University of Bergen
Department of Informatics

SmartSwarm - A Multi-Agent Reinforcement Learning based Particle Swarm Optimization Algorithm

Author: Herman Jangsett Mostein
Supervisors: Ahmad Hemmati
Co-supervisors: Ramin Hasibi

UNIVERSITETET I BERGEN
Det matematisk-naturvitenskapelige fakultet

June, 2023

Abstract

Particle Swarm Optimization is a renowned continuous optimization method that utilizes Swarm Intelligence to find solutions to complex non-linear optimization problems efficiently. Since its proposal, many developments have been put forward to improve its capabilities by enhancing the stochastic and tunable component of the algorithm. This thesis introduces SmartSwarm, a variant of Particle Swarm Optimization that utilizes Multi-Agent Reinforcement Learning to control the velocity of a swarm of particles. This framework has the capability of incorporating domain-specific information in the optimization process, as well as adapting a self-taught velocity function. We show how this framework has the ability to discover a velocity function to maximize the performance of the algorithm.

Keywords:
Particle Swarm Optimization · PSO · Multi-Agent Reinforcement Learning
· Reinforcement Learning · Continuous Optimization · PPO · Swarm Intelligence
· Deep Learning · Machine Learning

Acknowledgements

I want to thank my competent supervisor Ahmad Hemmati, who has provided me with much guidance and help throughout the writing of this thesis. I would also like to thank my co-supervisor, Ramin Hasibi, who has helped me with both code and writing, even while he was abroad. This thesis would not have been possible without them.

I also want to express my gratitude to my friends Thorarinn S. Gunnarsson, Sigurd Roll Solberg, Halvor Helland Barndon, Eskil Hamre Isaksen, Vegard Birkenes, and Fredrik Nestvold Larsen, who has helped me by giving me their knowledgeable insights into the writing and experiments of this thesis. Their thoughtful critiques are much appreciated. In addition, I would like to thank the rest of my co-students at Machine Learning for many rich technical discussions and other uplifting, hilarious, and rewarding moments that I will never forget.

I would also like to thank my girlfriend, Vanessa Marie Haaland, for providing moral support and motivation and dealing with many lengthy expositions on technical details. Finally, my gratitude goes out to my family for cheering me on throughout the work on this thesis.

`Herman Jangsett Mostein`
Thursday 1st June, 2023

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Particle Swarm Optimization (PSO) (Kennedy and Eberhart, 1995) is a widely-used continuous optimization method that can be applied to solve complex nonlinear problems. Drawing inspiration from nature, the algorithm mimics how animals communicate and collaborate to find resources and safety. PSO leverages a swarm of particles to search a defined space, guided by an objective function, to identify the global optimum. A simple set of rules directs each particle and make them follow each other strategically to find better objective values. PSO has been deployed in many applications such as training machine learning algorithms (Phong et al., 2022), finding optimal positions of wind turbines (Song et al., 2018), diagnosing Alzheimer's disease (Zeng et al., 2018), and more (Gad, 2022).

PSO is an example of Swarm Intelligence (SI). SI is a field of artificial intelligence in which we use numerous entities that perform simple actions and exploit cooperation to solve complex problems or tasks. There are many examples of SI in optimization, such as Ant Colony Optimization, Genetic Algorithms, and Bacterial Foraging Optimization Algorithm.

One of the disadvantages of PSO is that it often gets stuck in local minima when optimizing in high dimensional spaces and thus cannot find the global optima. Several versions of PSO have been developed, many of which involve dynamically changing the

parameters of the algorithm to improve the efficiency of the algorithm. One example of this is the TVAC-PSO, in which the parameters of PSO are changed during the search to find the optimal solution more effectively (Ratnaweera et al., 2004).

Reinforcement Learning (RL) is a branch of machine learning that has proved its capability in recent years through multiple exciting breakthroughs. In RL, an agent continually engages in an environment, adapting its behavior in response to a system of rewards. This differs from other types of machine learning in that we do not only want the model to perform a single action but rather to construct a sequence of actions that combine to perform a task. Examples of such tasks might be driving a car from one place to another, controlling the temperature in a closed environment by letting warm air in and out, or playing an Atari game. For most real-world applications of RL, especially in recent years, Neural Networks (NNs) have played a significant role. Neural Networks have many desirable attributes that aid RL, such as universal approximation and a robust capacity for handling high-dimensional spaces. These attributes have allowed us to attack large-scale problems that were previously impossible.

Numerous examples showcase the interaction between optimization and RL, most commonly seen in applying optimization techniques to train RL agents. Furthermore, there are also instances where RL is used to enhance optimization techniques. Some of these methods have tried to apply Reinforcement Learning to PSO by tuning the parameters of the algorithm dynamically to improve the performance of the algorithm.

In this thesis, we propose a novel variant of PSO in which we use Multi-Agent RL to determine particles' velocity directly. This distinguishes our approach from existing RL-incorporated PSO variants, where the agent indirectly influences the particles' behavior through the algorithm's parameters. This means we do not use a set velocity function in this framework but rather a continuous RL agent that observes the search process and makes decisions to improve the algorithm's performance. Furthermore, we wish to investigate whether a Multi-Agent RL algorithm can adapt to the SmartSwarm framework and learn its velocity function.

The optimization algorithm in this framework allows for incorporating domain-specific variables that may guide particles toward superior optima. Examples could be terrain features and environmental factors in placing wind turbines, risk and diversification metrics when optimizing financial portfolios, or activation functions and regularization measures

for machine learning model optimization. By enabling the RL agent to observe these domain-specific factors, it can potentially expedite the finding of optimal solutions.

Furthermore, using RL to control the velocity of particles directly allows for more flexibility in the optimization process. Traditional PSO algorithms often require manual tuning of parameters, which can be time-consuming and lead to suboptimal results. By using RL to dynamically adapt the velocity of particles based on the observed search process, we can avoid manually tuning these parameters.

Another potential advantage of our proposed method is that it has the potential to improve the interpretability of the optimization process. By observing the decisions made by the RL agent, we can gain insight into the underlying dynamics of the optimization process and identify areas for further improvement. This can lead to a better understanding of the problem being optimized and ultimately lead to more efficient solutions.

1.2 Thesis outline

Chapter 2 - Theoretical background covers the background theory and information relevant to our algorithm. This will include the basics of optimization and Particle Swarm Optimization (PSO), as well as machine learning and reinforcement learning (RL). We will go into depth about Proximal Policy Optimization (PPO) and cover the basics of Multi-Agent Reinforcement Learning (MARL).

Chapter 3 - Related work will present research related to our method, such as the various developments of PSO. We will also present work that incorporates RL in PSO.

Chapter 4 - SmartSwarm will cover the mechanisms of our proposal SmartSwarm in depth. We will cover pseudo algorithms, as well as the details of the model architecture and hyperparameters.

Chapter 5 - Experiments contains the experimental setup and the details of how our experiments have been conducted, as well as the experimental results and discussions. The algorithm has been tested on three benchmarks, and its performance will be compared to PSO.

Chapter 6 - Conclusion and future work will conclude the thesis and discuss areas for future research and opportunities related to our work.

Chapter 2

Theoretical background

2.1 Optimization

Optimization is a specialized area within the field of informatics that employs mathematical techniques to discover optimal solutions for various optimization problems. These problems involve determining the lowest or highest point within an objective function, which typically represents a specific value like cost, time, distance, or any other relevant metric. The significance of optimization extends across numerous industries, including green tech, transportation and shipping, and finance, where it plays a pivotal role in enhancing efficiency and effectiveness.

Often in the field of optimization, we come across problems that are easy to verify, but hard to solve. The famous traveling salesman problem is one example. In this problem, we seek to find a cycle in a complete graph that minimizes the total cost of that cycle. For every cycle in such a graph, it is easy to calculate what the total cost is, but hard to find out what the cheapest cycle is. There is no known algorithm that can compute this in a reasonable time scale, and it is assumed to be an impossible feat. However, there are several techniques that can find an adequate solution to the problem, without guaranteeing an optimal solution. A class of algorithms that can do this is **Meta-Heuristics**.

Meta-Heuristics are high-level frameworks for applying heuristics to solve an optimization problem. This means that these Meta-Heuristics use heuristics to solve problems, and provides a framework to use these heuristics in an effective way. There are many examples of Meta-Heuristics such as Simulated Annealing (Kirkpatrick et al., 1983), Particle

Swarm Optimization (Kennedy and Eberhart, 1995), and Tabu Search (Glover, 1989). These methods do not guarantee to find optimal solutions to optimization problems but aim to find good solutions. Often, these solutions are close to the global optimum in objective value, and sometimes they are not. This usually depends on the size of the problems. Large problems with many possible solutions are harder to search effectively, and it might be challenging to find something close to the optimum.

Meta-Heuristics are often divided into two categories, Generative Meta-Heuristics, and Perturbative Meta-Heuristics. Generative Meta-Heuristics will try to generate a solution to the problem from scratch, while a perturbative method will start with a solution that it changes iteratively over time. The latter category is the one we will mainly investigate in this thesis.

When employing a perturbative Meta-Heuristic approach, the initial solution serves as a starting point from which we iteratively modify and refine it in pursuit of an improved solution. In this search, we often refer to the concept of neighborhoods to assess the proximity or similarity between solutions. When one solution can be transformed into another by applying our heuristics, we consider them as neighbors. Conversely, if the intersection of their neighborhoods is small, they are deemed distant or far apart.

The neighborhood of a solution s can be defined as:

$$N_h(s) = \{s' \mid \text{Prob}(s' = h(s)) > 0\} \tag{2.1}$$

where h is the heuristic that is used in the search. Different heuristics have different neighborhoods for the same solution, and if we use a Meta-Heuristic that uses many heuristics, then the neighborhood is the union of the neighborhoods of each heuristic. This can be expressed mathematically as:

$$N_H(s) = \bigcup_{h \in H} N_h(s) \tag{2.2}$$

where H is the pool of heuristics used in the search.

If an algorithm starts its search with a particular solution and confines itself to a close vicinity, focusing solely on nearby solutions, it inevitably overlooks a broad spectrum of

alternative solutions. Nevertheless, there is a reasonable assurance that the algorithm will discover one of the finest solutions within this localized neighborhood.

Conversely, when an algorithm consistently ventures far away from the current solution without fine-tuning its trajectory, it may encounter a diverse range of solutions, but they might not possess the same level of quality. This balance between seeing many solutions, and finding the best ones in a neighborhood is what we call the balance between **diversification** and **intensification**.

The balance between diversification and intensification is important for creating efficient optimization algorithms. Diversification is the aforementioned large step in the search space, that finds a large variety of solutions, while intensification is the finetuning of an already found solution. We typically want to use diversification to find a nice neighborhood to search, and later intensify on this neighborhood to find the best solutions in this area. The optimal balance of diversification and intensification can be quite tricky, and there are several methods that aim to find efficient ways of balancing the two.

We often divide the field of optimization into two subfields. One is **discrete optimization**, in which we want to optimize a function with respect to discrete variables. This could mean the order of operation in a given problem or scheduling problems. The other subfield of optimization is **continuous optimization**, in which we wish to optimize some function with regard to continuous variables. This is the subfield in which this thesis will focus. In the domain of continuous optimization, we can think of most problems as discovering optima in some subspace of a scalar field in $\mathbb{R}^n$. For many, this might seem familiar from calculus, where most functions are differentiable, which makes finding an optimum easier. However, in many cases, we might not have sufficient information to solve the problem with these classical mathematical methods. This poses a challenge that requires other optimization methods. In the following subsection, we will discuss one such technique called Particle Swarm Optimization.

2.1.1 PSO - Particle Swarm Optimization

PSO is an example of an algorithm based on Swarm Intelligence (Kennedy, 2006). These types of algorithms are based on a certain degree of chaos and messiness but rely mainly on the cooperation of simple agents. The concept is based on the natural phenomena in which animals cooperate. For example, if many animals stay together in groups, they may follow each other and benefit from each other's discoveries of food and resources. If one animal finds a valuable resource, the animals that follow it will also find this, and the group stays stronger. In PSO, we want the particles to discover points in a space where a certain objective function has a minimum. We create a setting in which the particles can communicate with each other, and share information. As the particles share information, they may utilize this to find optima.

The algorithm works as follows. Start off by defining a bounded space $\mathbb{X} \subset \mathbb{R}^n$. Now we can define every point $x \in \mathbb{X}$ to represent a solution to an optimization problem with an objective function $f : \mathbb{X} \mapsto \mathbb{R}$. The intention of the algorithm is to find the solution x where $f(x) \leq f(x') \quad \forall\, x' \in \mathbb{X}$.
Now we place a set of particles $\mathbb{S}$ in the space and assign them a coordinate $x_i \in \mathbb{X}$. We keep track of which of these positions is the best position each particle has seen, and the best position of the swarm, and call them p_i and g respectively. Every particle is then assigned a velocity v_i that changes the particle's position for each timestep in the algorithm. This velocity is given by:

$$v_{i,d} \leftarrow \omega v_{i,d} + \phi_p r_p(p_{i,d} - x_{i,d}) + \phi_g r_g(g_{i,d} - x_{i,d}) \tag{2.3}$$

for particle i in dimension d where r_p, r_g are random numbers and ϕ_p, ϕ_g, ω are tuned hyperparameters. See Algorithm 1 for full pseudo code. Through this mechanism, the particles will move around the search space and see many different solutions to the problem. As they move toward the best solutions, the assumption is that they will find other low-objective solutions nearby.
Upon inspecting formula 2.3 we might notice how the velocity of one particle is influenced by the best position the particle has seen (p_i), and also the best position the swarm has found together (g).

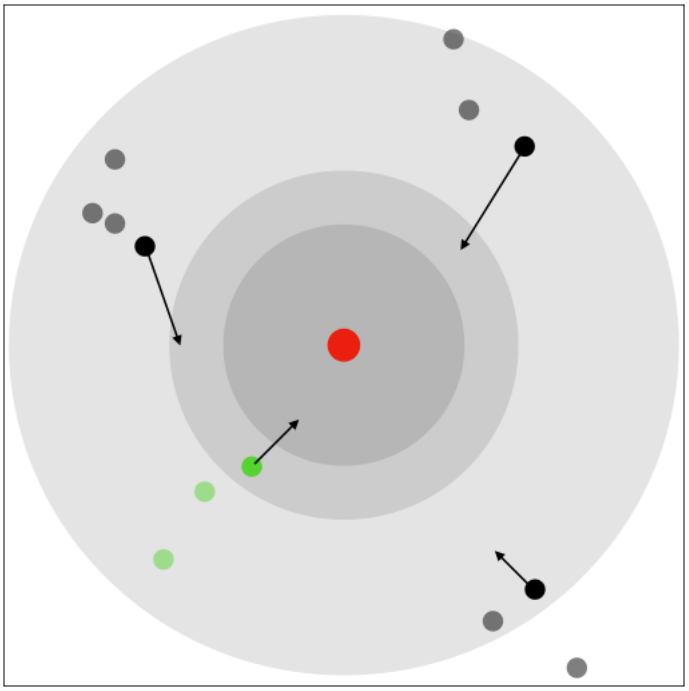


Figure 2.1: Illustration of particles in a 2D field, using PSO to find the global optimum (red dot). The green dot is the best-found solution (g), with opaque dots being historic positions.

Just like the animals described at the beginning of this section, these particles will follow each other and benefit from each other's success. Notice that the user has some control over the algorithm by tuning the hyperparameters ϕ_p, ϕ_g. The relationship and size of these parameters control how much intensification and diversification the algorithm will attain. With a high ϕ_g the particles will follow the best position more closely and thus intensify, while ϕ_p will play the same role for diversification. ϕ_p and ϕ_g are often called the cognitive and the social parameters respectively.
In our proposed algorithm, we will substitute 2.3 for a trained machine learning model that will define a new velocity function based on other features of the search process.

There have been many other versions of PSO that change the velocity function (Poli et al., 2007). These methods often involve finding better ways of controlling the balance of diversification and intensification, as well as the magnitude of the velocity vectors. The inertia weight ω in equation 2.3 was not included in the original proposal (Kennedy and Eberhart, 1995), but rather an addition to controlling the velocity of the particles (Shi and Eberhart, 1998).

Algorithm 1 Particle Swarm Optimization

for each particle i = 1, ..., S **do**
    Initialize the position of the particle: $x_i \sim U(b_l, b_u)$
    Initialize the particle's best-known position: $p_i \leftarrow x_i$
    if $f(p_i) < f(g)$ **then**
        Update the best position in the swarm: $g \leftarrow p_i$
    end if
    Initialize the velocity of the particle: $v_i \sim U(-|b_u - b_l|, |b_u - b_l|)$
end for
repeat until stopping criterion met
    for each particle i = 1, ..., S **do**
        for each dimension d = 1, ..., n **do**
            Pick random numbers: $r_p, r_g \sim U(0, 1)$
            Update velocity by: $v_{i,d} \leftarrow \omega v_{i,d} + \phi_p r_p(p_{i,d} - x_{i,d}) + \phi_g r_g(g_{i,d} - x_{i,d})$
        end for
        Update position of the particle: $x_i \leftarrow x_i + v_i$
        if $f(x_i) < f(p_i)$ **then**
            Update best-known position of the particle: $p_i \leftarrow x_i$
            if $f(p_i) < f(g)$ **then**
                Update best-known position of the swarm: $g \leftarrow p_i$
            end if
        end if
    end for
until stopping criterion met

2.1.2 Objective functions

Objective functions are the functions we want to find a minimum in when doing optimization. The objective functions usually reflect a certain value or cost that we want to minimize. In the transport sector for example, we would like to minimize the cost, the carbon footprint, the time, and many other values in our transporting efforts. Thus, we need a route or a plan that allows for low costs.

Note that we could also consider maximization when doing optimization. For example, a financial manager would like to maximize their expected return or their sales. However, maximization and minimization are mathematically equivalent. This follows from the observation that $\min_x f(x) = \max_x -f(x)$. Because of this equivalence, we will only consider minimization in this thesis, as there is no loss of generality.

For performing experiments on optimization methods, it is not uncommon to use standardized benchmark functions. These are functions that we know the global optima of, and they we can test and compare optimization methods on. Such benchmark functions can come in many shapes, such as Bowl-shaped, Valley shaped, Plate shaped, or functions with many local minima. These shapes pose different challenges for the optimization techniques, and some methods might be more suitable for certain search objective functions.

When we do optimization, we often talk about the search space or, a solution space. This is the space of all the possible solutions to our problem. In continuous optimization, this space is a subset of $\mathbb{R}^n$. So every solution to a continuous optimization can be denoted as a vector of real values. In combinatorial optimization, these solutions and their representations are typically more complex and specific to each problem.

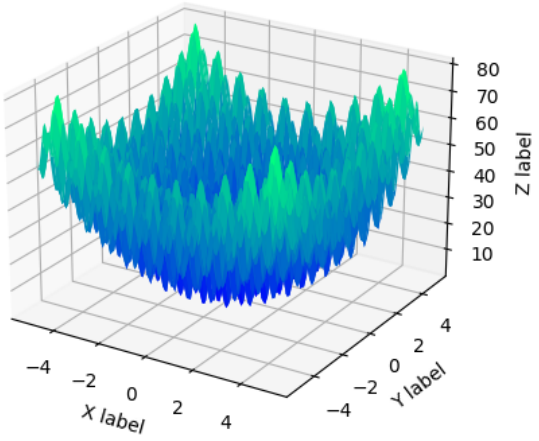


Figure 2.2: 3D Plot of the Rastrigin Function

A frequently used benchmark function is the Rastrigin function (Rastrigin, 1974):

$$f(\mathbf{x}) = An + \sum_{i=0}^{n} \left[x_i^2 + Acos(2\pi x_i)\right] \tag{2.4}$$

where n is the dimension of $\mathbf{x}$. As we can see from Figure 2.2, this function has many optima, that would typically throw off optimization methods. We often want to pick a function that can handle many different dimensional spaces, as is the case for Rastrigin. This allows us to use the same functions but at different levels of complexity. Typically, higher dimensional functions will be harder to optimize.

2.2 Reinforcement Learning

Machine learning is often divided into three subfields. Supervised Learning, Unsupervised Learning, and Reinforcement Learning. Supervised Learning is concerned with learning tasks by seeing multiple samples of labeled data and approximate functions that can match the data with its labels. Unsupervised Learning aims to process data without any labels and learn from it by finding patterns. Reinforcement Learning works in a different way. Instead of taking a single data point and giving a certain label, Reinforcement Learning wants to give a sequence of actions, that are appropriate for the observations

the agent encounters. This could, for example, be a self-driving car that encounters an obstacle on the road, and that wants to maneuver away from the dangerous situation. It could be a walking robot in a maze that wants to find its way out, or perhaps a chess-playing AI. In all of these examples, it is not sufficient for the agent to perform a single advantageous move, but rather a sequence of moves that leads to a beneficial outcome in the future.
In recent years, Reinforcement Learning has been applied in numerous areas, including well-known games like chess (Silver et al., 2017), Go (Silver et al., 2016), and Starcraft (Vinyals et al., 2019). Moreover, it has been utilized in practical applications, such as language models (Ouyang et al., 2022), self-driving cars (Jebessa et al., 2022), and healthcare (Yu et al., 2021).

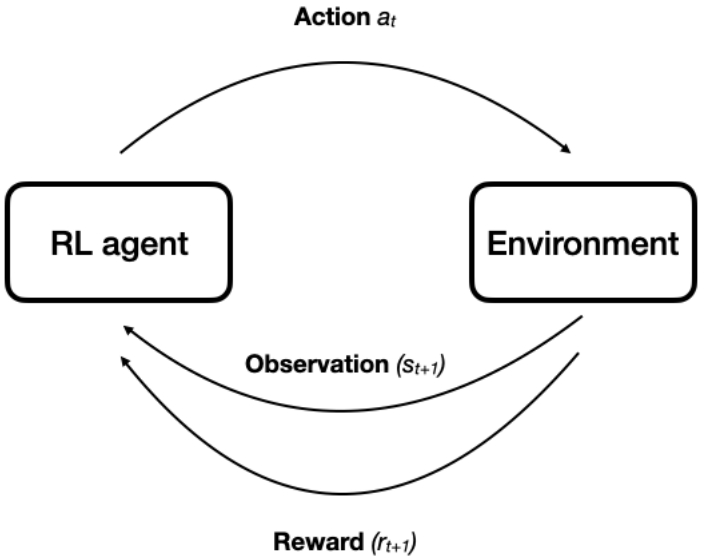


Figure 2.3: An illustration of the Reinforcement Learning training cycle. The agent receives observations ($s \in \mathcal{S}$) and a reward ($r \in \mathbb{R}$), and produces a new action ($a \in \mathcal{A}$)

The way Reinforcement Learning works is by the agent being placed in an environment. When in this environment, the agent will receive observations about the state of the environment and will perform an action that changes the state. The agent is free to try different actions in different situations and explore the space. If an agent does something desirable, we can give it a reward. This reward can be larger or smaller depending on how good the action was, and negative if the action is undesirable. In this way, the agent will learn from the rewards it is given and will start to act as we want it to.

Problems that can be solved by RL are usually modeled by a **Markov Decision Process** (MDP). The formal definition of a MDP is a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R})$. Here, the state

space $\mathcal{S}$ is a set of states. The action space $\mathcal{A}$ is a set of actions, that the agent can perform. $\mathcal{P}$ is a transition matrix, denoting the probability of one state being followed by another, given a certain action $a \in \mathcal{A}$. The reward function $\mathcal{R}$ is a map from $(s, s', a) \to \mathbb{R}$ where the s, s' is a state and the state following it, and a is the action performed for that transition. The reward function is the previously described indicator of how well the agent does. (Sutton and Barto, 2018)

In an MDP, we can create a **policy** $\pi : \mathcal{S} \to \mathcal{A}$. This policy is the "actor" and determines the behavior of the agent. Policies can be deterministic, where a certain state will lead to the same action every time. They can also be probabilistic and may perform different actions in the same state, following a certain probability distribution. The agent's policy is trained to act in a way that obtains the most rewards during an episode.

For the agent to learn, we need the agent to sometimes choose actions that are not optimal. Reinforcement Learning is based on trial and error, and the only way for the agent to learn the mechanics of the environment is for the agent to try actions that it might consider suboptimal at first. Since the agent has not had the time to explore the space, its evaluation of a certain state-action pair might be unreliable. This leads to a central element in training RL agents, exploration vs exploitation. When training, we want to explore the environment to learn how it works, but we do not want to explore all the time, because this will lead to less reward. There are several methods of balancing exploration and exploitation, and we will discuss some of these in later sections.

Reinforcement Learning agents often rely on a **value function** v_π. The value function is an estimation of the **return**. The return is defined as the sum of rewards from a certain timestep until the end of the episode. This sum is usually discounted with a factor γ. This reflects that most applications will be more concerned with a reward in the short run than the same reward in the future. Thus the return is defined as $\sum_{k=1}^{\infty} \gamma^k R_{t+k+1}$. With this definition, the value function can be defined as:

$$v_\pi(s) = \mathbb{E}_\pi[G_t \mid S_t = s] \; = \; \mathbb{E}_\pi \left[\sum_{k=1}^{\infty} \gamma^k R_{t+k+1} \;\middle|\; S_t = s \right] \tag{2.5}$$

Where G_t is the return from the timestep t, S_t is the state in the current timestep, s is the observed state, and R_t is the reward in timestep t. When a RL agent utilizes a value function, it is easier to pick an action, since the agent can find the action that will lead to more rewards.

Another way of estimating the future rewards is by using a **q-function**. This works in much the same way as the value function, but instead of estimating the value of a state, the q-function estimates the value of a state-action pair. So if the agent receives a certain observation, the agent uses the q-function to clarify what action will give the best return. This is done by estimating the q-value of each action in that given state and picking the action with the highest q-value. The q-function is given by:

$$q_\pi(s,a) = \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a] \;=\; \mathbb{E}_\pi\left[\sum_{k=1}^{\infty} \gamma^k R_{t+k+1} \;\middle|\; S_t = s, A_t = a\right] \tag{2.6}$$

We often categorize the methods of Reinforcement Learning into three categories. **Policy-based**, **Value-based** , and **Actor-Critic** algorithms.

Policy-based RL algorithms use a learned policy to perform actions instead of relying on an evaluation of the state-action pairs. This means that the policy is changed directly from training, and the agent is not concerned with estimating the value function during the learning process. One famous example of a policy-based RL method is REINFORCE (Williams, 1992). This algorithm calculates the returns of an episode and adjusts its policy directly, to make better decisions. This adjustment relies on the Policy-Gradient theorem that shall be addressed in the next subsection.
Policy-based RL has the advantage of directly optimizing the policy, instead of having to go the long way of updating a value function. The disadvantage of these algorithms is that you usually have to go through an entire episode before updating parameters, thus increasing the variance during training.

Value-based RL is mainly leaning on a value function to perform actions. This is typically done by tuning the value function and having a policy that simply picks the action that the value function recommends. One example of this is SARSA (Rummery and Niranjan, 1994). This algorithm uses a q-function to estimate the values of taking a certain action in a certain state and adjusts this estimate based on the value of the following state and action. While value-based methods can help reduce the variance problems with policy-based methods, they can suffer from some instability since we are not acting directly on the policy, but rather indirectly optimizing it.

Actor-Critic methods are a mixture of policy-based and value-based RL. In these methods, we use a parameterized policy that we do updates on, while we also use an estimated value function that assists the policy in performing parameter updates and learning. These methods remove some of the problems with using only a policy by having a policy that relies heavily on a value function. Proximal Policy Optimization is one example of an Actor-Critic method, that we will discuss in detail in Section 2.2.1.

2.2.1 DRL - Deep Reinforcement Learning

Deep Reinforcement Learning is a subfield of Reinforcement Learning in which we use neural networks to create efficient RL methods. Many applications of Reinforcement Learning have obstacles that are hard to overcome with traditional RL methods but can be drastically improved by the use of neural networks. The most common of these obstacles is when the state space is too large to handle with tabular methods. This is especially the case for continuous state spaces, where it is impossible to assign a value or an action to each state in a table. With neural networks, we can take a vectorized representation of the state as an input to the network, and have it approximate a function that returns an action or a value.

ANN - Artificial neural networks

Deep neural networks are function approximators that apply a sequence of non-linear operations on a vectorized input and are trained to produce a certain output. Each of these operations is called a layer in the neural network and can be written as:

$$\mathbf{h}^{(n)} = g^{(n)} \left(\mathbf{W}^{(n)}\mathbf{h}^{(n-1)} + \mathbf{b}^{(n)}\right) \tag{2.7}$$

where $\mathbf{h}^{(n-1)}$ is the output from the previous layer and $\mathbf{h}^{(0)}$ is the input vector $\mathbf{x}$. $\mathbf{W}^{(n)}$ and $\mathbf{b}^{(n)}$ are called the weights and the biases of the layer n in the neural network. They are trained iteratively to improve the performance of the network. The function g^n is the activation function of the layer n in the network. This is a non-linear function, that is meant to improve the capacity of the network, or the ability to approximate a larger class of functions. There are many activation functions, but some of the most popular are

the sigmoid function $\sigma = \frac{1}{1+e^{-x}}$, the ReLU $f(x) = max(0, x)$, and the hyperbolic tangent function $f(x) = \tanh(x)$. Note that without the activation functions, a composition of the linear operation $\mathbf{Wx} + \mathbf{b}$ will itself be a linear function, and will not be an efficient function approximator for non-linear relations.

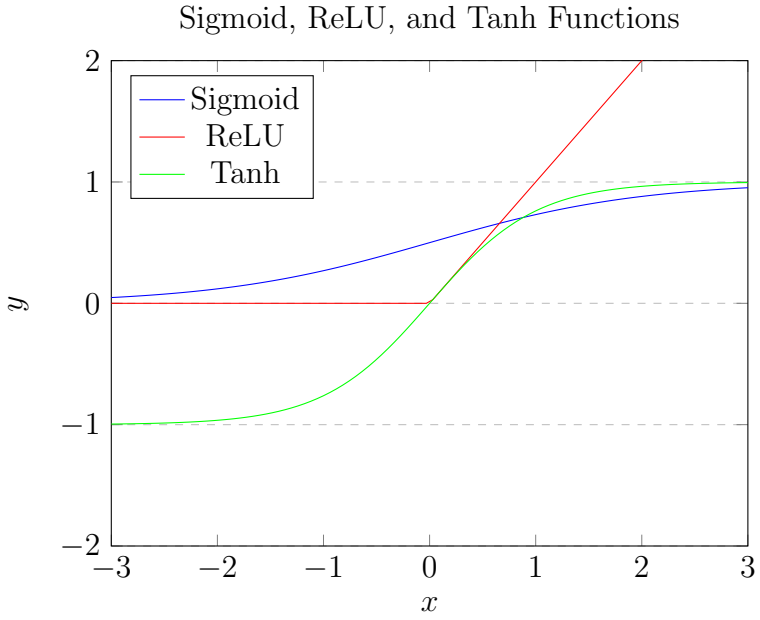


Figure 2.4: Plot of the Sigmoid, ReLU, and Tanh functions.

For computing an output we will apply 2.7 to the input $\mathbf{x}$. This result is considered the output from the first layer in the neural network. We will then apply equation 2.7 to that result, and continue this iterative process n times, one for each layer of the network. The final $\mathbf{h}^n$ will be the final output. Depending on the task at hand, one can consider $\mathbf{h}^n$ itself to be the output from the network, or one can consider the index of the largest value in $\mathbf{h}^n$ to represent a class that the network outputs. The process of iteratively applying 2.7 is called **forward-propagation**.

For the neural network to shape the output to match our task, we assign it a **loss function**. This function is meant as an indicator of how far the neural network is from performing in an optimal way. If the loss is high, that means that the neural network is not performing close to optimal on the task. During training, we optimize the loss function with regards to its weights $\mathbf{W}^{(n)}$ and $\mathbf{b}^{(n)}$. This way, the neural network will gradually improve its performance and the loss will decrease. The loss function can be constructed in many different ways, and we will return to the relevant loss functions for

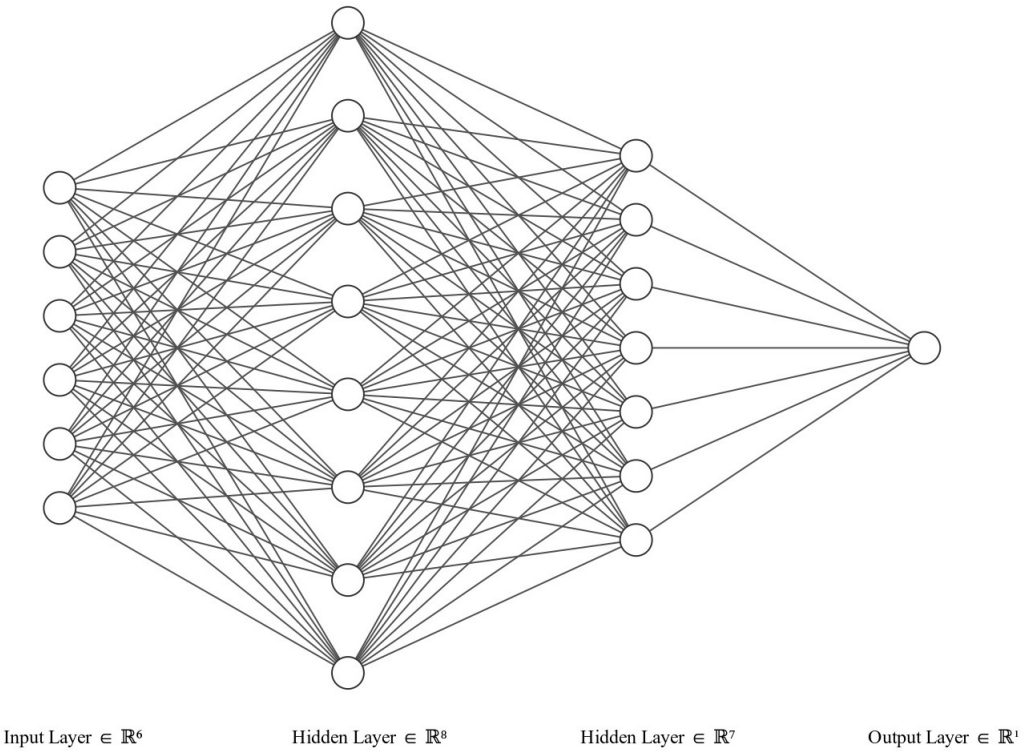


Figure 2.5: An illustration of a neural network with 4 layers.

this thesis in a later section. The optimization of the loss function is usually done by a gradient-based optimization method. This is because they have shown high performance, and take advantage of the fact that the loss function often is differentiable with regard to the weights and biases. There are many gradient-based optimization algorithms, but the most frequently used is a variant of the aforementioned Stochastic Gradient Descent (SDG) algorithm. The act of computing the gradient of the network parameters and updating the weights based on these is called **back-propagation**. The equation for SDG is

$$\theta_{t+1} \;=\; \theta_t - \alpha \nabla \hat{J}(\theta_t) \tag{2.8}$$

where θ is the parameters of the neural network, α is a learning rate, and J is a cost function that we wish to minimize. Usually, this cost function is a sum of loss functions, for multiple forward propagations.

Policy Gradient Theorem

Since neural networks allow us to predict high-dimensional and complicated, non-linear relationships accurately, they are precious to the modern Reinforcement Learning world. In practice, they can be used in several ways, but the most common is by having a neural network function as a value function, an action-value function, or as the policy itself. For Actor-Critic methods, we usually use neural networks for both the policy and the value function. It is not obvious how we can use backpropagation in the RL case, and especially not in the policy-based methods, but as we will see, there is a theoretical foundation for this.

One critical theorem that provides a theoretical framework for many of the Deep Reinforcement Learning applications is the policy gradient theorem. The policy gradient theorem gives the equation:

$$\nabla J(\theta) \propto \sum_{s} \mu(s) \sum_{a} q_{\pi}(s,a) \nabla \pi(a|s,\theta) \tag{2.9}$$

where μ is the distribution of state probabilities and J is some function to indicate how well the model performs. J is usually set to be $v_{\pi_\theta}(s_0)$ as this is the expected return for the entire episode. This provides a nice indication of the performance of our agent and may function similarly to an "inverse" loss function. The main takeaway from this theorem is that we can apply this to a general update function like:

$$\theta_{t+1} \;=\; \theta_t + \alpha \nabla \hat{J}(\theta_t) \tag{2.10}$$

This provides a way for us to update the policy weights with gradient-based optimization algorithms such as SGD, and will then again make Deep Reinforcement Learning easier to implement. Note that this formula is more representative of gradient ascent, and it is different than 2.8. Since J is set to be $v_{\pi_\theta}(s_0)$, the expected episodic return, we would like to maximize J.

This is important, not only for the policy-based methods but also for the Actor-Critic based methods. In these methods, we will need to update both the policy and the estimator of the value function. In fact, for updating the value function, we can use a similar update method to 2.10 but use a different J to optimize:

$$\mathbf{w}_{t+1} \;=\; \mathbf{w}_t - \alpha \left[v_\pi(S_t) - \hat{v}(S_t, \mathbf{w}) \right] \nabla \hat{v}(S_t, \mathbf{w}_t) \tag{2.11}$$

where α is a constant learning rate, $\hat{v}$ is the approximated value function and v_π is the actual value function. Note that this update function attempts to minimize the difference between the estimated value function and the actual value function. In most cases we do not know the actual value function in a given environment, we would not need to estimate it at all if it was known. However, we can find its value in a given state by calculating the discounted return after the episode and updating the weights accordingly. We can also use bootstrapping to perform this update function.
In the next section, we will investigate a modern algorithm, called Proximal Policy Optimization.

2.2.2 PPO - Proximal Policy Optimization

Proximal Policy Optimization (Schulman et al., 2017) is an on-policy Actor-Critic learning algorithm that has become popular due to its stability and ease of training. It was developed by OpenAI and has been widely adopted for different applications, one of which is the algorithm behind the famous language model ChatGPT (Ouyang et al., 2022). PPO aims to stabilize learning by restricting how fast the model can change over time. Since PPO is an on-policy model, if the changes to the policy are too large, the agent's behavior will change drastically. This will cause the agent to observe a different part of the state space, and this way, the agent will struggle to learn. However, by using clipping, or KL-divergence, PPO can prevent this from happening.

The main source of the success of PPO comes from the construction of the loss function:

$$L_t^{CLIP+VF+S}(\theta) \;=\; \hat{\mathbb{E}}_t[L_t^{CLIP}(\theta) + c_1 L_t^{VF}(\theta) + c_2 S[\pi_t](s_t)] \tag{2.12}$$

(Schulman et al., 2017)

where θ is the model parameters, c_1, c_2 are coefficients, $S[\pi_t]$ is the statistical entropy on the policy output. Having the entropy in the loss is meant to ensure exploration. L^{VF} is the mean-squared error loss of the value function.

L^{CLIP} is a clipped policy loss defined by:

$$L^{CLIP}(\theta) \;=\; \hat{\mathbb{E}}_t[min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t] \tag{2.13}$$

where $r_t = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$ and $\hat{A}_t$ is the advantage, as calculated by a chosen advantage function. The term $clip(r_t(\theta), 1-\epsilon, 1+\epsilon)$ will limit $r_t(\theta)$ to stay within the range $(1-\epsilon, 1+\epsilon)$. This means that the relationship between the current policy output and the previous policy output is limited, thus restricting how different the behavior is in the current policy versus the old policy. Note that we also include a minimum function of the weighted advantage and the clipped term. What this effectively does, is that it can revert bad decisions. If the advantage is negative and r_t is large, that means that the current policy will pick a bad action with a high probability. This will result in poor performance, so we allow the loss function to escape the clipping and do a large reverting step from this suboptimal behavior.

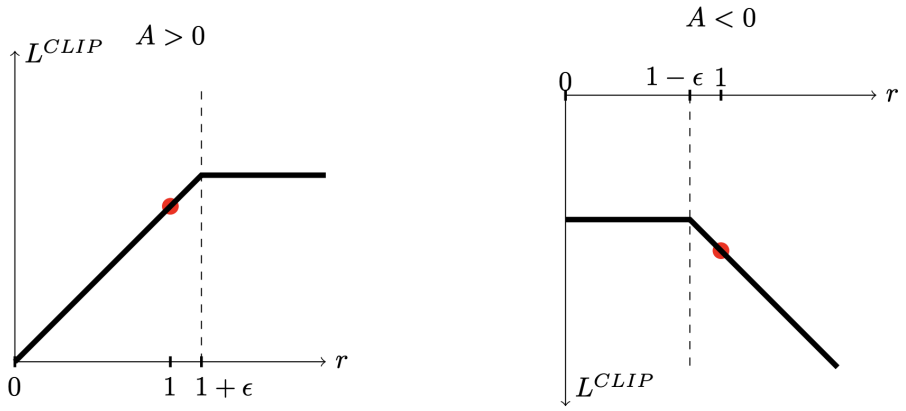


Figure 2.6: L^{CLIP} as a function of r by equation 2.13. The loss is bounded from above in both cases, but when A is negative, we allow for a larger negative loss. Figure from (Schulman et al., 2017)

Another way of restricting rapid change in the policy is by using a KL-divergence penalty. This can be achieved by replacing the L^{CLIP} term in the loss function 2.12 with

$$L^{KLPEN}(\theta) \;=\; \hat{\mathbb{E}}_t[r_t(\theta)\hat{A}_t - \beta KL(\pi_{\theta_{old}}(\cdot|s_t), \pi_\theta(\cdot|s_t))] \tag{2.14}$$

where β is a coefficient and KL is the KL divergence. β is usually adjusted by setting a target value for the KL-divergence term, decreasing β if the divergence is too low, and increasing it if not.
While this method is common and can be used, either in combination with or instead of clipping, it is not considered to be as effective as clipping (Schulman et al., 2017).

The aforementioned advantage function can be chosen in many ways. One of the simplest ways is to calculate $A(s,a) = Q(s,a) - V(s)$, where $Q(s,a)$ is the calculated q-value, and $V(s)$ is the estimated value in the current state. However, the most common advantage function for PPO is a version of generalized advantage estimation:

$$\hat{A}_t \;=\; \delta_t + (\gamma\lambda)\delta_{t+1} + ... + (\gamma\lambda)^{T-t+1}\delta_{T-1} \tag{2.15}$$

where $\delta_t = R_t + \gamma V(s_{t+1}) - V(s_t)$, in which R_t denotes the reward in timestep t, γ is the discount rate, and λ is a tuned parameter. V is the learned value estimation from our critic. The strength of this advantage function is that it looks ahead, and also uses the estimated value function, instead of the Monte Carlo method, which can often lead to unstable training.

PPO should be considered a learning framework rather than a specific learning algorithm, because of its many versions, and optional features. We can apply the framework to both continuous and discrete settings, as well as Recurrent Neural Networks (RNN), and other more concrete applications. In this thesis, the main focus will be the continuous version of PPO in which we only wish to obtain a single one-dimensional continuous value from the agent. This is done by having the policy network output a single number, that will represent a mean value for a normal distribution. This normal distribution can have a standard deviation that is a fixed constant, a trainable parameter, or an output from the actor network. When the agent acts in the environment, we will simply sample an action from the continuous normal distribution, and use this as our action. The reason why we sample from this normal distribution is to explore the search space.

Another promising feature of PPO is the fact that it can be used efficiently in Reinforcement Learning problems in which we have more than one agent acting in the environment (Yu et al., 2022). This is called Multi-Agent Reinforcement Learning and will be discussed further in the following section.

2.3 MARL - Multi-Agent Reinforcement Learning

Some Reinforcement Learning problems require more than one agent. This could for example be games in which two or more agents play against each other or other settings in which multiple agents cooperate on a common goal. These kinds of settings are often different from traditional Reinforcement learning problems because the environment is affected by other Learning agents.

We usually categorize Multi-Agent RL into three categories: fully cooperative, fully competitive, and mixed settings. The cooperative settings involve cases where we want different agents to cooperate on achieving the same thing. Often in these settings, each agent benefits from the success of others and they have an incentive to help each other. In competitive settings, we have different agents, that do not benefit from the success of the other agents. We usually model these environments as zero-sum games, in which the gain of one agent is the loss of another. These agents often have the incentive to get into the other agents' way, and only succeed by themselves. In the mixed settings, we may have elements from both the cooperative setting and the competitive settings. In this thesis, we will mainly focus on the cooperative setting, since our work is built on this principle.

For Multi-Agent systems, it is natural for us to define our problem in a different way from the single-agent problems. Thus we define the **Markov game** (Zhang et al., 2019):

Definition 2.3.1. A Markov Game is defined as a tuple: $(\mathcal{N}, \mathcal{S}, \{\mathcal{A}^i\}_{i\in\mathcal{N}}, \mathcal{P}, \{\mathcal{R}^i\}_{i\in\mathcal{N}})$ where $\mathcal{N} = \{1, ..., N\}$ is the set of $N > 1$ agents, $\mathcal{S}$ is the state space, $\mathcal{A}^i$ is the action space for agent i in the environment, and thus we also define $\mathcal{A} = \mathcal{A}^1 \times ... \times \mathcal{A}^N$.
$\mathcal{P} : \mathcal{S} \times \mathcal{A} \to \Delta(\mathcal{S})$ is the transition probability, and $\mathcal{R}^i$ is the reward function for agent i.

This definition is very similar to the MDP of the single-agent settings, except for the inclusion of the set of agents, and the allowance for an action space and reward that is different for each agent. While the problem may look similar, the inclusion of many agents will often complicate training quite extensively. This can be seen clearly when revisiting the definition of the value function, but taking many agents into account:

$$V_{\pi^i,\pi^{-i}} \;=\; \mathbb{E}\left[\sum_{t\geq 0}\gamma^t R^i(s_t,a_t,s_{t+1}) \;\middle|\; a_t^i \sim \pi^i(\cdot|s_t), s_0 = s\right] \tag{2.16}$$

Where $-i$ denotes every index except for i, and $a_t \in \mathcal{A}$ is the joined action of all agents. When there are several agents in the environment, each agent has to take the other agents' behavior into account to maximize its own reward. As every agent is training and improving, the dynamics of the environment will also change, thus making the agent's knowledge obsolete. This makes the training unstable.

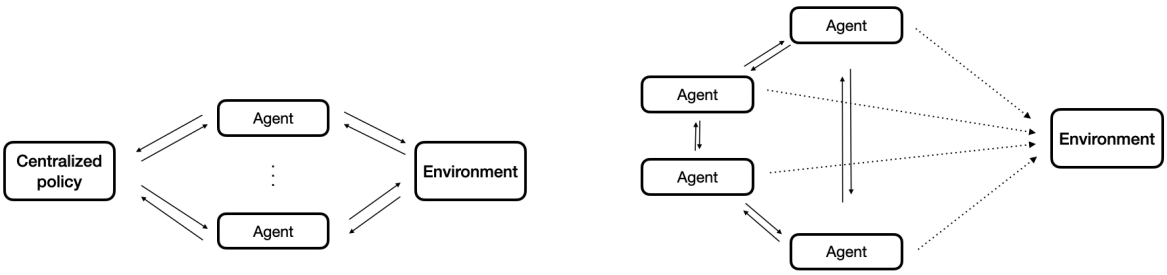
Because of the aforementioned interdependence of the agents, we often have to take game theory into account. We would like the agents to find an optimal policy that lives in equilibrium with the policies of the other agents. One concept that may help us here is the **Nash equilibrium** Zhang et al. (2019):

Definition 2.3.2. A Nash Equilibrium of a Markov game $(\mathcal{N},\mathcal{S},\{\mathcal{A}^i\}_{i\in\mathcal{N}},\mathcal{P},\{\mathcal{R}^i\}_{i\in\mathcal{N}})$ is a joint policy $\pi^* = (\pi^{1,*},...,\pi^{N,*})$ such that for any $s \in \mathcal{S}$ and $i \in \mathcal{N}$

$$V^i_{\pi^{i,*},\pi^{-i,*}}(s) \geq V^i_{\pi^i,\pi^{-i,*}}(s),\ \forall \pi^i \tag{2.17}$$

This concept illustrates a situation in which none of the agents are incentivized to change their policy. Note that this does not necessarily guarantee that every agent will receive its maximum reward, because the reward depends on the other policies. However, in a Nash equilibrium, the agents have maximized their rewards given the policy of the other agents.

Another important distinction in Multi-Agent Reinforcement Learning is the communication framework. Often, we would like for our agents to be able to communicate with each other. This is especially true in cooperative settings. This can be achieved in different ways.
In **centralized settings** each agent is controlled by a single controller. This would



(a) Centralized setting

(b) Decentralized setting with networked agents

Figure 2.7: Different setting for MARL. In (a) we observe how we can use a centralized policy to receive observations from the agents and distribute actions. In (b) the agents share information with each other and pick their actions by themselves.

involve a joint policy, a joint reward system, and joint observations. Because the centralized controller can access the joint observation of all agents and can control all of their behaviors, this makes it much easier to develop cooperation between agents. The practical weakness in such a scheme is that such a controller can get quite large. It would need a joint observation space $\mathcal{O} = \prod_{i \in \mathcal{N}} \mathcal{O}^i$ and a joint action space $\mathcal{A} = \prod_{i \in \mathcal{N}} \mathcal{A}^i$. For practical applications, with many agents, this can be hard to train.

Another way of doing this is to remove the joint controller and have the agents communicate with each other in a network. This can be done by including information from other agents in the observation of an agent so that it can take the other agents' information into account. This is called a **decentralized setting with networked agents**. These types of learning schemes may be more sensitive to the instability that often arises in MARL. When one agent updates its policy, its neighbor will experience a different observation than before. This will make it harder for the neighbor agent to learn, and we can sometimes see an oscillating behavior in the training process. This is a more managable learning scheme than the centralized setting. We do not have to deal with the large joint observations and action spaces, which makes learning easier. One way of dealing with these settings is to use **shared parameters**, in which every agent uses the same policy map from the observation to the action space. This simplifies implementation and training but requires a symmetric setup for each agent.

Sometimes we do not need the agents to communicate with each other at all. This is called a **decentralized setting**. This is typically the case for competitive settings, in which we do not want the contenders to have information about each other.

Chapter 3

Related work

One of the most interesting developments of the PSO algorithm is the previously mentioned work of Shi and Eberhart (1998). They introduced the inertia weight (see equation 2.3) that limits the momentum of the velocity of the particles. What they realized is that the original velocity function without the inertia weight can create a compounding effect that makes the velocities quite large if the particles move in the same direction many timesteps. By using an inertia weight $\omega < 1$, they controlled this compounding effect to a larger degree, and the particles will not end up with such a large velocity. The inertia weight can be defined in different ways, where one of the alternatives is a constant weight that is less than one. Another alternative is to use a linearly varying inertia weight defined by the following equation.

$$\omega = (\omega_1 - \omega_2) \times \frac{iter_{max} - iter}{iter_{max}} + \omega_2 \tag{3.1}$$

where ω_1, ω_2 are constants, $iter$ is the current iteration in the search, and $iter_{max}$ is the maximum number of iterations. We can consider ω_1 to be the initial inertia weight and ω_2 the final inertia weight, and ω gets linearly changed from one to the other. The thought behind this is to use a larger weight at the beginning of the search to encourage diversification and decrease it to a more intensifying, lower value towards the end. This made a large improvement to PSO and is widely applied today.

This work inspired another large breakthrough in the development of PSO, which is the work of Ratnaweera et al. (2004). They realized that the social and cognitive parameters should be adjusted to make the particles diversify at the beginning of the search and intensify later. This will help find better optima because the particles will have discovered better areas to intensify on when they have searched larger portions of the search space first. This was done by adjusting the social and cognitive parameters in a similar way as the work of Shi and Eberhart (1998) did for the inertia weight. The social parameter was increased over the search, while the cognitive parameter was decreased. This work explored two versions of their new proposal, where one of the two is a mutation scheme for the particles (MPSO-TVAC). The idea is to prevent particles from getting stuck in local optima, by perturbing their velocities if the particles are stuck. This means, that the velocity will be changed randomly in a random direction if the number of timesteps since the last global improvement gets large. This makes the particle escape the local minima, and they are free to keep exploring the search space. The other version, named HPSO-TVAC, does not include the velocity from the previous timestep in the velocity function and only uses the social and cognitive components. However, if the velocity is zero, they will reinitialize the velocity to a random value. This makes sure that the particles do not stagnate at a local minimum, and also prevents exploding velocities because of the momentum.

There are several different applications of Reinforcement Learning to PSO. Samma et al. (2016) proposed a variant of PSO in which a reinforcement learning algorithm indirectly influenced the velocity of the particles. In this algorithm, the reinforcement learning agent was tasked to pick from five actions: exploration, convergence, high jump, low jump, and fine-tuning. Each of these actions corresponds to a particular update equation for the position of the particles. Using this approach, the reinforcement learning model is not responsible for picking the actual velocity but rather the velocity function appropriate for each timestep. In this work, the authors have used a tabular Q-learning method to pick these actions.

A similar setup has been used for combinatorial optimization in the work of Kallestad et al. (2023). This reinforcement learning application had the agent pick between heuristics to apply to a particular optimization problem on a specific solution. This agent was given information about the search process that could assist the agent in picking what kind of heuristic was appropriate. In this algorithm, the agent can choose from a pool of diversifying heuristics and intensifying heuristics. While this algorithm is not a version of PSO, it has a similar use of Reinforcement Learning as the application mentioned above, as well as what will be proposed in this thesis.

In the work by Yin et al. (2023), the authors present an approach in which the parameters of PSO are adjusted by Reinforcement Learning. These parameters include the ones in the velocity function 2.3 and play a big part in the algorithm's efficiency. By tuning these parameters during the search, we can optimize the algorithm to work optimally at all times during the search. In this paper, they utilize the DDPG algorithm and input a state representation of the swarm and the search process. This state representation consists of measures of how far in the search process we currently are, as well as measures of the shape of the swarm, and how far apart the particles are in relation to each other. This is processed by an RL policy to output how to change the parameters of the PSO algorithm. The swarm is divided into five groups, where each group receives a 4-dimensional action to use for tuning the parameters. As a reward function, they give the agent a positive reward value of 1 if the swarm has found a better global solution, and -1 if not.

The work of Liu et al. (2019). demonstrates a Q-learning-based version, in which intensifying and diversifying parameters are tuned by Reinforcement Learning. They use a tabular Q-learning model, that is adjusted with their reward signal. This way, the agent has a set of possible variables, and the combinations of variables are given a value based on their performance. This is a method of finding the optimal set of variables to be used for optimization.

We have not seen any other algorithms that apply Reinforcement Learning to decide the particles' velocity in PSO directly. However, this is what we wish to investigate in this thesis.

Chapter 4

SmartSwarm

In this section, we will present the SmartSwarm algorithm, a Multi-Agent Reinforcement Learning-based version of Particle Swarm Optimization. The algorithm aims to use Reinforcement Learning to improve the velocity function of the particles in PSO, and thus improve the algorithm's performance.

4.1 The algorithm

The main difference between our algorithm and the traditional PSO algorithm is how the velocity of each particle is computed. The traditional PSO uses an equation 2.3, that utilizes information about the other agents and the particle's history to compute a velocity. The aim of our algorithm is to learn a new velocity function. This is done by having every particle be a Reinforcement Learning agent and having the RL agents learn a policy that will function as their velocity function. These agents will, as the traditional PSO algorithm, receive information about the environment and compute the velocity based on this. This information can consist of any value that will help the performance. Since every particle is an agent of its own, and they are communicating with each other, this is an example of a cooperative Multi-Agent Reinforcement Learning setting.

SmartSwarm considers one search as one episode for the RL algorithm, and one timestep is one move of the particles in a certain dimension. Thus, the agent will receive one state representation, output one velocity, and receive one reward each timestep. Since this is a Multi-Agent setting, this will happen once for each of the particles within a timestep.

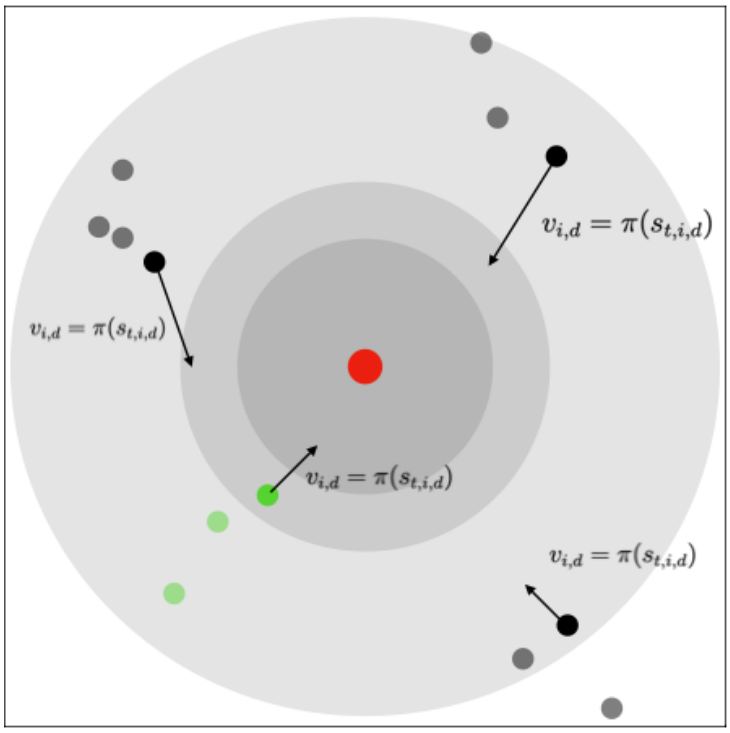


Figure 4.1: A figure of how the SmartSwarm algorithm works. The velocities of the particles are defined by a policy that all the particles use.

The SmartSwarm uses a networked agent communication structure between particles, with shared parameters. This means that every agent shares the same policy, and this policy will learn from all agents. This is to simplify the algorithm and to exploit the fact that every agent has the same goals and objectives. If every particle has its own policy, we would have a much longer and more complicated training process, and we would lose invariance to the number of particles in the algorithm.

The networked communication between particles lies in the state representation. It includes information about the other particles, such as the best-found solution, and how much better the best solution is, relative to the current particle. This provides useful information to the agent, with regard to diversification and intensification. If the agent receives information from the other agents that there are much better solutions to be found, the agent may choose to diversify. However, if the agent is in the best-found solution, it might choose to intensify. Note that this is also incorporated in the PSO velocity function (equation 2.3), but in our version, the policy is free to act in a more complex manner based on these signals. The agent may decide to keep diversifying if it does not think it is beneficial to follow the best solution found.

Algorithm 2 SmartSwarm

for each particle i = 1, ..., S **do**
  Initialize the position of the particle: $x_i \sim U(b_l, b_u)$
  Initialize the particle's best-known position: $p_i \leftarrow x_i$
  if $f(p_i) < f(g)$ **then**
    Update the best position in the swarm: $g \leftarrow p_i$
  end if
  Initialize the velocity of the particle: $v_i \sim U(-|b_u - b_l|, |b_u - b_l|)$
end for
while a termination criterion is not met **do**
  for each particle i = 1, ..., S **do**
    for each dimension d=1, ..., n **do**
      Define current state: $s_{t,i,d}$
      Update velocity with RL agent $v_{i,d} = \pi(s_{t,i,d})$
    end for
    Update position of the particle: $x_i \leftarrow x_i + v_i$
    if $f(x_i) < f(p_i)$ **then**
      Update best-known position of the particle: $p_i \leftarrow x_i$
      if $f(p_i) < f(g_i)$ **then**
        Update best-known position of the swarm: $g \leftarrow p_i$
      end if
    end if
  end for
end while

4.2 The agent

The Reinforcement Learning framework used to train the SmartSwarm policy is the Proximal Policy Optimization framework as described in section 2.2.1. We apply a continuous version of PPO where the output of the actor is a one-dimensional scalar value, that is further applied as the mean of a normal distribution the action is drawn from. This section will cover the details behind this RL agent.

4.2.1 The state representation

The state representation is meant to convey as much information about the environment as possible to the agent. This information should allow the agent to make qualified decisions and allow the agents to understand their position in the search progress. It includes features such as best found solution, current solution, or how far we have come in the search process. This allows the agent to make more strategic choices since it can understand easily when to intensify or when to diversify.
As in the traditional PSO algorithm, our agent should receive information about the other particles, to get an understanding of the function space, and where to go. This may however cause instability since it is impossible for the agent to predict the value of these features in the coming timesteps. Because of this, we have limited the number of such features.

Feature	**Description**
dist_to_global_best	The relative distance from the particle to the global best-found solution in this dimension.
dist_to_particle_best	The relative distance from the particle to the particle's best-found solution in this dimension.
v	The current velocity of this particle in this dimension
dim	The current dimension
ratio	The ratio between the objective in this solution and the initial best-found solution.
best_ratio	The ratio between the objective in the global best-found solution and the initial best-found solution
iter	Share of iterations performed

Table 4.1: State features and their descriptions

For ease of training, the state will be normalized prior to its input into the agent. This is to avoid the instability that follows with outliers, and large changes in the state. The states are normalized by

$$X_i = \frac{X_i - \overline{X}}{\sigma} \tag{4.1}$$

where X_i is an observation, $\overline{X_i}$ is the mean of observations, and σ is their standard deviation.

4.2.2 Action

In our version of this algorithm, we have chosen to only output the velocity in a single dimension for each particle. The reason for this is that one trained agent can be used in problems with different dimensionality than what it was trained on. This does however assume some level of independence of the function in the different dimensions. Neural networks have the ability to output a high dimensional velocity, but this requires a fixed dimension in the problems to which the agent is applied. It is possible to apply a Recurrent Neural Network to solve this issue, but that is beyond the scope of this thesis.

For our agent to be able to optimize any objective function with any boundary restrictions, it is important to normalize the outputs of the agent with the size of the space the particles may move in. Because of this, we include a *tanh* function as the particles are passed to the environment. This could be done in the model architecture, but this may cause some gradient vanishing in the neural network.

As previously described, the action is sampled from a normal distribution created by the output of the actor network. This works by having the PPO agent output a value that is considered the mean of the distribution. The variance of this distribution is a learned parameter, that is not dependent on the state of the input. Then, the action that is used in the environment is drawn randomly from this distribution.
This is done to ensure exploration during the training process. Usually, since the models will get better over time during training, the model will learn to decrease the variance of the distribution through the learnable parameter. This means that the normal distribution usually will become sharper during training, and the model will rely more on the mean output. It is not uncommon to simply use the outputted mean as the action during

validation.
In our case, it is not necessarily a bad thing to include the normal distribution also in the evaluation. As randomness plays a role in PSO, this could ensure the diversification of our particles in our algorithm, and we will therefore keep it. The standard deviation is a trainable parameter in our agent, so it will naturally be decreased as the agent learns. If we were to remove it completely, this would be an obstructing change to the policy, as the agent has adapted its policy to the normal distribution during training.

To make sure the action is sized proportionally to the space in which the particles move, we need to scale up the action after it is normalized. This is done by using the previously mentioned *tanh* function and then scaling with the diameter of the space.

$$\begin{aligned} v = & \quad \frac{\text{diam}(S)}{\beta} \cdot \tanh(a) \\ = & \quad \frac{\sqrt{d}}{\beta} \cdot (x_{\max} - x_{\min}) \cdot \tanh(a) \end{aligned}$$

where S is the search space, a is the action from the agents, d is the dimension of the search space, and x_{max}, x_{min} is the boundaries of the search space. β is the least number of steps we want the agent to use to get across the search space. If it is set low, the particles will move slowly, and if set high, they will move fast.

4.2.3 Reward function

For the agent to learn to perform the search effectively, it is essential for the agent to receive a reward signal it can understand. The reward function is perhaps the most important part of RL development. We need a reward function that reflects the final goal we want the agent to achieve, while also giving the agent enough information along the way for it to find a high-functioning policy. We give the agent a reward based on the objective value of the current solution that the particle has found. This will eventually make the particle search for the best possible solution because it will receive a higher reward for doing so.

One weakness of this reward function is that the particles spawn randomly, and some particles might get lucky and spawn in a low-objective location. This will cause the agent to receive a high reward without doing any work to achieve it. This can cause instability in learning because the agent gets high rewards without having learned a good policy. To fight this problem, we scale the objective by the initial objective, as seen in equation 4.2. To give more rewards for very small objective values, we use a negative logarithm. The reward function is given as

$$R_i^t = -\log \frac{f(x_i^t)}{\overline{f(x^0)}} \tag{4.2}$$

where f is the objective function, x_i^t is the position of particle i on timestep t. $\overline{f(x^0)}$ represents the average objective value of all the particles at the beginning of the search. This value is used, because it will be more stable every search, than a single objective value by itself. This will give a reward only when the agent actually acts in a way that improves the solution, and not if it is lucky with its initial position. Note that this reward function will also incentivize the agent to find a nice solution as fast as possible since the RL agent wants to maximize its return and not the reward of a single timestep.

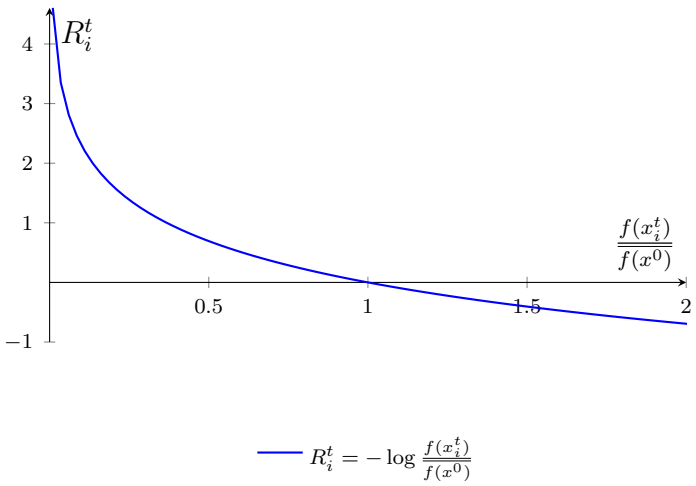


Figure 4.2: Plot of the reward function as a function of the ratio between the current solution and the initial solution.

4.2.4 Architecture and hyperparameters

PPO uses an Actor-Critic structure that applies neural networks as both the policy and the value function. Our architecture uses a neural network of two hidden layers of 64 nodes, as well as an output layer with 64 nodes. Both the actor and the critic have this architecture and an output dimension of one. We use ReLU activation functions between all layers.

Table 4.2: PPO Hyperparameters

Hyperparameter	Value
Learning rate (decreasing)	5×10^{-4}
Clip range	0.1
Value function coef.	1
Entropy coef.	0
Batch size	64
Number of steps per rollout	4096
Optimizer	Adam

4.3 Training setup

The training algorithm starts with the agent performing $n_{episodes}$ searches, and saving information such as the observations, rewards, and actions it performs. Then the agent will go back into the experience replay and update the weights of the model according to the PPO update process described in the background section.

When training the SmartSwarm algorithm, one should be aware of overfitting to objective functions. For the agent to be able to generalize to other spaces, the agent needs to train on a variety of spaces. This may include different function classes, or simply scaling or dragging the functions. This process will make the agent more robust in different objective functions, but may not be as specialized. The agent has the capability of training on different dimensional problems as well.

Algorithm 3 SmartSwarm training algorithm

Require: Hyperparameters: $n_{agents}, n_{episodes}, n_{steps}, n_{epochs}$
Initialize policies π_{θ_i} and value functions V_{ϕ_i} for $i = 1, \ldots, n_{agents}$
for *update* $\leftarrow 1...n_{updates}$ **do**
for *episode* $\leftarrow 1...n_{episodes}$ **do**
Initialize environment and get initial states s_i for all agents i
for $t \leftarrow 1, n_{steps}$ **do**
for $i \leftarrow 1, n_{agents}$ **do**
Choose action $a_i \sim \pi_{\theta_i}(a_i|s_i)$ for agent i
end for
Execute actions a_i for all agents and observe rewards r_i and next states s_i'
Compute advantage estimates and returns for current states and rewards for all agents i
Update states $s_i \leftarrow s_i'$ for all agents i
end for
Add episode information to replay log
end for
for *episode* $\leftarrow 1...n_{episodes}$ **do**
for *epoch* $\leftarrow 1, n_{epochs}$ **do**
for $i \leftarrow 1, n_{agents}$ **do**
Update policy π_{θ_i} and value function V_{ϕ_i} for agent i using PPO and the computed advantage estimates and returns
end for
end for
end for
end for

Chapter 5

Experiments

5.1 Experimental setup

5.1.1 Experimental environment

The experiments were performed on a Macbook Pro with an Apple M2 Pro chip with 16 GB memory running Ventura 13.2 operating system.

5.1.2 Baseline

Particle Swarm Optimization

The first baseline we compare our model to is the standard PSO algorithm as described in PSO - Particle Swarm Optimization. This is chosen because our algorithm aims to expand on and improve PSO.

PSO has a few parameters that will influence the performance of the algorithm. These are set as the values in Table 5.1. We will use a version of PSO where we include a constant inertia weight as proposed in the work of Shi and Eberhart (1998). Other parameters such as the number of particles, and the number of iterations of the search will be the same as described for SmartSwarm in each experiment, as these parameters have a similar influence on the two algorithms.

Table 5.1: Hyperparameters for PSO Algorithm

Hyperparameter	Value
Inertia weight	0.7
Cognitive parameter (ϕ_p)	1.5
Social parameter (ϕ_g)	2.0

Random Swarm

We will also use a random baseline, to verify that the agent is better than what it would be if it performed random actions. This works by using an action that is sampled from a uniform distribution (equation 5.1):

$$v_i^t \sim \mathcal{U}(-1, 1) \tag{5.1}$$

This action will function in the exact same way that the actions from the SmartSwarm policy does. It will also be scaled in the same way as the SmartSwarm action does. If our agent does not learn, it is expected to perform similarly to a random agent, since there is no intention behind the actions that are performed. However, if the agent does in fact learn from the environment, and makes intentional attempts to find a better optimum, it will exceed the performance of the random baseline.

This baseline has no parameters, except those related to the environment. This includes the number of agents, dimensionality, objective functions, etc. All of these will be kept the same as for PSO, and SmartSwarm, to create a realistic comparison.

5.1.3 Implementation

All experiments and algorithms are implemented with Python 3.10.11. The Reinforcement Learning model is a PPO model implemented with PyTorch (Paszke et al., 2019). The environment is developed using the Petting Zoo API. Petting Zoo (Terry et al., 2020) is a Python framework for Multi-Agent RL that is built on Gym. This drastically simplifies the implementation and allows for vectorized environments for ease of training.

5.1.4 Experiment structure

These experiments were conducted by training the model on several instances of the objective functions we experiment on. During the training of the algorithm, we translate the functions by a randomized value, to ensure that the model learns the function space and not just certain coordinates. In other words, we want to prevent overfitting by doing it this way.

Each experiment is tested by doing 100 searches for each of the three algorithms. We will evaluate the performance of these algorithms by noting the mean of the objective values of the global best-found solutions, as well as their standard deviation. The standard deviation may give an indication of the consistency of the algorithms.

In our experiments, we use a relatively small number of timesteps and low dimensionality compared to other literature. This will create a larger difference between the high-performance methods, and the low-performance methods since the low-performance will not have the time to randomly find low-objective solutions by coincidence. It will also benefit methods that can find a low-objective solution fast.

5.2 Experiments on the Rastrigin Function

The Rastrigin function is a highly noisy function with many local minima close together. It is a function on which it is hard to find the global optima because of the many distracting local optima.

The function is given by

$$f(\mathbf{x}) = An + \sum_{i=0}^{n} \left[x_i^2 + Acos(2\pi x_i) \right] \tag{5.2}$$

where n is the dimension of $\mathbf{x}$, and A is a parameter that can be tuned. We have set A to be 10.

The global minimum is in $x = [0, ..., 0]$ with objective value 0. It is restricted by $x_i \in [-5.12, 5.12]$ $\forall i$.

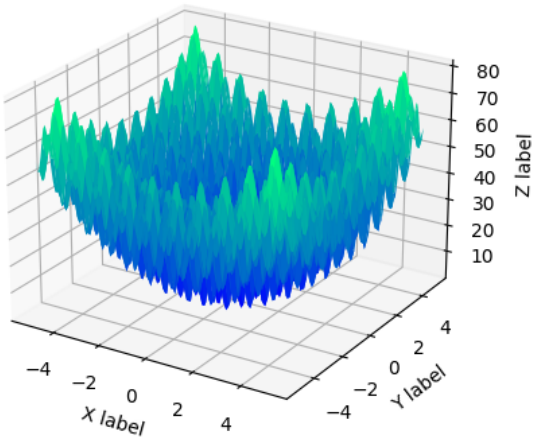


Figure 5.1: A 3D Plot of the Rastrigin Function

Table 5.2 displays the hyperparameters that are used for each of the experiments. All the parameters are the same for all of the three methods tested, except for the Initial Standard deviation, which indicates the standard deviation of the action of the SmartSwarm actor-network. The table displays three different experiments, with increasing complexity and size of the search space.

5.2.1 Experiment results

Table 5.3 shows the mean objective values and the standard deviation in the experiments conducted on the Rastrigin functions. In all cases, the SmartSwarm is outperformed by PSO, and in the larger instances, the method does not show signs of learning, since it is outperformed by the Random algorithm as well.

Table 5.2: Experiment parameters for Rastrigin function

	Parameters				
Experiment	Particles	Dimensions	Timesteps trained	Iterations	Initial std
1	50	2	35 mill	100	e^{-1}
2	100	5	50 mill	100	$e^{-0.5}$
3	100	10	100 mill	100	$e^{-0.5}$

Table 5.3: Objective values for SmartSwarm and baselines on Rastrigin

Experiment	Objective (Standard deviation)		
	SmartSwarm	PSO	Random
1	5.55×10^{-3} (7.333×10^{-3})	1.77×10^{-3} (1.99×10^{-3})	0.531 (0.441)
2	13.602 (3.296)	2.816 (1.025)	12.661 (2.835)
3	70.154 (10.674)	26.050 (4.376)	64.921 (7.365)

In Table 5.4 we can see the improvement of the objective value of the final best solution compared to the best initial solution. As can be seen from the table, the SmartSwarm algorithm is close to the performance of PSO in the smallest instance, but not on the experiments with higher dimensional spaces.

Table 5.4: Improvement values for SmartSwarm and baselines on Rastrigin

Experiment	Improvement in % (Standard deviation)		
	SmartSwarm	PSO	Random
1	99.900 (1.477×10^{-3})	99.970 (0.0414)	88.3324 (14.7240)
2	62.457 (0.136)	92.175 (3.730)	65.8658 (9.7913)
3	31.884 (0.124)	75.797 (4.967)	38.6044 (9.3050)

5.2.2 Discussion

Our experiments revealed that during the training process, the global best-found solution discovered in each episode consistently decreased in objective value. As the agents found better solutions in the searches, we saw the standard deviation decrease in experiment 1. This indicated that the agents got more confident in their ability to control the particles, and wanted less variability

This effect is depicted in Figure 5.2, which illustrates the decreasing standard deviation over time. By learning to adjust the variability as a parameter, the agent recognized the benefits of lower standard deviation, which gave it more control over the output. This finding indicated that the agent is in fact learning, as it became less reliant on randomness.

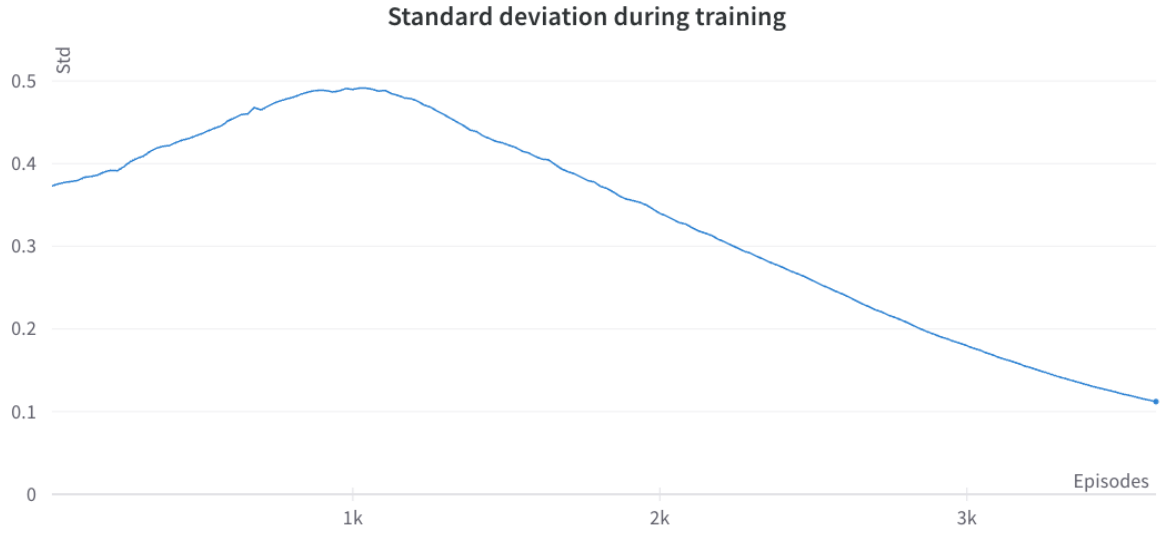


Figure 5.2: Plot showing the learned variability in the action of PPO over time on experiment 1.

Similarly, the gradual improvement in the quality of solutions discovered by the agents was also noticeable, as depicted in Figure 5.3. This graph highlights the progression of the agent and its performance improvement throughout the training process.

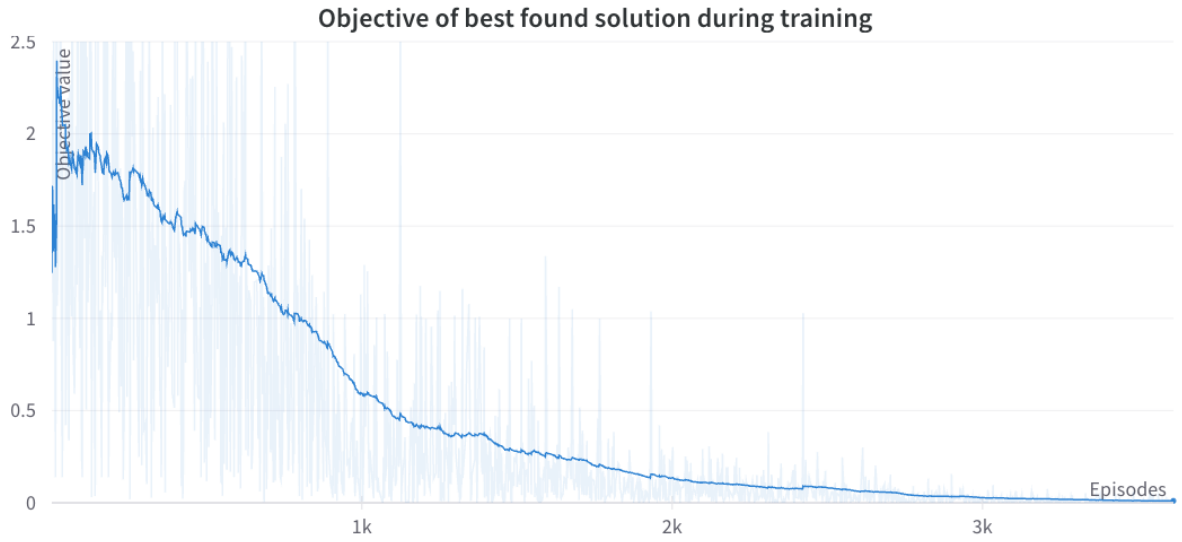


Figure 5.3: Plot illustrating the objective value of the best-found solution during the training phase of experiment 1.

As for the chosen strategy of the model, the training data indicates that the agent is aware that it should diversify at the start of an episode, and increase its intensification towards the end of an episode. As previously described, we would like the particles to disperse at the start of the episode while moving toward each other in the final stages of the search. This allows the agents to gather as much information as possible before intensifying, thus finding the best minimum to focus on. During the training process, it is apparent that the best-found solution is found later in the search when the performance of the algorithm is increased. As can be seen from 5.4, during the final stages of training, the model finds the global best solution around iteration 60 out of 100.

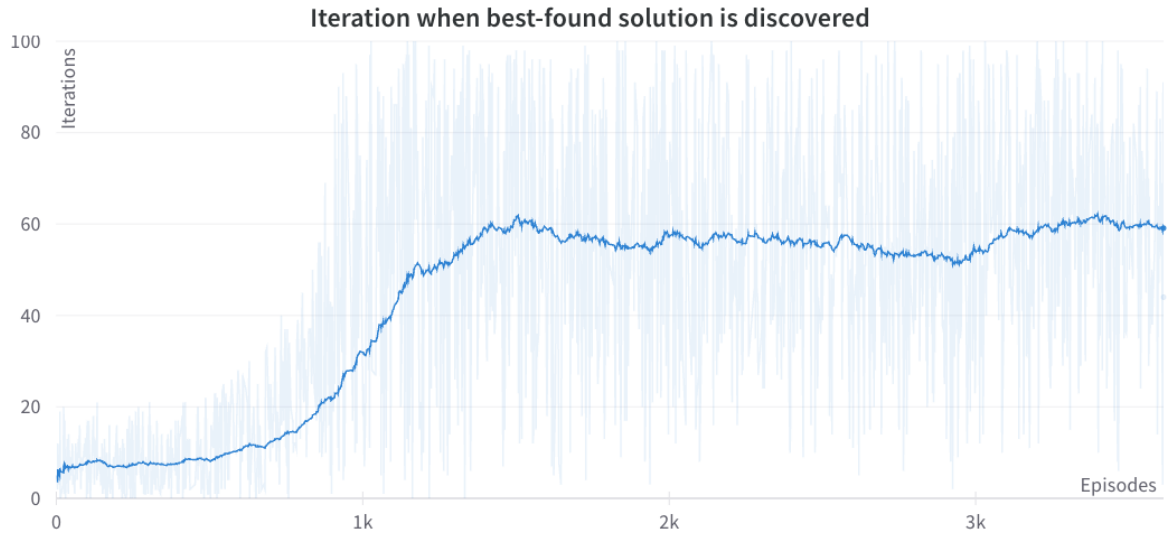


Figure 5.4: Plot illustrating the iteration of which the best-found solution is found on experiment 1. This model is training with searches lasting for 100 iterations. High values, mean that the best-found solution is found late in the search.

As we can see, the SmartSwarm algorithm learns and adapts to the problem in the smaller instance. However, it does not perform as well in the larger instances. This is likely caused by the fact that the problem is complex and hard to adapt to. As can be seen from the performance of the larger problems (figure 5.5), the agent is struggling to learn the environment dynamics and the expected returns, which causes trouble as the agents attempt to pick the optimal actions. While this might be improved with more training time and resources, it seems to be a tough learning task.

The agent seems to be able to improve its strategy slightly, but the trend breaks very early. The agent learns to improve its objective value up to a point, but then the agent cannot progress anymore. This stagnation of progress signals that the agent struggles to learn in more challenging environments, such as the high-dimensional Rastrigin function.

It is likely that the algorithm would benefit from longer training sessions than what was possible under the time constrictions of this thesis. Longer training sessions would allow the agents to explore more, and adapt to the more complex environments as well as the simpler ones.

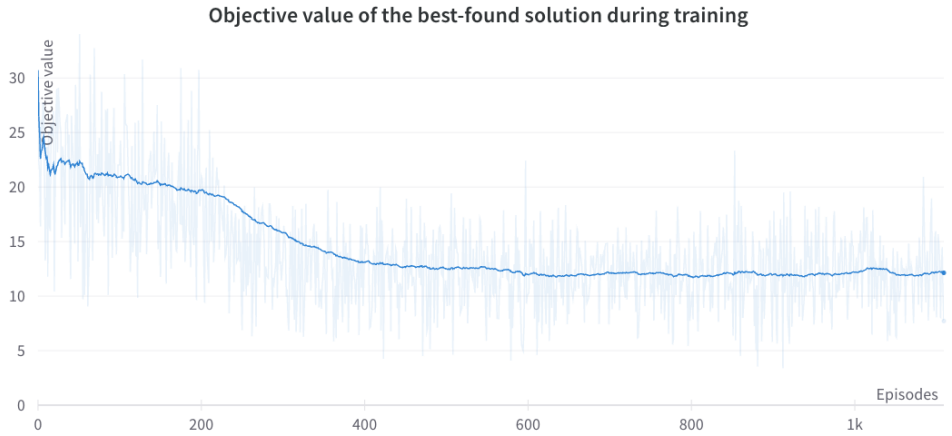


Figure 5.5: Plot illustrating the objective of the best-found solution during training of the medium-large instance (experiment 2). The objective value declines up to a point and then stagnates at approximately the same value.

During the experiments on the larger instances, we can see an opposite effect from what is observed in the smallest instance, when it comes to the standard deviation of the output. Instead of decreasing, as a result of confidence, it increases. This is likely because the agent cannot find an effective policy in the complex environment, and when the model tries to decrease its variability in the outputs, the agent receives less reward. In this situation, the agent adjusts its weights to maximize the expected reward, thus increasing the randomness.

5.3 Experiments on the Sphere Function

This is a very stable function with no noise or unpredictability. This makes it easy to find the global optimum because there are no local optima except the global one. This objective function belongs to a category of functions called Bowl-shaped functions.

$$f(\mathbf{x}) = \sum_{i=0}^{n} x_i^2 \tag{5.3}$$

The Sphere function has a global optimum in $x = [0, ..., 0]$ with an objective value of 0. It is restricted by $x_i \in [-5.12, 5.12]\ \forall i$.

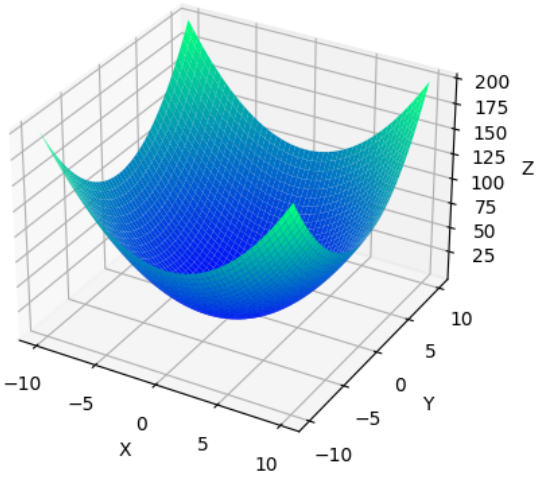


Figure 5.6: A 3D Plot of the Sphere Function

5.3.1 Experimental results

As we can see in the relative improvement in Table 5.7, the SmartSwarm has a dip in performance as we go into higher dimensions, but it is not as large as for the Rastrigin function. For the Sphere function, we can see the performance staying relatively high in both dimensions 2 and 5, and getting worse on the 10-dimensional instance.

Table 5.5: Experiment parameters for Sphere function

	Parameters				
Experiment	Particles	Dimensions	Timesteps trained	Iterations	Initial std
1	50	2	35 mill	100	e^{-1}
2	100	5	50 mill	100	$e^{-0.5}$
3	100	10	100 mill	100	$e^{-0.5}$

Table 5.6: Objective values for SmartSwarm and baselines on Sphere

Experiment	Objective (Standard deviation)		
	SmartSwarm	PSO	Random
1	2.571×10^{-4} (2.691×10^{-4})	2.447×10^{-5} (2.923×10^{-5})	0.01771 (0.0194)
2	2.178 (1.228)	0.0199 (0.0111)	4.9256 (2.9129)
3	51.719 (14.659)	0.796 (0.232)	50.431 (13.555)

Table 5.7: Improvement values for SmartSwarm and baselines on Sphere

Experiment	Improvement in % (Standard deviation)		
	SmartSwarm	PSO	Random
1	99.933 (0.00321)	99.997 (0.00899)	96.278 (11.702)
2	89.804 (0.0925)	99.877 (0.1512)	80.160 (14.670)
3	56.443 (0.157)	99.289 (0.2684)	56.029 (14.587)

5.3.2 Discussion

The results of the experiments on the Sphere function verify the claim in the previous section about complex objective functions. The SmartSwarm algorithm seems to learn much quicker and finds better optima on the Sphere function than what it did on the Rastrigin function. However, the results are still not as good as what PSO achieves. Note that the Random baseline is behind in experiments 1 and 2 while being on par with SmartSwarm in the largest instance. This indicates the same tendency as for Rastrigin that SmartSwarm struggles in higher dimensions.

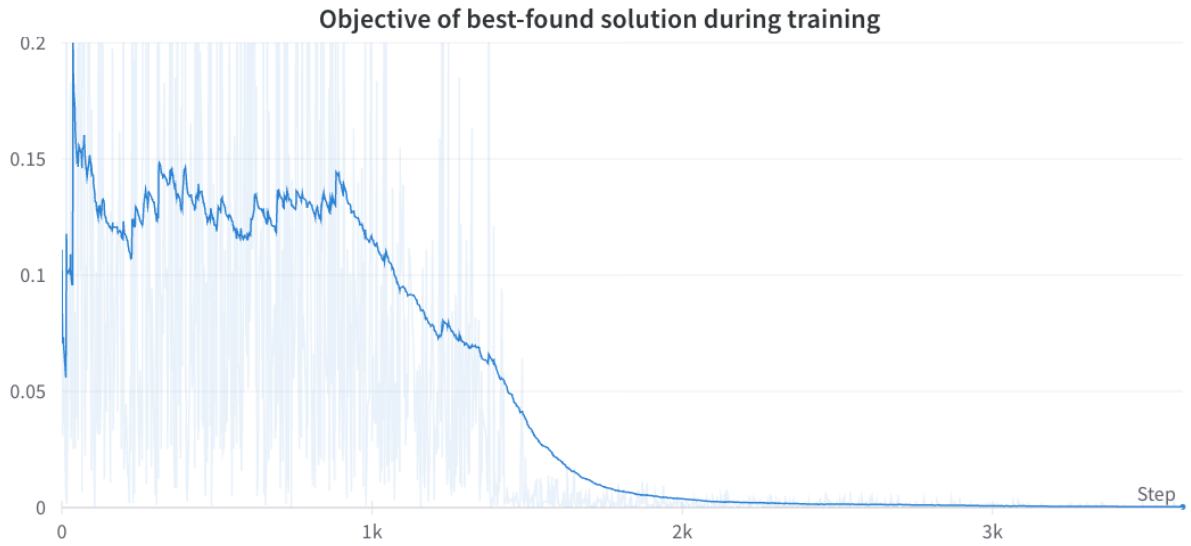


Figure 5.7: Plot of the objective of the best-found solution on the smallest instance of the sphere function.

Part of the reason that the agent can learn in experiment 2 on the Sphere function, but not in the Rastrigin functions, is that the objective values are much more predictable. For the agent to learn, it is necessary that the agent can estimate the return of the states it observes, and that is very hard to do in noisy functions. The Sphere function is much easier to approximate and estimate the value of under a certain action.

From Figure 5.8 we can interpret some of the agent's behavior in the smallest instance of the Sphere function. As we can see, in the areas where there is a large positive relative distance to the global best solution (y-axis in Figure 5.8), the agent prefers to output a larger valued action. Indicating that the agents try to move towards the global best solution.

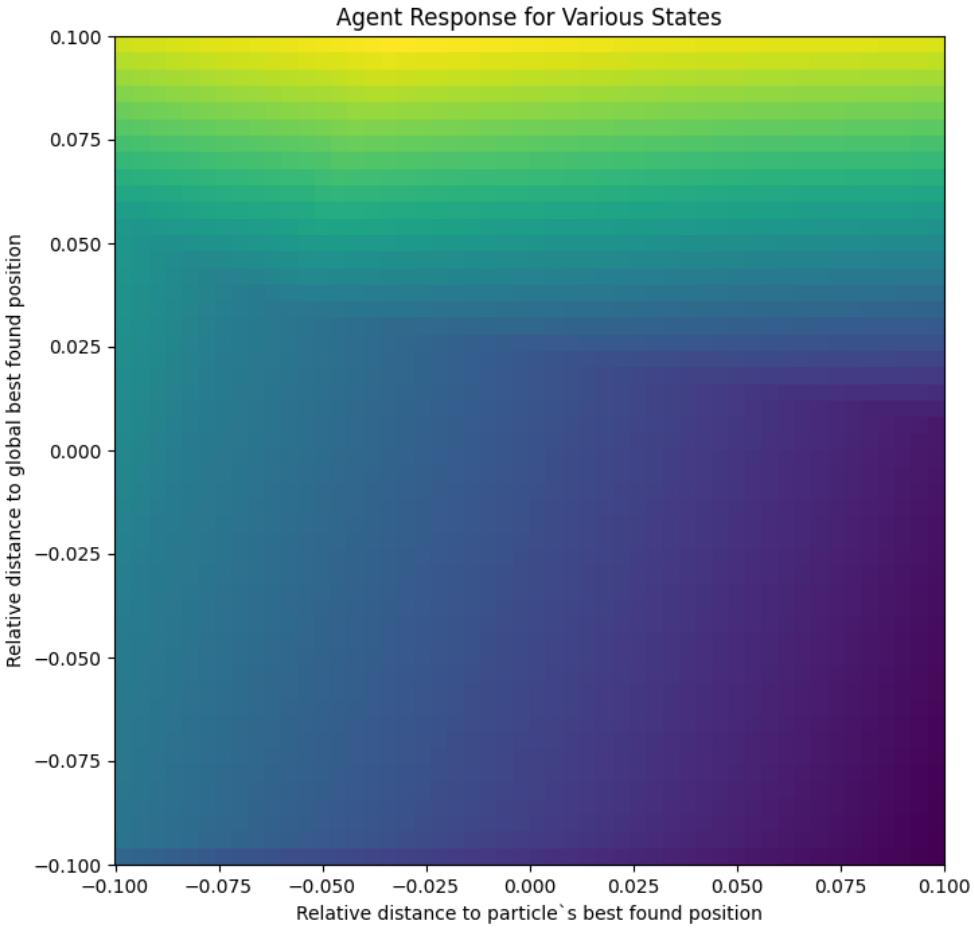


Figure 5.8: A plot of the agent's response to different values of relative distance to the global best position, and its own best position. Brighter areas indicate that the policy would output a large value for these values, and darker indicate a smaller response.

5.4 Experiments on the Rosenbrock Function

The Rosenbrock function is a function with little noise, but many points that have a zero gradient in some directions. There is a band of points around the global minima that is flat in one direction but not in another, which might cause some difficulty for optimization methods. This function belongs to a category of functions called Valley-shaped functions. This classification becomes clear when inspecting the shape of the function in Figure 5.9.

$$f(\mathbf{x}) = \sum_{i=1}^{n-1} [100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2] \tag{5.4}$$

The global optimum of this function is in $\mathbf{x} = [1, ..., 1]$ with objective value 0. While this function can sometimes be restricted by $x_i \in [-2.048, 2.048]\ \forall i$, we have used the more frequently used alternative, which is $x_i \in [-5, 10]\ \forall i$.

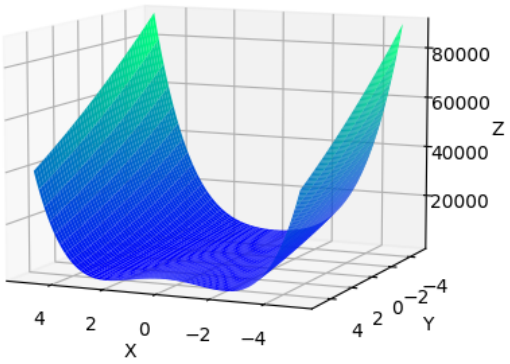


Figure 5.9: A 3D Plot of the Rosenbrock Function

5.4.1 Experimental results

Table 5.8: Experiment parameters for Rosenbrock function

	Parameters				
Experiment	Particles	Dimensions	Timesteps trained	Iterations	Initial std
1	50	2	35 mill	100	e^{-1}
2	100	5	50 mill	100	$e^{-0.5}$
3	100	10	100 mill	100	$e^{-0.5}$

As with the other benchmark functions, we can see that the difference between the PSO and SmartSwarm is larger in the bigger instances.

Table 5.9: Objective values for SmartSwarm and baselines on Rosenbrock

Experiment	Objective (Standard deviation)		
	SmartSwarm	PSO	Random
1	0.0250 (0.0683)	3.158×10^{-4} (3.23×10^{-4})	0.05613 (0.0669)
2	13.416 (5.908)	2.625 (0.954)	93.934 (62.259)
3	5089.612 (2722.441)	45.790 (15.109)	5625.6569 (2407.835)

Table 5.10: Improvement values for SmartSwarm and baselines on Rosenbrock

Experiment	Improvement in % (Standard deviation)		
	SmartSwarm	PSO	Random
1	99.144 (0.0250)	99.937 (0.487)	96.603 (11.138)
2	97.285 (0.0443)	99.702 (0.287)	86.645 (16.871)
3	66.533 (0.221)	99.639 (0.297)	58.699 (22.0410)

5.4.2 Discussion

As can be seen from the experimental results, the method still struggles in higher dimensions. This is most probably still a symptom of the learning problems related to complicated high-dimensional objective functions.

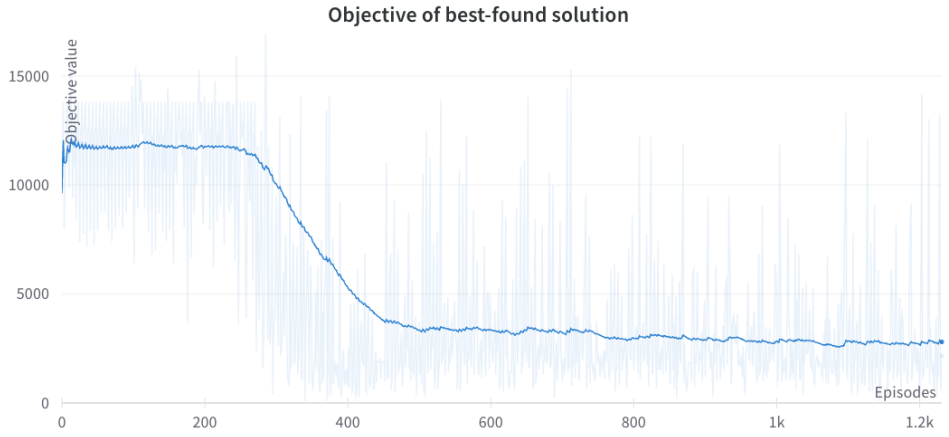


Figure 5.10: Plot illustrating the objective value of the best-found solution during the training phase of the largest instance on the Rosenbrock function.

As can be seen from Figure 5.10, the agent learns for a while, and then the learning speed drastically goes down. The same effect is observed in the reward the agent receives in Figure 5.11, as the reward stops improving. This happens around episode 500, which is the same time as the agent stops improving its objective value. At this point, it is likely that the agent has found some policy that it struggles to improve on, and that cannot match the performance of PSO.

Figure 5.11: Plot illustrating the mean reward pr agent per timestep for each batch during training.

Chapter 6

Conclusion and future work

6.1 Conclusion

In this thesis, we have proposed the novel PSO variant SmartSwarm. This is a Multi-Agent Reinforcement Learning based version of PSO that uses continuous models to dictate the velocity of particles in PSO. Furthermore, the algorithm utilizes Multi-Agent RL's cooperative and complex nature to allow for more advanced behaviors in the swarm. This framework aims to adapt to the specific optimization task and search for an optimal policy to determine the best velocity for each particle in every setting.

In the process of conducting experiments using three widely-recognized benchmarks—Rastrigin, Sphere, and Rosenbrock functions—across a range of problem sizes, we have successfully demonstrated the capacity of our Multi-Agent Reinforcement Learning algorithm to adapt and learn within some of these environments. Furthermore, our observations suggest that SmartSwarm exhibits greater ease of learning and improvement in simpler environments than in more complex ones. While our algorithm has not been able to surpass the performance of PSO in our experiments, we regard this as a stepping stone for future work.

6.2 Future work

Future work should explore domain-specific uses of SmartSwarm in which we can use additional parameters that will give the algorithm more information than PSO can exploit. To include more information in the model, we need parameters that relate to the objective function we want to optimize, such that the agent can find patterns in the observations and thus search the space more efficiently. This should be compared to PSO to investigate whether such parameters will aid the optimization process.

One exciting area of research is to use this algorithm as a meta-learning optimization technique. This could involve using SmartSwarm instead of a typical gradient-based method. Such a learning scheme could incorporate different learning parameters, such as regularization parameters or model architecture, to find an optimal model more efficiently.

SmartSwarm has the potential to employ several other learning frameworks that could potentially improve its performance. An example of this could be a Recurrent Neural Network that could handle all dimensions of the problem in a single forward pass for each agent. A Recurrent Neural Network could create a more complex policy that will consider all dimensions when determining the velocities of the particles.

Other Reinforcement Learning methods should be explored to enhance the SmartSwarm algorithm further. This may include off-policy learning methods that could learn a fixed velocity function and attempts to fine-tune this policy. This way, we can ensure learning to a higher degree of certainty but may suffer from bias towards the velocity function used for pretraining the model. In addition, it might be worth exploring longer training times for the models investigated in this work, as this might allow PPO to learn in higher dimensional spaces.

As SmartSwarm has a learned policy, it could potentially hold some interesting strategies to be explored further. When studying these strategies, we could gain a deeper understanding of the optimal movement of a swarm. Furthermore, by mimicking these learned strategies, we might find algorithms with enhanced performance.

Bibliography

Ahmed G Gad. Particle swarm optimization algorithm and its applications: A systematic review. 29:2531–2561, 2022. doi: 10.1007/s11831-021-09694-4.
URL: `https://doi.org/10.1007/s11831-021-09694-4`.

Fred Glover. Tabu search—part i. *ORSA Journal on Computing*, 1(3):190–206, 1989. doi: 10.1287/ijoc.1.3.190.
URL: `https://doi.org/10.1287/ijoc.1.3.190`.

Estephanos Jebessa, Kidus Olana, Kidus Getachew, Stuart Isteefanos, and Tauheed Khan Mohd. Analysis of reinforcement learning in autonomous vehicles. In *2022 IEEE 12th Annual Computing and Communication Workshop and Conference (CCWC)*, pages 0087–0091, 2022. doi: 10.1109/CCWC54503.2022.9720883.

Jakob Kallestad, Ramin Hasibi, Ahmad Hemmati, and Kenneth Sörensen. A general deep reinforcement learning hyperheuristic framework for solving combinatorial optimization problems. *European Journal of Operational Research*, 309:446–468, 8 2023. ISSN 0377-2217. doi: 10.1016/J.EJOR.2023.01.017.

J. Kennedy. Swarm intelligence. In A.Y. Zomaya, editor, *Handbook of Nature-Inspired and Innovative Computing*. Springer, Boston, MA, 2006. doi: 10.1007/0-387-27705-6_6.

J. Kennedy and R. Eberhart. Particle swarm optimization. In *Proceedings of ICNN'95 - International Conference on Neural Networks*, volume 4, pages 1942–1948 vol.4, 1995. doi: 10.1109/ICNN.1995.488968.

S. Kirkpatrick, C. D. Gelatt Jr, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983. doi: 10.1126/science.220.4598.671.

Yaxian Liu, Hui Lu, Shi Cheng, and Yuhui Shi. An adaptive online parameter control algorithm for particle swarm optimization based on reinforcement learning. In *2019 IEEE Congress on Evolutionary Computation (CEC)*, pages 815–822, 2019. doi: 10.1109/CEC.2019.8790035.

Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback, 2022.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.
URL: `http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf`.

Nguyen Huu Phong, Augusto Santos, and Bernardete Ribeiro. PSO-convolutional neural networks with heterogeneous learning rate. *IEEE Access*, 10:89970–89988, 2022. doi: 10.1109/access.2022.3201142.
URL: `https://doi.org/10.1109%2Faccess.2022.3201142`.

R. Poli, J. Kennedy, and T. Blackwell. Particle swarm optimization. *Swarm Intelligence*, 1:33–57, 2007. doi: 10.1007/s11721-007-0002-0.

L. A. Rastrigin. *Systems of Extreme Control*. Nauka, Moscow, 1974.

A. Ratnaweera, S.K. Halgamuge, and H.C. Watson. Self-organizing hierarchical particle swarm optimizer with time-varying acceleration coefficients. *IEEE Transactions on Evolutionary Computation*, 8(3):240–255, 2004. doi: 10.1109/TEVC.2004.826071.

Gavin Rummery and Mahesan Niranjan. On-line q-learning using connectionist systems. Technical Report CUED/F-INFENG/TR 166, Cambridge University, 1994.

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, 6 2016. ISSN 15684946. doi: 10.1016/J.ASOC.2016.01.006.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. 7 2017. doi: 10.48550/arxiv.1707.06347.
URL: `https://arxiv.org/abs/1707.06347`.

Y. Shi and R. C. Eberhart. A modified particle swarm optimizer. In *Proceedings of the IEEE international conference on evolutionary computation*, pages 69–73, Piscataway, 1998. IEEE.

D Silver, A Huang, C Maddison, and et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016. doi: 10.1038/nature16961.

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy P. Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *CoRR*, abs/1712.01815, 2017.
URL: `http://arxiv.org/abs/1712.01815`.

MengXuan Song, Kai Chen, and Jun Wang. Three-dimensional wind turbine positioning using gaussian particle swarm optimization with differential evolution. *Journal of Wind Engineering and Industrial Aerodynamics*, 172:317–324, Jan 2018. doi: 10.1016/j.jweia.2017.10.032.

Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018.
URL: `http://incompleteideas.net/book/the-book-2nd.html`.

Justin K. Terry, Benjamin Black, Ananth Hari, Luis S. Santos, Clemens Dieffendahl, Niall L. Williams, Yashas Lokesh, Caroline Horsch, and Praveen Ravi. Pettingzoo: Gym for multi-agent reinforcement learning. *CoRR*, abs/2009.14471, 2020.
URL: `https://arxiv.org/abs/2009.14471`.

Oriol Vinyals, Igor Babuschkin, Wojciech M. Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H. Choi, Richard Powell, Timo Ewalds, Petko Georgiev, Junhyuk Oh, Dan Horgan, Manuel Kroiss, Ivo Danihelka, Aja Huang, Laurent Sifre, Trevor Cai, John P. Agapiou, Max Jaderberg, Alexander S. Vezhnevets, Rémi Leblond, Tobias Pohlen, Valentin Dalibard, David Budden, Yury Sulsky, James Molloy, Tom L. Paine, Caglar Gulcehre, Ziyu Wang, Tobias Pfaff, Yuhuai Wu, Roman Ring, Dani Yogatama, Dario Wünsch, Katrina McKinney, Oliver Smith, Tom Schaul, Timothy Lillicrap, Koray Kavukcuoglu, Demis Hassabis, Chris Apps, and David Silver. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature 2019 575:7782*, 575:350–354, 10 2019. ISSN 1476-4687. doi: 10.1038/s41586-019-1724-z.
URL: `https://www.nature.com/articles/s41586-019-1724-z`.

Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(3-4):229–256, May 1992. doi: 10.1007/BF00992696.
URL: https://doi.org/10.1007/BF00992696.

S. Yin, M. Jin, H. Lu, et al. Reinforcement-learning-based parameter adaptation method for particle swarm optimization. *Complex Intell. Syst.*, 2023.
URL: https://doi.org/10.1007/s40747-023-01012-8.

Chao Yu, Jiming Liu, Shamim Nemati, and Guosheng Yin. Reinforcement learning in healthcare: A survey. *ACM Computing Surveys (CSUR)*, 55(1):1–36, 2021.

Chao Yu, Akash Velu, Eugene Vinitsky, Jiaxuan Gao, Yu Wang, Alexandre Bayen, and Yi Wu. The surprising effectiveness of ppo in cooperative, multi-agent games, 2022.

Nianyin Zeng, Hong Qiu, Zidong Wang, Weibo Liu, Hong Zhang, and Yurong Li. A new switching-delayed-pso-based optimized svm algorithm for diagnosis of alzheimer's disease. *Neurocomputing*, 320:195–202, 2018.

Kaiqing Zhang, Zhuoran Yang, and Tamer Başar. Multi-agent reinforcement learning: A selective overview of theories and algorithms. 11 2019.
URL: http://arxiv.org/abs/1911.10635.

Appendix A

Notes on the standard deviation of output under learning

The experiments have shown a large dependence on the initial standard deviation in the policy network of the SmartSwarm algorithm. As mentioned in earlier chapters, the action of the PPO algorithm is sampled from a normal distribution to ensure exploration during training. When the velocity is drawn from the normal distribution of the actor, the probability of the actor picking that action is adjusted according to how good that action was. So even if the agent never would have picked a certain velocity, the algorithm still learns from the outcome.

Note that the standard deviation of the output is actually a learnable parameter, and it is tuned by gradient descent as the other parameters of the model are. This value will be set to what benefits the agent most. This thinking makes it easy to ignore the initial standard deviation as it will be tuned. But the initial value of the standard deviation has a lot of influence on the progression of the training.

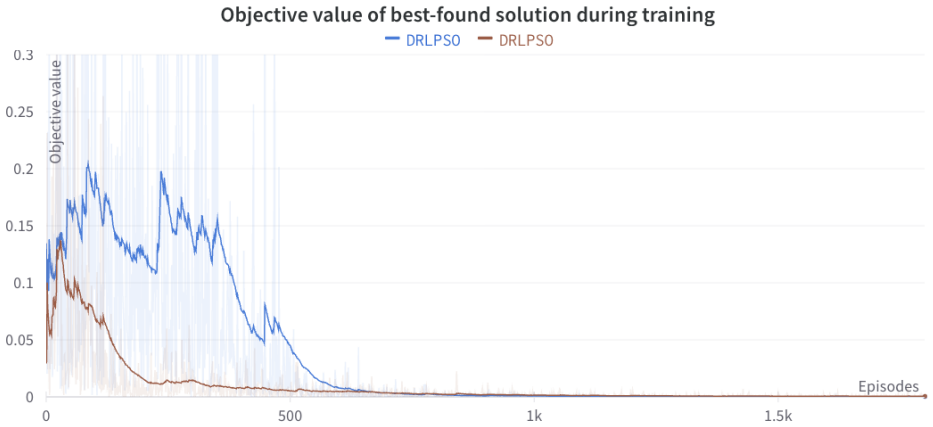


Figure A.1: A figure of the training progression of one training session using a high initial standard deviation on the model (brown color), and one training session with a smaller standard deviation (blue color). The initial standard deviation of the agents displayed by a brown line was 1, while the standard deviation of the agents behind the blue line was e^{-1}.

From Figure A.1 we can see that in the session in which we used a smaller standard deviation, the agent gives an impression of it not learning. But this is countered by a very rapid decrease in the objective values of the solutions found in the search and a performance that is better than the other agent with a larger standard deviation. This is an effect that comes from the agent's ability to precisely determine its actions. When the standard deviation is high, it is hard for the agents to accurately control their behavior, and this affects their performance. It also follows that the agents will encounter a larger number of states and that they might learn more from these observations with a higher standard deviation.

Striking a beneficial balance between exploration and exploitation through the standard deviation is critical and has a large effect on the final model. A low initial standard deviation will often lead to high-performance agents but will learn slower. Conversely, a high initial standard deviation will have the opposite effect.