A Continuous Reinforcement Learning Approach to Self-Adaptive Particle Swarm Optimisation

Duncan Tilley

A dissertation submitted to the Faculty of Science, University of the Witwatersrand, Johannesburg, in fulfilment of the requirements for the degree of Master of Science.

Signed on August 30, 2023 in Johannesburg

Declaration

I declare that this dissertation is my own, unaided work. It is being submitted for the Degree of Master of Science in Computer Science at the University of the Witwatersrand, Johannesburg. It has not been submitted before for any degree or examination at any other University.

Duncan Tilley
2522818
August 30, 2023 at Johannesburg

Abstract

Particle Swarm Optimisation (PSO) is a popular black-box optimisation technique due to its simple implementation and surprising ability to perform well on various problems. Unfortunately, PSO is fairly sensitive to the choice of hyper-parameters. For this reason, many self-adaptive techniques have been proposed that attempt to both simplify hyper-parameter selection and improve the performance of PSO. Surveys however show that many self-adaptive techniques are still outperformed by time-varying techniques where the value of coefficients are simply increased or decreased over time. More recent works have shown the successful application of Reinforcement Learning (RL) to learn self-adaptive control policies for optimisers such as differential evolution, genetic algorithms, and PSO. However, many of these applications were limited to only discrete state and action spaces, which severely limits the choices available to a control policy, given that the PSO coefficients are continuous variables. This dissertation therefore investigates the application of continuous RL techniques to learn a self-adaptive control policy that can make full use of the continuous nature of the PSO coefficients. The dissertation first introduces the RL framework used to learn a continuous control policy by defining the environment, action-space, state-space, and a number of possible reward functions. An effective learning environment that is able to overcome the difficulties of continuous RL is then derived through a series of experiments, culminating in a successfully learned continuous control policy. The policy is then shown to perform well on the benchmark problems used during training when compared to other self-adaptive PSO algorithms. Further testing on benchmark problems not seen during training suggest that the learned policy may however not generalise well to other functions, but this is shown to also be a problem in other PSO algorithms. Finally, the dissertation performs a number of experiments to provide insights into the behaviours learned by the continuous control policy.

Supervisor : Prof. Christopher Cleghorn
Department : Computer Science & Applied Mathematics
Degree : Master of Science

Acknowledgements

This work would not have seen completion without the following people:

- My supervisor, Christopher Cleghorn, who is always available to help out and give advice on anything at all, despite his busy schedule.
- My wife, Annika, who is always there to help me when I'm having a tough time, and always keeps me moving forward.
- My colleagues at EPI-USE and 5DT that allowed me the flexibility to work on this dissertation and showed continued interest in my work.

Contents

List of Figures

List of Tables

List of Symbols

Particle Swarm Optimisation Symbols

f	A real-valued problem function, $f : \mathbb{R}^D \to \mathbb{R}$
D	Problem function dimensionality
N	Swarm size
$\boldsymbol{x}_i$	Position vector of particle i
$\boldsymbol{v}_i$	Velocity vector of particle i
y_i	The current fitness (i.e. objective value) of particle i
t	The current iteration count of the PSO run
T	The iteration limit of the PSO run
w	The inertia weight coefficient
c_1	The cognitive weight coefficient
c_2	The social weight coefficient
$\boldsymbol{q}_i$	The personal best position of particle i
$\hat{\boldsymbol{q}}$	The global best position
$\boldsymbol{r}_{i1}, \boldsymbol{r}_{i2}$	Random vectors sampled from $U(0,1)^D$ for particle i
x_{min}, x_{max}	The min and max boundaries of the search space
v_{min}, v_{max}	The min and max component-wise velocity used with velocity clamping

Reinforcement Learning Symbols

$\mathcal{S}$	The set of possible states (i.e. the state space)
$\mathcal{A}$	The set of possible actions (i.e. the action space)
$\mathcal{R}$	The set of possible reward values
$\mathcal{S}_0$	The possible starting states
t	The current step count
T	The total number of steps in a given episode
a_t	The action taken at step t
s_t	The state observed at step t
r_t	The reward received at the end of step $t-1$
G_t	The total discounted episodic return as at step t
π	The policy function
θ_π	The policy approximation function parameters
v_π	The state-value function
q_π	The action-value function, or Q-function
α	The learning rate
γ	The reward discount factor
ϵ	The fraction of steps to be taken greedily with ϵ-greedy algorithms
ϵ	The clipping range applied to the PPO policy gradient
$\hat{A}$	The generalised advantage function estimate
λ	The GAE bias-variance trade-off parameter

1 | Introduction

Particle Swarm Optimisation (PSO) is a popular population-based optimisation technique which was first proposed by Kennedy and Eberhart in 1995 [25, 12]. Based on the flocking of birds, PSO is easy to implement and performs well on various problems. Unfortunately, PSO is sensitive to the choice of its coefficient values which are also problem dependent, which is a common issue with metaheuristic algorithms. This thus makes the optimiser difficult to use at times since a practitioner must first tune these values to get a good solution to a problem.

Self-Adaptive PSO (SAPSO) algorithms are variants of PSO that adaptively control the values of the algorithm coefficients during a run in an attempt to both improve the performance, and to remove the burden of choosing coefficient values from the practitioner. Many different SAPSO variants have been proposed (refer to Chapter 3), though many of them frequently lead to divergent behaviours where particles leave the search space [18, 4], and are still outperformed by simpler techniques on most problems [55]. This makes these techniques unappealing to use in practice, and so practitioners are still burdened with coefficient choice.

The research put forward in this dissertation will attempt to apply modern Reinforcement Learning (RL) techniques [52] to learn a policy for self-adaptive coefficient control, with the hope that a learnt policy may outperform existing SAPSO techniques. The use of RL to improve adaptive control of optimisation techniques is not novel. A similar approach has recently been used on Differential Evolution [46], various other evolutionary algorithms [24] and even PSO [30, 42], however these studies only limited their use of RL to discrete state and action spaces. This dissertation will therefore focus on the use of reinforcement learning with fully continuous action- and state-spaces, as this should enable more complex and performant behaviour to be learnt, given that both the problem domain and the PSO coefficients are real-valued. This technique will be denoted as the Continuous Reinforcement Learning PSO technique (CRLPSO) throughout this dissertation.

For the purposes of this study, the term *self-adaptive PSO* is used to refer to PSO algorithms that rely exclusively on information from the swarm to adapt coefficient values, where *adaptive PSO* is considered to be a broader term which may include adaptation techniques that use other sources of information such as gradient or other prior information of the objective function. These terms are sometimes used interchangeably in the literature. Furthermore, this study does not consider variants of PSO that make structural changes to the algorithm (e.g. by introducing additional factors to the velocity update equation), and also only considers single-objective, real-valued optimisation problems.

1.1 Objectives

The primary objectives of this dissertation are summarized as follows:

- To investigate whether continuous Reinforcement Learning (RL) can be applied suc-

cessfully to Particle Swarm Optimisation (PSO) in order to learn a CRLPSO policy for the self-adaptation of the coefficients w, c_1, and c_2 (as to be defined in Chapter 2).

- To formulate an effective learning environment for learning such a policy, including but not limited to the following sub-objectives:
  - To design a set of possible reward functions and to evaluate the efficacy of each.
  - To formulate the real-valued observations of the swarm state that make up the state space used as input to the policy.
  - To decide on the RL algorithm used, as well as the design of related elements such as the normalisation of state, action, and reward values.
  - To perform a hyper-parameter optimisation exercise of the RL algorithm to ensure effective training of the policy.
- To identify an effective representation of a policy that is able to control the coefficients (w, c_1, and c_2) by considering different artificial neural network architectures and different representations of the continuous action space.
- To determine what impact the dimensionality of problem functions used during training has on the optimisation performance of the CRLPSO policy.
- To evaluate the optimisation performance of the learnt CRLPSO policy on the problem functions used during training, and to compare this performance to other existing PSO and SAPSO algorithms.
- To evaluate the ability of the learnt CRLPSO policy to generalise to unseen problems by evaluating the optimisation performance of the policy on problem functions that were not seen during training, and to again compare this performance to other existing PSO and SAPSO algorithms.
- To analyse the behaviours learnt by the CRLPSO policy to better understand the observed performance of the policy, and to provide additional insights for possible future work.

1.2 Dissertation Outline

This dissertation is roughly divided into three parts. The first part is introductory, and includes this chapter along with Chapters 2 to 4. The first part also covers the necessary background in the form of a literature survey. The second part is the main content of this dissertation, with Chapters 5 to 7 presenting the contributions of this study. The final part concludes the dissertation in Chapter 8. In particular, the remainder of this dissertation is organised as follows:

- Chapter 2 first introduces the canonical PSO algorithm and discusses the typical design choices that need to be made in practice. The chapter then continues with a brief overview of the theoretical properties of the PSO algorithm. This portion of the chapter first covers the criteria under which PSO exhibits stable, convergent behaviour, and then discusses theory surrounding the particle behaviour and movement.

- Chapter 3 continues the discussion of PSO by first introducing the two time-variant PSO algorithms. The remainder of the chapter then introduces a number of self-adaptive PSO algorithms that vary in the way that the coefficient values are chosen. The combination of the canonical algorithm introduced in Chapter 2, and the time-varying and self-adaptive PSO algorithms introduced in this chapter form the baselines to which the learnt policy will be compared.

- Chapter 4 introduces the field of reinforcement learning. The chapter first defines the various components of the environment in which RL operates in the context of a Markov Decision Process. The chapter then provides a brief overview of discrete RL techniques which are used by the Q-learning PSO algorithm introduced in Chapter 3. Next, the chapter introduces function approximation in the context of RL, which is necessary for the real-valued domain which is the focus of this dissertation. Finally, the chapter introduces the policy gradient techniques, and specifically the Proximal Policy Optimisation (PPO) algorithm, which is used to train the continuous control policy.

- Chapter 5 provides the first novel contribution of this dissertation and places the PSO algorithm in the context of RL by defining the learning environment. This can be considered the main framework that the CRLPSO technique comprises of. The chapter also provides a number of small experiments to justify design choices where applicable. Specifically, the chapter first defines the continuous action space to be used by the learnt policy. Next, the 28 real-valued features are defined that make up the state space. The chapter then introduces 7 possible reward functions that will be considered when training the control policy.

- Chapter 6 firstly provides the implementation details of the PPO RL algorithm and the policy model architecture before conducting three series of experiments to narrow down the possible design choices to a single effectively trained CRLPSO policy. The first experiment explores the various reward functions and PPO hyper-parameters, the second experiment explores different policy model architectures, and the last experiment investigates the impact of problem function dimensionality on the performance of the trained policy.

- Chapter 7 takes the most performant CRLPSO policy trained in Chapter 6 and performs a number of analyses to evaluate and better understand the optimisation performance of the policy. The first part evaluates the performance in comparison to the other PSO algorithms on the CEC problem suite [2], which is used throughout training. The second part evaluates the performance on the BBOB problem suite [17], which was not considered during training and therefore provides an unbiased measurement of the CRLPSO policy's performance. The final part then analyses the policy's behaviour first in terms of the coefficient control that was learnt, and then in terms of the stability and convergent behaviour of the swarm.

- Chapter 8 finally provides a summary of the dissertation's findings along with some concluding thoughts. The chapter ends off the dissertation by mentioning possible areas for future work to build upon the contributions of this study.

In addition, the dissertation is also supplemented with the following appendix:

- Appendix A provides a listing of the CEC 2017 benchmark functions that are frequently used throughout the study for training and evaluation of the policy.

2 | Particle Swarm Optimisation

Particle Swarm Optimisation (PSO) was first introduced in 1995 by Eberhart and Kennedy [25, 12]. PSO was initially designed as an abstract simulation of human psychological and social behaviour, strongly influenced by Reynolds' Boids bird flocking simulation. After performing a few simulations, the ability of the technique as an optimiser was noted by its authors and was then reinvented as an optimisation technique.

The PSO paradigm consists of two components, namely the *social* and *cognitive* components. The social component influences an individual to follow the behaviour of the population, and the cognitive component influences an individual to follow behaviour that it currently believes to be the best. The details of the algorithm are described next in Section 2.1. Sections 2.2 and 2.3 then discuss theoretical aspects of the technique's convergence criteria and movement, respectively.

2.1 The Particle Swarm Optimisation Algorithm

Particle Swarm Optimisation (PSO) is a black box optimiser, i.e. it does not require any information or make any assumption on the properties of the underlying objective function, such as gradient or landscape information. Furthermore, PSO is traditionally used for single-objective, real-valued optimisation where the objective function is of the form $f\colon \mathbb{R}^D \to \mathbb{R}$, where $D \geq 1$ is the dimensionality of the problem. Variants have been proposed that allow PSO to optimise both discrete problems and multi-objective functions, however this study will focus on the canonical algorithm and therefore only consider single-objective, real-valued optimisation problems. Furthermore, this study will assume minimisation problems, however all techniques discussed will also be applicable to maximisation problems.

The population or *swarm* is a set of *particles*, each representing a possible solution to the underlying objective function. The number of particles in the swarm is known as the *swarm size*, $N \in \mathbb{N}$. Each particle is given a position and a velocity, which is what lead to the name of the technique. The position and velocity of particle i are the vectors $\boldsymbol{x}_i, \boldsymbol{v}_i \in \mathbb{R}^D$, respectively. The position of a particle therefore represents a candidate solution to the objective function, such that the objective value $y_i = f(\boldsymbol{x}_i)$ measures the fitness of particle i.

PSO is executed iteratively, with $t = 0, 1, 2, \ldots$ used to represent the current iteration. At each iteration, the particle velocities are updated according to the social and cognitive components, as well as maintaining some degree of its current trajectory (i.e. an inertia component). The updated velocities are then used to calculate new particle positions. The update equation for particle i is defined as the following recurrent set of equations:

$$\boldsymbol{v}_i(t+1) = w\boldsymbol{v}_i(t) + c_1\boldsymbol{r}_{i1}(t) \otimes [\boldsymbol{q}_i(t) - \boldsymbol{x}_i(t)] + c_2\boldsymbol{r}_{i2}(t) \otimes [\hat{\boldsymbol{q}}(t) - \boldsymbol{x}_i(t)] \tag{2.1}$$

$$\boldsymbol{x}_i(t+1) = \boldsymbol{x}_i(t) + \boldsymbol{v}_i(t+1) \tag{2.2}$$

where w, c_1 and c_2 are the inertia weight, cognitive, and social coefficients, respectively. These scalar values are the hyper-parameters of PSO and the impact of their values are discussed in Sections 2.2 and 2.3. Note that the inertia weight, w, was only later introduced by Shi and Eberhart [47] but is considered as part of the canonical PSO algorithm.

The first term of Equation (2.1) is the inertia component, which maintains some of the particle's previous velocity from the previous iteration, weighted by w. The second and third terms are the cognitive and social components, respectively. Here, $\boldsymbol{r}_{i1}, \boldsymbol{r}_{i2} \in \mathbb{R}^D$ are vectors where each component is sampled uniformly at random within the range $[0, 1]$ at each iteration [36], i.e. $\boldsymbol{r}_1, \boldsymbol{r}_2 \sim U(0, 1)^D$. The $\otimes$ operator represents component-wise vector multiplication.

The cognitive component attracts the particle towards $\boldsymbol{q}_i$, which is the *personal best*, or P-best, position of particle i. The personal best position, $\boldsymbol{q}_i$, is updated to $\boldsymbol{x}_i$ at the end of the iteration if it is better than the previous personal best position and is within the bounds of the search space,

$$\boldsymbol{q}_i(t+1) = \begin{cases} \boldsymbol{x}_i(t) & \text{if } f(\boldsymbol{x}_i(t)) < f(\boldsymbol{q}_i(t)) \text{ and } \boldsymbol{x} \in [x_{min}, x_{max}]^D \\ \boldsymbol{q}_i(t) & \text{otherwise,} \end{cases} \tag{2.3}$$

where x_{min} and x_{max} represent the minimum and maximum bounds of the search space, respectively. This study assumes a hypercube search boundary with the same bounds on every dimension without loss of generality.

The social component attracts the particle towards $\hat{\boldsymbol{q}}$, which is the *global best*, or G-best, position of the swarm. The global best position is simply the best of the personal bests and represents the best solution found so far, calculated as

$$\hat{\boldsymbol{q}}(t) = \underset{i}{\operatorname{argmin}}\, f(\boldsymbol{q}_i(t)). \tag{2.4}$$

It is also possible to replace the global best with a *neighbourhood best* position, whereby $\hat{\boldsymbol{q}}$ becomes $\hat{\boldsymbol{q}}_i$, and is taken as the best personal position in the neighbourhood of particle i,

$$\hat{\boldsymbol{q}}_i(t) = \underset{j \in \mathcal{N}_i}{\operatorname{argmin}}\, f(\boldsymbol{q}_j(t)). \tag{2.5}$$

where $\mathcal{N}_i$ is a subset of particles that form particle i's neighbourhood.

Various studies have investigated the impact of the neighbourhood topology on the performance of PSO [26], however this study will focus only on the global best approach (also known as the fully-connected or star neighbourhood topology), as it was proposed for the canonical PSO algorithm and is the approach most commonly used in self-adaptive PSO techniques.

The swarm is usually initialized by sampling the initial positions uniformly at random across the search space, i.e. $\boldsymbol{x}(0) \sim U(x_{min}, x_{max})^D$. The velocity can be initialized in various ways, however a good approach is to simply set the velocity to zero at the start ($\boldsymbol{v}(0) = \boldsymbol{0}$) [14]. These are the initialization methods that will be used throughout this study.

Another consideration is what swarm size (N) to use. In many cases, practitioners and PSO variants simply choose a swarm size in the range of 20-50 particles, as that is what

was used by the original PSO authors [25]. In many cases this can be sufficient, however a study done by [38] shows that a size of 70-500 is more appropriate for most problems and PSO variants, however the exact impact depends on the dimensionality of the problem being solved and is generally not as important as the values of the coefficients w, c_1, and c_2.

Finally, it is worth noting that some techniques apply *velocity clamping*, whereby every component of the velocity vectors are limited to a maximum value before applying Equation (2.2),

$$v(t)_{ik} = \text{clip}(-v_{max}, v(t)_{ik}, v_{max}). \tag{2.6}$$

The choice of v_{max} can be arbitrary, however it is usually set as the size of the search space, $v_{max} = x_{max} - x_{min}$. Velocity clamping is used to avoid the "explosion" of particles that may occur if the velocities become too large, causing particles to leave the search space and stop contributing to the swarm. However, as more research was put towards analysing the convergence criteria of PSO (see Section 2.2), it has become less common to clamp the velocity as sensible choices of coefficient values will prohibit explosive behaviour.

The particle positions are repeatedly updated as described above until some stopping criteria is met. The simplest criteria is to limit the number of iterations, or equivalently to limit the number of objective function evaluations. Other criteria can be used such as measuring the average swarm velocity or inter-particle distances and stopping once the swarm has reached a desired equilibrium state. The PSO algorithm as described in this section is presented in full in Algorithm 1, with an iteration limit used as stopping condition.

Algorithm 1 Canonical Particle Swarm Optimisation

Input: Dimensionality D, swarm size N, objective function ($f\colon \mathbb{R}^D \to \mathbb{R}$), coefficients (w, c_1, c_2), search space (x_{min}, x_{max}), iteration limit T.
for $i \in 1, 2, \ldots, N$ **do**
  $\boldsymbol{x}_i(0) \sim U(x_{min}, x_{max})^D$
  $\boldsymbol{v}_i(0) \leftarrow \mathbf{0}^D$
  $\boldsymbol{q}_i(0) \leftarrow \boldsymbol{x}_i(0)$
end for
$\hat{\boldsymbol{q}}(0) \leftarrow \text{argmin}_i\, f(\boldsymbol{q}_i(0))$
for $t \in 1, 2, \ldots T$ **do**
  for $i \in 1, 2, \ldots, N$ **do**
    Set $\boldsymbol{v}_i(t)$ according to Equation (2.1)
    Set $\boldsymbol{x}_i(t)$ according to Equation (2.2)
    Set $\boldsymbol{q}_i(t)$ according to Equation (2.3)
  end for
  $\hat{\boldsymbol{q}}(t) \leftarrow \text{argmin}_i\, f(\boldsymbol{q}_i(t))$ {Note that the value of $f(\boldsymbol{q}_i(t))$) does not have to be reevaluated as it has already been evaluated in Equation (2.3) from the previous step.}
end for
Return: $\hat{\boldsymbol{q}}(T)$

PSO is a popular optimisation algorithm due to its simple implementation and (sometimes surprising) ability to optimise problems, however the success of PSO largely depends on the selection of the coefficients, c_1, c_2 and w. Many empirical and theoretical studies have been performed in order to determine guidelines on choosing these values. The remainder of this section is dedicated to discussing the most current results of these studies.

2.2 Stability Criteria

This section discusses the most current stability criteria for PSO. These criteria define boundaries for the values of the coefficients (w, c_1, and c_2) that will guarantee that particles will reach a convergent, equilibrium state. Achieving a stable configuration is important in practice, as unstable and divergent particles adversely affect the optimisation ability of PSO [6].

Stability analyses focus on two different types of convergence: first-order and second-order convergence. First-order convergence investigates the convergence of a particle's position to a fixed point in expectation, while second-order convergence considers the convergence of the variance of particle positions. Therefore, for a PSO configuration to be both order-1 and order-2 stable, the position of a particle must converge to a fixed point in both expectation and variance.

Historically, PSO stability analyses started with simplified models of PSO and have improved by systematically relaxing modelling assumptions. The earliest analyses were performed on deterministic models of PSO (i.e. by omitting $\boldsymbol{r}_1$ and $\boldsymbol{r}_2$ or replacing them with their expected values of $\frac{1}{2}$) and under the assumption that the personal and global best positions were fixed [9, 53, 54, 39]. Later works were performed under the more relaxed assumption that w, $\theta_1 = r_1 c_1$, and $\theta_2 = r_2 c_2$ are random variables with a well-defined expectation and variance [15], allowing inclusion of the stochastic components. A more thorough discussion of the stability analysis history can be found in [5].

The most recent result by Cleghorn and Engelbrecht [7] further relaxed these assumptions by modelling the personal and global bests as convergent sequences of random variables. These are currently the most realistic assumptions for which both order-1 and order-2 stability criteria have been derived, and were also proved to be the minimal assumptions necessary for the derivation thereof. The remainder of this section thus discusses the stability boundaries derived under these assumptions.

This result [7] allows derivation of stability criteria for any PSO variant for which the componentwise update equation can be written in the following form:

$$x_k(t+1) = x_k(t)\alpha + x_k(t-1)\beta + \gamma_t, \tag{2.7}$$

where α and β are random variables with a well-defined expectation and variance, and γ_t is a sequence of random variables with a well-defined expectation and variance. The stability criteria of any PSO variant that can be written in the form of Equation (2.7) (which includes canonical PSO) can be derived by using Theorem 2.2.1. Refer to [7] for a proof thereof.

Theorem 2.2.1
The following properties hold for all PSO variants of the form described in Equation (2.7), where $\rho(\cdot)$ represents the spectral radius of a matrix:

1. *Assuming $\boldsymbol{i}_t$ converges, particle positions are order-1 stable for every initial condition if and only if $\rho(\boldsymbol{A}) < 1$, where*

$$\boldsymbol{A} = \begin{bmatrix} \mathbb{E}[\alpha] & \mathbb{E}[\beta] \\ 1 & 0 \end{bmatrix} \text{ and } \boldsymbol{i}_t = \begin{bmatrix} \mathbb{E}[\gamma_t] \\ 0 \end{bmatrix} \tag{2.8}$$

2. *Particle positions are order-2 stable if $\rho(\boldsymbol{B}) < 1$ and $\boldsymbol{j}_t$ converges, where*

$$\boldsymbol{B} = \begin{bmatrix} \mathbb{E}[\alpha] & \mathbb{E}[\beta] & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & \mathbb{E}[\alpha^2] & \mathbb{E}[\beta^2] & 2\,\mathbb{E}[\alpha\beta] \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & \mathbb{E}[\alpha] & 0 & \mathbb{E}[\beta] \end{bmatrix} \text{ and } \boldsymbol{j}_t = \begin{bmatrix} \mathbb{E}[\gamma_t] \\ 0 \\ \mathbb{E}[\gamma_t^2] \\ 0 \\ 0 \end{bmatrix} \tag{2.9}$$

under the assumption that the limits of $\mathbb{E}[\gamma_t\alpha]$ *and* $\mathbb{E}[\gamma_t\beta]$ *exist.*

3. *If particle positions are order-1 stable, then the following is a necessary condition for order-2 stability:*

$$1 - \mathbb{E}[\alpha] - \mathbb{E}[\beta] \neq 0 \tag{2.10}$$

$$1 - \mathbb{E}[\alpha^2] - \mathbb{E}[\beta^2] - \frac{2\,\mathbb{E}[\alpha\beta]\,\mathbb{E}[\alpha]}{1 - \mathbb{E}[\beta]} > 0 \tag{2.11}$$

4. *The convergence of* $\mathbb{E}[\gamma_t]$ *is a necessary condition for order-1 stability, and the convergence of both* $\mathbb{E}[\gamma_t]$ *and* $\mathbb{E}[\gamma_t^2]$ *is a necessary condition for order-2 stability.*

In order to apply Theorem 2.2.1 to the canonical PSO algorithm as described in Section 2.1, the standard update equation in Equations (2.1) and (2.2) must be written in the form of Equation (2.7). This is achieved by setting the coefficients α, β, and γ_t to the following:

$$\begin{aligned} \alpha &= (1 + w) - \theta_{1k} - \theta_{2k} \\ \beta &= -w \\ \gamma_t &= \theta_{1k} q_k(t) + \theta_{2k} \hat{q}_k(t) \end{aligned} \tag{2.12}$$

where $\theta_{1k} = r_{1k}c_1$ and $\theta_{2k} = r_{2k}c_2$. In order to derive stability boundaries, the convergence of $\boldsymbol{i}_t$ and $\boldsymbol{j}_t$ must first be proven, whereafter the expressions $\rho(\boldsymbol{A}) < 1$ and $\rho(\boldsymbol{B}) < 1$ can be expanded to get order-1 and order-2 stability boundaries.

Proof of the convergence of $\boldsymbol{i}_t$ and $\boldsymbol{j}_t$ as well as the derivation of the stability conditions can be found in [7]; the results are simply restated here. Under the commonly used simplification of requiring that $c_1 = c_2$, the criteria for order-1 stability are

$$-1 < w < 1 \quad \text{and} \quad 0 < c_1 + c_2 < 4(w + 1), \tag{2.13}$$

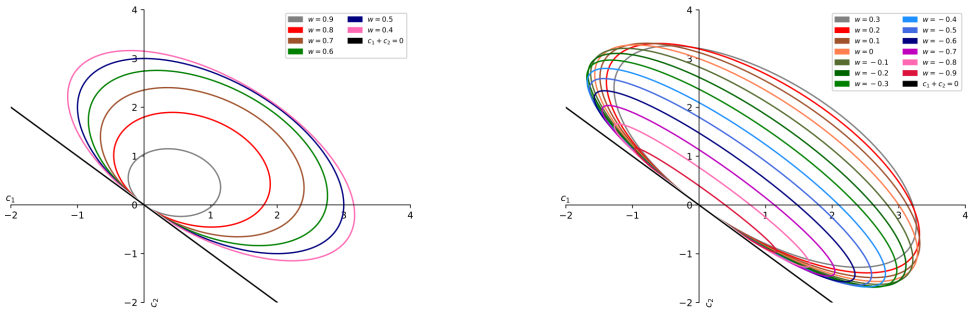
and for both order-1 and order-2 stability are

$$-1 < w < 1 \quad \text{and} \quad 0 < c_1 + c_2 < \frac{24(1 - w^2)}{7 - 5w}. \tag{2.14}$$

The full order-2 stability criteria not requiring $c_1 = c_2$ were later reported in [8], which are

$$-1 < w < 1 \quad \text{and} \quad 0 < c_1 + c_2 < \frac{4(1 - w^2)}{1 - w + \frac{(c_1^2 + c_2^2)(1+w)}{3(c_1 + c_2)^2}}. \tag{2.15}$$

The full criteria from Equation (2.15) are visualised in Figure 2.1, where the ellipticals show the boundaries for values of c_1 and c_2 at different values of w. These criteria are



(a) Order-2 stable regions for $w = 0.4$ to 0.9. (b) Order-2 stable regions for $w = -0.9$ to 0.3.

Figure 2.1: Second-order stability regions of PSO under fixed values of w. The interior of the ellipticals correspond to the stable boundaries defined by Equation (2.15). The above figures are taken from [8] with permission from the authors.

important for practitioners of PSO to choose stable combinations of coefficients which will lead to convergent behaviour.

Stability, however, is not the only property of the swarm to consider. Stability criteria help to ensure convergent behaviour, but it does not guarantee sufficient exploration prior to convergence.

2.3 Particle Behaviour and Movement

In addition to the convergence of the swarm, it may be useful to consider the impact that the value of coefficients have on the movement of the swarm. A practitioner may be able to choose coefficients such that the swarm movements are, for example, more sporadic or more monotonous to affect the exploration ability of particles.

Though not as well-studied as the convergence criteria, there have been various theoretical works focussing on the movement of the swarm. A recent work by Bonyadi [4] provides theoretical guidelines for the design of adaptive PSO algorithms and it is these that are explored in this section. A thorough discussion on the history of movement analysis can be found in [5]. Note that proofs to all theorems in this section can be found in [4].

Three different kinds of movement patterns are of concern, namely the correlation between subsequent particle positions (i.e. particle *autocorrelation*), the expected movement distance and search range, and the focus of the search towards either global or personal bests.

Being able to calculate the correlation between a particle's positions at different iterations allows one to understand what sort of behaviour a particle will have. The Pearson correlation [37] between positions can be measured according to Theorem 2.3.1, which provides a correlation coefficient in the range of $[-1, 1]$.

Theorem 2.3.1
The Pearson correlation between positions $x_k(t)$ and $x_k(t-i)$, denoted as ρ_i, at the equilibrium point is calculated by

$$\rho_i = \mathbb{E}[l]\rho_{i-1} - \mathbb{E}[w]\rho_{i-2} \tag{2.16}$$

for any $i > 1$, where $l = 1 + w - r_{1k}c_1 - r_{2k}c_2$, $\rho_1 = \frac{\mathbb{E}[l]}{\mathbb{E}[w]+1}$, and $\rho_0 = 1$. Note that the particle index is omitted in the above without loss of generality.

A coefficient further away from 0 means that there is a correlation between consecutive positions, meaning more consistent and predictable movement of a particle. A correlation close to 0 indicates more random behaviour, as it will be impossible to predict a particle's next position given its current position.

A practitioner might opt for coefficients that provide weak correlations (i.e. more random behaviour) during the early stages of a run to explore a larger part of the search space, but change to coefficients with a stronger correlation during local search to allow particles to maintain their current trajectories and improve exploitative behaviour.

The next aspect of particle movement to consider is the expected movement distance, or the expected range of the particle search. The expected movement distance is defined as the Euclidean distance between consecutive positions,

$$\mathbb{E}[d(t)] = \mathbb{E}[x(t) - x(t-1)]^2 = \mathbb{E}[v(t)]^2. \tag{2.17}$$

Note that the component index has been dropped in Equation (2.17) without a loss of generality. Using this definition, the expected movement distance can be calculated according to Theorem 2.3.2.

Theorem 2.3.2
The expected movement distance at iteration t, $\mathbb{E}[d(t)]$, at the equilibrium point is calculated as

$$\mathbb{E}[d(t)] = 2\,\mathbb{V}[x](1 - \rho_1), \tag{2.18}$$

where $\mathbb{V}[x]$ is the variance of the particle's position, which reaches a fixed point in equilibrium.

A larger expected movement distance of course increases the search range of a particle, so this result may be used by a practitioner to select coefficients that would lead to either better exploration, or better exploitation by either increasing or decreasing the expected movement distance, respectively.

The final and perhaps most intuitive aspect of particle movement is whether a particle focuses its search around the global best position, or around its personal best position. In order to quantify this focus, Bonyadi introduces the focus of a particle, F, as defined in Theorem 2.3.3.

Theorem 2.3.3
The focus of a particle is defined as $F = \left(\frac{\mathbb{E}[r_2 c_2]}{\mathbb{E}[r_1 c_1]}\right)^2$. Particle positions are more concentrated around $\hat{\boldsymbol{q}}(t)$ if $F > 1$, and more concentrated around $\boldsymbol{q}_i(t)$ otherwise.

The results of Theorem 2.3.3 are very intuitive, since if $\mathbb{E}[r_1 c_1] > \mathbb{E}[r_2 c_2]$ (and therefore $F > 1$), the social (global best) attractor will be stronger and thus cause the particle to move closer to the global best position, and the opposite case causes the particle to move close to its own personal best position. Again, this result can be used to guide the swarm towards either exploration or exploitation.

Taking the three aspects of movement described above into consideration, Bonyadi proposes a way to calculate the value of coefficients when given a desired correlation, expected movement distance, and focus [4]. This however is outside the scope of this study, but the above theorems can prove useful to use as tools for analysing the swarm behaviour of different self-adaptive PSO techniques.

3 | Self-Adaptive Particle Swarm Optimisation

Self-adaptive PSO (SAPSO) algorithms are variants of PSO that use information about the swarm to dynamically adapt the value of the coefficients c_1, c_2 and w. In the literature, these are sometimes also referred to as self-tuning or simply adaptive PSO algorithms.

This study restricts its scope to include only SAPSO algorithms that do not modify the original PSO algorithm beyond the adaptation of the coefficients. I.e., if an algorithm makes changes such as adding new particle behaviour or mutation operators such that it does not fit in the framework presented in Chapter 2, it will not be considered for this study.

This section provides an overview of existing SAPSO algorithms and is by no means exhaustive. The algorithms presented in this section were chosen based on performance measured in surveys [55, 18], and includes some newer techniques that have been proposed since.

The SAPSO techniques discussed in this section are only a handful of the many that have been proposed, and if allowed to include those that make other adjustments to the PSO algorithm or to include hybrid techniques, an exhaustive discussion would be near impossible. The intention is to use the techniques presented here as a baseline to which the proposed technique will be compared.

The first subsection provides a background of two time-varying algorithms which are technically not self-adaptive, but historically they provided a good starting point for other algorithms and have been shown to still outperform most techniques. The remainder of the section covers various self-adaptive approaches, and discusses issues that are present in most, if not all, self-adaptive approaches in Section 3.6.

3.1 Time-Varying PSO Algorithms

PSO with time-varying inertia weight (PSO-TVIW) was introduced by [48] and is possibly the first variant of PSO that does not use fixed coefficient values. The inertia weight coefficient, w, is linearly decreased throughout the run in order to allow higher velocity and better global search capability at the start of the run, and lead to lower velocity and better local search nearer to the end. The value of $w(t)$ at iterations $t = 0, 1, \ldots, T-1$ is calculated as

$$w(t) = w_s + \frac{t}{T-1}(w_f - w_s), \tag{3.1}$$

and the values for c_1 and c_2 remain fixed throughout the run. The parameters ω_s and ω_f are respectively the starting and ending values for the inertia weight. The authors used values of $w_s = 0.9$ and $w_f = 0.4$.

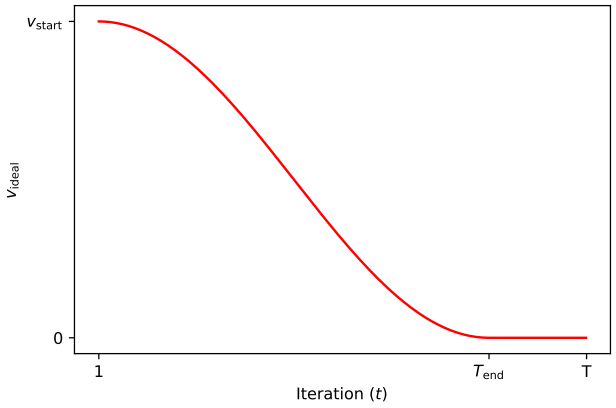


Figure 3.1: The sinusoidal ideal velocity curve as defined in Equation (3.3).

PSO with time-varying acceleration coefficients (PSO-TVAC) was later introduced by [40] and builds on the same premise. In addition to a linearly decreasing inertia weight, PSO-TVAC introduces a linearly decreasing c_1 and linearly increasing c_2, which further aids in guiding the swarm to an initial global search and late local search. The values of $c_1(t)$ and $c_2(t)$ are calculated in the same fashion as $w(t)$ in Equation (3.1), with w_s and w_f being replaced by c_{1s} and c_{1f}, and c_{2s} and c_{2f}, respectively. The authors found that decreasing c_1 from 2.5 to 0.5 and increasing c_2 from 0.5 to 2.5 provided improved results over using only the decreasing inertia weight.

3.2 PSO with Velocity Information

PSO with velocity information (APSO-VI) proposes the idea of an *ideal velocity* [59] that the swarm should attempt to maintain in order to promote good exploration at the start and local search at the end to promote convergence. The proposed approach adapts the inertia weight in order to keep the average velocity of the swarm close to the ideal velocity. The average velocity is measured as

$$v_{\text{avg}}(t) = \frac{1}{N \cdot D} \sum_{i}^{N} \sum_{k}^{D} |v_{ik}(t)| \,. \tag{3.2}$$

The ideal velocity, v_{ideal}, is defined as a sinusoidal curve, starting at v_{start} and ending at 0 near the end of the run. The curve is defined as

$$v_{\text{ideal}}(t) = \begin{cases} v_{\text{start}} \cdot \frac{1+\cos(\frac{t}{T_{\text{end}}}\pi)}{2} & \text{if } t < T_{\text{end}} \\ 0 & \text{otherwise} \end{cases} \tag{3.3}$$

and is illustrated in Figure 3.1. The value of $T_{\text{end}} \leq T$ indicates the iteration after which the ideal velocity should reach 0, as indicated by Equation (3.3).

The adaptive strategy to control the inertia weight is simply to increase or decrease w by a fixed step size according to the relationship between v_{avg} and v_{ideal}. The value of w is also clamped to the range $[w_{\min}, w_{\max}]$. This is formally defined as

$$w(t+1) = \begin{cases} \max\{w(t) - \Delta w, w_{\min}\} & \text{if } v_{\text{avg}}(t) \geq v_{\text{ideal}}(t+1) \\ \min\{w(t) + \Delta w, w_{\max}\} & \text{otherwise,} \end{cases} \tag{3.4}$$

where Δw is the step size and $w(0)$, the initial inertia weight, can be chosen arbitrarily. APSO-VI therefore decreases the inertia if the swarm is considered to have a too high velocity, and increases the inertia if it is too low. The values of c_1 and c_2 again are fixed throughout the run.

3.3 Improved PSO

Improved PSO by Li and Tan (IPSO) proposes adapting the inertia weight according to a calculated state of the swarm [28]. Different to the previously discussed techniques, the value for w is also calculated individually for each particle, meaning that every particle i has a unique inertia weight, $w_i(t)$.

In order to determine the value, IPSO calculates the *convergence factor* and *diffusion factor* of each particle. The convergence factor measures the amount of improvement of the particle's personal best over the previous iteration according to

$$c_i(t) = \frac{|f(\boldsymbol{q}_i(t-1)) - f(\boldsymbol{q}_i(t))|}{f(\boldsymbol{q}_i(t-1)) + f(\boldsymbol{q}_i(t))}, \tag{3.5}$$

where a value of 0 indicates no improvement. The diffusion factor then measures the performance of the particle in relation to the current global best, with a lower value indicating that the particle is performing better. The diffusion factor is defined as

$$d_i(t) = \frac{|f(\boldsymbol{q}_i(t)) - f(\hat{\boldsymbol{q}}(t))|}{f(\boldsymbol{q}_i(t)) + f(\hat{\boldsymbol{q}}(t))}. \tag{3.6}$$

Armed with the above descriptors of a particle's state, the inertia weight is calculated as

$$w_i(t) = 1 - \left| \frac{\alpha \cdot (1 - c_i(t))}{(1 + d_i(t)) \cdot (1 + \beta)} \right|, \tag{3.7}$$

where α and β are newly introduced scalar coefficients, typically set in the range of $[0, 1]$. The values of c_1 and c_2 once again remain fixed throughout. An increase in either the diffusion or convergence factor leads to a higher inertia weight. The justification provided by the authors is that this adaptive technique helps prevent premature convergence to local minima or non-optimal point.

3.4 Fuzzy Self-Tuning PSO

Fuzzy Self-Tuning PSO (FST-PSO) uses fuzzy logic to adaptively determine the coefficient values for a specific particle [35]. In addition to controlling the values of w, c_1, and c_2, FST-PSO introduces the variables $\mathscr{U}$ and $\mathscr{L}$ which are used to control the degree of maximum and minimum velocity clamping respectively, per individual particle. Velocity is clamped on each axis as in Equation (2.6), with

$$v_{max} = \mathscr{U} \cdot (x_{max} - x_{min}) \tag{3.8}$$

$$v_{min} = \mathscr{L} \cdot (x_{max} - x_{min}). \tag{3.9}$$

In order to determine the values of w, c_1, c_2, $\mathscr{U}$, and $\mathscr{L}$ for each particle at each iteration, FST-PSO makes use of a Fuzzy Rule-Based System (FRBS), based on the concepts of distance from the global best, and the fitness improvement of a particle in relation to the previous iteration.

The distance from the global best is simply defined as the Euclidean distance $\delta_i(t) = ||\boldsymbol{x}_i(t) - \hat{\boldsymbol{q}}(t)||_2$. The value δ_{max} is used to denote the diagonal of the hyper-rectangular search space, defined as $\delta_{max} = \sqrt{\sum_k^D (x_{max_k} - x_{min_k})^2}$. The second concept, the particle fitness improvement, is then defined as

$$\phi_i(t) = \frac{||\boldsymbol{x}_i(t) - \boldsymbol{x}_i(t-1)||_2}{\delta_{max}} \cdot \frac{\min\{f(\boldsymbol{x}_i(t)), f_\Delta\} - \min\{f(\boldsymbol{x}_i(t-1)), f_\Delta\}}{|f_\Delta|}, \tag{3.10}$$

where f_Δ is the worst fitness value calculated right after initializing the swarm (i.e. $f_\Delta = \max_i f(\boldsymbol{x}_i(0))$). The second factor measures the improvement of the particle's fitness compared to its previous position, normalized by f_Δ, with a lower value indicating that the particle's position is improving. The first term weights the result according to the size of the particle's movement.

The final result of Equation (3.10) is a value $\phi_i(t) \in [-1, 1]$, where a lower value indicates faster improvement. The distance from the global best is a value $\delta_i(t) \in [0, \delta_{max}]$. *Membership functions* are then defined for each of these values, that provide a value in the range $[0, 1]$ for different linguistic values.

The authors define membership functions for the linguistic values of *Same*, *Near*, and *Far* for the distance δ_i, and for the linguistic values of *Better*, *Same*, and *Worse* for the fitness improvement ϕ_i. The exact definition of these membership functions can be found in [35], however it is enough to know that the memberships can overlap such that, for example, $\phi_i(t)$ can be both *Better* and *Same* to some degree. The membership degree across all values however always sum to 1.

The fuzzy rules are listed in Table 3.1. The membership degree for every rule is first calculated based on the linguistic value memberships. The values for the coefficients are then calculated as the average of all rules that contribute to the coefficient, weighted by the membership degree:

$$value = \frac{\sum_r^R \rho_r z_r}{\sum_r^R \rho_r}, \tag{3.11}$$

where ρ_r is the membership degree of rule r, and z_r is the coefficient value specified by the rule. A more detailed explanation of the fuzzy rule system, the membership function definitions, and the reasoning behind the choice of rules can be found in [35].

3.5 Q-Learning PSO

A SAPSO variant using a form of Reinforcement Learning (RL) was introduced by Liu et. al. [30] in 2019. The technique, aptly named Q-learning PSO (QLPSO), uses Q-learning to learn a table that is used to map a particle's state to the values of its coefficients. This section discusses the state and action spaces and the reward function formulation, which are the components required for Q-learning. Refer to Chapter 4 for a background on RL, and Section 4.4 specifically for Q-learning.

The table that is learned (the Q table) maps a particle's state to the expected reward that the particle can expect when taking that action given the current state. It is assumed

Table 3.1: Fuzzy logic rules used by FST-PSO.

Rule Number	Rule Description
1	if (ϕ_i is *Worse* or δ_i is *Same*) then w is 0.3
2	if (ϕ_i is *Same* or δ_i is *Near*) then w is 0.5
3	if (ϕ_i is *Better* or δ_i is *Far*) then w is 1.0
4	if (ϕ_i is *Better* or δ_i is *Near*) then c_2 is 1.0
5	if (ϕ_i is *Same* or δ_i is *Same*) then c_2 is 2.0
6	if (ϕ_i is *Worse* or δ_i is *Far*) then c_2 is 3.0
7	if (ϕ_i is *Far*) then c_1 is 0.1
8	if (ϕ_i is *Worse* or ϕ_i is *Same* or δ_i is *Same* or δ_i is *Near*) then c_1 is 1.5
9	if (ϕ_i is *Better*) then c_1 is 3.0
10	if (ϕ_i is *Same* or ϕ_i is *Better* or δ_i is *Far*) then $\mathscr{L}$ is 0
11	if (δ_i is *Same* or δ_i is *Near*) then $\mathscr{L}$ is 0.001
12	if (ϕ_i is *Worse*) then $\mathscr{L}$ is 0.01
13	if (δ_i is *Same*) then $\mathscr{U}$ is 0.1
14	if (ϕ_i is *Same* or ϕ_i is *Better* or δ_i is *Near*) then $\mathscr{U}$ is 0.15
15	if (ϕ_i is *Worse* or δ_i is *Far*) then $\mathscr{U}$ is 0.2

Table 3.2: Boundaries used to determine the QLPSO agent's decision state value.

Relative Distance	Description	Value
$0 \leq d_i < 0.25\Delta R$	Nearest	0
$0.25\Delta R \leq d_i < 0.5\Delta R$	Nearer	1
$0.5\Delta R \leq d_i < 0.75\Delta R$	Farer	2
$0.75\Delta R \leq d_i$	Farthest	3

that QLPSO first learns the Q table and then uses a learned table to optimise any problem, however this is not explicitly stated. It is also possible that the Q table is learned during every run, however this could lead to random and suboptimal behaviour at the start of a run.

It is important to note that the Q table is meant to be used by each particle individually, i.e. at each iteration, the state of each particle is calculated and an action is chosen greedily to determine the coefficients $w_i(t)$, $c_{1i}(t)$, and $c_{2i}(t)$ for every particle i.

The state space of QLPSO attempts to observe the state of an individual particle as two discrete variables, each taking one of 4 possible values (i.e. the state can be one of $4 \times 4 = 16$ unique values). The two variables are respectively called the *decision* state space and *objective* state space.

The decision state space categorises the distance between the particle i's current position and the global best position by measuring a relative Euclidean distance $d_i = ||\boldsymbol{x}_i - \hat{\boldsymbol{q}}||_2$. This distance is then discretised to 4 values by comparing it to the search space range ($\Delta R = x_{max} - x_{min}$) according to the boundaries as shown in Table 3.2.

The objective state space then categorises the particle's current fitness in relation to the global best using a relative fitness, $f_i = f(\boldsymbol{x}) - f(\hat{\boldsymbol{q}})$. This is discretised in a similar manner as the decision state space: by comparing the value against ΔF, which is the difference between the worst fitness seen so far and the best fitness, $f(\hat{\boldsymbol{q}})$, as shown in Table 3.3. Note that the objective state space can easily become a poor indicator of a particle's state if ΔF becomes too large or if the swarm is in a convergent state, as that

Table 3.3: Boundaries used to determine the QLPSO agent's objective state value.

Relative Distance	Description	Value
$0 \leq f_i < 0.25\Delta F$	Smallest	0
$0.25\Delta F \leq f_i < 0.5\Delta F$	Smaller	1
$0.5\Delta F \leq f_i < 0.75\Delta F$	Larger	2
$0.75\Delta F \leq f_i$	Largest	3

Table 3.4: QLPSO action space and the associated coefficient values.

Value	Description	w	c_1	c_2
0	Rough exploration	1	2.5	0.5
1	Fine exploration	0.8	2	1
2	Slow convergence	0.6	1	2
3	Fast convergence	0.4	0.5	2.5
–	Local search	0	0	3

will cause all particles to take on the smallest value of this variable.

Next is the action space of QLPSO. Each particle has a choice of 4 discrete actions, which are each mapped to a set of coefficient values as shown in Table 3.4. Note that the fifth action is not included in the Q table, as it is instead forced for all particles during the last 10% of iterations. The Q table is thus only used during the first 90% of any run.

Finally, the reward function of QLPSO is used the indicate the "goodness" of a specific action choice and is what the RL agent uses to learn the Q table. The reward given after an action depends on three factors: the current iteration, the previous action taken, and whether the particle's personal best improved after taking the action.

The reward is formulated as a set of linear equations with differing slopes with respect to the current iteration for each action as shown in Figure 3.2. These slopes indicate that the authors want the agent to prefer certain actions at different times during a run. The intersection of the slope depends on whether the action lead to an improvement in fitness, such that the agent is punished with a negative reward if there was no improvement, and rewarded with a positive reward otherwise. The reward function is formally defined as

$$R_{t,i} = c + \begin{cases} -3\frac{t}{T} & \text{if } A_{t-1,i} = 0 \\ -\frac{t}{T} & \text{if } A_{t-1,i} = 1 \\ \frac{t}{T} & \text{if } A_{t-1,i} = 2 \\ -3\frac{t}{T} & \text{if } A_{t-1,i} = 3, \end{cases} \tag{3.12}$$

where t is the current iteration, T is the maximum number of iterations, and A_{t-1} is the action taken at the previous iteration. Note that c, which simply adjusts the intercept of the function to match Figure 3.2, is not formally defined here for the sake of brevity.

This reward function can be criticised as being unnecessarily biased to prefer certain actions over others. It may be beneficial to provide the same degree of punishment or reward regardless of the action taken in order to avoid biasing the agent towards certain actions.

Unfortunately the authors do not provide any analysis on what the outcome is of the learning, as that would provide an indication of whether the bias in the reward function heavily impacts the outcomes.

This is currently the technique most directly related to what is proposed by this study,

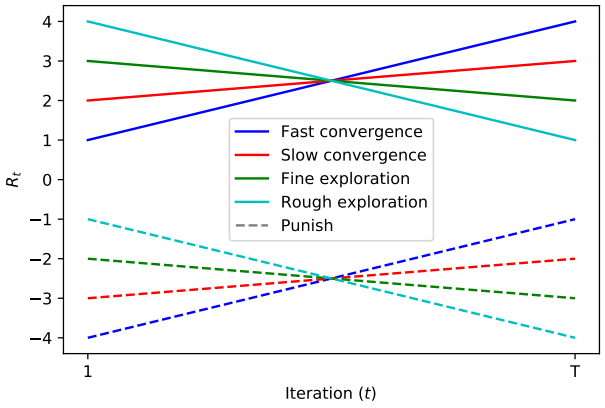


Figure 3.2: The QLPSO reward function as defined in Equation (3.12). The positive reward (solid line) is given if the action lead to a fitness improvement, and the negative punishment (dashed line) reward is given otherwise.

however it is still limited to its use of discrete state and action spaces. In addition to this limitation, there are many possible criticisms of this technique as briefly mentioned in the relevant discussions above, all of which will also be addressed to some extent by the proposed research. Note that another similar technique was proposed by [42], however further modifications were proposed to the algorithm which disqualifies it from this study, and the application of RL is fairly similar to QLPSO in both design and limitation. The CRLPSO technique put forward in this dissertation is strongly related to QLPSO, however the technique is not limited to discrete spaces.

3.6 A Word on SAPSO Performance

The handful of SAPSO techniques discussed above should give an indication of the type of self-adaptive techniques tried by the research community. The techniques can range from using simple equations such as that of APSO-VI [59], to using elaborate rules as that of FST-PSO [35].

Despite this, surveys have shown that the performance of self-adaptive techniques simply do not live up to the expectation, preventing them from being useful to practitioners. Although most original studies showed promising performance of their new technique, further independent studies have shown that the simple time-variant techniques outperform self-adaptive techniques on unseen benchmark functions [55].

A possible reason for the poor performance of these techniques is that many of these techniques tend towards coefficient configurations that lead to divergent behaviour or premature convergence [18]. The techniques are generally not designed according to the theoretical constraints and guidelines discussed in Sections 2.2 and 2.3 [4].

4 | Reinforcement Learning

Reinforcement Learning (RL) is a part of the broader field of machine learning that is concerned with training an agent to act within an environment in such a way as to maximize a received reward signal. This section first explains the concepts used in the context of RL and then provides a background on commonly used reinforcement learning algorithms. A more comprehensive introduction to RL can be found in the Sutton & Barto's book on the subject [52].

The main components of reinforcement learning are the *agent*, that interacts with the *environment* by taking a certain *action*. The inputs that the agent can observe within the environment forms the current *state* of the environment. After taking an action, the agent is given a *reward*, which is a real-valued signal that states how favourable the current state of the agent is, or how close the agent is to completing an objective.

The goal of RL, as stated succinctly by Sutton & Barto [52], is "learning what to do". RL uses optimisation techniques to train an agent to choose the action in a given state that will maximize the reward received. The power of RL is the ability to not only consider immediate rewards, but also learn to maximize future reward, i.e. to learn to act in a way that may not be rewarding immediately, but lead to a greater future outcome.

4.1 Markov Decision Process

A finite Markov Decision Process, or finite MDP, is a way to formalize the decision making process, which contains the context of a RL problem. A finite MDP is formally defined as the quintuple $(\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{S}_0, p)$ [52].

$\mathcal{S}$ is the set of states which comprises of all the observations that the agent may encounter. The representation of the set of states has a large impact on the type of RL algorithm that can effectively teach an agent to act in the environment. Tabular reinforcement learning focuses on *discrete state* problems, where there is a finite number of states such that $\mathcal{S} \subset \mathbb{N}$. This representation is however very limiting, so many modern techniques focus on *continuous state* environments where observations are encoded as $d_{\mathcal{S}}$-dimensional real-valued vectors. In this case, $\mathcal{S} \subseteq \mathbb{R}^{d_{\mathcal{S}}}$. The distinction between these state representations is explored further in Section 4.5.

$\mathcal{A}$ is the set of actions that are available to the agent. The set of actions may be dependent on the state such that $\mathcal{A}(s)$ may be different from $\mathcal{A}(s')$ if $s \neq s'$, however this study will assume that all actions are available in all states. Similar to the set of states, the representation of $\mathcal{A}$ has an impact on the type of RL algorithms that can be used. A distinction is again made between *discrete action* environments where $\mathcal{A} \subset \mathbb{N}$ and *continuous action* environments, where $\mathcal{A} \subseteq \mathbb{R}^{d_{\mathcal{A}}}$.

$\mathcal{R} \subseteq \mathbb{R}$ is the set of possible reward values. On its own, this is not a particularly useful property of the MDP, though some consideration should be given to the range of reward values used. $\mathcal{S}_0 \subseteq \mathcal{S}$ is simply the set of possible starting states, such that the initial state

is sampled from $\mathcal{S}_0$.

Finally, the function $p\colon (\mathcal{S}, \mathcal{R}, \mathcal{S}, \mathcal{A}) \to \mathbb{R}$ encodes the dynamics of the MDP environment. This function defines a probability distribution of state-reward pairs, giving the probability of ending up in a next state s' and receiving the reward r, provided that the agent is currently in state s and takes action a:

$$p(s', r|s, a) = P\{S_t = s', R_t = r \mid S_{t-1} = s, A_{t-1} = a\} \tag{4.1}$$

This function thus not only encodes the dynamics of the environment (i.e. how the action affects the state), but also defines the reward distribution.

When interacting with a finite MDP, the interaction is normally divided into independent *episodes* which starts at an initial state and continues until a terminating state is reached. The exact path followed during an episode is called the *trajectory*, which can be represented as the following path:

$$S_0, A_0, R_1, S_1, A_1, R_2, \ldots, S_{T-1}, A_{T-1}, R_T, S_T$$

where T is used to denote the length of an episode, with S_T referred to as the termination state. Following [52], the subscript for reward will be taken to start at 1. This emphasises the fact that the reward is rather a "response" to the previous state and action pair, and is returned together with the next state. I.e., reward R_t is the reward received in response to taking action A_{t-1} in state S_{t-1}. Note that it is possible for the path to never terminate, i.e. to have an episode of infinite length, however this study will focus on the episodic case.

With the definition of the environment defined by the finite MDP, the agent in the context of RL follows a certain *policy* to choose actions in every state. The policy is formally defined as the function $\pi\colon (\mathcal{A}, \mathcal{S}) \to \mathbb{R}$, which provides a probability distribution over the actions in every state, defined as

$$\pi(a|s) = P\{A_t = a|S_t = s\} \tag{4.2}$$

An agent can act stochastically according to the policy by sampling the action from the policy's probability distribution, i.e. $A_t \sim \pi(\cdot|S_t)$, or can act *greedily* such that $A_t = \operatorname{argmax}_{a\in\mathcal{A}} \pi(a|S_t)$. The purpose of reinforcement learning therefore boils down to learning an *optimal policy* from experience gained by interacting with the environment. In order to compare two policies, a way to measure the efficacy of a policy must be defined. This is what the *value functions* are for.

The *state-value function*, $v_\pi\colon \mathcal{S} \to \mathbb{R}$, estimates the total *discounted* reward (called the *return*) that an agent will receive when starting in a state $s \in \mathcal{S}$ and then following the policy π. It therefore assigns a measurable value to the current policy when starting in a specified state. More formally, the state-value function is defined as either an expectation as in Equation (4.3), or as a recursive formula as in Equation (4.4), where $\gamma \in [0, 1]$ is the discount factor.

$$v_\pi(s) = \mathbb{E}\left[R_{t+1} + \gamma R_{t+2} + \ldots + \gamma^{T-t} R_T \;\middle|\; S_t = s\right] \tag{4.3}$$

$$= \sum_{a\in\mathcal{A}} \pi(a|s) \sum_{s'\in\mathcal{S}, r\in\mathcal{R}} p(s', r|s, a)[r + \gamma v_\pi(s')] \tag{4.4}$$

The recursive definition is known as the *Bellman equation* for v_π, which expresses the dependency of a state's value on the value of future states.

In addition to the state-value function, the *action-value function*, or Q-function, is also commonly used. The action-value function, $q_\pi \colon (\mathcal{S}, \mathcal{A}) \to \mathbb{R}$, instead estimates the return when starting in a state $s \in \mathcal{S}$, taking action $a \in \mathcal{A}$ and only *thereafter* following the policy π. The action-value function can also be defined as an expectation as in Equation (4.5) or in terms of the state-value function as in Equation (4.6).

$$q_\pi(s,a) = \mathbb{E}\left[R_{t+1} + \gamma R_{t+2} + \ldots + \gamma^{T-t} R_T \mid S_t = s, A_t = a\right] \tag{4.5}$$

$$= \sum_{s',r} \left[p(s', r|s, a)r + \gamma v_\pi(s')\right] \tag{4.6}$$

Equation (4.6) above expresses the relationship between the action- and state-value functions. The state-value function can also be expressed in terms of q_π as

$$v_\pi(s) = \sum_a \pi(a|s) q_\pi(s,a). \tag{4.7}$$

By substituting v_π in Equation (4.6) with Equation (4.7), it is also possible to express q_π as a purely recursive formula.

With the value functions defined, it is now possible to define exactly what an optimal policy looks like. The optimal policy is the policy, π^*, which has the optimal value functions where, for each state s,

$$v_{\pi^*}(s) = v^*(s) = \max_\pi v_\pi(s) \tag{4.8}$$

$$q_{\pi^*}(s,a) = q^*(s,a) = \max_\pi q_\pi(s,a) \tag{4.9}$$

The optimal policy is then simply the policy that acts greedily according to the optimal value function,

$$\pi^*(a|s) = \begin{cases} 1 & \text{if } a = \operatorname{argmax}_{a' \in \mathcal{A}} q^*(s,a') \\ 0 & \text{otherwise.} \end{cases} \tag{4.10}$$

Note that it is possible that multiple optimal policies exist for a problem if multiple actions in a state have the same, maximal return. A policy will still be optimal if it chooses randomly between these actions, or chooses a specific one.

In order to find an optimal policy, it is therefore sufficient to learn only the optimal state-value or action-value function, from which the optimal policy can be derived. This is the approach followed by many RL techniques, however it is also possible to directly learn the policy's action distribution.

Unfortunately, it is difficult to find the true value of any value function, let alone the optimal one. The value function (state or action) for a given policy can be closely approximated using dynamic programming techniques, however this is only possible in the tabular case where both the state and action spaces are discrete, and is furthermore only practical if the state space is not too large.

To complicate matters further, the purpose of RL is of course to *improve* the value function towards optimality in addition to approximating its value, which causes the underlying function that is being optimised to be non-stationary, as the policy continually changes. These are some of the issues that make the application of RL difficult, however many techniques have been proposed and iteratively improved to work around these issues.

The remainder of this section provides an introduction to different RL techniques. The section is by no means exhaustive, but covers the techniques that will be essential for the execution of the proposed research.

4.2 Tabular Reinforcement Learning

In early reinforcement learning techniques [22, 3, 51, 56, 50, 41] (now commonly referred to as classical), the value function is represented as a table that maps from the current state (or state-action pairs) to its value. This is only possible if both the state and action spaces are discrete, so these techniques, though well studied, are not applicable to many real-world problems. Nevertheless, it is good to have an idea of how these algorithms work as the general structure of most modern techniques are still reminiscent of the tabular techniques.

Reinforcement learning algorithms generally have two steps: policy *evaluation*, and policy *improvement*. Sometimes these are more generally referred to as *prediction* and *control*, respectively. Policy evaluation involves estimating the value of v_π or q_π under the current policy. With an estimate for the value of the current policy, policy improvement then involves updating the policy to be more optimal.

Perhaps the most accurate way to estimate the value function of a current policy is through *dynamic programming* [3], which is an iterative method to estimate the value functions using the recursive Bellman equation (Equation (4.4)). Unfortunately, this relies on knowing the dynamics of the environment (the function $p(s', r|s, a)$) which is rarely known in practice, and also becomes very expensive with larger state spaces. Most modern techniques instead make use of Monte Carlo techniques [50] or Temporal Difference (TD) learning [51], which are introduced in the next sections.

Policy improvement is simply done by constructing a new *greedy* policy, π', based on the current state-value function estimate as

$$\pi'(a|s) = \begin{cases} 1 & \text{if } a = \operatorname{argmax}_{a'} q_\pi(s, a') \\ 0 & \text{otherwise.} \end{cases} \tag{4.11}$$

Acting greedily according to the previous policy's value function ensures that the value of the new policy is strictly better or equal in all states. This result is known as the *Policy Improvement Theorem*, stated in Theorem 4.2.1. A proof thereof can be found in [3] or [52].

Theorem 4.2.1
For all states $s \in \mathcal{S}$, if

$$q_\pi(s, \pi'(s)) \geq v_\pi(s), \tag{4.12}$$

where $\pi'(s)$ represents the action taken by the deterministic policy π', then the policy π' must be as good as, or better than, π. That is, for all states $s \in \mathcal{S}$, it follows that

$$v_{\pi'}(s) \geq v_\pi(s). \tag{4.13}$$

If the policy can no longer be improved (i.e. if the policy π' is as good as, but not better than, π), then the policy must be optimal [3]. Using the policy improvement theorem and a method of evaluating the policy, this process of evaluation and improvement, called *policy iteration*, is repeated until the policy can no longer be improved or some other stopping condition is met.

As mentioned before, dynamic programming is a very reliable technique to evaluate the policy, but requires having a model of the environment's dynamics and can be infeasible for large state spaces. The following two sections respectively explore Monte Carlo and TD-learning techniques, which provide alternative ways to evaluate the policy.

4.3 Monte Carlo Techniques

The use of Monte Carlo techniques [50] to evaluate a policy (estimate its value function) is one of the alternatives to dynamic programming. Firstly, it does not require exhaustively iterating over all states and is thus more computationally efficient. Secondly, and perhaps more important than the first, is that it does not require a model of the environment's dynamics. That is, it does not require $p(r, s'|s, a)$. The class of RL algorithms that do not require this function are referred to as *model-free* RL techniques.

The downside, however, is that the true value function is now only estimated which introduces problems during policy improvement, as the policy is updated based on estimates instead of true values. This is a recurring theme with all other policy iteration algorithms discussed in this chapter outside of dynamic programming.

The premise of Monte Carlo for policy evaluation is very simple. The value function is simply estimated by using the true returns of episodes:

$$v_\pi(s) \approx R_{t+1} + \gamma R_{t+2} + \ldots + \gamma^{T-t} R_T | S_t = s \tag{4.14}$$

In order to collect these returns, Monte Carlo performs multiple episodes to collect the trajectories $(S_0, A_0, R_1, S_1, \ldots)$ of sample episodes. This process of collecting trajectories is called the *rollout* phase.

Once collected, the trajectories are used to calculate estimates of $v_\pi(s)$ for every state s that was visited as the average of the true returns. Multiple episodes may be performed to collect more sample trajectories.

Another unfortunate effect of not forcefully visiting every state, is that there is no guarantee that every state will be visited, especially if the policy and environment are deterministic. The solution is to instead create an ϵ-greedy policy during policy improvement, defined as

$$\pi'(a|s) = \begin{cases} 1 - \epsilon + \frac{\epsilon}{|\mathcal{A}|} & \text{if } a = \text{argmax}_{a' \in \mathcal{A}}\, q_\pi(s, a') \\ \frac{\epsilon}{|\mathcal{A}|} & \text{otherwise,} \end{cases} \tag{4.15}$$

which means that the policy will choose a non-greedy action with a probability of ϵ. This allows the rollout phase to collect returns of all states. The value of ϵ is usually small, such as 0.05 and is many times reduced at later iterations of the algorithm to reduce the amount of exploration.

This idea of using a non-greedy policy leads to a separation between *on-policy* and *off-policy* algorithms. On-policy algorithms (which includes Monte Carlo) only estimate the value function of the policy they are acting according to, which is normally the ϵ-greedy policy. This means that the optimal policy is not necessarily learned as the optimal value function is not necessarily the same as that of the ϵ-greedy policy. Off-policy techniques such as Q-learning (explained in the next section) are able to estimate the value function of the optimal policy while acting non-optimally.

The complete algorithm for policy iteration using Monte Carlo is provided in Algorithm 2, which makes use of the Monte Carlo rollouts for on-policy learning of an ϵ-greedy

policy. The next section introduces TD-learning, which is another alternative method for policy evaluation.

Algorithm 2 On-policy Monte Carlo control for ϵ-greedy policies

Input: $\epsilon \in [0, 1]$, $\gamma \in [0, 1]$
$\pi \leftarrow$ arbitrary ϵ-greedy policy
$q(s, a) \leftarrow$ arbitrary, for all $s \in \mathcal{S}, a \in \mathcal{A}$
$Returns(s, a) \leftarrow$ empty list, for all $s \in \mathcal{S}, a \in \mathcal{A}$
repeat
  Generate a trajectory, $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T, S_T$, using π
  $G \leftarrow 0$
  for $t = T - 1, T - 2, \ldots, 0$ **do**
    $G \leftarrow \gamma G + R_{t+1}$
    Append G to $Returns(S_t, A_t)$
    $q(S_t, A_t) \leftarrow$ average$(Returns(S_t, A_t))$
    Update $\pi(\cdot|S_t)$ to be ϵ-greedy
  end for
until stopping condition met
Return: Policy π

4.4 Temporal Difference Learning

Temporal Difference (TD) learning is a central idea in reinforcement learning that, like Monte Carlo, allows evaluating a policy directly from experience instead of using a model of the environment [51]. The difference between TD learning and Monte Carlo techniques is the use of *bootstrapping*. TD learning techniques can update the value function estimates after every step without having to wait for an episode to complete, and thus does not have a separate rollout phase.

After every step in the environment, the value function estimate for the previous state is updated according to

$$v_\pi(s_t) = v_\pi(s_t) + \alpha[R_{t+1} + \gamma v_\pi(s_{t+1}) - v_\pi(s_t)] \tag{4.16}$$

where α is the learning rate, which determines how drastic subsequent updates change the estimate of the value function. The term $[R_{t+1} + \gamma v_\pi(s_{t+1}) - v_\pi(s_t)]$ is known as the *TD error*, and is often annotated as δ_t. This term bootstraps the error by guessing the true value of the expected return based on the current estimate.

The remainder of this section discusses SARSA and Q-learning, which respectively provide on-policy and off-policy control methods using TD learning.

4.4.1 SARSA

SARSA [41] is an on-policy technique that uses the bootstrapping of TD learning to estimate the action-value function, $q_\pi(s, a)$. The name comes from the fact that the technique requires choosing two actions before performing an update, requiring the sequence $S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1}$. Note that the last action does not have to be taken yet, but must be decided before updating. The estimate of the action-value function is then updated according to

$$q_\pi(s_t, a_t) = q_\pi(s_t, a_t) + \alpha[R_{t+1} + \gamma q_\pi(s_{t+1}, a_{t+1}) - q_\pi(s_t, a_t)], \tag{4.17}$$

which is similar to the TD update of v_π in Equation (4.16). SARSA is a straightforward adaptation of TD learning to estimating the action-value function. Similar to Monte Carlo, policy improvement creates a new ϵ-greedy policy which is required to ensure that all states can be visited. The full algorithm is provided in Algorithm 3.

Algorithm 3 On-policy TD (SARSA) control for ϵ-greedy policies

Input: $\epsilon \in [0, 1]$, $\alpha \in [0, 1]$, $\gamma \in [0, 1]$
$\pi \leftarrow$ arbitrary ϵ-greedy policy
$q(s, a) \leftarrow$ arbitrary, for all $s \in \mathcal{S}, a \in \mathcal{A}$
repeat
  $S \leftarrow S_0$
  $A \sim \pi(\cdot|S)$
  repeat
    Take action A, observing reward R and next state S'
    $A' \sim \pi(\cdot|S)$
    $q(S, A) \leftarrow q(S, A) + \alpha\left[R + \gamma q(S', A') - q(S, A)\right]$
    Update $\pi(\cdot|S)$ to be ϵ-greedy
    $S \leftarrow S'$, $A \leftarrow A'$
  until S is terminal
until stopping condition met
Return: Policy π

As SARSA is on-policy, this again means that the learned value function is that of the ϵ-greedy policy and may not necessarily be the optimal value function for a greedy policy. For this reason, Q-learning, described next, is generally preferred in practice.

4.4.2 Q-learning

Q-learning [56] is an off-policy TD learning technique that, like SARSA, estimates the action-value function. Unlike SARSA, Q-learning directly estimates the optimal action-value function, meaning that the technique is free to act non-optimal for the sake of exploration without affecting the end result. In addition to this, Q-learning does not require choosing a second action, only requiring the sequence of $S_t, A_t, R_{t+1}, S_{t+1}$. This makes Q-learning a more attractive choice in practice when implementing TD learning.

The estimate of the action-value function after every step is updated according to

$$q_\pi(s_t, a_t) = q_\pi(s_t, a_t) + \alpha[R_{t+1} + \gamma \max_a q_\pi(s_{t+1}, a) - q_\pi(s_t, a_t)]. \tag{4.18}$$

Note that the term $q_\pi(s_{t+1}, a_{t+1})$ from Equation (4.17) is replaced with the previous estimate for the greedy action, $\max_a q_\pi(s_{t+1}, a)$. The full algorithm for Q-learning is provided in Algorithm 4.

The techniques discussed thus far are still considered as "classic" techniques. Though Q-learning is still readily used in practice, the limitation of these techniques to the tabular case (discrete state and action spaces) makes them inapplicable to many real-world problems. The next step discusses the idea of using function approximation to extend these techniques further.

Algorithm 4 Off-policy TD (Q-learning) control for ϵ-greedy policies

Input: $\epsilon \in [0,1]$, $\alpha \in [0,1]$, $\gamma \in [0,1]$
$\pi \leftarrow$ arbitrary ϵ-greedy policy
$q(s,a) \leftarrow$ arbitrary, for all $s \in \mathcal{S}, a \in \mathcal{A}$
repeat
  $S \leftarrow S_0$
  repeat
    $A \sim \pi(\cdot|S)$
    Take action A, observing reward R and next state S'
    $q(S,A) \leftarrow q(S,A) + \alpha\left[R + \gamma \max_a q(S',a) - q(S,A)\right]$
    Update $\pi(\cdot|S)$ to be ϵ-greedy
    $S \leftarrow S'$
  until S is terminal
until stopping condition met
Return: Policy π

4.5 Function Approximation

The techniques discussed thus far are applicable in the tabular case, i.e. where both the state and action spaces are discrete. Unfortunately, many real-world problems do not fall within this category. Instead of having a tabular representation of the value function, it is possible to instead approximate the function by using a parameterized functional representation.

The approximated value function for state s with a set of parameters $\boldsymbol{\theta}$ is notated as $\hat{v}_\pi(s;\boldsymbol{\theta}) \approx v_\pi(s)$. The objective of reinforcement learning then becomes to find a set of parameters that minimizes the difference between $\hat{v}_\pi(s)$ and $v_\pi(s)$, from which an optimal greedy or ϵ-greedy policy can be derived as before.

It is also common to instead label the set of parameters per policy, i.e. to instead use $\hat{v}(s,\boldsymbol{\theta}_\pi)$. This will be the preferred notation in this study as it emphasises the fact that only the parameters change in subsequent iterations. Furthermore, it is assumed from now on that the state is represented as a vector of d real-valued features, $\boldsymbol{s} \in \mathbb{R}^d$.

Many different functional forms exist that can be used as value function representations. Examples include decision trees, many forms of multivariate regression, and artificial neural networks which are introduced in this section. Perhaps the most well-studied form is the simple linear function,

$$\hat{v}(\boldsymbol{s};\boldsymbol{\theta}_\pi) = \boldsymbol{\theta}_\pi \cdot \boldsymbol{s} = \sum_{i=1}^{d} \theta_{\pi i} s_i, \tag{4.19}$$

where the parameters, $\boldsymbol{\theta} \in \mathbb{R}^d$, are a vector with the same dimensionality as the state vector. Although a linear representation has strong theoretical guarantees [51], it is fairly limited in the complexity of functions it can approximate and are therefore not commonly used in practice.

Arguably, the most common functional representation used today is that of Artificial Neural Networks (ANN). A comprehensive introduction to ANNs is presented in [16]. ANNs are able to approximate complex non-linear functions, which make them very strong candidates for the representation of value functions.

A common way to visualise neural networks are as a connected graph, as shown in

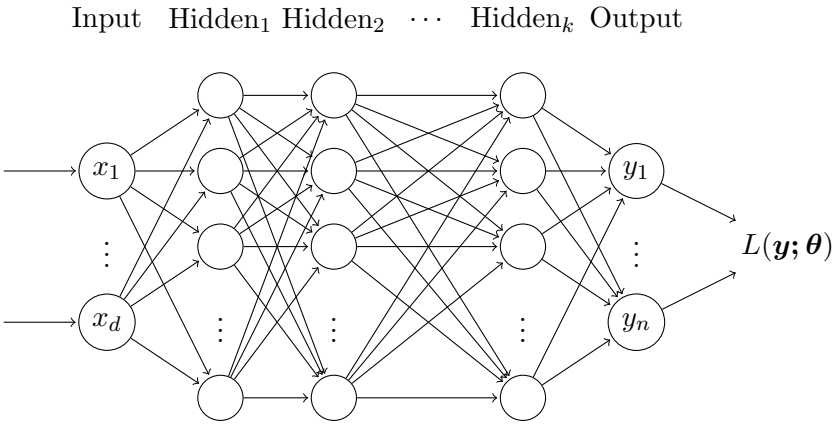


Figure 4.1: An illustration of an Artificial Neural Network (ANN) comprising of an input, an output, and k hidden layers. Each node (excluding the input layer) consists of a weighted sum of the previous layer's nodes, and the resulting sum is passed through an activation function. The final node, $L(\boldsymbol{y}, \boldsymbol{\theta})$, represents the loss function for output $\boldsymbol{y}$ with weights $\boldsymbol{\theta}$.

Figure 4.1. When looking at the graph, each vertical line of nodes, or *units*, is referred to as a *layer* of the network. Each unit is itself a linear function with its own weight vector, where the inputs to the unit is the vector of outputs from the previous layer's units. The output of each linear unit is then passed through a non-linear *activation function*, before being passed to the next layer.

Mathematically, it is more convenient to use a matrix to represent the parameters of each layer. Each layer can then be represented as a single matrix multiplication:

$$\boldsymbol{y}_i = f(\boldsymbol{y}_{i-1}^T \boldsymbol{W}_i), \tag{4.20}$$

where $\boldsymbol{y}_i$ is the output and $\boldsymbol{W}_i$ is the parameter matrix of layer i. Each column of $\boldsymbol{W}_i$ can be thought of as the weights of each linear unit in the layer. The resulting output vector is then passed through the activation function ($f\colon \mathbb{R}^n \to \mathbb{R}$). Note that the activation functions typically only take a scalar value as input, in which case the function is applied element-wise to the vector resulting from the matrix multiplication.

The input layer can be thought of as an "identity layer", where $\boldsymbol{x}_0 = \boldsymbol{x}$ is the input to the neural network. The final layer is referred to as the output layer, and all layers between the input and output layers are referred to as hidden layers. The efficacy of a neural network depends on the choice of its *architecture*, which mainly involves configuring the number of hidden layers, the number of units per hidden layer, and the choice of activation functions.

The number of hidden layers and number of units per layer affect the representational power of the neural network. A layer with an m-dimensional input vector and n units will have an $(m \times n)$ weight matrix. Although the representational power increases, the number of individual parameters that need to be optimised also increases as more layers and units are added. This makes it more difficult to optimise the neural network, meaning that the challenge is often to find as few units and layers as possible to effectively approximate a function.

Finally, the choice of activation functions also impact the ANN's representational power and may affect how easy it is to train or optimise the ANN. Commonly used activation functions include tanh, relu and sigmoid, defined as follows:

$$\tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1} \tag{4.21}$$

$$\text{relu}(x) = \max\{0, x\} \tag{4.22}$$

$$\text{sigmoid}(x) = \frac{e^{x}}{e^{x} + 1} \tag{4.23}$$

Feature engineering is also an important factor in the successful application of a neural network. Feature engineering involves choosing the values that form the input vector $\boldsymbol{x}$, as well as optional steps such as whether or not to normalize or clamp the ranges of values. In the context of reinforcement learning, this involves choosing observations that will make up the state vector.

Optimising the parameters of an ANN (also referred to as training or fitting the model) usually involves minimizing a loss function, which defines the magnitude of error between the output of the ANN and the expected ground-truth output. The loss function is minimized using a process of *gradient descent*.

This involves first calculating the partial derivatives of the loss function's output with respect to each individual parameter using the chain rule. This step is called *back-propagation* and the result is a gradient vector for the set of parameters:

$$\nabla_{\boldsymbol{\theta}} L(\boldsymbol{x}; \boldsymbol{\theta}) = \left[\frac{\partial L}{\partial \theta_1}, \frac{\partial L}{\partial \theta_2}, \ldots, \frac{\partial L}{\partial \theta_n} \right] \tag{4.24}$$

where $L(\boldsymbol{x}; \boldsymbol{\theta})$ is the loss value of the ANN's output given the input $\boldsymbol{x}$. With the gradient computed, the parameters are then iteratively updated using

$$\boldsymbol{\theta}' = \boldsymbol{\theta} + \alpha \nabla_{\boldsymbol{\theta}} L(F(\boldsymbol{x})), \tag{4.25}$$

where α is a learning rate coefficient. Other variants of gradient descent exist such as Adam [27] and AdaGrad [11], but a discussion thereof is not within the scope of this study.

Note that the form of neural network discussed in this section is referred to as a *feed forward* neural network (FFNN), as it contains no graph connections that feed into previous layers and contains only *fully connected* layers, where all units of the previous layer are connected to all units of the next. Other neural networks involve recurrent neural networks (RNN) and convolution neural networks (CNN), but are outside the scope of this discussion.

A noteworthy example of using an ANN with the Q-learning algorithm is the famous Deep Q-Network (DQN) used to teach an agent to play Atari's Breakout directly from a video feed, which was able to learn super-human performance on the arcade game [34]. The remainder of this study however directs its attention to a different way of applying function approximation – by directly learning a policy that maps states to actions.

4.6 Policy Gradient

Policy gradient methods [58] take a different approach to reinforcement learning. Techniques discussed thus far attempt to learn a value function and then use the value of a state-action pair to choose the best action. Policy gradient techniques instead directly learn a parameterized policy function, $\pi(a|s; \boldsymbol{\theta})$, where $\boldsymbol{\theta}$ again represents the set of parameters. This section first discusses how a policy can be represented by a function, then

discusses the policy gradient theorem and the first technique that used policy gradients. It looks at a more recent policy gradient technique called Proximal Policy Optimisation (PPO).

4.6.1 Policy Representation

In the case of discrete actions, the output of the policy function is a real-valued vector, denoted as $\boldsymbol{h} \in \mathbb{R}^{|\mathcal{A}|}$, with the same dimensionality as there are actions. The elements of this vector can be thought of as preferences for each action, where a higher number indicates a stronger preference for one action over another. The probability of taking each action can then be calculated using the soft-max function, as follows:

$$\pi(a|s;\boldsymbol{\theta}) = \frac{e^{h(a|s;\boldsymbol{\theta})}}{\sum_{a' \in \mathcal{A}} e^{h(a'|s;\boldsymbol{\theta})}} \tag{4.26}$$

where $h(a|s;\boldsymbol{\theta})$ is the output of the vector $\boldsymbol{h}$ for the action a. This representation ensures that the individual action probabilities sum to 1 to form a probability distribution.

As noted by [52], it is possible to use the approximated action-values as preferences, however the action-values will converge on their true values and therefore limit the amount by which two actions' preferences may differ. By instead using the policy gradient approach, there is no limit on how much one action can be preferred over another, allowing a policy to eventually become deterministic.

One of the great advantages of policy gradients is that it is possible to represent continuous actions, i.e. where $\boldsymbol{a} \in \mathbb{R}^d$. This is impossible (or at least, very difficult) to do using approaches that learn value functions. A common representation is to learn a Gaussian distribution, such that the output of the policy function is the mean and standard deviation of a d-dimensional distribution. The action vector can then be sampled from this distribution as

$$a_k \sim N(\mu_k, \sigma_k), \tag{4.27}$$

where μ_k and σ_k are outputs from the policy function. Some applications instead fix the value of σ_k and only learn the means of the distribution.

Another result of using policy gradient is that it is not required to construct an ϵ-greedy policy. The policy is inherently stochastic and facilitates sufficient exploration to visit all states, especially during early stages of training.

4.6.2 Policy Gradient Theorem

In order to learn an optimal policy, the parameters are updated using a gradient-based algorithm, as discussed in the case of ANNs in Section 4.5. However, unlike in the traditional case of supervised learning for neural networks, there is no loss function to minimize using gradient descent. Instead, gradient *ascent* is used to maximize the parameters with respect to some performance measure of the policy.

The simplest performance criteria for a policy is the value of the initial state in an episode, i.e. $v_\pi(s_0)$. The parameters can therefore be optimised by estimating the gradient $\nabla_{\boldsymbol{\theta}} v_\pi(s_0)$, and updating the parameters according to Equation (4.28), where α is the learning rate.

$$\boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} + \alpha \nabla_{\boldsymbol{\theta}} v_\pi(s_0). \tag{4.28}$$

The state distribution, denoted as $\mu(s)$, defines the proportion of time that is spend in state s under the current policy, with the distribution over all states summing to 1. The difficulty of using Equation (4.28) is in finding the gradient $\nabla v_\pi(s_0)$, as this relies on knowing how changing the policy parameters will affect the state distribution, which is typically unknown.

Fortunately, a result known as the *Policy Gradient Theorem* provides a way to calculate a gradient that is proportional to $\nabla v_\pi(s_0)$. The result is provided in Theorem 4.6.1. A proof thereof can be found in [32] and [52].

Theorem 4.6.1
Given that π is a parameterized policy with parameters $\boldsymbol{\theta}$, the following proportion holds:

$$\nabla_{\boldsymbol{\theta}} v_\pi(s_0) \propto \sum_{s \in \mathcal{S}} \mu(s) \sum_{a \in \mathcal{A}} q_\pi(s,a) \nabla_{\boldsymbol{\theta}} \pi(a|s;\boldsymbol{\theta}), \tag{4.29}$$

where $\mu(s)$ is the state distribution under the policy π.

As a result of Theorem 4.6.1, the proportional gradient can be calculated without having to find $\nabla\mu(s)$, however the value of $\mu(s)$ still needs to be estimated. Furthermore, the summation over all actions in Equation (4.29) is less than ideal as it will become an expensive integral in the case of continuous actions.

The first policy gradient algorithm, called REINFORCE, was introduced by [58]. The update equation of REINFORCE replaces both the summation over states and actions in Equation (4.29) with expectations. The first replacement is to simply replace the summation over actions weighted by $\mu(s)$ with an expectation:

$$\begin{aligned} \nabla v_\pi(s_0) &\propto \sum_s \mu(s) \sum_a q_\pi(s,a) \nabla \pi(a|s;\boldsymbol{\theta}) \\ &= \mathop{\mathbb{E}}_{\pi} \left[\sum_a q_\pi(s_t,a) \nabla \pi(a|s_t;\boldsymbol{\theta}) \right] \end{aligned} \tag{4.30}$$

Replacing the action summation is not as simple, as the summation is not weighted by the probability of each action (i.e. it can not be readily replaced by an expectation). This is changed into an expectation by multiplying and then dividing by the probability of selecting each action, $\pi(a|s;\boldsymbol{\theta})$, which provides an equation that relies only on expectations:

$$\begin{aligned} \nabla v_\pi(s_0) &\propto \mathop{\mathbb{E}}_{\pi} \left[\sum_a q_\pi(s_t,a) \nabla \pi(a|s_t;\boldsymbol{\theta}) \right] \\ &= \mathop{\mathbb{E}}_{\pi} \left[\pi(a_t|s_t;\boldsymbol{\theta}) \sum_a q_\pi(s_t,a) \frac{\nabla \pi(a_t|s_t;\boldsymbol{\theta})}{\pi(a_t|s_t;\boldsymbol{\theta})} \right] \\ &= \mathop{\mathbb{E}}_{\pi} \left[q_\pi(s_t,a_t) \frac{\nabla \pi(a_t|s_t;\boldsymbol{\theta})}{\pi(a_t|s_t;\boldsymbol{\theta})} \right] \end{aligned} \tag{4.31}$$

REINFORCE then continues to use Monte Carlo techniques to estimate the expectation and, as a result, the proportional policy gradient. This is shown in Equation (4.32) below, whereby the expected value of $q(s_t,a_t)$ is simply estimated as the true return of the current episode. The final result of Equation (4.32) is a more succinct form following from the identity $\nabla \ln x = \frac{\nabla x}{x}$.

$$\begin{aligned}\nabla v_\pi(s_0) &\propto \mathop{\mathbb{E}}_{\pi}\left[q_\pi(s_t, a_t)\frac{\nabla\pi(a_t|s_t;\boldsymbol{\theta})}{\pi(a_t|s_t;\boldsymbol{\theta})}\right] \\ &\approx G_t\frac{\nabla\pi(a_t|s_t;\boldsymbol{\theta})}{\pi(a_t|s_t;\boldsymbol{\theta})} \\ &= G_t\nabla\ln\pi(a_t|s_t;\boldsymbol{\theta})\end{aligned} \tag{4.32}$$

This approximation of the gradient can now be plugged into Equation (4.28) to create the resulting REINFORCE update equation. The full REINFORCE algorithm is presented in Algorithm 5.

Algorithm 5 REINFORCE policy gradient control for policy π

Input: $\alpha \in [0,1]$, $\gamma \in [0,1]$, policy representation $\pi(a|s,\boldsymbol{\theta})$
$\theta_k \leftarrow$ arbitrary, for all $k \in 1,\ldots,|\boldsymbol{\theta}|$
repeat
  Generate a trajectory, $s_0, a_0, r_1, \ldots, s_{T-1}, a_{T-1}, r_T, s_T$, using $\pi(\cdot|\cdot;\boldsymbol{\theta})$
  $G \leftarrow 0$
  for $t = 0, 1, \ldots, T-1$ **do**
    $G \leftarrow \sum_{k=t+1}^{T}\gamma^{k-t-1}r_k$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha\gamma^t G\nabla\ln\pi(a_t|s_t;\boldsymbol{\theta})$
  end for
until stopping condition met
Return: Policy π

4.6.3 Advantage Estimation

Before continuing the discussion of policy gradient methods, a brief discussion on *advantage estimation* [44] is required. The use of the action-value function ($q_\pi(s_t, a_t)$) in the formulation of the policy gradient in Equation (4.30) can be replaced with a number of equivalent expressions. One particularly useful expression is the *advantage function*, defined as

$$A_\pi(s_t, a_t) = q_\pi(s_t, a_t) - v_\pi(s_t), \tag{4.33}$$

and the policy gradient can then be formulated as

$$\nabla v_\pi(s_0) \propto \mathop{\mathbb{E}}_{\pi}\left[\sum_a A_\pi(s_t, a)\nabla\pi(a|s_t;\boldsymbol{\theta})\right]. \tag{4.34}$$

The use of the advantage function provides lower variance policy gradients compared to using only the action-value function, but the advantage function must also be estimated since neither of the true state- or action-value functions (v_π or q_π, respectively) will be known. The generalized advantage estimator (GAE) provides a method of estimating the advantage function when given an approximation of the state-value function, $v_\pi(s_t)$ [44].

Firstly, define the TD residual of the state-value function as $\delta_t = r_t + \gamma v_\pi(s_{t+1}) - v_\pi(s_t)$. The generalised advantage estimate is then

$$\hat{A}_t = \sum_{l=0}^{\infty}(\gamma\lambda)^l\delta_{t+l}, \tag{4.35}$$

where λ is a parameter in the range $[0,1]$ that controls the trade-off between higher variance or increased bias as a result of bootstrapping. Use of a higher lambda value reduces bias,

and conversely a lower value of lambda increases bias but reduces variance. A value of λ close to 1 is typically used to introduce a small amount of bias from bootstrapping, but reduces the variance sufficiently to improve the stability of policy gradient techniques.

The generalised advantage estimator still requires an approximation of the state-value function, which is typically done by learning an approximate function using an Artificial Neural Network, which takes the state s_t as input, and estimates the value of $v_\pi(s_t)$.

4.6.4 Proximal Policy Optimisation

A problem with optimising a policy using policy gradients is the non-stationarity. As the policy is updated, the state distribution and value functions change and thus the true direction of the gradient changes from what is originally estimated. This leads to policy updates possibly being detrimental if too many policy optimisation iterations are performed before estimating the value of the new policy.

The first technique to address this issue was Trust-Region Policy Optimisation (TRPO), which uses a theoretical result to put a constraint on the policy optimisation as to limit the effect of the policy update [43]. As in the case of REINFORCE, the objective is to maximize the performance of the policy, measured as $v_\pi(s_0)$. The objective and the constraint is formulated as

$$\begin{aligned} &\underset{\boldsymbol{\theta}}{\text{maximize}} \ \sum \mu_{\boldsymbol{\theta}'}(s) \sum \pi(a|s;\boldsymbol{\theta}) \hat{A}(s,a) \\ &\text{subject to } \bar{D}_{KL}^{\mu_{\boldsymbol{\theta}'}}(\pi_{\boldsymbol{\theta}} || \pi_{\boldsymbol{\theta}'}) < \delta, \end{aligned} \tag{4.36}$$

where $\boldsymbol{\theta}'$ denotes the previous set of parameters. Note that the maximization objective is simply the expansion of $v_\pi(s_0)$. The left side of the constraint is an approximation of the Kullback-Leibler Divergence (D_{KL}) between the old and new policy probability distributions, which provides a metric of how different the distributions are. The constraint therefore limits the D_{KL} between $\boldsymbol{\theta}$ and $\boldsymbol{\theta}'$ to the specified value of δ.

It is important to note that both the objective and constraint in Equation (4.36) are changed to approximations for use with Monte Carlo techniques to arrive at the final, practical TRPO algorithm, however the details thereof are outside the scope of this discussion.

Although the addition of the trust region successfully addresses the issue at hand, it is rarely used in practice due to being limited to second order gradient techniques. A more recent technique, known as Proximal Policy Optimisation (PPO), addresses this by instead framing the optimisation constraint as a penalty on the objective function [45].

First, let $r_t(\boldsymbol{\theta}) = \frac{\pi_{\boldsymbol{\theta}}(a_t|s_t)}{\pi_{\boldsymbol{\theta}'}(a_t|s_t)}$, where $\boldsymbol{\theta}'$ is the previous set of policy parameters. Maximizing the following surrogate objective function, known as conservative policy iteration, was proposed by [23]:

$$\underset{\boldsymbol{\theta}}{\text{maximize}} \ \mathbb{E}\left[\frac{\pi_{\boldsymbol{\theta}}(a_t|s_t)}{\pi_{\boldsymbol{\theta}'}(a_t|s_t)} \hat{A}(s_t,a_t)\right] = \mathbb{E}\left[r_t(\boldsymbol{\theta}) \hat{A}(s_t,a_t)\right] \tag{4.37}$$

The PPO authors note that optimising this objective without a constraint as put in place by TRPO can lead to possibly divergent updates. The authors propose a penalty on the above objective as follows:

$$\underset{\boldsymbol{\theta}}{\text{maximize}} \ \mathbb{E}\left[\min(\ r_t(\boldsymbol{\theta})\hat{A}(s,a),\ \text{clip}(r_t(\boldsymbol{\theta}), 1-\epsilon, 1+\epsilon)\hat{A}(s,a)\ )\right] \tag{4.38}$$

Note that the first case of the min is the objective function from Equation (4.37). The second case is where the penalty is applied. This case is only different from the first if

$r_t(\boldsymbol{\theta}) > 1+\epsilon$ or $r_t(\boldsymbol{\theta}) < 1-\epsilon$, where ϵ is a hyper-parameter, usually set to 0.2. The second case therefore penalises the objective function if the new policy becomes too different from the old, removing the incentive to make overly large changes to the policy parameters.

Perhaps the most important difference between the penalty approach from PPO and the constraint used by TRPO is that the objective in Equation (4.38) can still be solved using first order optimisation techniques such as gradient ascent. As a result, PPO is easier to apply in practice, and has seen extensive application in recent years.

This concludes the discussion on both particle swarm optimisation and reinforcement learning. The background provided in this section will be used to guide the proposed research, which is further explained in the remaining sections.

5 | The Continuous Reinforcement Learning Particle Swarm Optimisation Environment

This chapter discusses the different components that make up the CRLPSO environment in which an agent can act. This chapter is closely related to Chapter 6, which focuses on the design of the policy and the application of the RL algorithm.

As there are many choices for the RL components discussed in this chapter and the next, some sections provide small experimental results to motivate the design choices. These experiments typically rely on the optimisation of a policy for which details are only discussed in the next chapter, however these details are not important for the interpretation of the results.

The goal of an RL agent's policy in the context of PSO is to choose the values of w, c_1, and c_2 at each iteration in order to maximize the optimisation ability (i.e. the ability to achieve a good fitness value in the duration of a run) of the PSO algorithm. A single step of the RL agent therefore corresponds to a single iteration of the PSO algorithm. A single episode then corresponds to a single run of the PSO algorithm on a specific benchmark function. The implementation of the CRLPSO environment is provided in Algorithm 6, which is based on the REINFORCE algorithm [58] provided previously in Chapter 4 as Algorithm 5. Note that Algorithm 6 can be easily adapted for other policy gradient algorithms such as PPO by changing the policy gradient calculations and policy update step, but is shown with REINFORCE for clarity due to its simplicity.

Note that this implementation of the PSO algorithm makes use of velocity clamping. During experiments it was found that velocity clamping prevents the case where velocities grow too large during training when a policy's coefficient choices cause divergent behaviour.

The remaining sections of this chapter describe the pieces of the CRLPSO environment, namely the action space in Section 5.1, the state space in Section 5.2, and the reward function definitions in Section 5.3.

5.1 Action Space

The main purpose of this study is to investigate the feasibility of a policy that uses a continuous action space. It is however important to first consider discrete action spaces in order to understand what the benefit of the more complex continuous action space may be. Therefore, this section first investigates and discusses discrete action spaces in Section 5.1.1 before describing the continuous action space that is used in the remainder of this study in Section 5.1.2.

Algorithm 6 Policy gradient control for a Particle Swarm Optimisation policy π

Input: Learning rate $\alpha \in [0, 1]$, discount factor $\gamma \in [0, 1]$, policy representation $\pi(a|s, \boldsymbol{\theta})$, objective function $f\colon \mathbb{R}^D \to \mathbb{R}$ where D is the function dimensionality, swarm size N, search space (x_{min}, x_{max}), iteration limit T.

for $i \in 1, 2, \ldots, N$ **do**
  $\boldsymbol{x}_i(0) \sim U(x_{min}, x_{max})^D$ {Initialise positions}
  $\boldsymbol{v}_i(0) \leftarrow \mathbf{0}^D$ {Zero-initialise velocities}
  $\boldsymbol{q}_i(0) \leftarrow \boldsymbol{x}_i(0)$ {Initialise personal best}
end for
$\hat{\boldsymbol{q}}(0) \leftarrow \operatorname{argmin}_i f(\boldsymbol{q}_i(0))$ {Initialise global best}
$v_{max} \leftarrow x_{max} - x_{min}$ {Determine velocity clamping range}
$v_{min} \leftarrow -v_{max}$

$\theta_k \leftarrow$ arbitrary, for all $k \in 1, \ldots, |\boldsymbol{\theta}|$ {Initialise policy parameters}

repeat
  Calculate state S_0 (see Section 5.2)
  for $t = 0, 1, \ldots, T - 1$ **do**
    $A_t = (w(t), c_1(t), c_2(t)) = \pi_{\boldsymbol{\theta}}(s_t)$ {Choose coefficients according to policy π}
    for $i \in 1, 2, \ldots, N$ **do**
      Set $\boldsymbol{v}_i(t)$ according to Equation (2.1)
      Clamp $\boldsymbol{v}_i(t)$ according to Equation (2.6)
      Set $\boldsymbol{x}_i(t)$ according to Equation (2.2)
      Set $\boldsymbol{q}_i(t)$ according to Equation (2.3)
    end for

    $\hat{\boldsymbol{q}}(t) \leftarrow \operatorname{argmin}_i f(\boldsymbol{q}_i(t))$
    Calculate state S_{t+1}
    Receive reward R_{t+1} (see Section 5.3)
  end for

  $G \leftarrow 0$
  for $t = 0, 1, \ldots, T - 1$ **do**
    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \gamma^t G \nabla \ln \pi(A_t|S_t; \boldsymbol{\theta})$ {REINFORCE policy gradient ascent}
  end for
until stopping condition met
Return: Policy π

Table 5.1: QLPSO action space copied forward from Table 3.4.

Value	Description	w	c_1	c_2
0	Rough exploration	1	2.5	0.5
1	Fine exploration	0.8	2	1
2	Slow convergence	0.6	1	2
3	Fast convergence	0.4	0.5	2.5

5.1.1 Discrete Action Spaces

For the PSO environment, the coefficients w, c_1, and c_2 are continuous variables. These variables therefore have to be discretised in some way in order to create a discrete action space. There are typically two ways to achieve this, namely to either

1. select a finite number of coefficient combinations and allow an agent to choose among them, or

2. quantise each coefficient to a set of discrete values, and let the action space be the Cartesian product of these sets.

The disadvantage of option 1 is that the coefficient values still have to be chosen carefully, and the choice of these values may have a large impact on the agent's performance.

Option 2 is simpler as it does not require choosing coefficient combinations, and merely requires the inclusion of reasonable values for each individual coefficient. In theory, the agent will also have more freedom as each coefficient can be adjusted individually. The disadvantage however is that the size of the action space becomes exponentially larger as you include more options per coefficient, which may require more samples to accurately calculate the policy gradient during training [49], thereby reducing the sample efficiency of the RL algorithm.

Now consider the following two discrete action spaces, defined using each of the options above. The first action space is the same as that used by QLPSO (see Section 3.5), which is an example of option 1. The action space includes 4 carefully chosen coefficient combinations. The action space is shown in Table 5.1 (brought forward from Table 3.4). The local search step that is performed in the last 10% of iterations by QLPSO is excluded for these experiments, as the purpose is to investigate the ability of a policy optimised using RL without relying on manually chosen steps.

For the second option, consider the following product of quantised sets:

$$\begin{aligned} W &= \{0.4,\ 0.8,\ 1.0,\ 1.1\} \\ C_1 &= \{0.5,\ 1.0,\ 2.0,\ 2.5\} \\ C_2 &= \{0.5,\ 1.0,\ 2.0,\ 2.5\} \\ \mathcal{A} &= W \times C_1 \times C_2 \end{aligned} \tag{5.1}$$

Even with only 4 choices for each coefficient, the action space contains a total of $4^3 = 64$ actions to choose from, which is already significantly larger than the 4 actions of the first action space.

Figure 5.1 shows the training curves of agents during training. The agents are trained using the DQN [34] algorithm with the parameters shown in Table 5.2. Each epoch of training is equivalent to one full run of the PSO algorithm, and the agents are provided a dense ranking reward, which is introduced later in Section 5.3 as R_2. For now, it is enough

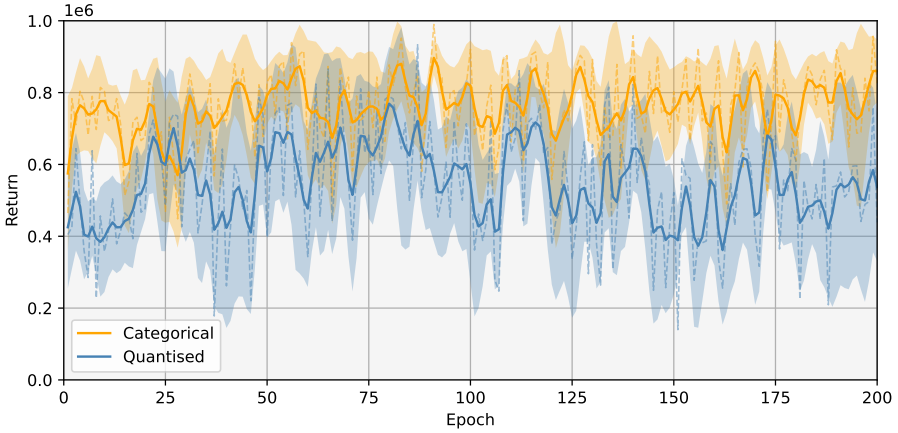


Figure 5.1: Total return of agents during training, with the only difference being the use of either quantised or categorical discrete actions. Results are averaged across 3 independent training runs. Results are smoothed using a Gaussian filter to improve clarity. The dashed lines show the unfiltered results and the shaded areas represent the standard deviation.

Table 5.2: DQN parameters used for conducting discrete action space experiments.

Parameter	Value
Learning Rate	0.0005
Discount Factor (γ)	0.99
Exploration (ϵ)	Linearly annealed from 1 to 0.02 over the first 25 epochs

to know that the total return of the epoch correlates to the optimisation performance of the agent on the given benchmark function. From the figure, it is clear that the categorical actions from option 1 achieve greater returns and thus perform better than the quantised actions.

Clearly it is the case here that providing the agent with coefficient combinations is better than the quantised coefficients, however keep in mind that the performance achievable by the agent will depend greatly on the quality of the coefficients provided. A further investigation of discrete actions is however outside the scope of this research, since the focus is on investigating the use of continuous action spaces. These experiments do however show that discretisation of the action space is not a viable alternative to a continuous action space, because of the additional effort to choose "good" values, which are likely to be problem dependent and therefore will not allow equivalent performance on unseen functions.

5.1.2 Continuous Action Space

Unlike the discrete action spaces above, the use of a continuous action space will allow an agent to learn optimal policies without requiring careful design of the action space. For the continuous case, and for the remainder of this study, the action space of the agent is the real-valued vector $\boldsymbol{a} \in \mathbb{R}^3$, corresponding to a vector of the three coefficient values, $\boldsymbol{a}(t) = [w(t), c_1(t), c_2(t)]$.

Although it is possible to allow the agent direct control of the coefficients, it may be beneficial to still limit the ranges of each of these actions. A simple approach is to clip

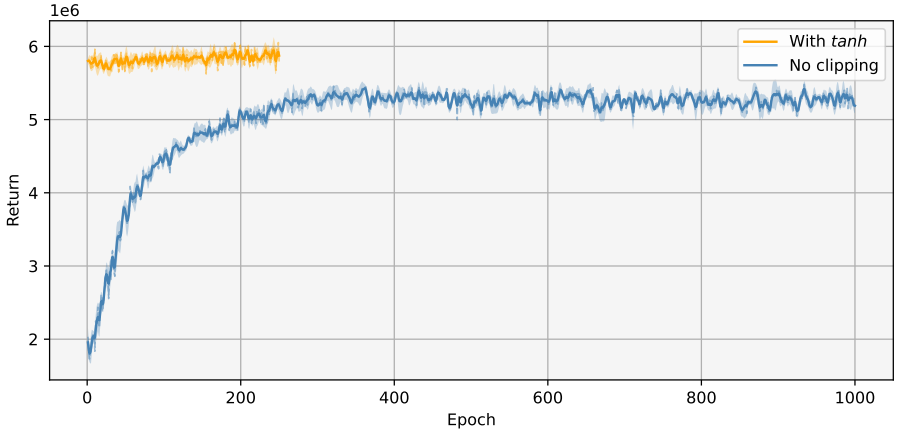


Figure 5.2: Total return of agents during training, with the only difference being the use of tanh clipping or not. Results are averaged across 3 independent training runs. To improve the clarity of the graph, results are smoothed using a Gaussian filter. The dashed lines show the unfiltered results and shaded areas represent the standard deviation.

the values of each action (e.g. $w = \text{clip}(a_1, -1, 1)$). An alternative is to instead pass the values through a tanh function and scale the output to the correct range. This approach will be used as it still allows the policy to make use of the full range of real values.

To disambiguate between the actual output of the policy and the final action values, the policy output will be denoted as $\boldsymbol{h}_a = [h_w, h_{c_1}, h_{c_2}]$, and the final mapped values are denoted as $\boldsymbol{a} = [w_1, c_1, c_2]$. Note that the values of these coefficients will vary between iterations, but the time index t is omitted for the sake of clarity.

For the purposes of this study, the policy outputs are mapped according to

$$\begin{aligned} w &= 1.1 \cdot \tanh(h_w), \\ c_1 &= 1.25 \cdot (\tanh(h_{c_1}) + 1), \\ c_2 &= 1.25 \cdot (\tanh(h_{c_2}) + 1). \end{aligned} \tag{5.2}$$

This limits the values of w, c_1, and c_2 to $(-1, 1)$, $(0, 2.5)$, and $(0, 2.5)$, respectively. The choice of these values is based on analyses of the typical PSO hyperparameter landscape as discussed in Section 2.2.

In order to motivate this design choice, Figure 5.2 shows the training curves of two different agents. The setup of the agents are identical, except for the mapping of action values. The first agent uses no mapping, i.e. $\boldsymbol{a} = \boldsymbol{h}_a \in \mathbb{R}^3$, and the second uses the mapping from Equation (5.2). As is evident from the figure, the returns achieved by the policy using tanh clipping is higher than that achieved without clipping. The policy without clipping is not even able to achieve the same returns after 1000 epochs of training than an untrained policy with clipping achieves at the first epoch.

5.2 State Space

This section describes the features that make up the state space of the environment. A number of features are proposed that describe the current state of the PSO algorithm, while adhering to the following two constraints:

- The feature may not require any prior information about the underlying function, other than the bounds of the search space.
- The range of each feature must be independent of the PSO hyper-parameters, such as the swarm size or number of iterations.

The first constraint allows the trained policy to be applied to unknown functions, which is of course a strict requirement for any black-box optimisation technique. The second constraint provides additional flexibility by allowing a practitioner to tune the PSO hyper-parameter values for additional performance gains without having to retrain the policy. All features are also normalized using a running average normalization as described in Section 6.1 to avoid any bias towards a specific feature.

The remainder of this section defines the 28 features that make up the state space. The state space is therefore a 28-dimensional vector, and the values of each feature are evaluated at the end of each PSO iteration as was described in Algorithm 6. The features are also listed in Table 5.4.

The first feature, $S_1(t) = \frac{t}{T}$, calculates the progress of the run. The next three, $S_2(t)$ to $S_4(t)$, are the values of the coefficients chosen for the previous iteration, $c_1(t-1)$, $c_2(t-1)$, and $w(t-1)$ respectively.

The following two features provide metrics of how fast the swarm is converging and how widely spread the particles are, using the convergence and diffusion factors defined by Improved PSO [28]. The features are the mean convergence and mean diffusion factors across the swarm,

$$S_5(t) = \sum_{i \in N} \frac{c_i(t)}{N}, \text{ and} \tag{5.3}$$

$$S_6(t) = \sum_{i \in N} \frac{d_i(t)}{N}, \tag{5.4}$$

where $c_i(t)$ is the convergence factor of particle i as defined in Equation (3.5), and $d_i(t)$ is the diffusion factor of particle i as defined in Equation (3.6).

The next feature is the dimensionality of the function as a fraction of D_{max}, $S_7(\cdot) = \frac{D}{D_{max}}$, to enable the policy to learn different behaviours for different dimensionalities. For this study, D_{max} is set to 100, which implies that the agent should not be used to optimise higher dimension problems after training as this may lead to unexpected behaviours.

The following two features measure the stagnation and improvement rate of the swarm and are closely related. The stagnation is measured as the number of timesteps that the global best does *not* improve, defined as

$$\begin{aligned} S_8(0) &= 0 \\ S_8(t) &= \begin{cases} S_8(t-1) + 1 & \text{if } f(\hat{\boldsymbol{q}}(t)) = f(\hat{\boldsymbol{q}}(t-1)) \\ S_8(t-1) & \text{otherwise.} \end{cases} \end{aligned} \tag{5.5}$$

The improvement rate is then the number of timesteps that the global best *does* improve, defined as

$$\begin{aligned} S_9(0) &= 0 \\ S_9(t) &= \begin{cases} S_9(t-1) + 1 & \text{if } f(\hat{\boldsymbol{q}}(t)) < f(\hat{\boldsymbol{q}}(t-1)) \\ S_9(t-1) & \text{otherwise.} \end{cases} \end{aligned} \tag{5.6}$$

The following two features provide the mean and standard deviation of the particle velocities, respectively. Note that the mean and standard deviation are calculated using the absolute component values of the velocity vectors.

$$S_{10}(t) = \frac{1}{N} \sum_{i \in N} ||\boldsymbol{v}_i||_1 \tag{5.7}$$

$$S_{11}(t) = \sqrt{\sum_{i \in N} \sum_{k \in D} \frac{\left[|v_{i,k}| - S_{10}(t)\right]^2}{D \cdot N}} \tag{5.8}$$

The next set of four features use inter-particle distances to provide metrics of how far the swarm is spread apart and how much of the search space is covered. The first is the swarm diversity, which is calculated as the mean inter-particle distance,

$$S_{12}(t) = \frac{1}{N^2} \sum_{i \in N} \sum_{j \in N} ||\boldsymbol{x}_i - \boldsymbol{x}_j||_2. \tag{5.9}$$

The next two measure the mean distance of particles from two points: the swarm center,

$$\begin{aligned} S_{13}(t) &= \frac{1}{N} \sum ||\boldsymbol{x}_i - \boldsymbol{\mu}_{\boldsymbol{x}}||_2, \\ \text{where } \boldsymbol{\mu}_{\boldsymbol{x}} &= \frac{1}{N} \sum \boldsymbol{x}_i, \end{aligned} \tag{5.10}$$

and the global best position,

$$S_{14}(t) = \frac{1}{N} \sum ||\boldsymbol{x}_i - \hat{\boldsymbol{q}}||_2. \tag{5.11}$$

The final distance metric measures the mean distance between each particle and its own personal best position,

$$S_{15}(t) = \frac{1}{N} \sum ||\boldsymbol{x}_i - \boldsymbol{q}_i||_2. \tag{5.12}$$

The following two features provide the mean and standard deviation of the fitness, normalized using the current global best and worst fitness values (f_{best} and f_{worst}, respectively). The worst fitness is calculated only once at $t = 0$ as the worst fitness value measured after the swarm is initialized.

$$S_{16}(t) = \frac{\frac{1}{N} \sum y_i - f_{best}}{f_{worst} - f_{best}} \tag{5.13}$$

$$S_{17}(t) = \frac{1}{\sigma_{max}} \sqrt{\sum \frac{(y_i - \mu_y)^2}{N}}, \tag{5.14}$$

where σ_{max} is the standard deviation when half of the population has fitness equal to the global best, and half equal to the global worst.

The next three features, S_{18} to S_{20}, are the coefficient values as would be calculated by PSO-TVAC, as in Equation (3.1). The values of $(c_{1s}, c_{1f}) = (2.5, 0.5)$, $(c_{2s}, c_{2f}) = (0.5, 2.5)$, and $(w_s, w_f) = (0.9, 0.4)$ are used.

The remaining features are derived from the theoretical discussion of PSO in Sections 2.2 and 2.3. The next five, S_{21} to S_{25}, are the five eigenvalues of the matrix $\boldsymbol{B}$

defined in Equation (2.9). The values of α and β are used as defined in Equation (2.12). Since the expected values in the matrix (e.g. $\mathbb{E}[\alpha]$) are not known, the immediate values are calculated with $c_1(t-1)$, $c_2(t-1)$, and $w(t-1)$ instead.

The appendix of [7] contains equations to directly calculate the 5 eigenvalues. In the CRLPSO implementation, the matrix $\boldsymbol{B}$ is instead first calculated, with the eigenvalues then being provided by the `eigvals`[1] function of the Numpy package.

The next feature is the autocorrelation defined in Equation (2.16) for $i = 1$, i.e. ρ_1. The expected values $\mathbb{E}[l]$ and $\mathbb{E}[w]$ are again not known and the immediate values are substituted instead:

$$S_{26}(t) = \frac{1 + w(t) - \frac{c_1(t)}{2} - \frac{c_2(t)}{2}}{w(t) + 1} \tag{5.15}$$

Next is the expected movement distance, $S_{27}(t) = \mathbb{E}[d(t)]$ (Equation (2.18)). The value of ρ_1 is calculated as in Equation (5.15), and $\mathbb{V}[\boldsymbol{x}]$ is estimated using the last $n = 10$ particle positions. The final feature is then the search focus $S_{28}(t) = F(t)$ as defined in Theorem 2.3.3, using immediate values for r_1, r_2, c_1, and c_2.

5.3 Reward Functions

This section defines the reward functions that will be considered. The choice of reward function has a direct impact on the final performance of a trained agent, and also impacts the difficulty of training the agent as some reward functions may be easier to optimise for than others. This section therefore proposes 7 different reward functions with different characteristics, and the efficacy of each will be empirically evaluated in Chapter 6.

Section 5.3.1 first discusses possible issues that arise when choosing reward functions specifically in the context of PSO, and the remainder of the section defines the various reward functions. The rewards defined in this section are summarised in Table 5.3.

5.3.1 Difficulties of Reward Functions

In the case of the PSO environment, the goal is to optimise the given objective function as well as possible. To that end, a reward function must provide a signal to the agent that, when maximised, will increase the optimisation ability of the PSO algorithm. Perhaps the simplest reward function would be to reward the agent with the resulting objective value at the end of a run, negating the value in the case of minimisation problems. This conceptually simple reward however suffers from two problems.

The first problem is that this reward will have a large variance since different benchmark functions have very different objective value ranges. For example, a good value for f_1 can be orders of magnitude larger than for f_2, and by extension, an improvement of say 1000 on the objective value of f_1 is much less significant than an improvement of the same magnitude on f_2.

The second problem is the sparsity of the signal. Many steps have to be taken by the agent before receiving a non-zero reward signal. One possible solution to the sparsity of signal is to reward the agent at every step according to the current global best objective value. This however is no longer a direct representation of the agent's underlying goal and will likely cause an unwanted bias; consider as an example that the reward function may

[1] https://numpy.org/doc/stable/reference/generated/numpy.linalg.eigvals.html#numpy-linalg-eigvals

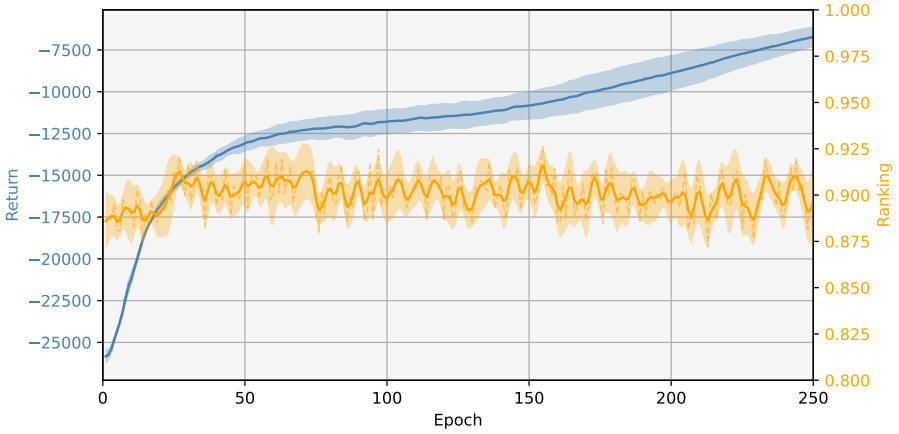


Figure 5.3: The reward and ranking during training of a policy using the ideal velocity reward defined in Equation (5.16). The shaded area represents the standard deviation across three independent experiments.

provide a stronger signal when a good fitness value is kept throughout the run, even if a worse fitness at the start might lead to an even better fitness at the end of a run.

For these reasons, other reward functions must be designed that work around the two above-mentioned problems while also trying to minimise the amount of bias present in the signal. A reward function can be considered biased if it is possible to learn to optimise the reward without actually increasing the optimisation ability of the PSO agent. Most reward functions that do not directly correlate to the result of a PSO run are likely biased, and the bias of each reward function is investigated experimentally in Chapter 6.

As an example of a biased reward function, consider a reward that is maximized if the agent follows the ideal velocity curve of PSO with Velocity Information (see Section 3.2), defined as

$$R(t) = -\left|v_{\text{avg}}(t) - v_{\text{ideal}}(t)\right|, \tag{5.16}$$

where $v_{\text{avg}}(t)$ and $v_{\text{ideal}}(t)$ are as defined in Equations (3.2) and (3.3), respectively.

The training curve of an agent learning to optimise this reward function is shown in Figure 5.3. The agent is trained using the state and action space provided in this chapter and the training setup provided in the next chapter. The optimisation ability of the agent is also measured using the ranking metric, which is defined in the following section. For the purposes of this experiment, it is enough to know that the ranking is a close indication of the optimisation performance, and an increased ranking means that the agent was able to find a better objective value on the given benchmark function.

It is clear that the policy is able to optimise the reward function very well, however the ranking only increases somewhat during the first 25 epochs, but remains nearly stagnant thereafter. This is a clear indication of a bias in the reward, since a significant increase in the return does not improve the optimisation ability of the agent. Ideally the increase in return should be at least somewhat correlated to an increase in ranking, such that an optimised policy provides a better PSO algorithm.

Unfortunately, it is difficult to design a reward function that does not suffer from high variance, has dense signal, and is unbiased. For this reason, biased reward functions are still worth investigating. The remainder of this section first introduces a novel performance metric, and then continues to define a set of 7 reward functions that will be investigated

in Chapter 6.

5.3.2 Ranking-Based Performance Evaluation

In order to provide a lower-variance and unbiased metric of optimisation performance, the *ranking* of a position $\boldsymbol{x}$ for an objective function f is defined as $\text{rank}_f \colon \mathbb{R}^D \to [0, 1]$. The ranking effectively maps the current objective value into the range $[0, 1]$ to provide a metric that has the same range for all objective functions, while maintaining the property that if $f(\boldsymbol{x}) < f(\boldsymbol{x'})$, then $\text{rank}_f(\boldsymbol{x}) \geq \text{rank}_f(\boldsymbol{x'})$. A higher ranking is thus desirable.

The ranking for an objective function is defined by first collecting the results of a number of canonical PSO runs on the objective function with different values for the coefficients w, c_1, and c_2. For the purposes of this study, 25 values for $c = c_1 = c_2$ and 23 values for w are used. The values for c are in the range $[0.1, 2.5]$ and spaced 0.1 apart, and for w are in the range $[-1.1, 1.1]$ and spaced 0.1 apart. A grid of size $(25, 23)$ is then constructed of pairs of (c, w) values, and 10 runs of canonical PSO are performed for each parameter combination and for each objective function. These results are collected separately for different dimensionalities. Note that the choice of only 10 runs is sufficient as these results are used only to collect data points to define the ranking metric, and are not used for any statistical tests or to determine the best coefficients.

This process is performed for each of the 30 benchmark functions in the CEC 2017 benchmark suite [2] (listed in Appendix A) and repeated for dimensions 10, 30, 50, and 100. A swarm size of 30 is used for all runs with 1000, 1500, 2500, and 5000 iterations for each dimension, respectively. With this setup, results of 5750 independent runs for each function and each dimensionality are collected.

The results of these runs (i.e. the object values of the final global best positions) are then listed in ascending order for each objective function. The ranking of a position $\boldsymbol{x}$ on objective function f is then calculated by first finding the index of the first value that is *worse* than the value of $f(\boldsymbol{x})$. The ranking is then calculated as $1 - \frac{i}{n}$ where i is the index found, and n is the number of results collected for the objective function f, which in the case of the setup defined above will always be 5750.

With this definition, it is possible to find positions worse than the lowest ranking, and better than the highest ranking. In these cases, the rankings are calculated as 0 and 1, respectively. However, based on experience, results from a run will very rarely fall outside of these ranges.

Besides providing a performance metric for different objective functions with a consistent range, the ranking is also more reliable in the following sense. When considering only the objective values or a linearly normalized objective value (i.e. $\frac{f(\boldsymbol{x}) - f_{\text{best}}}{f_{\text{worst}} - f_{\text{best}}}$)), there are cases where a small improvement in optimisation capability can lead to large increases in the objective value due to the function landscape. By instead comparing the objective value to other obtained values, the ranking is less prone to such cases. This is illustrated by the plots in Figure 5.4, where the difference between a linear normalization and the ranking are apparent. For a linear ranking, the graphs are expected to form a straight, descending line, but the graphs from Figure 5.4 differ significantly in shape between the functions.

5.3.3 Reward Function Definitions

The first two reward functions are defined to make use of the ranking metric described in Section 5.3.2. The first returns the ranking achieved at the end of a run of T iterations, defined as

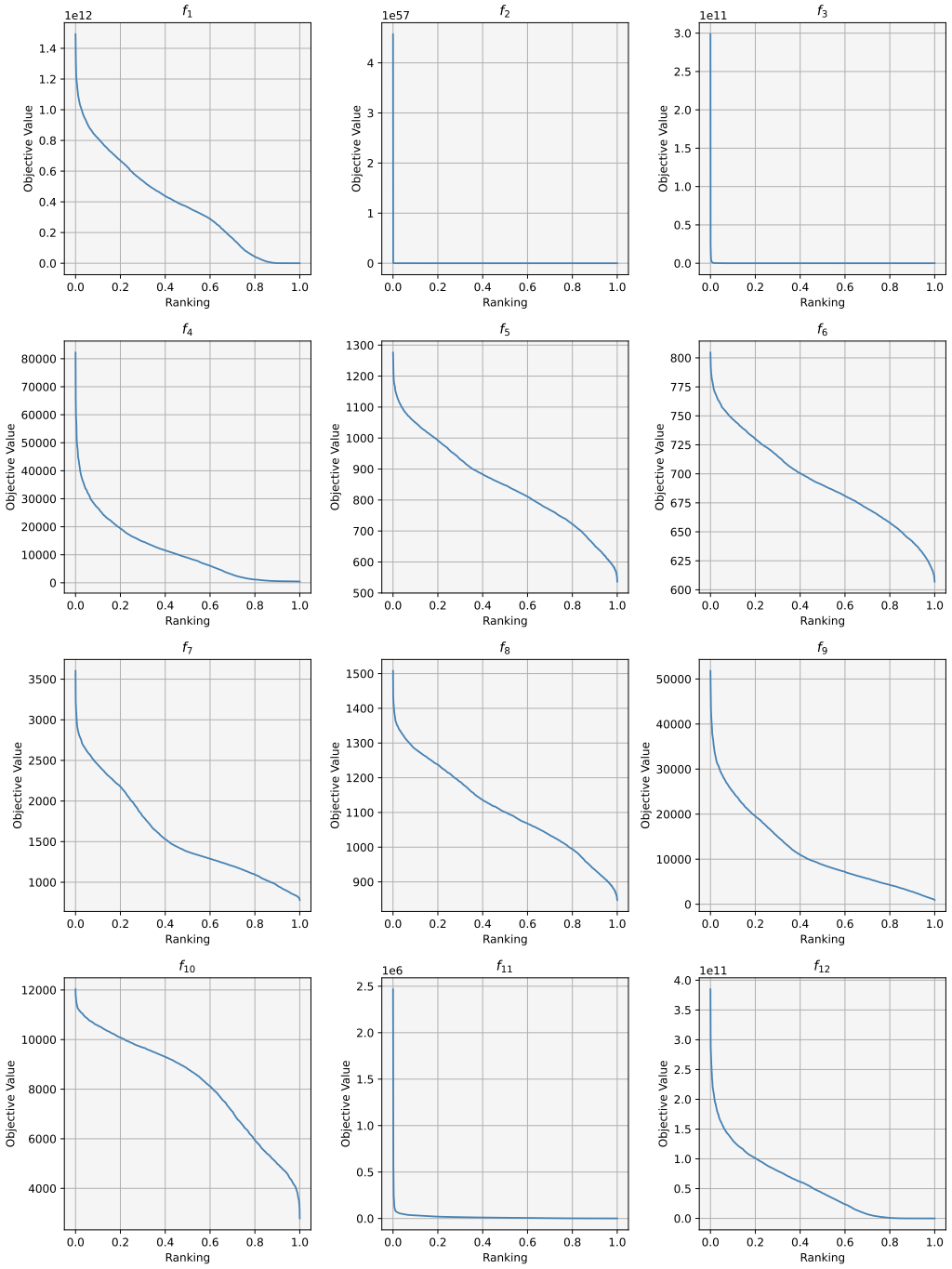


Figure 5.4: Curves showing the objective value as a function of the ranking achieved at that value for the first 12 CEC 2017 objective functions. The non-linearity of the curves shows how the ranking metric is able to provide a more reliable performance metric for each function as the characteristics of the function can be considered.

$$R_1(t) = \begin{cases} \text{rank}(\hat{\boldsymbol{q}}(t)) & \text{if } t = T \\ 0 & \text{otherwise.} \end{cases} \tag{5.17}$$

This reward is perhaps the most unbiased representation of the agent's goal, but provides a very sparse signal. As an alternative, the dense ranking reward is defined as

$$R_2(t) = \text{rank}(\hat{\boldsymbol{q}}(t)), \tag{5.18}$$

which provides the ranking at every iteration.

Both the sparse and dense ranking rewards provide the performance of the run as a reward signal. As an alternative, a number of rewards that instead provide a positive signal whenever the current global best position is improved are also defined. These rewards will incentivise the agent to always take actions that improve its position, without explicitly rewarding the agent for the final result.

The first reward is similar to the reward function used by Sharma et. al. in the application of RL to differential evolution [46]. The reward returns a positive signal for every improvement in the best position, defined as

$$R_3(t) = \begin{cases} 10 & \text{if } \hat{\boldsymbol{q}}(t) < \hat{\boldsymbol{q}}(t-1) \\ 0 & \text{otherwise.} \end{cases} \tag{5.19}$$

This reward may however be easy to exploit; consider an agent that learns to make slow, insignificant improvements to the fitness value (perhaps by choosing coefficient values that lead to slow particle velocities), thereby receiving a positive reward at each step while barely improving the best position. The reward can rather include information on the significance of the improvement by incorporating the improvement in the objective value,

$$R_4(t) = f(\hat{\boldsymbol{q}}(t-1)) - f(\hat{\boldsymbol{q}}(t)), \tag{5.20}$$

however this reward will instead suffer from high variance between different objective functions. An improvement reward is therefore also defined which is based on the improvement in the ranking as

$$R_5(t) = \text{rank}(\hat{\boldsymbol{q}}(t)) - \text{rank}(\hat{\boldsymbol{q}}(t-1)). \tag{5.21}$$

The next reward function attempts to rather penalize the agent according to how far away the current global best position is from the optimum. In order to make no assumptions on the knowledge of the global optimum, the optimum position is defined as $\boldsymbol{p}_f, \text{rank}_f(\boldsymbol{p}) = 1$. As mentioned before, there may be multiple positions that have a ranking of 1. The best position found throughout all runs used to collect samples for the ranking function is used as the optimum position, since this position will necessarily have a ranking equal to 1, but may not be the true global optimum.

The reward function is then defined as a penalty based on the Euclidean distance of the current best from the position $\boldsymbol{p}_f$,

$$R_6(t) = -||\hat{\boldsymbol{q}}(t) - \boldsymbol{p}_f||_2. \tag{5.22}$$

The final reward function takes a different approach. The agent is penalized at every iteration where the ranking of the best position is below 0.9, and provides a large positive reward as soon as a ranking of 0.9 is achieved. Once achieved, the PSO run is ended. Formally, the reward is defined as

Table 5.3: Summary of CRLPSO reward functions considered throughout this study.

Symbol	Name	Definition
R_1	Sparse Ranking	Equation (5.17)
R_2	Dense Ranking	Equation (5.18)
R_3	Fixed Improvement	Equation (5.19)
R_4	G-Best Improvement	Equation (5.20)
R_5	Ranking Improvement	Equation (5.21)
R_6	Distance from Optima	Equation (5.22)
R_7	Target-Based	Equation (5.23)

$$R_7(t) = \begin{cases} -1 & \text{if } \text{rank}(\hat{\boldsymbol{q}}(t)) < 0.9 \\ T & \text{otherwise.} \end{cases} \tag{5.23}$$

This target-based reward should incentivise the agent to achieve a good ranking in as few iterations as possible.

5.4 Summary

This chapter defined the elements that make up the CRLPSO environment, namely the action space, the state space, and the choice of reward function. The state space features are summarised in Table 5.4, and the reward function definitions are summarised in Table 5.3. The following chapter discusses the setup of the reinforcement learning agent used to optimise the environment defined in this chapter, and performs a number of experiments to train an optimal agent.

Table 5.4: Summary of the 28 CRLPSO state space features.

Number	Description	Definition
1	Time	t/T
2-4	Previous coefficients	$w(t-1),\ c_1(t-1),\ c_2(t-1)$
5	Mean convergence factor	Equation (5.3)
6	Mean diffusion factor	Equation (5.4)
7	Problem dimension	D/D_{max}
8	Stagnation	Equation (5.5)
9	Improvement rate	Equation (5.6)
10	Mean velocity	Equation (5.7)
11	Velocity variance	Equation (5.8)
12	Diversity	Equation (5.9)
13	Distance from swarm centroid	Equation (5.10)
14	Distance from G-Best	Equation (5.11)
15	Distance from P-Best	Equation (5.12)
16	Mean fitness	Equation (5.13)
17	Fitness variance	Equation (5.14)
18-20	TVAC coefficients	$w(t)$, $c_1(t)$, $c_2(t)$ as per (3.1)
21-25	$\boldsymbol{B}$ Eigenvalues	Eigenvalues from (2.9)
26	Autocorrelation	Equation (5.15)
27	Expected movement	Equation (2.18)
28	Search focus	Theorem 2.3.3

6 | Learning a Continuous Particle Swarm Optimisation Policy

Chapter 5 formulated continuous PSO parameter control as the CRLPSO environment that a Reinforcement Learning (RL) agent can interact with. The next important topic is the actual training of a control policy that maximizes the optimisation performance of the underlying PSO algorithm. This chapter details the implementation of the RL process and provides a number of experiments which determine an effective setup for training a CRLPSO policy.

Sections 6.1, 6.2, and 6.3 discuss the details of the learning environment, the choice and implementation of the RL algorithm, and the policy model architecture, respectively. Sections 6.4 through 6.6 then provide experimental results. Section 6.4 first performs hyper-parameter selection and evaluates the efficacy of the different reward functions. Section 6.5 then refines the policy by exploring the different policy model architectures. Section 6.6 finally evaluates the efficacy of learning for different function dimensionalities.

6.1 Reinforcement Learning Setup

For the purposes of this study, the Proximal Policy Optimisation (PPO) [45] algorithm, as was described in Section 4.6.4, is used to train the agent. This decision was made based on early experimentation, where PPO was found to be the only algorithm among those investigated capable of optimising for the PSO control environment. Other algorithms evaluated include Asynchronous Advantage Actor Critic (A3C) [33] and Deep Deterministic Policy Gradient (DDPG) [29].

The implementation of PPO provided by the Stable-Baselines [20] library is used[1]. This implementation differs slightly from the original PPO algorithm as the clipping is also applied to the value function updates, and not only the policy updates as in Equation (4.38). Furthermore, the advantage values collected during the rollout phase are normalised before updating the policy, which is not described in the original paper.

Normalisation is also applied to both the state vector and the reward values based on the recommendations of [1]. It was found that both state and reward normalisation are crucial to successful optimisation of the policy. Normalisation is performed by keeping running averages and variances of the state and reward using Welford's online algorithm [57]. For the state vector, a running average and variance is kept for each individual feature. The values are then normalised according to

$$y'(t) = \frac{y(t) - \mu_y(t)}{\sqrt{\sigma_y^2(t) + \epsilon}}, \tag{6.1}$$

[1]Specifically the PPO2 GPU implementation. See https://stable-baselines.readthedocs.io/en/master/modules/ppo2.html

where y is the state or reward value in question, $\mu_y(t)$ and $\sigma_y^2(t)$ are the running average and variance of y, respectively, and ϵ is a small constant used to avoid division by zero. State values are normalised *before* being input into the policy and value functions. Normalisation of the sparse ranking reward (R_1) only considered the non-zero values when updating the running average, as the sparsity of the reward signal caused the running average to be dominated by zero values. For all other reward functions, all values are considered.

The episode rollouts are essential to collect experience of the current policy performance in order to adjust the policy weights. As the PPO algorithm does not make use of experience replay to keep experience from previous epochs, multiple episodes are executed at each epoch to collect sufficient experience. Each individual episode corresponds to a PSO run of a single benchmark function, using the current control policy. This was found to be crucial; performing only a single episode between successive policy updates proved incapable of optimising any reward function. For the purposes of this study, 40 episodes are executed in parallel at each epoch.

After rollouts, the experience samples (i.e. samples of (S_t, A_t, R_{t+1})) are stored in a buffer and used to update the policy and value function model parameters as described in Section 4.6.4. The experience buffer is then looped over 4 times to perform the model updates using gradient ascent between each successive rollout phase. The Adam optimiser [27] is used to perform gradient ascent. To clarify, the term *epoch* is used in the remainder of this chapter to refer to a single iteration of rollout and the successive policy/value function parameter updates. During experiments, a select number of epochs are executed which then makes up the training duration.

6.2 Learning Environment

In order to train an agent, experience is gained by executing full PSO runs during the rollout phase of each epoch. For this to be effective, a diverse set of benchmark functions should be used to expose the agent to different environments. For the purposes of this study, the CEC-2017 benchmark suite [2] is used. This suite defines 30 n-dimensional benchmark functions that have varying properties such as uni- or multi-modality, symmetry or asymmetry, etc. The functions are listed in Appendix A. All functions are strictly treated as black-box functions, with no information of the function definition being used in any way.

In order to measure the ability of the trained policy to generalise to new functions, only 20 of the 30 benchmark functions are used during training. Whether or not the function is used during training is also indicated in Appendix A. During training, the 20 functions are equally distributed amongst the 40 episodes executed in the rollout of each epoch. This ensures that each epoch gains an equal amount of experience from each function.

At the end of each epoch during training, the mean ranking and episodic returns achieved across the 40 optimisation runs (i.e. episodes) are recorded. Furthermore, the average of the ranking over the last 30 epochs is kept using a sliding window, and the model parameters at the epoch with the highest value of this average are saved. This allows restoration of the best performing policy parameters (including the running mean and variances used for state and reward normalisation) in cases where the ranking deteriorates after a number of epochs. Note that the parameters that achieve the best *ranking* are restored rather than the best returns, as the purpose is ultimately to learn a policy that maximises the ranking metric.

Once training completes and the best performing parameters are restored, the policy is evaluated on the full suite of 30 benchmark functions by performing 30 independent

runs for each function. At evaluation, the running averages and variances used for state and reward normalisation are no longer updated. Also note that the policy is still used stochastically (i.e. the actual action values are sampled from the normal distribution produced by the policy). An alternative method is to assume a variance of 0 and use the mean of the distribution predicted by the policy directly, however this was found to produce worse results. The purpose of model evaluation is to get a final ranking metric for the training experiment, which allows a fair comparison between different trained policies.

As a final note on the learning environment setup, the PSO swarm size is set to 30, and the run length is set to 1000 for $D = 10$, 1500 for $D = 30$, 2500 for $D = 50$, and 5000 for $D = 100$. Note that the number of function evaluations is then equal to the swarm size times the run length (e.g. $30 \times 1000 = 30000$ function evaluations for $D = 10$). Importantly, these are the same values used during collection of results for the ranking metric as discussed in Section 5.3.2. For the purposes of this study, the values have been fixed as this ensures that any observed difference in performance is only due to differences in policy behaviour. However, as mentioned in Section 5.2, all of these values can be changed in practice even after training a policy.

6.3 Policy Model Architecture

As discussed previously in Section 4.5, the purpose of using a parameterized model to represent both the policy and value function is to allow real-valued state and action values. This section first describes how the policies are configured to allow continuous state and action values, and then describes the three different model architecture types that will be explored experimentally in Section 6.5.

The general idea of using neural networks as a form of function approximation for both the policy and value function was discussed in Sections 4.5 and 4.6. The state vector $\mathbf{S}(t) \in \mathbb{R}^{28}$ is provided as input to the neural network after normalising the state features, as discussed in Section 6.2.

As the PPO algorithm makes use of advantage estimation, both the policy and value functions ($\pi(\boldsymbol{S})$ and $v_\pi(\boldsymbol{S})$) are estimated. Instead of using two separate neural networks to approximate each of these functions, the parameters from a number of the first layers can also be shared between both functions to both reduce the number of learnable parameters and allow sharing of learnt capabilities between the two function approximations. The exact number and type of layers shared between both differ based on the model architecture.

The output for the value function approximation is a single real-valued value, and is optimised using a mean-squared error loss function by using the value function estimates of the current rollout as ground-truth targets. For the policy function approximation, the output is a vector of 3 real-valued values. This vector provides the means of 3 normal distributions from which the values of $\boldsymbol{h}_a$ are sampled and thereafter mapped to the coefficient values as per Equation (5.2). The standard deviations of the 3 normal distributions are provided as parameters of the neural network and are also learnable via gradient ascent.

For the design of the neural network, this study will explore different options that differ both in architectural design and in the way that information from previous timesteps can be propagated forward. The remainder of this section discusses the three neural network model architecture types, namely the multilayer perceptron, a recurrent neural network using LSTM modules, and a convolutional neural network.

6.3.1 Multilayer Perceptron

The first architecture is a regular feed-forward neural network, implemented using a multilayer perceptron (MLP). This is the standard architecture that was introduced in Section 4.5, where each layer is a fully-connected layer followed by an activation function. This is perhaps the simplest type of neural network design, and does not make use of any historical information. The only information provided to the network is the current state vector, $\boldsymbol{S}(t)$.

For this study, the MLP will make use of two hidden layers followed by the output layer, and tanh activation functions are applied to the outputs of each hidden layer, based on the recommendations of [1]. No weights are shared between the value and policy functions; i.e. each function has its own neural network with an identical number of layers, and differ only in the design of their respective output layers. The exact number of units per hidden layer will be explored experimentally.

6.3.2 Recurrent Neural Network

The next architecture is a long short-term memory (LSTM) neural network [21], which is a type of recurrent neural network. This network provides a form of feedback by feeding the output of a node back into itself, which allows propagation of information from the previous timestep to the next. In this sense, the LSTM neural network considers the series of states $\boldsymbol{S}(1), \boldsymbol{S}(2), \ldots, \boldsymbol{S}(t)$ to estimate the value and policy at timestep t. This should allow more complex behaviours that could not be learnt if considering only the state at the current timestep.

The main component of the LSTM network is the LSTM *memory cell*, which can be considered as a single recurrent node that keeps a memory. Internally, each cell keeps a short-term and long-term memory, denoted as $\boldsymbol{h}(t)$ and $\boldsymbol{c}(t)$ respectively. Each cell also uses a forget gate, an input gate, and an output gate which learn to filter input data and previous states to keep only relevant information. After filtering, both $\boldsymbol{h}(t+1)$ and $\boldsymbol{c}(t+1)$ are then recurrently passed to the cell in the next timestep.

The LSTM architecture used first has two hidden layers of 64 units each followed by tanh activation functions. The output of these are then passed on to a layer of LSTM cells. The exact number of cells will be explored experimentally. The weights of these layers are shared between the value and policy functions. The output vector produced by the layer of LSTM cells is then passed to the respective fully-connected value and policy output layers. To clarify, the input to this network is still only a single state vector $\boldsymbol{S}_t$, however the recurrent nature of the layer of LSTM cells propagates information to the LSTM cells in the next timestep.

6.3.3 Convolutional Neural Network

For the last architecture, a different approach is taken to include historical information. At each timestep, the state vector is collected into a rolling buffer of the last 100 states. This provides a 2-dimensional matrix of size (28×100) that can be used as input to a neural network. The buffer is initialised to all zeros.

To process this matrix, a 1-dimensional convolutional neural network (CNN) [16, Ch. 9] is used. This type of network is a feedforward neural network that uses convolutions instead of simple matrix multiplications as the MLP does. The weights of the convolution filters are learnable such that the neural network is able to learn to extract features from the input sequence. CNNs are most popularly applied to the computer vision domain where

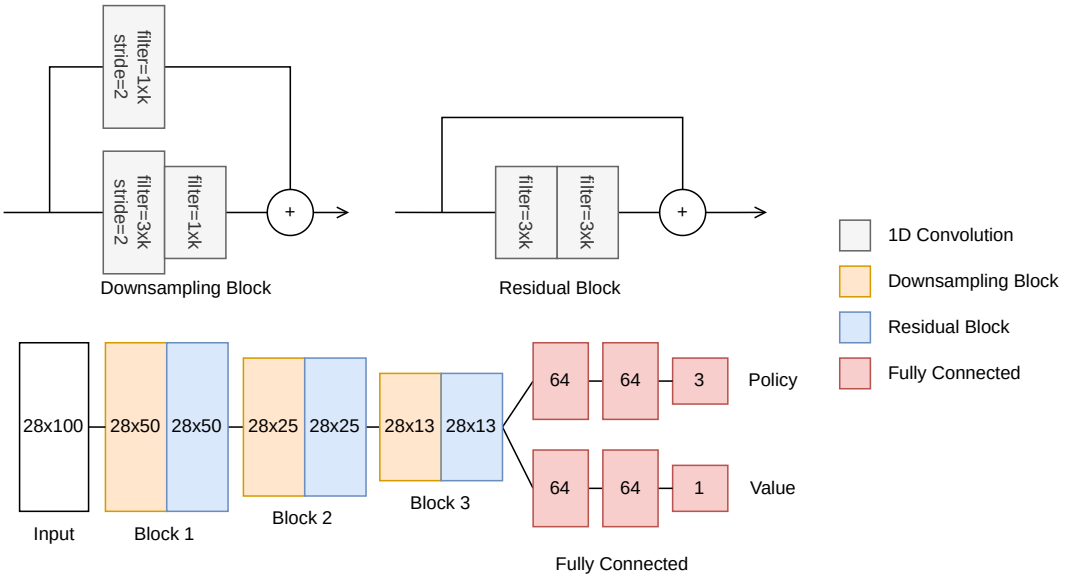


Figure 6.1: An illustration of the ResNet-based CNN architecture. The dimensions on the layers indicate the size of the layer's output. The convolution layers show the filter dimensions as $m \times k$ where m is the filter size and k is the number of filters, which is left to be determined.

2-dimensional filters are used to extract features from images. What makes a convolution perform well in this domain is the fact that the filter considers only the spatial neighbours in a small area of the input sequence, which enables the network to learn to extract spatial features. In order to apply a CNN to the historical state information, a 1-dimensional filter is used and applied to the sequence of states.

Based on the 2-dimensional residual blocks used by the ResNet architecture [19], the network organises a number of convolutional layers into blocks and adds residual (or "skip") connections between subsequent blocks. This allows better propagation of the gradient during back-propagation, aiding in training the large number of parameters of the model. Also note that typically multiple filters are used at each layer to increase the number of learnable parameters and allow extraction of multiple features at each layer. The CNN layers are shared between the policy and value functions. The output of the CNN is then passed through two fully connected layers with 64 units each followed by tanh activation functions, and a final fully connected output layer. The fully connected layers do not share weights between the policy and value functions. The full architecture of the CNN is illustrated in Figure 6.1, which also indicates the sizes of the filters. The number of filters to use for each convolution layer will be investigated experimentally, and is thus indicated as the parameter k in Figure 6.1.

6.4 Reward Function and Hyper-Parameter Selection

The purpose of this first experiment is to both determine the reward functions that work well and to perform hyper-parameter selection for each reward function. This experiment only considers the PPO algorithm hyper-parameters, namely the discount factor, clipping range, learning rate, and bias-variance trade-off. All other hyper-parameters, including those of the PSO environment, remain fixed during experimentation and are as provided

in Sections 6.1 and 6.2.

6.4.1 Experimental Setup

This experiment focusses on evaluating the efficacy of the different reward functions. For each reward function, a number of policies are trained using different combinations of the four hyper-parameters shown in Table 6.1, as these have been identified through early testing as having a high impact on the training outcome. For each hyper-parameter, two values are tested. The combinations of these values are then tested, providing a total of $2^4 = 16$ combinations. Additionally, 3 repeat experiments are run for each combination to compensate for the variance caused by the stochastic PSO environment and PPO algorithm, for a total of $16 \times 3 = 48$ training runs per reward function.

The policy architecture used for this experiment is the multilayer perceptron (MLP) described in Section 6.3.1, with 64 units per hidden layer. Benchmark functions used during training are also restricted to only dimensionality of $D = 10$. Other architectures and the impact of function dimensionality will be considered in Sections 6.5 and 6.6, respectively.

The outcome of each training run is an optimised policy. Each optimised policy is then evaluated on all 30 of the CEC-2017 benchmark functions with $D = 10$ using the ranking metric defined in Section 5.3.2. Each benchmark function is repeated for 30 independent runs, and the mean ranking is recorded. After collecting the mean rankings for each of the 48 policies, the rankings of repeat experiments are averaged together to get a final mean ranking for each parameter combination. The following section discusses the outcomes of these experiments.

6.4.2 Results

The best ranking achieved by each reward function along with the best hyper-parameter values are shown in Table 6.2. From the results, it is evident that the target reward (R_7) was able to achieve the best ranking. However, it is also clear that the rankings achieved by the different reward functions are not very different. The worst results achieved by using R_2 and R_4 have a mean ranking of 0.8659, which is only marginally lower than the best ranking of 0.8693 achieved by R_7. To understand why, consider the training graphs provided in Figure 6.2. In nearly all cases, the ranking does not improve at all during training, so the set of parameters used for evaluation may have only undergone a small number of training epochs and is therefore not much different than a randomly initialised, untrained policy. This causes most experiments to result in a similar ranking.

Given the above, one may wonder whether the training then has any effect at all, given that the ranking does not improve during training. A statistical investigation in the following section provides evidence that the policy does in fact learn during training.

Table 6.1: Range of PPO hyper-parameters considered during hyper-parameter optimisation.

Symbol	Description	Values	Reference
γ	Discount factor	0.99, 0.999	Equation (4.4)
λ	GAE bias-variance trade-off	0.95, 1.0	Equation (4.35)
α	Learning rate	0.000025, 0.001	Equation (4.25)
ϵ	Clipping range	0.05, 0.2	Equation (4.38)

Table 6.2: The most performant PPO hyper-parameters per reward function and the associated ranking achieved by the combination.

Reward Function	γ	λ	α	ϵ	Mean Ranking ($\pm\sigma$)
R_1 (Sparse Ranking)	0.999	1.00	0.001	0.2	0.86630 ± 0.08831
R_2 (Dense Ranking)	0.99	1.00	0.000025	0.05	0.86589 ± 0.08924
R_3 (Fixed Improvement)	0.999	0.95	0.000025	0.05	0.86726 ± 0.09193
R_4 (G-Best Improvement)	0.999	0.95	0.001	0.05	0.86589 ± 0.08953
R_5 (Ranking Improvement)	0.99	0.95	0.000025	0.05	0.86633 ± 0.08963
R_6 (Distance from Optima)	0.99	0.95	0.001	0.2	0.86629 ± 0.09064
R_7 (Target-Based)	0.999	1.00	0.001	0.2	0.86931 ± 0.08781

Thereafter, Section 6.4.2 provides a more detailed discussion on the performance of each individual reward function.

Training Efficacy

In order to determine whether or not the training of an agent has any significant effect on the policy, two statistical tests are conducted. The purpose is to compare the performance of a model trained with the target-based reward (R_7) to the performance of a random model (i.e. a model that has only been initialised and undergone no training). A sample of 30 models are trained using R_7 for 250 epochs as before, and 30 random models are initialised. Each of these 60 models are then evaluated, with 30 independent runs on each of the 30 benchmark functions.

For the first test, the optimisation results of all 30 random or trained models are combined for each function (providing a total of 900 optimisation results per function), and the distribution of results are compared between the random and target models. A test for distribution normality using D'Agostino and Pearson's test [10] confirms that none of the results follow a normal distribution, so the test is conducted using the Mann-Whitney U test [31] under the alternative hypothesis H_a that the target models' distribution is less than that of the random models, and the null hypothesis H_0 that the target models' distribution is equal to or worse than that of the random models. The null-hypothesis is rejected for $p < \alpha = 0.05$. The U statistics and p-values for each function are listed in Table 6.3.

The results show that the distribution of results by the target reward trained models are significantly better (i.e. less) than the distribution of results from the random models on 7 of the 30 functions, indicated in boldface in Table 6.3. This indicates that there is some improvement as a result of training. There is however no significant difference on the other 23 functions, and in fact performs significantly worse on the two functions F_6 and F_9. Nonetheless, the overall effect of training does show an improvement.

For the next test, the mean ranking achieved is calculated for each of the 30 normal and trained models, providing two distributions of 30 samples each. The test for distribution normality indicates that both the target reward ranking and random ranking distributions are normally distributed. The Student's t-test is therefore used under the alternative hypothesis H_a that the target reward models' mean ranking distribution is greater than that of the random models (which in the case of the ranking metric would indicate significantly better performance with the target reward models), and the null hypothesis H_0 that the target reward models' mean ranking distribution is equal to or less than that of the random models (which would indicate equal or significantly worse performance). The null

Table 6.3: The Mann-Whitney U statistic for all CEC benchmark functions for models trained with the target reward versus randomly initialised models. Functions where the target reward performed significantly better are highlighted in boldface.

Function	U_{target}	U_{random}	p-Value
F_1	443092	366908	**0.0003**
F_2	600488	209512	**0.0000**
F_3	542730	267270	**0.0000**
F_4	409535	400465	0.3404
F_5	399884	410116	0.6787
F_6	381719	428281	0.9826
F_7	416758	393242	0.1431
F_8	391739	418261	0.8855
F_9	385294	424706	0.9631
F_{10}	394823	415177	0.8220
F_{11}	399597	410403	0.6880
F_{12}	439568	370432	**0.0009**
F_{13}	408092	401908	0.3896
F_{14}	416354	393646	0.1516
F_{15}	402821	407179	0.5783
F_{16}	393004	416996	0.8617
F_{17}	393566	416434	0.8501
F_{18}	398153	411847	0.7327
F_{19}	407930	402070	0.3952
F_{20}	392405	417595	0.8733
F_{21}	388034	421966	0.9381
F_{22}	395592	414408	0.8033
F_{23}	415456	394544	0.1715
F_{24}	427669	382331	**0.0199**
F_{25}	412166	397834	0.2579
F_{26}	413694.5	396305.5	0.2152
F_{27}	391106	418894	0.8962
F_{28}	405794	404206	0.4713
F_{29}	426118	383882	**0.0277**
F_{30}	433460	376540	**0.0049**

hypothesis is rejected if $p < \alpha = 0.05$.

The test provides a t-statistic of 1.7265 and a p-value of 0.04478. The distribution of mean rankings achieved by models trained using the target reward is therefore significantly greater than those achieved by random models, indicating again that training does have a positive effect on the policy performance.

In summary of the above, both tests show that there is in fact an improvement in results obtained by a trained policy, however minor the difference is. Also note that the test is not repeated for the other, less performant reward functions due to the computational cost of training 30 models.

Reward Function Performance

The training curves of top performing hyper-parameter combinations are shown in Figure 6.2. Each graph shows the actual returns and the ranking achieved at each epoch. Note that these values are the results aggregated across results from 40 independent episodes on 20 different benchmark functions as described in Section 6.2, and are averaged across the 3 repeat experiments.

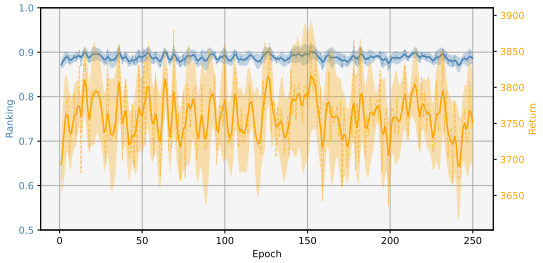
Sparse Ranking (R_1): Experiments show that the sparse ranking reward is difficult to optimise for. The best performing optimisation runs which are shown in Figure 6.2a were unable to achieve any significant improvement in the return. Most hyper-parameter combinations either have a similar training curve to Figure 6.2a, or fail completely and result in decreasing returns. Also note that there is a direct correlation between the ranking and the return, as is expected since the reward provided at the end of each run is equal to the ranking.

Dense Ranking (R_2): The dense ranking reward is possible to optimise, however the change from a sparse to a dense ranking introduces a bias that can be exploited by the agent (e.g. an agent can learn to be overly exploitative to get a "good enough" value early on to improve the total return, however a worse value at the start may lead to better results towards the end of a run but will decrease the total return). As is evident in the trajectory of the returns versus that of the ranking in Figure 6.2b, the ranking decreases as the returns improve significantly during the first 100 epochs. Furthermore, the reward is apparently difficult to optimise beyond a certain point, with the returns starting to decrease after epoch 100, at which point the ranking also continues to decrease.

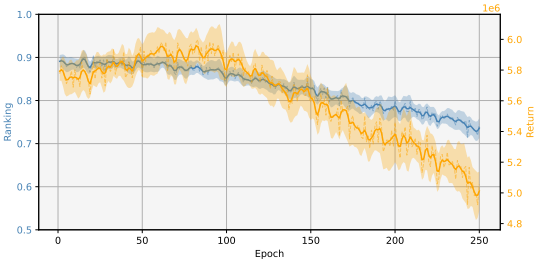
Fixed Improvement (R_3): Similar to the dense ranking, the fixed improvement reward proved to be even easier to exploit by the agent. Figure 6.2c shows an apparent inverse relationship between the returns and the ranking achieved by the agent. The returns are steadily increased as the policy is optimised to improve the reward, however the ranking steadily decreases. It is likely that the policy is able to learn to make frequent but small improvements to the global best position, which would increase the returns but lead to worse outcomes for the PSO run.

Global Best Improvement (R_4): The global best improvement reward has an interesting trajectory, as seen in Figure 6.2d. There are no noticeable changes in the returns, however the ranking steadily declines over the training period. This is likely due to the high variance in the reward values caused by different benchmark function value ranges, which makes the reward function difficult to optimise and also leads to an undesirable policy.

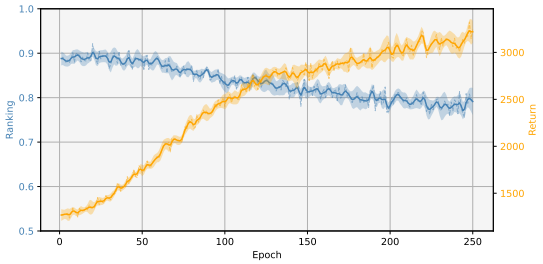
Ranking Improvement (R_5): The ranking improvement reward proved to be easy to optimise for, as is evident in Figure 6.2e. The returns are optimised very well throughout the training run and do not deteriorate after some time as in the case of the target-based



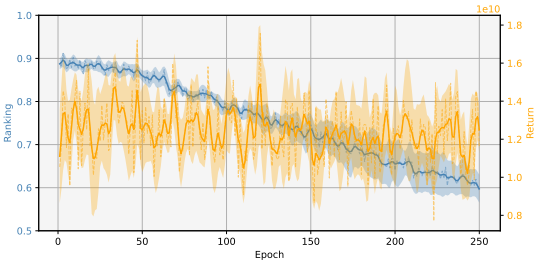
(a) Sparse Ranking (R_1)



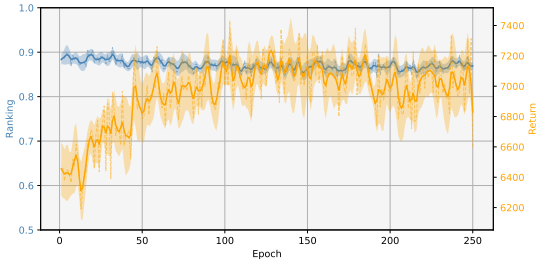
(b) Dense Ranking (R_2)



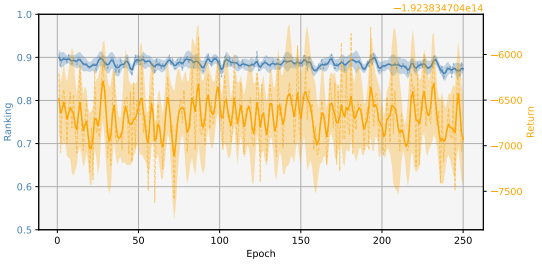
(c) Fixed Improvement (R_3)



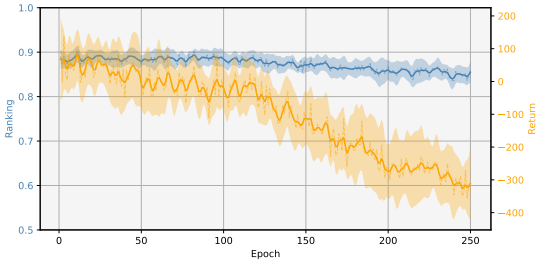
(d) G-Best Improvement (R_4)



(e) Ranking Improvement (R_5)



(f) Distance from Optima (R_6)



(g) Target-Based (R_7)

Figure 6.2: Mean rankings and episodic returns achieved by agents over the course of 250 epochs of training using different reward functions. Results are averaged across 3 independent runs. To improve the clarity of the graph, results are smoothed using a Gaussian filter. The dashed lines show the unfiltered results and shaded areas represent the standard deviation.

and dense ranking rewards. The ranking also does decrease slightly as the returns are optimised, which suggests a small bias. The bias is however not as strong as in the dense ranking (R_2) and fixed improvement (R_3) rewards.

Distance from Optima (R_6): Similar to the sparse ranking reward, the distance from optima reward is difficult to optimise but shows a strong correlation with the ranking, as is evident in Figure 6.2f.

Target-Based (R_7): The target reward also proved difficult to optimise, with the returns decreasing over the duration of training, as in Figure 6.2g. The target reward also seems to have a positive correlation with the ranking, with the ranking decreasing along with the returns over the training duration.

In order to further formalise the notion of reward function bias, the Pearson correlation between the ranking and the total returns achieved during training is calculated for each reward function. A positive correlation indicates that the reward function is a good proxy for optimisation performance, and a negative correlation indicates that the reward function, when optimised, leads to a degradation of optimisation performance. A correlation close to zero instead shows that there is no clear relationship between the reward function and the policy's optimisation ability. Note, however, that the correlation coefficient alone does not indicate whether a reward function has an exploitable bias or not, as not all policies necessarily learn to exploit this bias. It is also possible that the correlation changes over the duration of training as the policy starts to exploit the bias, such as the case of dense ranking in Figure 6.2b.

The calculated correlations are shown in Table 6.4. Note that these correlations were calculated across all runs performed during this experiment, and not only the runs illustrated in Figure 6.2. The most noticeable result is that the only negative correlation is with the fixed improvement reward (R_3), which confirms that optimisation of R_3 leads to a degradation in optimisation performance. The global best improvement reward has a weak positive correlation, which suggests that it is not a good indicator of optimisation performance.

The remaining reward functions all have strong positive correlations to the ranking metric. The correlation of the sparse ranking function is also 1.00 as expected, as it is equivalent to the ranking metric. Despite these positive correlations, the dense ranking and ranking improvement reward functions (R_2 and R_5) do have an exploitable bias as discussed previously.

In summary of the above discussion, the dense ranking, fixed improvement, and global best improvement reward functions proved to be unsuitable for training a PSO control policy. The dense ranking and global best improvement reward functions (R_2 and R_4) due to their biases and difficulty of optimisation, and the fixed improvement reward function

Table 6.4: The Pearson correlation between the ranking and returns for the different reward functions during training.

Reward Function	Correlation ($\pm\sigma$)
R_1 (Sparse Ranking)	1.000 ± 0.000
R_2 (Dense Ranking)	0.735 ± 0.239
R_3 (Fixed Improvement)	-0.817 ± 0.039
R_4 (G-Best Improvement)	0.171 ± 0.175
R_5 (Ranking Improvement)	0.585 ± 0.234
R_6 (Distance from Optima)	0.589 ± 0.166
R_7 (Target-Based)	0.737 ± 0.112

(R_3) due to its extreme exploitable bias. For this reason, these three reward functions will not be considered during further experimentation. The ranking improvement reward also contains some exploitable bias, but is not as severe as in the case of R_3 and proved much easier to optimise than R_2 and R_4. The sparse ranking, distance from optima, and target-based reward functions all show little to no bias, but were difficult to optimise given the setup used in this experiment. Section 6.5 will further attempt to optimise these functions by exploring the selection of policy architecture.

6.5 Architecture Selection

This next experiment performs selection of the policy model architecture, since this may have a significant impact on the training and resulting optimisation performance of the agent. Ideally, the model architecture and hyper-parameter values would be selected together since the best hyper-parameters may depend on the architecture; however, the search space would be too large, given the computational cost of training a single agent. Therefore the best hyper-parameter values found for each reward function in the previous experiment (Section 6.4) are used, and only the model architecture is considered in this experiment. Furthermore, for this experiment, only the sparse ranking, ranking improvement, distance from optima, and target-based reward functions are considered based on the conclusions reached in the previous experiment.

6.5.1 Experimental Setup

This experiment evaluates the differences in performance, if any, between the MLP, LSTM, and CNN policy architectures, as were described in Section 6.3. The size of the architecture may impact the performance, and must also be considered. Therefore, a set of 6 experiments are executed for each reward function, wherein two configurations of each of the 3 policies – one large and one small – will be tested. As in the previous experiment, each run is repeated 3 times to compensate for variance caused by the stochastic PSO and PPO algorithms. There is thus a total of $6 \times 3 = 18$ training runs per reward function.

For this experiment, the 6 architectures tested are MLP networks with either 64 units or 128 units in the two hidden layers, recurrent LSTM networks with either 64 or 256 LSTM cells, and CNNs with either 8 or 32 filters per 1D convolutional layer. These are of course not an exhaustive list of the possible choices, but serve well enough to identify whether or not the choice of architecture has any significant impact, rather than attempting to determine the most optimal architecture.

As already mentioned, the hyper-parameter values identified in the previous experiment will be used as listed in Table 6.2. Benchmark functions used during training and evaluation are again restricted to a dimensionality of $D = 10$. The trained policies are evaluated on all 30 of the CEC-2017 benchmark functions with 30 independent runs and reported using the ranking metric. A discussion of the outcomes of these experiments follow.

6.5.2 Results

The ranking metric of each model architecture per reward function is shown in Table 6.5. There are no clear patterns across the reward functions, however the MLP architecture performed best on three of the four reward functions (R_1, R_5, and R_7), with only R_6 performing better with the use of an LSTM network. This suggests that the inclusion of

Table 6.5: Rankings achieved by different policy model architectures and model sizes. The ranking of the best performing architecture is highlighted in boldface for each reward function.

Reward Function	Architecture	Size	Ranking
R_1 (Sparse Ranking)	MLP	64 Units	0.86630 ± 0.08831
	MLP	128 Units	$\mathbf{0.86772 \pm 0.08954}$
	MLP	64 Cells	0.85957 ± 0.09194
	LSTM	256 Cells	0.86769 ± 0.08984
	CNN	8 Filters	0.86017 ± 0.09299
	CNN	32 Filters	0.86133 ± 0.09326
R_5 (Ranking Improvement)	MLP	64 Units	$\mathbf{0.86633 \pm 0.08963}$
	MLP	128 Units	0.85556 ± 0.09624
	LSTM	64 Cells	0.86073 ± 0.09790
	LSTM	256 Cells	0.85568 ± 0.09715
	CNN	8 Filters	0.86409 ± 0.09458
	CNN	32 Filters	0.86247 ± 0.09483
R_6 (Distance from Optima)	MLP	64 Units	0.86629 ± 0.09064
	MLP	128 Units	0.86335 ± 0.09317
	LSTM	64 Cells	$\mathbf{0.86870 \pm 0.09170}$
	LSTM	256 Cells	0.86700 ± 0.09187
	CNN	8 Filters	0.86230 ± 0.09170
	CNN	32 Filters	0.86157 ± 0.09516
R_7 (Target)	MLP	64 Units	$\mathbf{0.86931 \pm 0.08781}$
	MLP	128 Units	0.86871 ± 0.09197
	LSTM	64 Cells	0.84120 ± 0.09604
	LSTM	256 Cells	0.85484 ± 0.09443
	CNN	8 Filters	0.86434 ± 0.09417
	CNN	32 Filters	0.86461 ± 0.09578

historical state information does not provide any additional benefit for the agent. The CNN ResNet architecture was not the top-performing architecture in any case, but performed better than the LSTM network in the cases of R_5 and R_7.

Interestingly, the model size has a consistent impact on performance in most reward functions regardless of the type of architecture. The rankings of R_1 benefit from a larger model size, while R_5 and R_6 perform better with smaller models. In the case of R_7 this distinction isn't as clear, with performance being better with a smaller MLP, but better with larger CNN and LSTM networks. The performance of the MLP network for R_7 is however significantly better than either of the larger CNN or LSTM architectures, and is also the best performing model overall.

To have a better look at the impact of the architecture on training efficacy, consider the learning curves shown in Figure 6.3. Note that these show the curves achieved by the better of the two model sizes for each architecture. In the case of the sparse ranking reward (R_1), the difference between the architectures is very clear. The worst-performing architecture, the CNN, is difficult to train and diverges after the first 100 epochs. Both the MLP and LSTM networks maintain a consistent ranking during training and show little to no improvement (as was observed in the previous experiment), however the MLP network maintains a slightly better ranking throughout.

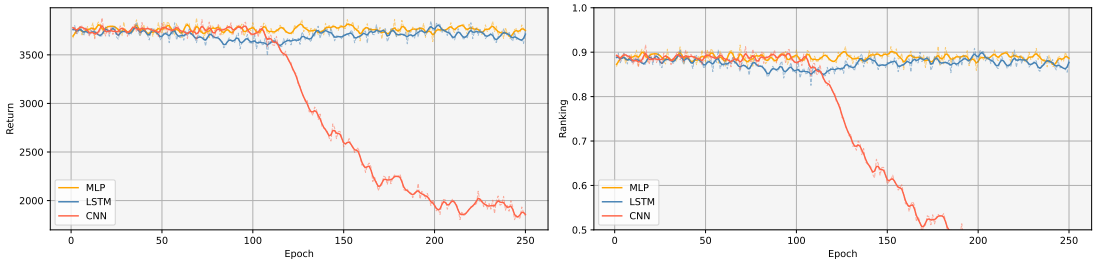
In the case of the ranking improvement reward (R_5), all architectures are able to successfully optimise the reward function. The MLP is however able to achieve higher returns and a better ranking compared to the LSTM and CNN architectures. The differences in the case of R_6 (the distance from optima reward) are not very clear. All three architectures show no improvement in either the reward or ranking, which was also observed in the previous experiment. Finally, in the case of the target-based reward (R_7), the MLP architecture clearly outperforms both the LSTM and CNN networks toward the end of training, but is still not able to provide a visible improvement in the ranking or returns.

This experiment concludes with the finding that an agent trained with the target reward (R_7) with a smaller MLP model architecture leads to the top-performing policy. Therefore, only this reward function and model architecture will be considered for the remainder of this study. It is however worth noting that R_6 is the only reward function among the four considered here that does not require the rankings to be calculated, but does require the positions of optima to be known. If the calculation of rankings as described in Section refsec:rlpso:ranking would be too expensive in some cases, R_6 could be a viable alternative. The next experiment will consider the last component relevant to the training of the agent: the inclusion of different benchmark function dimensions during training.

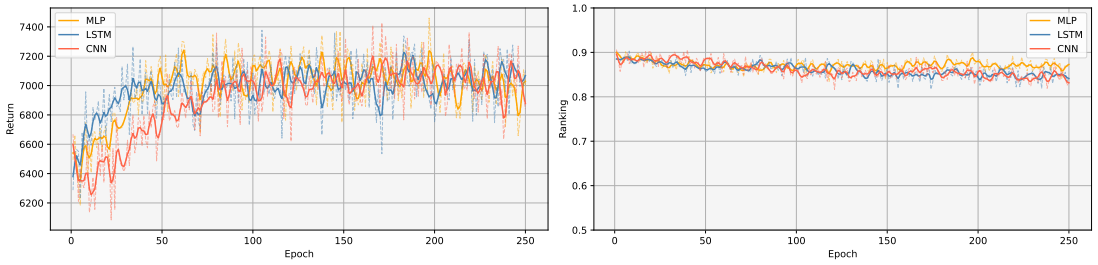
6.6 Impact of Dimensionality

Up until this point, all experiments exclusively considered benchmark functions with 10-dimensional parameters, both during training and evaluation. In practice however, problems of different dimensionalities are encountered and an optimiser should be able to work well in different cases. An agent that is trained only on functions of a specific dimensionality may not perform as well on other dimensions. In practice, an agent that is able to generalise to problems of different dimensions would be ideal as it would not require training a new agent for each problem dimension.

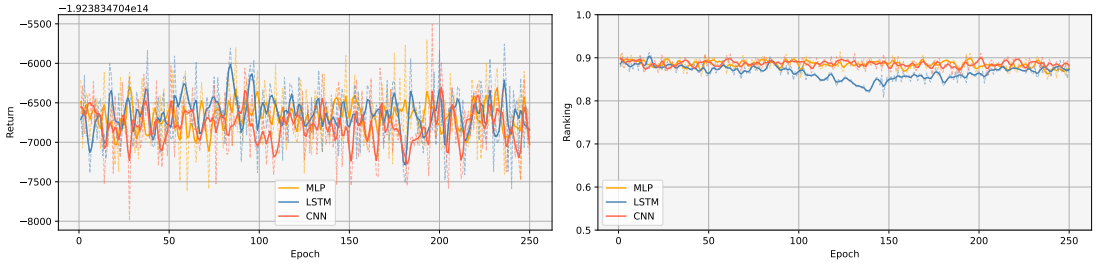
The purpose of this experiment is two-fold. Firstly, the experiment will determine how well agents trained on problems of a specific dimension generalise to other dimensions. Secondly, the experiment will assess the impact of introducing multiple dimensions during training on the agent's ability to optimise functions with different dimensionalities. Recall



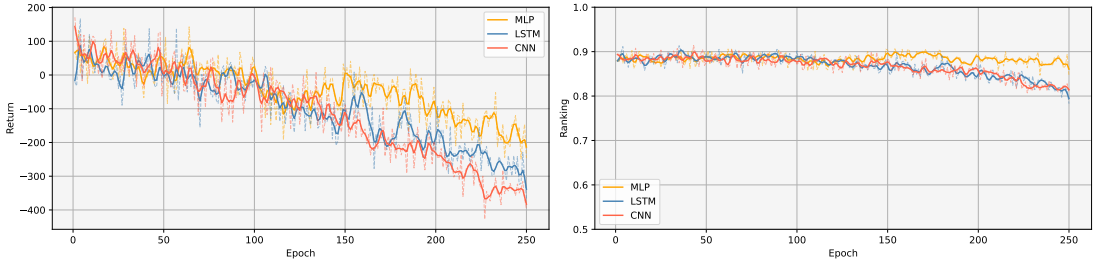
(a) Sparse Ranking (R_1)



(b) Ranking Improvement (R_5)



(c) Distance from Optima (R_6)



(d) Target-Based (R_7)

Figure 6.3: Learning curves of agents over the course of 250 epochs of training with different policy model architectures and reward functions. The episodic returns and mean rankings are shown on the left and right graphs, respectively. Results are averaged across 3 independent runs. To improve the clarity of the graph, results are smoothed using a Gaussian filter. The dashed lines show the unfiltered results.

that the state vector includes a normalised form of the current dimensionality (feature number 7 in Table 5.4), so an agent should in theory be able to learn different behaviour for different dimensionalities if exposed to different dimensionalities during training.

6.6.1 Experimental Setup

In order to evaluate the performance of models trained at specific dimensions, a set of agents will be trained on problems from the CEC-2017 benchmark suite with dimensions 10, 30, 50, and 100, respectively. As before, only the 20 functions listed as the training set in Appendix A will be used during training. Each trained policy will then be evaluated on all 30 functions and all 4 dimensionalities, for 30 independent runs, and the results are recorded using the ranking metric. These results can then be used to determine how well a policy trained at a single dimension is able to generalise to higher or lower problem dimensions. Next, a policy is trained using problems of all 4 dimensions during training (chosen randomly at the start of each episode), and then evaluated on all dimensions. This policy can be compared to the results of the other policies to determine whether the increased problem diversity during training improves the policy's optimisation ability on different dimensions.

Since impact of the problem dimension is the only focus of this experiment, the hyper-parameters, architecture, and reward function that were found to be the best performing in the previous experiment will be used. Specifically, the target-based reward function (R_7) with an MLP policy architecture of 64 nodes per hidden layer will be used, along with the hyper-parameters for R_7 listed in Table 6.2.

The number of iterations (i.e. the episode length) used for each dimension will differ to allow more iterations when optimising higher dimension problems. For dimensions 10, 30, 50, and 100, runs with 1000, 1500, 2500, and 5000 iterations are used, respectively, both during training and evaluation. Note that this is the same number of iterations used to collect the rankings as described in Section 5.3.2. The increase in episode length also increases the number of samples collected at each epoch, as well as increasing the amount of computation required per epoch. For this reason, the policies trained with 100-dimensional problems are trained for 100 epochs, while all other policies are trained for 250 epochs as before. A discussion on the outcomes of these experiments follows.

6.6.2 Results

The resulting mean rankings for each trained policy (averaged across 3 independent experiments) is provided in Table 6.6. When considering the policies trained using only a

D	$D = 10$ Policy	$D = 30$ Policy	$D = 50$ Policy	$D = 100$ Policy	General Policy
10	0.86931 ± 0.11	0.86500 ± 0.12	$\mathbf{0.87038 \pm 0.12}$	0.86351 ± 0.12	0.86461 ± 0.12
30	0.89016 ± 0.10	0.90801 ± 0.08	0.91291 ± 0.08	$\mathbf{0.91419 \pm 0.07}$	0.91189 ± 0.08
50	0.91292 ± 0.08	0.92848 ± 0.06	0.93322 ± 0.06	$\mathbf{0.93675 \pm 0.05}$	0.93233 ± 0.06
100	0.93491 ± 0.08	0.95133 ± 0.07	0.95165 ± 0.07	$\mathbf{0.96336 \pm 0.04}$	0.95806 ± 0.05
Overall	0.90183 ± 0.10	0.91320 ± 0.08	0.91704 ± 0.08	$\mathbf{0.91945 \pm 0.07}$	0.91672 ± 0.08

Table 6.6: Results of policies trained at different function dimensionalities. Each column represents policies trained using specific dimensions, with the last column showing the "general" policy trained using all dimensions. Each row represents the evaluation results for each dimension, and the last row shows the overall average ranking achieved across all dimensions.

single dimension, a clear pattern emerges in the overall ranking: training at higher dimensions improves the overall optimisation performance. This alone provides evidence that training at higher dimensions has the ability to generalise well to lower dimensions, but lower dimension training does not generalise as well to higher dimensions. This is likely due to the increased difficulty of high-dimension problems that allow the policy to learn more useful behaviours. It may however simply be due to the increased number of samples collected at each epoch due to a larger number of iterations. However, if that were the case, policies should also see increased performance with more training epochs, which is not the case. Further experimentation would be required to determine the cause, which is outside the scope of this work.

There is no clear evidence that a policy trained at a specific dimension actually performs better at that dimension, since in all cases the policy is outperformed at its trained dimension by policies trained at higher dimensions. For example, policies trained at $D = 10$ perform worse at $D = 10$ problems than the policies trained at $D = 50$. Likewise, the policies trained at $D = 30$ are outperformed on $D = 30$ problems by policies trained at both $D = 50$ and $D = 100$. This suggests that in order to get good performance on a specific problem, it is better to train at higher dimensions rather than at that specific dimension. As a result, the policy trained at $D = 100$ performs the best on problems of all dimensions except for $D = 10$.

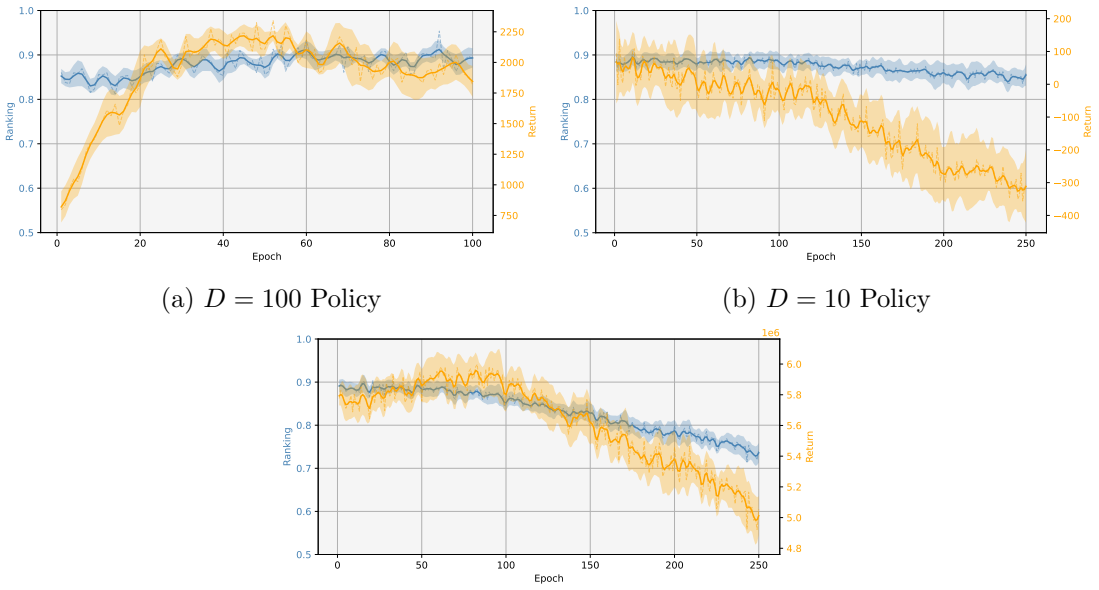
Now consider the general policy trained on multiple dimensions. Its overall ranking sits in the center of all other policies, being better than policies trained at $D = 10$ and $D = 30$, but worse than those trained at $D = 50$ and $D = 100$. This shows that it is better to train exclusively on higher dimension problems rather than a diverse set of dimensionalities. This also suggests that the agent is not able to learn any dimension-specific behaviour that improves its ability to optimise different dimensionalities.

For interest's sake, the training graphs for the policies trained at $D = 10$, $D = 100$, and the general policies are shown in Figure 6.4. The differences observed between the $D = 10$ and $D = 100$ policies are clear. The learning curve for $D = 100$ shows a significant increase in both the returns and the ranking, compared to the $D = 10$ curve which shows little to no increase in the ranking and no increase in the returns. This may suggest that the 10-dimensional problems are too simple for the agent to learn any significant behaviours.

In summary, this experiment showed that policies do not necessarily perform better on the dimensions that they are trained on, but instead gain increased performance from training on higher dimension problems. These policies are then able to generalise well to problems with lower dimensions. The experiment also showed that the inclusion of multiple dimensions during training does not increase performance compared to training only at the highest dimension problems available.

6.7 Summary

Through a series of experiments, this chapter investigated the efficacy of applying RL to learning a continuous coefficient policy for PSO as defined in Chapter 5. The first experiment (Section 6.4) highlighted the difficulty of choosing a good, unbiased reward function, and determined that the dense ranking, fixed improvement, and global best improvement reward functions were not effective reward functions, while the sparse ranking, ranking improvement, distance from optima, and target-based reward functions may be used to effectively train a policy. The experiment also determined a good set of hyper-parameters for the PPO algorithm for each reward function. Although the ranking metric showed little to no improvement in even the best cases, a statistical test provided evidence that



(a) $D = 100$ Policy

(b) $D = 10$ Policy

(c) General Policy

Figure 6.4: Mean rankings and episodic returns achieved by agents over the course of training using different problem dimensions. Results are averaged across 3 independent runs. To improve the clarity of the graph, results are smoothed using a Gaussian filter. The dashed lines show the unfiltered results and shaded areas represent the standard deviation.

the training does in fact provide a significant improvement, however small.

The second experiment (Section 6.5) explored the use of the different policy model architectures introduced in Section 6.3. The policies mainly differ in the way that historical state information is represented. The experiment showed that the best architecture is a regular MLP network for all reward functions except the distance from optima reward (R_6), for which an LSTM network performed better. The overall best performing policy is a small MLP trained using the target-based reward (R_7).

The third and final experiment (Section 6.6) evaluated what the impact is of changing the problem dimensions used during training, and determined whether trained policies are able to generalise to other dimensions. The experiment showed that trained policies generalise very well to lower dimensions, but not to higher dimensions. It also showed that training at a higher dimension improves performance on lower dimensions compared to policies trained at those same lower dimensions. Lastly, the experiment showed that the introduction of multiple dimensions during training does not improve the performance of the trained policy, and it is best to train the agent with higher dimensional problems.

In conclusion, this chapter determined that an effective setup to train a policy is to use the target-based reward, represent the policy using a small MLP network, and to use higher dimensional problems during training. The following chapter will further evaluate the optimisation performance of a policy trained in this way by comparing it against other self-adaptive PSO techniques, both on seen and unseen benchmark functions.

7 | Experimental Analyses

The previous chapter investigated the efficacy of training a policy for continuous control of PSO coefficients using RL. Given a successfully trained policy, this chapter will further investigate the properties of this agent in terms of its performance on various benchmark problems and provide an analysis of its learned behaviour.

In Section 7.1, the trained policy's optimisation performance is compared to other traditional and self-adaptive PSO algorithms on the same benchmark suite used during training and experimentation. In Section 7.2, the optimisation performance is again compared to the same set of techniques, but an unseen set of benchmark functions from a different benchmark suite is used to investigate the generalisation of these techniques to other problems. Finally, Section 7.3 provides some insight into the behaviours learnt by the agent, as well as conducting a convergence analysis to better understand the optimisation ability of the agent.

7.1 Optimisation Performance

In the previous chapter, the CEC 2017 benchmark suite [2] was used to both train and evaluate the performance of the various trained CRLPSO policies. In this section, the performance of the best policy found in Chapter 6 will be compared to other traditional and self-adaptive PSO algorithms as defined in Chapters 2 and 3 in order to evaluate the optimisation ability of the policy.

7.1.1 Experimental Setup

In order to measure the performance of the chosen PSO algorithms on the CEC benchmark suite, each algorithm is used in turn to optimise each of the 30 benchmark functions, as listed in Appendix A. Furthermore, the suite is tested at dimensionalities 10, 30, 50, and 100. Each optimiser is repeated on each function for 30 independent runs to compensate for the variance caused by the stochastic PSO algorithms, and the best fitness score is recorded for each run.

To compare the results between optimisers, the ranking metric proposed in Section 5.3.2 is used. In other words, the best fitness achieved at each run is ranked, which provides a number in the range of $[0, 1]$, with a higher value indicating a better performance. The average ranking per function and dimension is then calculated for each optimiser, and the overall average is also calculated. An optimiser with a higher overall ranking therefore performs better on average on the CEC benchmark suite. To provide additional evidence of any conclusions drawn using the above approach, a Mann-Whitney U-test [31] will also be conducted to measure significant differences between optimisers' performances on each function, with the null-hypothesis rejected for $p < \alpha = 0.05$.

All the PSO algorithms discussed in Chapter 3 will be used, as well as the canonical

PSO algorithm as discussed in Chapter 2. The hyper-parameters defined for each algorithm are set to default values provided by the author where applicable, or set to commonly-used suggested values otherwise. The algorithms, as well as their respective hyper-parameter values, are listed in Table 7.1. Note that for the canonical PSO, the coefficient values are the commonly-used parameters derived in [13]. An alternative choice of coefficients for the canonical PSO algorithm has also been included and denoted as PSO_{CEC}. These coefficient values were chosen as the best-performing set of parameters among those used during collection of the ranking data points (see Section 5.3.2), and can thus be considered as coefficients that have been "fine-tuned" for the CEC benchmark functions.

To further ensure fairness across all PSO algorithms, the swarm size, particle position and velocity initialisation, velocity clamping, and stopping conditions are the same for all algorithms. The only remaining difference between the chosen algorithms is thus the strategy used to choose coefficient values, since all algorithms also use the regular PSO update step as described in Section 2.1. For this experiment and all to follow, a swarm size of 30 is used. Particle positions are initialised uniformly at random within the search space, and velocities are initialised to zero [14]. The velocities are clamped according to Equation (2.6) with $v_{max} = 200$ (the size of the CEC functions' search space). Finally, the only stopping condition is to stop after a set number of iterations, which is equivalent to setting a computational budget using a fixed number of function evaluations (the number of function evaluations is equal to the swarm size times the number of iterations). Each algorithm is allowed 1000, 1500, 2500, and 5000 iterations respectively for dimensions 10, 30, 50, and 100. The above choices are the same as used for all experiments in Chapter 6.

The QLPSO algorithm also includes a training step, and is trained in a similar way to the CRLPSO algorithm. The QLPSO policy is learned using only the 20 functions marked for training (as indicated in Appendix A) and is trained on all dimensionalities. Training is performed for 250 epochs, where each epoch is a single run of QLPSO with the Q-table updated afterwards using the Q-learning algorithm discussed in Section 4.2, with $\epsilon = 0.05$, $\gamma = 0.95$, and $\alpha = 0.01$. Note that during inference, actions are chosen deterministically and the policy is not updated.

For the CRLPSO policy, the best performing policy found in Chapter 6 is used. See Section 6.6 for details on how this policy was trained. Also recall that in the experiments from Chapter 6, all evaluation results were a result of 3 repeated experiments. The policy chosen for this experiment is the most performant among these 3 policies, hence the

Table 7.1: PSO algorithms used for comparison along with the hyper-parameters used in the implementation of each.

Algorithm	Reference	Hyper-Parameters
PSO [25]	§ 2.1	$c_1 = c_2 = 1.49618$, $w = 0.7298$
PSO_{CEC}	§ 2.1	$c_1 = c_2 = 0.8$, $w = 0.9$
PSO-TVIW [48]	§ 3.1	$c_1 = c_2 = 1.49618, w_s = 0.9, w_f = 0.4$
PSO-TVAC [40]	§ 3.1	$c_{1s} = 2.5$, $c_{1f} = 0.5$, $c_{2s} = 0.5$, $c_{2f} = 2.5$, $w_s = 0.9$, $w_f = 0.4$
APSO-VI [59]	§ 3.2	$c_1 = c_2 = 1.49618$, $T_{\text{end}} = 0.95 \cdot T$, $\Delta w = 0.1, w_{\min} = 0.3$, $w_{\max} = 0.9$
IPSO [28]	§ 3.3	$c_1 = c_2 = 1.49618$, $\alpha = 0.5$, $\beta = 0.5$
FST-PSO [35]	§ 3.4	–
QLPSO [30]	§ 3.5	–

performance is expected to be slightly higher than what was reported previously.

7.1.2 Results

The average ranking achieved by each optimiser at each dimension as well as on average is shown in Table 7.2. The CRLPSO policy was able to achieve the second best overall performance, being beaten only by the PSO-TVAC algorithm. The QLPSO policy also does quite well in third place, with the PSO-TVIW and APSO-VI algorithms also still maintaining a ranking above 0.9. The remaining optimisers – FST-PSO, canonical PSO, and IPSO – have the three lowest scores, with IPSO being the worst overall. The number of wins and losses of the CRLPSO policy against the other optimisers are also shown in Table 7.3. The wins and losses indicate roughly the same performance relative to other optimisers as the rankings do, since there are generally more wins against optimisers with lower rankings. For example, at 30 dimensions where CRLPSO has a ranking of 0.924, there are 20 wins against FST-PSO which has a lower ranking of 0.873, and only 5 wins against the canonical PSO which has a higher ranking of 0.925.

When considering the results at each dimension separately, a few clear patterns emerge. Firstly, the CRLPSO policy does not perform as well relative to the other algorithms at 10 and 30 dimensions, with the two time-varying PSOs (PSO-TVAC and PSO-TVIW) and the canonical PSO outperforming the policy at these dimensions. This is especially true at the low 10-dimension problems, where most optimisers (except FST-PSO) have a similar score around 0.87, however CRLPSO is on the lower end here with a score of 0.868. It is also evident that CRLPSO performs worse on lower dimension problems from the wins and losses, where there are considerably fewer wins and more ties and losses at 10 dimensions compared to the wins and losses at higher dimensions.

At the 50 dimension problems, the performance of the canonical PSO, IPSO, and PSO-TVIW algorithms start to degrade, and at 100 dimensions these three perform significantly worse than the other algorithms. Most notably, the canonical PSO and IPSO algorithms have rankings below 0.5, where other algorithms still maintain good performance. Interestingly, it is only at this point where the tuned PSO_{CEC} results become significantly better than the regular PSO coefficients, and is able to maintain similar performance to PSO-TVIW at 100 dimensions. All of the above is a good indication that at higher dimensions, some algorithms require much finer tuning of hyper-parameters to achieve performance equal to what is achievable by the algorithm at lower dimensions.

The CRLPSO policy is consistently among the top 3 performing algorithms at all dimensions higher than 10, with the QLPSO and PSO-TVAC being strong contenders at all dimensions. It is however only at the 100-dimension problems where the CRLPSO policy significantly outperforms all other optimisers. This is especially evident when considering the wins and losses from Table 7.3, which shows that in most cases, performance of the CRLPSO policy is significantly better on most or even all 30 of the functions. The QLPSO and FST-PSO algorithms also perform relatively well at the higher dimension problems. APSO-VI and PSO-TVAC also perform well in terms of the ranking score, but is outperformed by the CRLPSO policy on most of the functions.

In conclusion, it is clear that the CRLPSO policy performs better at higher dimensions compared to the other optimisers, but has worse performance at lower dimensions. This could likely be due to the exclusive use of high-dimensional functions during training. However, the conclusions from Section 6.6 suggested that the performance is worse regardless of the problem dimension when trained at lower dimensions. It is also evident that the time-varying algorithms, and especially PSO-TVAC, perform significantly better than the

Table 7.2: Ranking metrics scored by the different PSO algorithms at different problem dimensions on the CEC benchmark problem suite. The top performing technique for each problem dimensionality is highlighted in boldface.

	PSO	PSO_{CEC}	PSO-TVIW
$D = 10$	0.88058 ± 0.12341	0.87271 ± 0.13013	0.88835 ± 0.11458
$D = 30$	0.92476 ± 0.07505	0.93223 ± 0.06566	0.91570 ± 0.08033
$D = 50$	0.92036 ± 0.07603	0.92946 ± 0.06775	0.90523 ± 0.07076
$D = 100$	0.42391 ± 0.30411	0.89348 ± 0.09940	0.89247 ± 0.07636
Average	0.78740 ± 0.27225	0.90697 ± 0.09774	0.90044 ± 0.08787
	PSO-TVAC	APSO-VI	IPSO
$D = 10$	$\mathbf{0.89347 \pm 0.11478}$	0.88616 ± 0.10412	0.86283 ± 0.12769
$D = 30$	$\mathbf{0.93987 \pm 0.06265}$	0.90580 ± 0.08343	0.88335 ± 0.10873
$D = 50$	$\mathbf{0.94475 \pm 0.04723}$	0.90808 ± 0.08033	0.75436 ± 0.21435
$D = 100$	0.94887 ± 0.03854	0.91587 ± 0.07884	0.39094 ± 0.24223
Average	$\mathbf{0.93174 \pm 0.07551}$	0.90398 ± 0.08796	0.72287 ± 0.26893
	FST-PSO	QLPSO	CRLPSO
$D = 10$	0.75940 ± 0.17592	0.86325 ± 0.12557	0.86759 ± 0.11248
$D = 30$	0.87371 ± 0.10168	0.91671 ± 0.09606	0.92381 ± 0.06535
$D = 50$	0.91467 ± 0.06798	0.94097 ± 0.06499	0.94454 ± 0.04359
$D = 100$	0.95660 ± 0.03777	0.95004 ± 0.06513	$\mathbf{0.96969 \pm 0.02914}$
Average	0.87609 ± 0.13127	0.91774 ± 0.09749	0.92641 ± 0.07959

self-adaptive algorithms and the canonical PSO algorithm. It would thus be beneficial to prefer these algorithms in practice, given their simplicity and excellent performance. In the next section, the performance of the CRLPSO policy is further investigated to determine whether the performance observed here also carries over to other benchmark problems.

7.2 Generalisation of Performance to Unseen Problems

As with training machine learning models, testing the performance on functions not used during training or development of the technique is important since a bias towards the functions used in training will very likely develop. This bias can exist firstly because the policy is trained on a specific set of functions (in this case a subset of the CEC 2017 benchmark suite), and may thus "overfit" its behaviour to work well on those, and only those, functions. This bias is well-known in machine learning research, and is commonly prohibited through the use of a separate validation set to measure performance during or after training a model. A second bias can also occur in this validation set however, since design decisions may be made based on the performance of a model on the validation set, which may not necessarily improve performance on other data. To avoid this, a "holdout" set is also a common practice, which is only used to evaluate a model's final performance.

Recall that the policy was trained using only a subset of 20 of the 30 functions, with 10 of the 30 functions being kept out as a sort of validation set. This section will firstly take a closer look at the performance difference between these two sets of functions in Section 7.2.1. This section will then also evaluate the performance of the policy on the BBOB function suite in Section 7.2.2. These functions were not considered in any way during the design of the training process or in the choice of the final policy, and can thus be considered as equivalent to a true holdout set.

Table 7.3: Number of CEC benchmark function wins and losses at different dimensions. The < column indicates functions where CRLPSO performed significantly better than the other optimiser, > indicates significantly worse performance, and = indicates no significant difference.

	$D = 10$			$D = 30$			$D = 50$			$D = 100$		
	<	=	>	<	=	>	<	=	>	<	=	>
PSO	1	17	12	5	16	9	14	9	7	30	0	0
PSO_{CEC}	2	21	7	2	20	8	11	10	9	27	1	2
PSO-TVIW	0	16	14	6	17	7	20	9	1	30	0	0
PSO-TVAC	0	13	17	1	16	13	6	15	9	24	3	3
APSO-VI	1	18	11	9	20	1	18	7	5	27	2	1
IPSO	3	22	5	16	11	3	30	0	0	30	0	0
FST-PSO	19	8	3	20	2	8	19	2	9	15	5	10
QLPSO	4	17	9	7	13	10	7	14	9	14	9	7

Table 7.4: Rankings scored by different PSO algorithms on only the CEC benchmark functions marked as training or evaluation, respectively. The ranking drift indicates the difference between the evaluation and training set scores.

	Train Ranking	Evaluation Ranking	Ranking Drift
PSO	0.78972	0.78276	-0.00696
PSO-TVIW	0.90061	0.90008	-0.00053
PSO-TVAC	0.92879	0.93765	0.00886
APSO-VI	0.90645	0.89904	-0.00741
FST-PSO	0.87277	0.88274	0.00997
IPSO	0.74415	0.68030	-0.06385
QLPSO	0.91566	0.92190	0.00625
CRLPSO	0.92558	0.92808	0.00250

7.2.1 Training Performance Bias

To determine whether there is a difference in performance between the 20 functions used for training and the 10 functions used only during evaluation, the difference between the average ranking achieved on each function set can be calculated. Any measured difference in performance may also be a result of the different function properties and landscapes, so the difference between these sets must be measured for all algorithms considered in the previous section, in order to put any performance difference of the CRLPSO policy into context.

The performances on both function sets for the CRLPSO policy and the average of the other optimisers are shown in Table 7.4. From these results, it is immediately clear that there is no significant difference in the rankings achieved on the two function sets, and they are both close to the average ranking reported in Table 7.2. In fact, the performance of the CRLPSO policy increases very slightly in the evaluation set compared to the training set, whereas most of the other optimisers had a slightly worse performance on the functions in the evaluation set. There is thus no evidence here to suggest that the policy overfit the training functions. However, it is still necessary to examine the performance on completely unseen benchmark functions, which is the topic of the following section.

7.2.2 Performance on Unseen Problems

For the purposes of this experiment, the BBOB benchmark suite available in the COCO testing framework [17] is used. This suite provides 24 functions with different properties, defined for 6 different dimensions, namely 2, 3, 5, 10, 20, and 40 dimensions. There is thus a total of 144 problem functions. The COCO framework also provides 15 "instances" per function, which are the same function but with different shifts and rotations applied to the parameter vectors. The framework then evaluates an optimiser by running the 15 instances of each function only once, rather than performing repeat runs on a single function. This is to allow consistent evaluation of deterministic and stochastic optimisers. However, since all PSO algorithms considered here are stochastic, only the first instance is used but repeated 30 times as in the case of the CEC benchmarks performed thus far.

Also note that the COCO framework includes a specific set of metrics and methods for measuring them in order to consistently compare different optimisers, however this experiment will not provide results in the format provided by COCO. This is because the COCO framework mainly benchmarks optimisers using the expected runtime (i.e. the number of function evaluations required to reach a specified target value for each problem function), whereas the experiments in this study have rather been focused on how optimal the value that can be found is when given a fixed number of function evaluations (or equivalently, a fixed number of iterations). Furthermore, in order to remain even more consistent with previous results, the process to collect data for the ranking metrics of the CEC benchmark functions (outlined in Section 5.3.2) was repeated for the 144 function and dimension pairs of the BBOB suite. The ranking metric is thus also used for this experiment.

This experiment makes use of the same PSO algorithms as in Section 7.1, as well as the same setup and hyper-parameter values as described there. This experiment considers 3 different CRLPSO policies, rather than only using the one that performed best during training. Recall that the policy used in the CEC benchmarks in Section 7.1 was the best performing policy trained using only 100-dimensional functions. In addition to this policy, this experiment also considers a policy trained using 10-dimensional functions, and a policy trained using a combination of 10-, 30-, 50-, and 100-dimensional functions. Refer to Section 6.6 for further details on how the training of these policies was conducted. These additional policies are included in order to better measure the ability of these policies to generalise to different functions, since the type of functions seen during training may impact the generalisation ability of the policy. The policies trained at 10, 100, and mixed dimensions are denoted as $\text{CRLPSO}_{\text{D10}}$, $\text{CRLPSO}_{\text{D100}}$, and $\text{CRLPSO}_{\text{General}}$, respectively.

Results

The results of the BBOB problems in terms of the ranking metric are shown in Table 7.5. As in the case of the CEC benchmarks, Table 7.6 also shows the number of wins and losses for each optimiser based on a Mann-Whitney U-test [31] with a p-value of 0.05. For the sake of brevity, the wins and losses are not grouped by dimension. To also make it easier to see the difference in performance between the CEC and BBOB suites, Table 7.7 shows the difference in the ranking score of each set for each optimiser. The difference in performance between the suites is also shown while excluding the results for the 100-dimensional CEC problems, since the BBOB suite goes only up to 40 dimensions. This difference can thus be considered a better indication of how the optimiser differs on both problem sets, since some optimisers (notably canonical PSO and IPSO) did poorly at higher dimensions.

The relative performance of the CRLPSO policy is quite different here than when com-

Table 7.5: Ranking metrics scored by the different PSO algorithms at different problem dimensions on the BBOB function suite. The top performing technique for each problem dimensionality is highlighted in boldface.

	PSO	PSO_{CEC}	PSO-TVIW	PSO-TVAC
$D = 2$	0.92809 ± 0.15208	0.83966 ± 0.17313	0.94271 ± 0.13069	$\mathbf{0.95289 \pm 0.11262}$
$D = 3$	0.87869 ± 0.16040	0.87029 ± 0.14714	0.90932 ± 0.12158	$\mathbf{0.91966 \pm 0.11586}$
$D = 5$	0.88320 ± 0.14208	0.87985 ± 0.13268	0.89739 ± 0.12676	$\mathbf{0.91075 \pm 0.11194}$
$D = 10$	0.91069 ± 0.10974	0.90903 ± 0.11107	0.91905 ± 0.09756	$\mathbf{0.93136 \pm 0.08841}$
$D = 20$	0.93279 ± 0.09185	0.92639 ± 0.08761	0.92306 ± 0.09014	$\mathbf{0.93689 \pm 0.07642}$
$D = 40$	0.93548 ± 0.09846	0.93297 ± 0.09014	0.92398 ± 0.08815	0.94956 ± 0.06658
Overall	0.91149 ± 0.13064	0.89303 ± 0.13157	0.91925 ± 0.11144	$\mathbf{0.93352 \pm 0.09839}$
	APSO-VI	IPSO	FST-PSO	QLPSO
$D = 2$	0.88046 ± 0.15391	0.91659 ± 0.17539	0.65194 ± 0.18967	0.80396 ± 0.19139
$D = 3$	0.88044 ± 0.12354	0.87041 ± 0.17442	0.71695 ± 0.19953	0.81172 ± 0.14117
$D = 5$	0.87897 ± 0.11375	0.86575 ± 0.13793	0.78427 ± 0.16361	0.83616 ± 0.10535
$D = 10$	0.88416 ± 0.09933	0.88559 ± 0.11419	0.84548 ± 0.12376	0.88186 ± 0.09990
$D = 20$	0.90030 ± 0.09792	0.89574 ± 0.10155	0.89784 ± 0.09429	0.91849 ± 0.07386
$D = 40$	0.91551 ± 0.10250	0.86186 ± 0.14892	0.94537 ± 0.06124	$\mathbf{0.95106 \pm 0.05316}$
Overall	0.88997 ± 0.11758	0.88266 ± 0.14602	0.80698 ± 0.17891	0.86721 ± 0.13155
	$\text{CRLPSO}_{\text{D100}}$	$\text{CRLPSO}_{\text{General}}$	$\text{CRLPSO}_{\text{D10}}$	
$D = 2$	0.87109 ± 0.15482	0.88829 ± 0.15264	0.90232 ± 0.13643	
$D = 3$	0.85777 ± 0.13329	0.86529 ± 0.15244	0.89625 ± 0.11988	
$D = 5$	0.85484 ± 0.12940	0.85860 ± 0.13949	0.88040 ± 0.11492	
$D = 10$	0.87127 ± 0.11990	0.88747 ± 0.11389	0.89688 ± 0.10384	
$D = 20$	0.88950 ± 0.11271	0.90181 ± 0.10609	0.89962 ± 0.10002	
$D = 40$	0.91586 ± 0.10658	0.92325 ± 0.10415	0.91791 ± 0.10004	
Overall	0.87672 ± 0.12879	0.88745 ± 0.13158	0.89890 ± 0.11381	

Table 7.6: Number of BBOB benchmark function wins and losses. The $<$ column indicates functions where the CRLPSO policy indicated in the column performed significantly better than the other optimiser, $>$ indicates significantly worse performance, and $=$ indicates no significant difference.

	$\text{CRLPSO}_{\text{D100}}$			$\text{CRLPSO}_{\text{General}}$			$\text{CRLPSO}_{\text{D10}}$		
	$<$	$=$	$>$	$<$	$=$	$>$	$<$	$=$	$>$
PSO	16	59	69	16	68	60	25	61	58
PSO_{CEC}	21	59	64	26	64	54	29	72	43
PSO-TVIW	10	56	78	14	70	60	11	79	54
PSO-TVAC	7	47	90	10	50	84	7	66	71
APSO-VI	11	88	45	38	85	21	41	83	20
IPSO	38	47	59	36	62	46	49	49	46
FST-PSO	76	23	45	79	28	37	85	21	38
QLPSO	59	55	30	73	49	22	72	48	24
$\text{CRLPSO}_{\text{D100}}$	-	-	-	49	92	3	58	81	5
$\text{CRLPSO}_{\text{General}}$	3	92	49	-	-	-	29	90	25
$\text{CRLPSO}_{\text{D10}}$	5	81	58	25	90	29	-	-	-

Table 7.7: Difference between rankings scored by different PSO algorithms on the BBOB and CEC benchmark functions. The second set of values excludes the 100-dimensional results from CEC to have more equal dimensionalities between the two function suites.

	All CEC dimensions			Excluding CEC $D = 100$		
	CEC	BBOB	Difference	CEC	BBOB	Difference
PSO	0.78740	0.91149	0.12409	0.90857	0.91149	0.00292
PSO_{CEC}	0.90697	0.89303	-0.01394	0.91147	0.89303	-0.01844
PSO-TVIW	0.90044	0.91925	0.01882	0.90309	0.91925	0.01616
PSO-TVAC	0.93174	0.93352	0.00178	0.92603	0.93352	0.00749
APSO-VI	0.90398	0.88997	-0.01400	0.90001	0.88997	-0.01004
IPSO	0.72287	0.88266	0.15979	0.83351	0.88266	0.04915
FST-PSO	0.87609	0.80698	-0.06912	0.84926	0.80698	-0.04228
QLPSO	0.91774	0.86721	-0.05053	0.90697	0.86721	-0.03976
$\text{CRLPSO}_{\text{D100}}$	0.92641	0.87672	-0.04969	0.91198	0.87672	-0.03526
$\text{CRLPSO}_{\text{General}}$	0.91810	0.88745	-0.03065	0.90534	0.88745	-0.01789
$\text{CRLPSO}_{\text{D10}}$	0.88342	0.89890	0.01548	0.87596	0.89890	0.02294

paring using the CEC benchmarks. Where before the top-performing optimisers were PSO-TVAC, QLPSO, and the CRLPSO policy, the top-performing optimisers on the BBOB functions are the canonical PSO and time-varying algorithms. PSO-TVAC is still the top-performing algorithm, which again shows that it is an excellent choice in practice. Surprisingly, the canonical PSO does very well, being beaten only by the two time-varying algorithms. The PSO_{CEC} (which is the canonical PSO with coefficients tuned for the CEC problems) also does relatively well, but is worse than the default set of coefficients. This also shows the sensitivity of PSO to its choice of coefficient values since the tuned coefficients did significantly better than the default values on the CEC problems.

Of all the self-adaptive optimisers, IPSO is the only one that performed better on the BBOB problems compared to CEC. This is however very likely because of the lower dimensionality of the BBOB suite compared to the CEC functions, and even on these problems the IPSO performance gets worse as the dimensionality increases (which is not the case with the canonical and time-varying algorithms). Interestingly, the remaining self-adaptive algorithms (APSO-VI, FST-PSO, QLPSO, and the CRLPSO policies) all do significantly worse on the BBOB problems. FST-PSO and QLPSO both have especially large drops in performance, which again may be attributed to the lower dimensionality as they evidently perform worse at the lower dimensions. The $\text{CRLPSO}_{\text{D100}}$ policy also has a significant drop in performance, but has a more consistent decrease across all dimensions. This can be seen by the fact that the QLPSO and FST-PSO algorithms perform better at the higher dimensions compared to $\text{CRLPSO}_{\text{D100}}$, but much worse at the lower dimensions. The $\text{CRLPSO}_{\text{D100}}$ policy however still outperforms both QLPSO and FST-PSO on average, which is further supported by the number of wins and losses reported in Table 7.6.

The difference in performance between the three CRLPSO policies is also significant. Looking at the wins and losses, it is clear that the $\text{CRLPSO}_{\text{D10}}$ policy performs the best, being marginally better than the $\text{CRLPSO}_{\text{General}}$ policy and significantly better than the $\text{CRLPSO}_{\text{D100}}$ policy. This actually suggests an inverse relationship between the CEC and BBOB performance, since the performance of these three policies on the CEC functions have the opposite ordering, as can be seen in Table 7.7. The $\text{CRLPSO}_{\text{D10}}$ policy in fact has a higher ranking on the BBOB functions than on the CEC suite.

This inverse relationship may be evidence of a bias towards the CEC functions devel-

oped during training, even though there was no significant difference between the training and evaluation sets as discussed in Section 7.2.1. It is possible that the CEC functions used in the training and evaluation sets were similar enough to show no evidence of overfitting, or that this is simply bias from the use of the CEC functions during the design of the training process. There are of course other factors than overfitting that may explain these differences, among them being the difference in dimensionality between the two function suites. The fact that the performance of all the other PSO algorithms also vary greatly between the sets seems to support this. It may be possible to provide sufficient evidence of overfitting by examining the performance on additional benchmark suites, however this is outside the scope of this study.

On a more general note, the properties of different benchmark suites can vary greatly and can greatly impact the performance of optimisers, as is evident from this experiment. It is therefore important for optimisers to be tested on different suites to properly evaluate the expected performance to allow practitioners to better decide which algorithms to use. This closely ties in to the discussion in Section 3.6, since many PSO algorithms claim good performance in the original paper, but are then found to perform poorly compared to other optimisers in later results. This is most likely because the algorithm was designed to perform well on that specific set of functions used in the paper, whether to the author's knowledge or not. The optimisation community can thus benefit greatly from adopting similar practices to the machine learning community with the use of multiple benchmark suites.

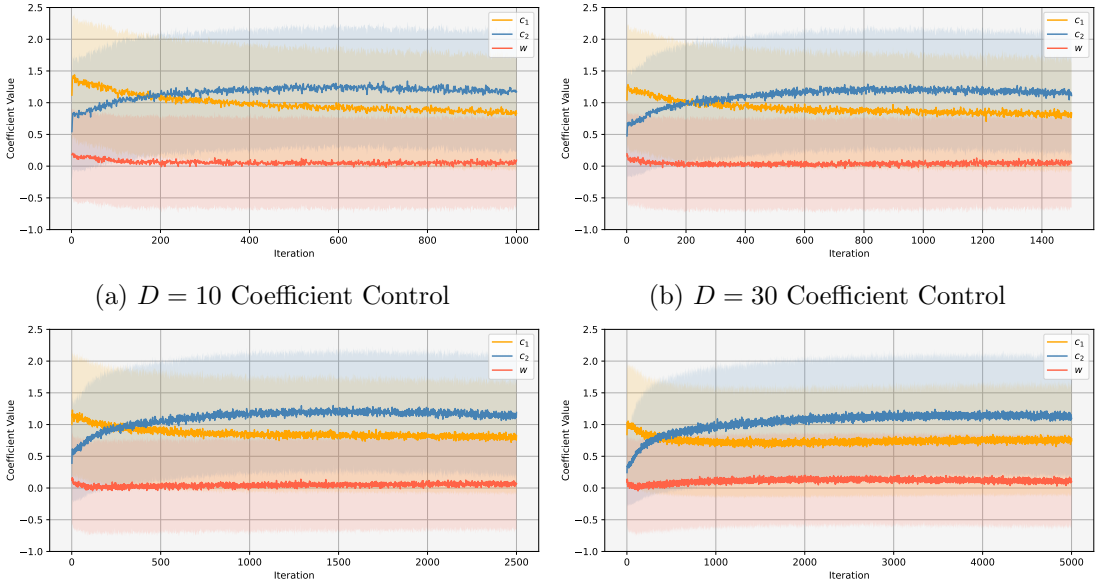
In conclusion of these results and those of Section 7.1, it is evident that the RL training can be applied successfully to learn a continuous PSO coefficient control policy, and is able to learn behaviours that enable better performance than the tested self-adaptive PSO algorithms. It is not however able to consistently beat all algorithms and does not seem to generalise well to unseen problems, as was shown in this section. In the next section, the behaviour of the policy will be investigated in more detail to get an understanding of what the RL agent has learnt.

7.3 Policy Behavioural and Convergence Analysis

As the final component of this study, this section provides a detailed investigation of the behaviour of the learnt CRLPSO policy. This is to gain a better understanding of whether the RL was really able to learn any interesting or useful behaviours, and may provide deeper insights on how to improve the technique in future. This analysis will again focus only on the best policy found in Chapter 6, which is the policy trained using only 100-dimensional problems, and which was denoted as $\text{CRLPSO}_{\text{D100}}$ in the previous experiment. See Section 6.6 for more details on how this policy was trained. The analysis will also only consider the CEC benchmark functions. The remainder of this section is separated into two parts: Section 7.3.1 first investigates the coefficient choices made by the policy, and Section 7.3.2 analyses the convergence properties of the swarm when controlled by the policy.

7.3.1 Coefficient Control

Recall that the CRLPSO agent is able to directly choose the values of the coefficients c_1, c_2, and w at each iteration of the PSO algorithm. As discussed in Section 6.3, the output of the model is 3 real values that form the means of normal distributions, from which the vector $\boldsymbol{h}_a \in \mathbb{R}^3$ is sampled. The variance of each normal distribution is also a trainable



(a) $D = 10$ Coefficient Control

(b) $D = 30$ Coefficient Control

(c) $D = 50$ Coefficient Control

(d) $D = 100$ Coefficient Control

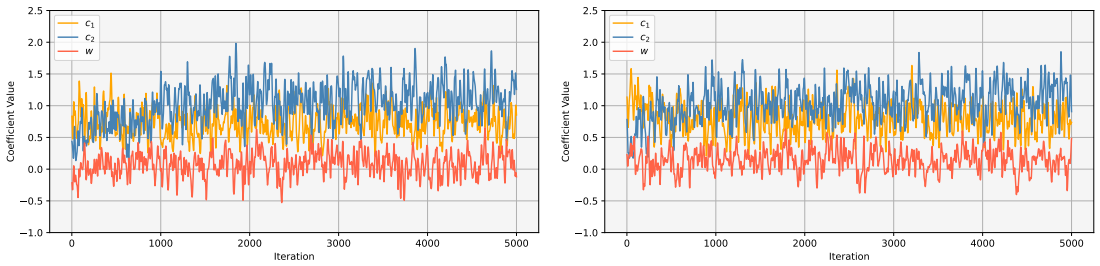
Figure 7.1: Coefficient values chosen by the CRLPSO policy on the CEC benchmark problems. Values are averaged across 30 runs per function and 30 different functions, and shown per dimension. Shaded areas represent the standard deviation.

parameter which is independent of the state. The values of the vector $\boldsymbol{h}_a$ are then clamped using the tanh function as in Equation (5.2) to obtain the final coefficient values. In this analysis, the actual values chosen by the policy will be investigated, as this can provide insights into whether the agent was able to successfully learn useful behaviours or not.

The coefficient values chosen by the CRLPSO policy at different dimensions are shown in Figure 7.1. These figures are the result of aggregation across both the 30 CEC functions and 30 independent runs per function, so each figure is the result of 900 data points. Two patterns are immediately obvious in all four dimensionalities, namely 1) the value of c_1 is decreased over the course of a run, and 2) the value of c_2 is increased over the course of a run. This is similar to the strategy used by PSO-TVAC [40], however the coefficients are linearly increased and decreased in the case of PSO-TVAC. The CRLPSO policy instead has a very sudden increase and decrease during about the first fifth of total iterations, with a slower change throughout the rest of the run.

In terms of values, c_1 decreases from between 1 and 1.5 to around 0.8 on average, and c_2 increases from around 0.5 to 1.5 on average, which in both cases is lower than the ranges of 2.5 to 0.9 and 0.9 to 2.5 used as defaults for PSO-TVAC for c_1 and c_2, respectively. Interestingly, the average starting values for both c_1 and c_2 used by the CRLPSO policy are lower as the dimensions increase, which suggests that there is some learnt behaviour that differs by dimension. There is however no consistent pattern in the choice of w across the four dimensions, but in all cases, w stays mostly around the same value (around 0.2 on average) throughout the run.

Another observation is that the variance of the coefficient values is very large, as is evidenced by the broad error bars in Figure 7.1. This high variance is due to high noise in the chosen coefficient values. Further calculation shows that the standard deviations of coefficients chosen by the policy are 0.894, 0.945, and 0.725 for c_1, c_2, and w respectively. This is very high given the ranges of $[0, 2.5]$ and $[-1.1, 1.1]$ for c_1 and c_2 and w allowed by



(a) F_{15} Coefficient Control

(b) F_{28} Coefficient Control

Figure 7.2: Coefficient values chosen by the CRLPSO policy during selected individual runs. To improve the clarity of the graphs, results have been significantly smoothed using a Gaussian filter with $\sigma = 5$.

the tanh clipping. This suggests that the agent has low confidence in its coefficient choices, since the variance of the normal distributions has not been reduced during training. This may be due to a difficult learning environment which does not allow the policy to converge. It is suspected that the high variance of the PSO environment due to the stochastic nature of the algorithm also adds much difficulty over and above the difficulties already caused by the so-called "deadly triad" of RL techniques [52, §11.3] (the deadly triad refers to divergence during training caused by the use of function approximation, bootstrapping, and off-policy training). Deterministic techniques such as DDPG [29] may be able to aid by reducing the variance of coefficients, however early testing with DDPG proved ineffective.

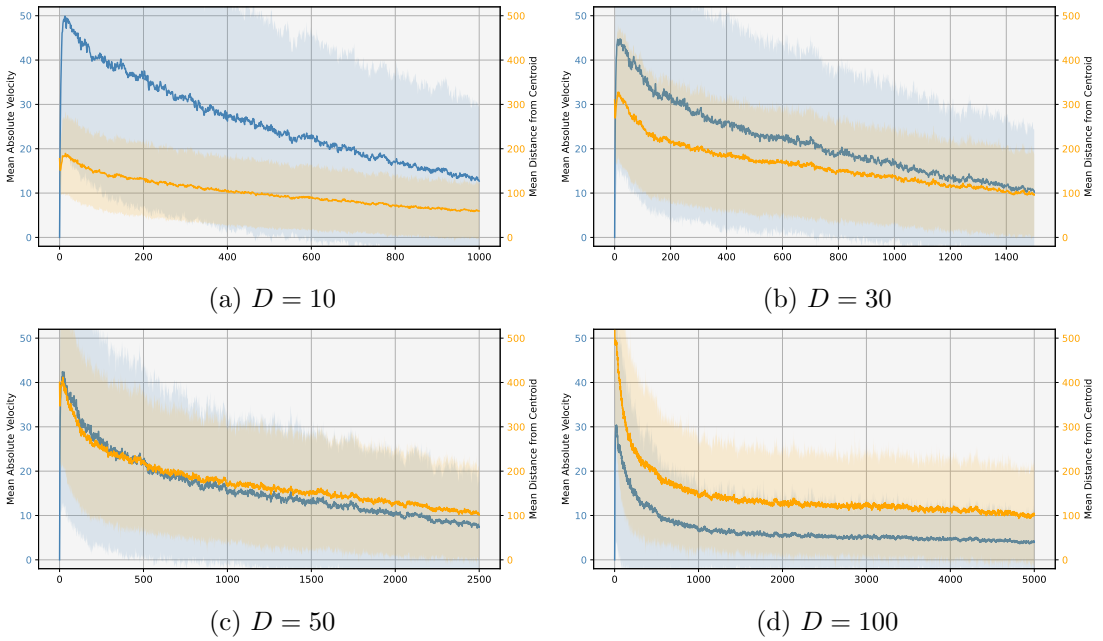
As a final part of this investigation into the coefficient choices, Figure 7.2 shows the coefficient choices for two individual runs on 100-dimensional F_{15} and F_{28} (arbitrarily chosen), since some of the nuance may be lost in the aggregated results of Figure 7.1. Note that these two graphs have been significantly smoothed using a Gaussian filter, and despite this the results still appear noisy due to the high variance discussed above.

In both cases, the increase of c_2 seen in Figure 7.1 is still quite pronounced despite the noise, however the decrease of c_1 is only visible on F_{28}, and is not as clear on F_{15}. Nevertheless, these behaviours are still present as expected. Besides the control of c_1 and c_2, it is difficult to identify any other nuanced behaviour due to the noise, however there are discernable patterns at some points. For example, at around iteration 3700 in Figure 7.2a (F_{15}), there is an increase in both w and c_2, and a decrease in c_1. This may still be no more than noise, and the actions will have to be considered alongside the values of the state features to better understand whether the policy has learnt any intelligent behaviour besides the decrease and increase of c_1 and c_2, respectively. This sort of in-depth analyses is however outside the scope of this study.

7.3.2 Convergence Analysis

This next part of the analysis focusses on the convergence and stability of the swarm. An overview of the theoretical stability criteria that guarantee convergent behaviour of the particle swarm was provided in Section 2.2. Convergent behaviour is important since divergent particles adversely impact the performance of PSO [7], and eventual convergence of the swarm leads to local search that can refine the best found positions. As a result, surveys of PSO algorithms frequently include some form of convergence analysis [55, 18]. This section therefore provides such analysis for the learnt CRLPSO policy.

Figure 7.3 shows both the mean velocity and the average distance from the swarm



(a) $D = 10$ (b) $D = 30$

(c) $D = 50$ (d) $D = 100$

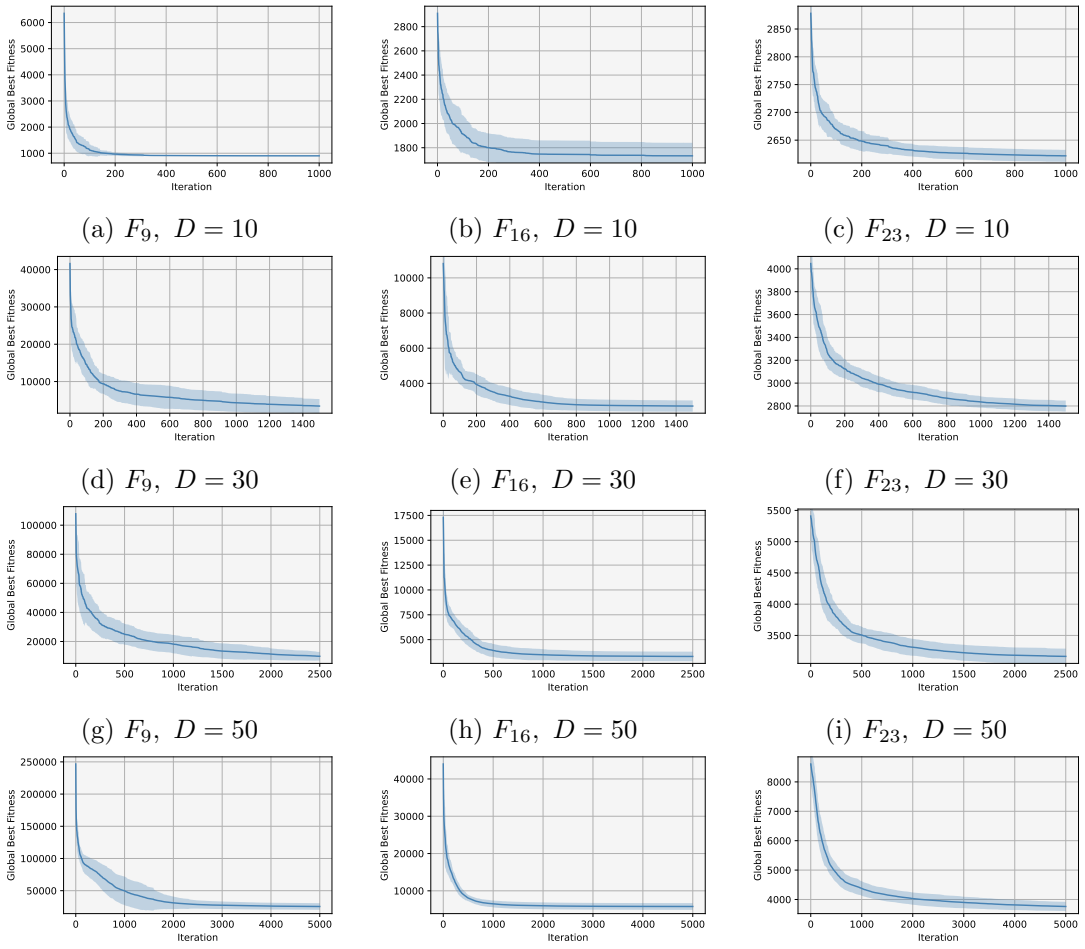
Figure 7.3: Average particle distance from the swarm centroid and mean absolute particle velocity maintained by the CRLPSO policy on the CEC benchmark problems. Values are averaged across 30 runs per function and 30 different functions, and shown per dimension. Shaded areas represent the standard deviation.

centroid, which both help describe the behaviour of the swarm. The mean velocity is the average L_1 norm of the particle velocity, and is expected to approach zero towards the end of the run. This is the concept used to define the "ideal" velocity curve of APSO-VI [59]. The velocity of the swarm controlled by the CRLPSO policy does go down as desired, however it still remains fairly high on average towards the end of the runs in all four dimensions. This suggests that the controller exhibits convergent behaviour, although faster convergence may lead to better performance. The fact that the mean velocity is still quickly declining towards the end for all but the 100-dimensional problems also indicates that the swarm has not converged yet at the end of the run.

The distance from the swarm centroid measures the average Euclidean distance from each particle to the center of the swarm (the average of all particle positions). The curve is also expected to decrease over time in the case of convergent behaviour, and again, this is in fact the case for the CRLPSO policy. Similar to the velocity however, the swarm distance declines steadily, but in all dimensions except 100 it is still declining quite quickly by the end of the run, which indicates that the swarm has not fully converged yet. All signs therefore point to insufficient convergence on average.

Also note that for both the velocity and distance from the swarm centroid, the variance is fairly high, which suggests that the swarm converges much faster on some functions than others. An ideal policy would be able to control the swarm well enough to have a high mean velocity and swarm distance at the start, but have a small velocity and distance towards the end, no matter what the underlying function is. This is indeed the goal that most, if not all, self-adaptive PSO algorithms strive towards. However, it seems that the CRLPSO policy is not able to achieve equally good convergence across all functions.

To provide further evidence of the above statement, Figure 7.4 shows the global best fitness value of three select functions over time, at all four dimensions. The functions F_9,



(a) $F_9,\ D = 10$ (b) $F_{16},\ D = 10$ (c) $F_{23},\ D = 10$

(d) $F_9,\ D = 30$ (e) $F_{16},\ D = 30$ (f) $F_{23},\ D = 30$

(g) $F_9,\ D = 50$ (h) $F_{16},\ D = 50$ (i) $F_{23},\ D = 50$

(j) $F_9,\ D = 100$ (k) $F_{16},\ D = 100$ (l) $F_{23},\ D = 100$

Figure 7.4: Global best fitness, $f(\hat{\boldsymbol{y}}(t))$, of select functions over the duration of runs using the CRLPSO policy. Results are averaged across 30 runs. Shaded areas represent the standard deviation.

F_{16}, and F_{23} were arbitrarily chosen, but show clear differences in the convergence of the swarm. In some cases there appears to be premature convergence, such as Figures 7.4a and 7.4k. In other cases, the swarm does not converge before the end of the run and therefore converges too slowly, as in Figures 7.4d and 7.4l.

Another way to analyse the convergent behaviour of the policy is to compare the choice of coefficients with the theoretical order-2 stability criteria of Equation (2.15), similar to the convergence analysis done in the survey of [18]. A quick calculation was done to see how frequently the coefficient choices made by the policy fall within the order-2 stability criteria. The outcome shows that on average, only 45.2% of iterations have coefficients that guarantee order-2 stability, and by extension, guarantee convergence. This helps to explain why the swarm does not converge as quickly, but further analysis would be required since this simple check does not take into consideration that some coefficient choices lead to faster convergence than others. A controller can therefore make some "divergent" coefficient choices, as long as the overall behaviour leads to eventual convergence. Such analysis is however outside the scope of this study.

Although it is interesting to consider the behaviour of the policy, it is unfortunately the

case that the actual behaviour learnt by the agent is outside the control of the designer. One can possibly introduce bias in the reward functions to favour certain convergent behaviours, as is the case of the QLPSO reward function. However, an unbiased reward function could in theory enable the agent to learn behaviours that the designer would not have imagined. Unfortunately most experiments in this study showed that the PSO environment is difficult to learn in, so it is likely that more biased reward functions will allow training more performant policies, which is a promising topic for future work.

7.4 Summary

This chapter provided many insights into the performance and behaviour of the policies trained using RL in Chapter 6. Benchmarks of the CEC benchmark suite in Section 7.1 showed that the CRLPSO policy performs very well compared to other PSO algorithms, especially at higher dimensions. Although the time-varying PSO-TVAC [40] algorithm has surprisingly good performance. However, further testing using the BBOB function set in Section 7.2, which was not considered during training or design in Chapter 6, showed that the policy does not necessarily generalise as well to new problems. It is difficult to conclude whether or not this is a result of bias from overfitting the CEC functions used during training, however the inverse relationship between performance on the CEC and BBOB functions observed on different CRLPSO policies seems to suggest that it may be the case.

In Section 7.3, the actual coefficient choices made by the CRLPSO policy were investigated, and it was discovered that the policy has learnt behaviours similar to that of PSO-TVAC. The coefficient choices however suffer from large variance, which suggests that the PSO environment is difficult to optimise using the RL techniques considered in this study. A convergence analysis also showed that although the swarm does eventually converge, convergence is sometimes too slow and the policy's performance may increase if coefficients were chosen in a way to guarantee faster convergence.

8 | Conclusion

This chapter concludes this dissertation and the investigations into the CRLPSO technique. Section 8.1 provides a summary and concluding thoughts of the study, and Section 8.2 suggests areas of improvement or further study as opportunities for future work.

8.1 Summary

The main elements of the CRLPSO environment was defined in Chapter 5. Firstly, the chapter defined the action space which allows the CRLPSO policy to choose values of w, c_1, and c_2 at each iteration of the PSO algorithm. The disadvantages of a discrete action space were discussed, and the continuous action space used by the CRLPSO policy was introduced. It was also found through an experiment that limiting the action space to a fixed range leads to significantly better results compared to allowing the agent to use the full range of real values.

Next, Chapter 5 introduced the state space of the agent that consists of 28 real-valued features. Each feature provides a different measure of the swarm's current state and thus allows the agent to choose coefficient values in the same fashion as a traditional self-adaptive PSO algorithm. The state described by these measures include the average velocity, the current computational budget remaining, features that help describe the stability of the swarm, etc. The features are listed in Table 5.4.

The last element introduced is a set of 7 different reward functions, differing largely in the amount of bias and variance contained in its reward signal. The chapter also discussed these issues of variance and especially bias, and experimentally showed how a biased reward signal can be optimised, but does not improve the optimisation performance of the agent. The chapter introduced the 7 reward functions (listed in Table 5.3) so that the most effective one can be identified via experimentation.

Chapter 5 also introduced the ranking metric, which is a novel way of directly comparing the performance between different problem functions. Different problem functions may have very different ranges, however the ranking metric is always within the range of $[0, 1]$, so results can be aggregated without being dominated by a single function with a large range. This metric was used throughout the study and was also key in the definition of some of the reward functions. It was also later shown in Chapter 7 that results derived using statistical methods lead to the same results as using the ranking metric.

With the environment defined, Chapter 6 focusses on the actual application of RL to learn a policy that acts within the CRLPSO environment. The chapter starts by describing the implementation details of the PPO RL algorithm and the training framework. Many of the design elements were found to be essential for effectively optimising any of the reward functions, which already hints at the difficulties of applying RL to continuous environments.

The first experiment of Chapter 6 performs a hyper-parameter optimisation of the PPO

algorithm, and repeats this for each of the 7 reward functions in order to also investigate which reward functions are most effective. The outcome shows that only four of the reward functions show any promise, namely the sparse ranking, ranking improvement, distance from optima, and target-based reward functions (R_1, R_5, R_6, and R_7, respectively). The other three reward functions, the dense ranking, fixed improvement, and G-best improvement reward functions (R_2, R_3, and R_4, respectively) were found to be either too easy to exploit or to have too high variance and thus did not lead to any improvement in the agent's optimisation ability.

The results of this first experiment also showed that none of the training runs have a very clear improvement in optimisation performance. Even in the case of the four promising reward functions identified above, the reward functions were either difficult to optimise or were eventually exploited to some degree, with the ranking metric not showing any improvement. However, a statistical experiment comparing the results of a CRLPSO policy trained using the target-based reward and an untrained policy showed that the trained policy performed significantly better, however marginally. This therefore provided evidence that the training is in fact effective.

Chapter 6 also defines three possible policy model architectures, namely a multilayer perceptron, an LSTM-based recurrent neural network, and a 1D convolutional neural network. The next experiment focussed on selecting the most effective of these policy architectures in combination with one of the four promising reward functions found in the previous experiment. The most effective architecture and reward function combination was found to be the target-based reward with a smaller multilayer perceptron of two layers with 64 units per layer. This combination of the policy architecture, reward function, and PPO hyper-parameters were thus chosen as the most effective training setup.

The last experiment of Chapter 6 investigated how the dimensionality of functions used during training impacts the final performance of the learnt CRLPSO policy. The most important result was that training at higher dimensions improve the overall performance of the agent, even at lower dimensions. On the other hand, training at lower dimensions does not generalise well to higher dimensions. The experiment also investigated the use of mixed dimensionality problems during training, and the results showed that the performance is worse than at training at only higher dimensions. Lower dimension problems thus do not aid as much during training, and may in fact hurt the performance of the agent.

The combination of hyper-parameters, architectures, etc. considered in Chapter 6 are by no means exhaustive, but the outcome of the chapter is a training setup that can effectively train the CRLPSO policy and lead to an improvement in the optimisation performance. The best performing policy was thus found to be the smaller multilayer perceptron trained using the target-based reward (R_7), and trained using only 100-dimensional problems.

Chapter 7 provided further insights into the performance of the learnt CRLPSO policy. The chapter first compared the performance of this policy to other self-adaptive, time-variant, and canonical PSO algorithms on the CEC benchmark suite. The performance of the CRLPSO policy was very competitive compared to the other algorithms at lower dimensions, but outperformed all other algorithms at the higher 100-dimensional problems. However, based on average performance across the 10-, 30-, 50-, and 100-dimensional problems, the CRLPSO policy only performed second best, with the time-variant PSO-TVAC performing better overall. This shows again that the time-varying PSO algorithms are able to outperform self-adaptive algorithms, despite the relative simplicity of the time-variant algorithms.

Recall that the CEC benchmark suite was used during training of the agent. As

indicated in Appendix A, only a subset of 20 of the 30 functions were used during training, with 10 functions used exclusively for evaluation. Chapter 7 investigated whether there is any significant difference in performance between these two sets, and found that the learnt policy performs roughly equally as well on both sets. This suggests that the agent does not overfit the training set, however there may still be an additional bias towards the evaluation functions since they were actively used to make design decisions.

Chapter 7 continued this investigation by evaluating the CRLPSO policy and the other PSO algorithms on an additional benchmark suite (the BBOB suite) which was not used during the experiments of Chapters 5 and 6. These additional experiments showed that the performance of the learnt policy does not generalise as well, since it performed worse relative to the other PSO algorithms. This may however be attributed to the lower dimensionality of the BBOB problems, and two of the other self-adaptive algorithms (QLPSO and FST-PSO) also performed significantly worse. Therefore no conclusive evidence of overfitting could be gathered and further experimentation would be required.

A final analysis in Chapter 7 investigated the behaviour of the learnt CRLPSO policy. The analyses first investigated the patterns of coefficient control that the policy has learnt, and shows that the policy has learnt similar behaviours to the time-variant PSO-TVAC algorithm with the control of c_1 and c_2. The value of c_1 is decreased over the average run, and the value of c_2 is increased, which promotes exploration early on and local search towards the end. There is however no clear pattern in the control of w. The investigation also shows high variance in the selection of coefficients, which implies low confidence in the policy and that the policy was not able to converge during training. This suggests that the RL training is not fully effective, and the suspected cause is the high variance of the stochastic PSO algorithm. Due to the noise, it is unfortunately not easy to pick out any more nuanced behaviours of the policy in a single run.

The final investigation of Chapter 7 performed a convergence analysis of the swarm when controlled by the CRLPSO policy. The analysis found that the policy does exhibit convergent behaviour, but the swarm converges too slowly, and faster convergence may be beneficial to the performance of the policy.

8.2 Future Work

The CRLPSO environment as defined in Chapter 5 can be expanded and modified in many different ways. Future work may consider changes to the action or state spaces, or define different reward functions and investigate how that affects the policy's performance. This dissertation also did not attempt to identify which features of the state space were really utilised by the agent (i.e. to analyse the feature importance), which may be the topic of future work. Such work may provide valuable insights into better selection of the state space features. Another interesting variation of the CRLPSO technique for future work would be to control the coefficients of each particle individually, which may enable more intricate behaviours. Future work could of course also benefit from using the CRLPSO environment and the findings of this dissertation as a starting point to apply RL to the control of optimisers other than PSO.

The experiments performed in Chapter 6 made use of the PPO algorithm based on early experimentation, and future work may provide more in-depth comparisons of training the CRLPSO agent with other RL algorithms. Specifically, RL algorithms that can handle large variance or noisy environments would be a good criteria for future work. The experiments were of course also not exhaustive, and future work could explore other policy architectures, hyper-parameter combinations, learning environment and RL algorithm

implementations, etc. Applications of the CRLPSO technique to other problem domains or benchmark sets would also be simple extensions of this study.

Future work may also extend on the analyses of the CRLPSO policy performed in Chapter 7. The performance of the policy was only compared to other PSO algorithms as the topic of this dissertation is only concerned with the self-adaptive control for PSO. Future work may include comparisons with other optimisers. Future work can also extend the convergence analysis to consider theoretical stability criteria or by investigating particle behaviour and movement.

Perhaps a necessary topic for future work is to define a general framework for evaluating optimisers with the different biases discussed in Chapter 7 in mind. Such a framework should provide a way for work to be evaluated in an unbiased manner before publication of results, and should not allow authors to design the algorithms specifically for this benchmark suite. Such framework may be very difficult to design, and would be a large undertaking to design and promote to the optimisation community.

A | CEC 2017 Benchmark Functions

The below table lists the symbols and names of the 30 functions defined by the 2017 CEC benchmark function suite. The last column also indicates whether the function was part of the 20 training functions, or whether the function was only used during evaluation.

The full definitions of each function can be found in [2]. For this study, the author's Python implementation of the suite was used, which is publicly available at `https://github.com/tilleyd/cec2017-py`. All functions are shifted and rotated using the fixed rotation matrices and shift vectors provided by the original problem definitions. All functions are limited to the search range $[-100, 100]^D$.

Table A.1: Summary of the 30 CEC-2017 benchmark functions and whether or not they are part of the CRLPSO training set.

Symbol	Function Name	Training Set
F_1	Bent Cigar Function	Yes
F_2	Sum of Different Power Function	No
F_3	Zakharov Function	No
F_4	Rosenbrock's Function	Yes
F_5	Rastrigin's Function	No
F_6	Schaffer's F7 Function	Yes
F_7	Lunacek Bi-Rastrigin's Function	Yes
F_8	Non-Continuous Rastrigin's Function	Yes
F_9	Levy Function	No
F_{10}	Schwefel's Function	Yes
F_{11}	Hybrid Function 1	Yes
F_{12}	Hybrid Function 2	Yes
F_{13}	Hybrid Function 3	Yes
F_{14}	Hybrid Function 4	Yes
F_{15}	Hybrid Function 5	No
F_{16}	Hybrid Function 6	Yes
F_{17}	Hybrid Function 7	No
F_{18}	Hybrid Function 8	Yes
F_{19}	Hybrid Function 9	No
F_{20}	Hybrid Function 10	Yes
F_{21}	Composition Function 1	Yes
F_{22}	Composition Function 2	Yes
F_{23}	Composition Function 3	Yes
F_{24}	Composition Function 4	Yes
F_{25}	Composition Function 5	No
F_{26}	Composition Function 6	Yes
F_{27}	Composition Function 7	No
F_{28}	Composition Function 8	Yes
F_{29}	Composition Function 10	Yes
F_{30}	Composition Function 9	No

Bibliography

[1] Marcin Andrychowicz, Anton Raichuk, Piotr Stańczyk, Manu Orsini, Sertan Girgin, Raphaël Marinier, Leonard Hussenot, Matthieu Geist, Olivier Pietquin, Marcin Michalski, et al. What matters for on-policy deep actor-critic methods? a large-scale study. In *International conference on learning representations*, 2020.

[2] N. H. Awad, M. Z. Ali, P. N. Suganthan, J. J. Liang, and B. Y. Qu. Problem Definitions and Evaluation Criteria for the CEC 2017 Special Session and Competition on Single Objective Bound Constrained Real-Parameter Numerical Optimization. Technical report, Nanyang Technological University, Singapore, 2016.

[3] Richard Bellman. Dynamic programming. *Science*, 153(3731):34–37, 1966.

[4] Mohammad Reza Bonyadi. A Theoretical Guideline for Designing an Effective Adaptive Particle Swarm. *IEEE Transactions on Evolutionary Computation*, 24(1):57–68, 2020.

[5] Mohammad Reza Bonyadi and Zbigniew Michalewicz. Particle swarm optimization for single objective continuous space problems: A review. *Evolutionary Computation*, 25(1):1–54, 2017.

[6] Christopher W. Cleghorn and Andries Engelbrecht. Particle swarm optimizer: The impact of unstable particles on performance. *2016 IEEE Symposium Series on Computational Intelligence, SSCI 2016*, 2017.

[7] Christopher W. Cleghorn and Andries P. Engelbrecht. Particle swarm stability: a theoretical extension using the non-stagnate distribution assumption. *Swarm Intelligence*, 12(1):1–22, 2018.

[8] Christopher W. Cleghorn and Belinda Stapelberg. Particle swarm optimization: stability analysis using n-informers under arbitrary coefficient distributions. *Swarm and Evolutionary Computation*, 71:101060, 2022.

[9] Maurice Clerc and James Kennedy. The particle swarm-explosion, stability, and convergence in a multidimensional complex space. *IEEE Transactions on Evolutionary Computation*, 6(1):58–73, 2002.

[10] Ralph D'Agostino and Egon S Pearson. Tests for departure from normality. *Biometrika*, 60(3):613–622, 1973.

[11] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 12(7), 2011.

[12] R. Eberhart and J. Kennedy. A new optimizer using particle swarm theory. In *Sixth International Symposium on Micro Machine and Human Science*, pages 39–43, Nagoya, Japan, 1995. IEEE.

[13] Russ C Eberhart and Yuhui Shi. Comparing inertia weights and constriction factors in particle swarm optimization. In *2000 IEEE Congress on Evolutionary Computation*, volume 1, pages 84–88. IEEE, 2000.

[14] A.P. Engelbrecht. Particle swarm optimization: velocity initialization. In *2012 IEEE Congress on Evolutionary Computation*, pages 1–8, Brisbane, QLD, 2012.

[15] Esperanza García-Gonzalo and Juan Luis Fernández-Martínez. Convergence and stochastic stability analysis of particle swarm optimization variants with generic parameter distributions. *Applied Mathematics and Computation*, 249:286–302, 2014.

[16] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. *Deep learning*, volume 1. MIT press Cambridge, 2016.

[17] N. Hansen, A. Auger, R. Ros, O. Mersmann, T. Tušar, and D. Brockhoff. COCO: A platform for comparing continuous optimizers in a black-box setting. *Optimization Methods and Software*, 2020.

[18] K.R. Harrison, A.P. Engelbrecht, and B.M. Ombuki-Berman. The sad state of self-adaptive particle swarm optimizers. In *2016 IEEE Congress on Evolutionary Computation*, pages 431–439, Vancouver, BC, 2016.

[19] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016.

[20] Ashley Hill, Antonin Raffin, Maximilian Ernestus, Adam Gleave, Anssi Kanervisto, Rene Traore, Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, and Yuhuai Wu. Stable baselines. `https://github.com/hill-a/stable-baselines`, 2018.

[21] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[22] Ronald A Howard. Dynamic programming and markov processes. 1960.

[23] Sham Kakade and John Langford. Approximately optimal approximate reinforcement learning. In *19th International Conference on Machine Learning*. Citeseer, 2002.

[24] G. Karafotias, A. E. Eiben, and M. Hoogendoorn. Generic parameter control with reinforcement learning. *2014 Genetic and Evolutionary Computation Conference*, 2014.

[25] J. Kennedy and R. Eberhert. Particle swarm optimization. In *International Conference on Neural Networks*, pages 1942–1948, Perth, WA, Australia, 1995.

[26] James Kennedy. Small worlds and mega-minds: effects of neighborhood topology on particle swarm performance. In *1999 IEEE Congress on Evolutionary Computation*, volume 3, pages 1931–1938. IEEE, 1999.

[27] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. dec 2014.

[28] Zhengwei Li and Guojun Tan. A self-adaptive mutation-particle swarm optimization algorithm. *4th International Conference on Natural Computation*, 1:30–34, 2008.

[29] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[30] Y. Liu, H. Lu, S. Cheng, and Y. Shi. An Adaptive Online Parameter Control Algorithm for Particle Swarm Optimization Based on Reinforcement Learning. In *2019 IEEE Congress on Evolutionary Computation*, pages 815–822, Wellington, New Zealand, 2019.

[31] Henry B Mann and Donald R Whitney. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics*, pages 50–60, 1947.

[32] Peter Marbach and John N Tsitsiklis. Simulation-based optimization of markov reward processes. *IEEE Transactions on Automatic Control*, 46(2):191–209, 2001.

[33] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International Conference on Machine Learning*, pages 1928–1937. PMLR, 2016.

[34] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint*, 2013.

[35] Marco S. Nobile, Paolo Cazzaniga, Daniela Besozzi, Riccardo Colombo, Giancarlo Mauri, and Gabriella Pasi. Fuzzy Self-Tuning PSO: A settings-free algorithm for global optimization. *Swarm and Evolutionary Computation*, 39(September 2017):70–85, 2018.

[36] E. T. Oldewage, A. P. Engelbrecht, and C. W. Cleghorn. Degrees of stochasticity in particle swarm optimization. *Swarm Intelligence*, 13(3-4):193–215, 2019.

[37] Karl Pearson. Note on regression and inheritance in the case of two parents. *The Royal Society of London*, 58(347-352):240–242, 1895.

[38] Adam P. Piotrowski, Jaroslaw J. Napiorkowski, and Agnieszka E. Piotrowska. Population size in Particle Swarm Optimization. *Swarm and Evolutionary Computation*, 58(March 2019):100718, 2020.

[39] Riccardo Poli. Mean and variance of the sampling distribution of particle swarm optimizers during stagnation. *IEEE Transactions on Evolutionary Computation*, 13(4):712–721, 2009.

[40] Asanga Ratnaweera, Saman K. Halgamuge, and Harry C. Watson. Self-organizing hierarchical particle swarm optimizer with time-varying acceleration coefficients. *IEEE Transactions on Evolutionary Computation*, 8(3):240–255, 2004.

[41] Gavin A Rummery and Mahesan Niranjan. *On-line Q-learning using connectionist systems*, volume 37. University of Cambridge, Department of Engineering Cambridge, UK, 1994.

[42] Hussein Samma, Chee Peng Lim, and Junita Mohamad Saleh. A new Reinforcement Learning-based Memetic Particle Swarm Optimizer. *Applied Soft Computing Journal*, 43:276–297, 2016.

[43] John Schulman, Sergey Levine, Philipp Moritz, Michael Jordan, and Pieter Abbeel. Trust region policy optimization. *31st International Conference on Machine Learning*, 37:1889–1897, 2015.

[44] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*, 2016.

[45] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[46] Mudita Sharma, Manuel López-Ibáñez, Alexandros Komninos, and Dimitar Kazakov. Deep reinforcement learning based parameter control in differential evolution. *2019 Genetic and Evolutionary Computation Conference*, pages 709–717, 2019.

[47] Y. Shi and R. Eberhart. A modified particle swarm optimizer. In *1998 IEEE International Conference on Evolutionary Computation*, pages 69–73, Anchorage, AK, USA, 1998.

[48] Y. Shi and R. Eberhart. Empirical study of particle swarm optimization. In *1999 Congress on Evolutionary Computation*, volume 3, pages 1945–1950, 1999.

[49] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. In *International Conference on Machine Learning*, pages 387–395. PMLR, 2014.

[50] Satinder P Singh and Richard S Sutton. Reinforcement learning with replacing eligibility traces. *Machine learning*, 22(1):123–158, 1996.

[51] Richard S Sutton. Learning to predict by the methods of temporal differences. *Machine learning*, 3(1):9–44, 1988.

[52] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[53] Ioan Cristian Trelea. The particle swarm optimization algorithm: convergence analysis and parameter selection. *Information processing letters*, 85(6):317–325, 2003.

[54] Frans Van den Bergh and Andries Petrus Engelbrecht. A study of particle swarm optimization particle trajectories. *Information sciences*, 176(8):937–971, 2006.

[55] E.T. Van Zyl and A.P. Engelbrecht. Comparison of self-adaptive particle swarm optimizers. In *2014 IEEE Symposium on Swarm Intelligence*, pages 1–9, Orlando, FL, 2014.

[56] Christopher John Cornish Hellaby Watkins. Learning from delayed rewards. 1989.

[57] BP Welford. Note on a method for calculating corrected sums of squares and products. *Technometrics*, 4(3):419–420, 1962.

[58] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

[59] Gang Xu. An adaptive parameter tuning of particle swarm optimization algorithm. *Applied Mathematics and Computation*, 219(9):4560–4569, 2013.