131.3590358.

[27] OpenAI. *Welcome to spinning up in deep RL!* URL: https://spinningup.openai.com/en/latest/.

[28] Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. "Stable-Baselines3: Reliable Reinforcement Learning Implementations". In: *Journal of Machine Learning Research* 22.268 (2021), pages 1–8. URL: http://jmlr.org/papers/v22/20-1364.html.

[29] Yoshitaka Sakurai, Kouhei Takada, Takashi Kawabe, and Setsuo Tsuruta. "A Method to Control Parameters of Evolutionary Algorithms by Using Reinforcement Learning". In: *2010 Sixth International Conference on Signal-Image Technology and Internet Based Systems*. 2010, pages 74–79. DOI: 10.1109/SITIS.2010.22.

[30] Gresa Shala, André Biedenkapp, Noor Awad, Steven Adriaensen, Marius Lindauer, and Frank Hutter. "Learning Step-Size Adaptation In CMA-ES". In: *Parallel Problem Solving from Nature PPSN XVI: 16th International Conference, PPSN 2020, Leiden, The Netherlands, September 5-9, 2020, Proceedings, Part I*. Leiden, The Netherlands: Springer-Verlag, 2020, pages 691–706. ISBN: 978-3-030-58111-4. DOI: 10.1007/978-3-030-58112-1_48. URL: https://doi.org/10.1007/978-3-030-58112-1_48.

[31] Mudita Sharma, Alexandros Komninos, Manuel Lopez Ibanez, and Dimitar Kazakov. *Deep Reinforcement Learning Based Parameter Control in Differential Evolution*. 2019. arXiv: 1905.08006 [cs.NE].

[32] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: A Bradford Book, 2018. ISBN: 0262039249.

[33] Hao Wang, Diederick Vermetten, Furong Ye, Carola Doerr, and Thomas Bäck. "IOHanalyzer: Detailed Performance Analyses for Iterative Optimization Heuristics". In: *ACM Trans. Evol. Learn. Optim.* 2.1 (Apr. 2022). ISSN: 2688-299X. DOI: 10.1145/3510426. URL: https://doi.org/10.1145/3510426.

[34] Thomas Weise and Zijun Wu. "Difficult Features of Combinatorial Optimization Problems and the Tunable W-Model Benchmark Problem for Simulating Them". In: *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. GECCO '18. Kyoto, Japan: Association for Computing Machinery, 2018, pages 1769–1776. ISBN: 9781450357647. DOI: 10.1145/3205651.3208240. URL: https://doi.org/10.1145/3205651.3208240.

[35] Liu Ziyin and Masahito Ueda. *Exact Phase Transitions in Deep Learning*. 2022. arXiv: 2205.12510 [cs.LG].