Carl Stermann-Lücke

Supervised Learning of Swarm Behaviour using Learning Classifier Systems

OTTO VON GUERICKE UNIVERSITÄT MAGDEBURG

INF FAKULTÄT FÜR INFORMATIK

Intelligent Cooperative Systems
Computational Intelligence

Supervised Learning of Swarm Behaviour using Learning Classifier Systems

Master Thesis

Carl Stermann-Lücke

July 14, 2020

Supervisor: Prof. Dr. Sanaz Mostaghim

Advisor: Dr. Christoph Steup

Carl Stermann-Lücke: *Supervised Learning of Swarm Behaviour using Learning Classifier Systems*
Otto-von-Guericke Universität
Intelligent Cooperative Systems
Computational Intelligence
Magdeburg, 2020.

Abstract

In Swarm Robotics, complex behaviour can emerge from the local interactions of multiple simple agents. This behaviour is generally robust, scalable and flexible. In this thesis, I investigate how the simple agents have to act locally for a given collective behaviour to emerge. For this task, I introduce Global-Local Swarm Learning (GLSL). Unlike previous approaches, GLSL uses supervised learning to recreate an existing behaviour that was created with information that the swarm does not have. As a classifier, I use a Michigan-Style Learning Classifier System (LCS). The research question is how the success of the learning depends on the difficulty of the task. I use a leader-follower task with different levels of difficulty. The results show no clear connection between the difficulty and the success. In its current configuration including the LCS, GLSL does not produce the desired behaviours reliably.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

Swarm Intelligence describes how to solve complex tasks with simple agents working together. Designing simple agents can be hard: Their design consists of local rules. Local rules are rules based on information of the local neighbourhood and sensor values only, not on global positioning and knowledge of all other agents. It is not clear how to reliably find such local behaviour rules for each agent, such that they emerge into the desired swarm behaviour. Previous approaches have drawbacks such as a tendency for overfitting, dependence on expert knowledge, (arguably) long learning cycles, and a lack of understandability or reproducibility. This thesis contributes to developing such a method by evaluating the performance and suitability of using Learning Classifier Systems in a supervised learning setting when applied on swarm intelligence tasks of varying difficulty. A reliable method to find local rules will make it possible to perform more tasks in the real world with a number of cheap robotic agents, and to perform these tasks robustly and scalably, independent of any global communication or central control.

Automatic ways to find such local rules have so far optimised rules by evaluating them according to the quality of the emerging behaviour, as described in section 2.3. In this thesis, however, I follow a different approach: supervised learning. Here, the local rules are optimised not to achieve the best quality emerging behaviour, but to best match an existing behaviour that solves the task. The supervised learning algorithm employed is a Michigan-style Learning Classifier System. Learning Classifier Systems (LCS) are an interesting family of learning algorithms to be applied to this problem, for three reasons:

- They produce explicit rules that are human-readable (unlike (Deep) Neural Networks in particular).

- They have an inherent generalisation pressure that tends to produce shorter and fewer rules.
- They have already successfully been applied to find swarm rules – but using reinforcement learning, not supervised learning.

The swarm tasks that I examine in this thesis include a few supposedly easy ones with just one or a few swarm members, to show whether learning them with an LCS is feasible in principle, and additionally a task of following a leader through a tunnel. This task is particularly interesting, because the agents that follow the leader make use of information in the training that is not available to them later on.

1.2 Goals

The goal of this thesis is to find out how the accuracy of learning a specific swarm behaviour using an LCS depends on the difficulty of the task. Accuracy in this sense means to what extent the learnt behaviour (i.e. the outcome) resembles the behaviour that was supposed to be learnt (i.e. the input). The difficulty of the task in turn depends to a great extent on the difficulty of the simulated environment. The swarm behaviour is following a leader through a tunnel without collision. The environment, and therefore the task, is said to be more difficult if the tunnel is longer and narrower, the leader is faster, the memory of the followers is shorter, their sensors are noisier or have poorer resolution, and if a smaller proportion of the followers can see the leader themselves.Since a key concept of swarm intelligence is that the swarm do not contain a leader [2], in this scenario, I only consider the followers to be the swarm, while the leader belongs to the environment which the swarm has to interact with.

1.3 Tasks

1. Implement a collision-free simulation of a swarm passing a tunnel where every robot can access global information. (section 3.3)
2. Collect training data of local sensors and actuators from this simulation. (also section 3.3)

3. Learn the connection between local sensors and actuators from this training data using a Michigan-style Learning Classifier System (LCS). (sections 2.1 and 3.5)
4. Simulate the learnt behaviour and compare it to the original behaviour. (section 3.6)
5. If the learning is considered successful, make the environment more difficult and go to 1.

2 Related Work

2.1 Learning Classifier Systems

For the LCS, I use an existing python software called "ExSTraCS" by Ryan Urbanowicz and Jason Moore [11]. It makes sense to use an existing package, because it is readily available, and because it is difficult to find errors in the implementation of a learning algorithm, so implementing it myself would be an additional risk factor. The input data can be categorical or continuous, but the output data is categorical, i.e. it predicts classes.

As any classifier, LCS needs training instances that come with a class label (*Dataset* in figure 2.1). These form the ground truth, which the classifier is supposed to learn to predict. The training instances are processed by the LCS one after the other. An LCS composes its classifier of single rules that tell the class for data samples they match. Each rule has certain attributes of the dataset specified and (usually) others unspecified. Attributes can be categorical or continuous, and both types can appear in the same rule - however, in the implementation that I use for this thesis, each attribute can be either categorical or continuous, not a mix of both. This is determined during a pre-analysis of the data by counting the number of values that an attribute has. If this number is below a threshold, the attribute is considered categorical, otherwise continuous.
A rule matches a data sample if and only if all of its specified attributes match. That means for categorical attributes, that the value of the attribute in the data sample must be the same as in the rule. For continuous attributes, the value of the attribute in the data sample must fall within the interval specified by the rule. Attributes that are not specified, match any value of the corresponding attribute in a data sample.

A rule consists of a number of different attributes. However, the important ones for us are the indices of specified attributes ("Specified"), the values of

InstanceID	D-180	D-90	D0	D90	Class
3397	0.89608730	8.0	8.0	8.0	0
2771	8.0	0.8897508	8.0	8.0	3

Table 2.1: These are two example data points. The *InstanceID* is not an attribute.

Listing 2.1: a rule including the names of its attributes

```
Specified          Condition          Phenotype          Fitness
   Accuracy           Numerosity         AveMatchSetSize
   TimeStampGA        InitTimeStamp      Specificity
   DeletionProb       CorrectCount       MatchCount
   CorrectCover       MatchCover         EpochComplete

[1]       [[0.16702957999999999, 3.100603082]]       3
   0.510015227831791          0.510015227831791          4
          22.835706760497693          999997   960758   0.25
        90.97316725311057          4354      8537      297
   583       1
```

the specified attributes ("Condition"), the predicted class ("Phenotype") and the accuracy. The accuracy is the quotient of the number of matches where the rule predicted the correct class (*CorrectCount*), over the number of matches of the rule in total (*MatchCount*). The fitness value is in general a power of the accuracy (the exponent is a parameter that can be set by the user). However, if the algorithm generates new rules by covering, mutation and crossover (see below), the new rules get a fixed fitness value (which is also a parameter).
Consider the two rules in table 2.1 and the rule in listing 2.1. Only the second attribute ($D-90$) is specified (since the indices are zero-based) and must be between 0.167 and 3.101 to match. The first data point (ID 3397) has a value of 8 for attribute $D-90$, which is not within this range, so it does not match. The second one (ID 2771) has a value of 0.890, which is within the range specified in the rule. So it matches. When the rule sees the second data point for the first time, its *MatchCount* will increase by 1. Since both the data point and the rule predict class 3, the *CorrectCount* also increases by 1. The

other parameters, particularly accuracy and fitness, get updated accordingly.

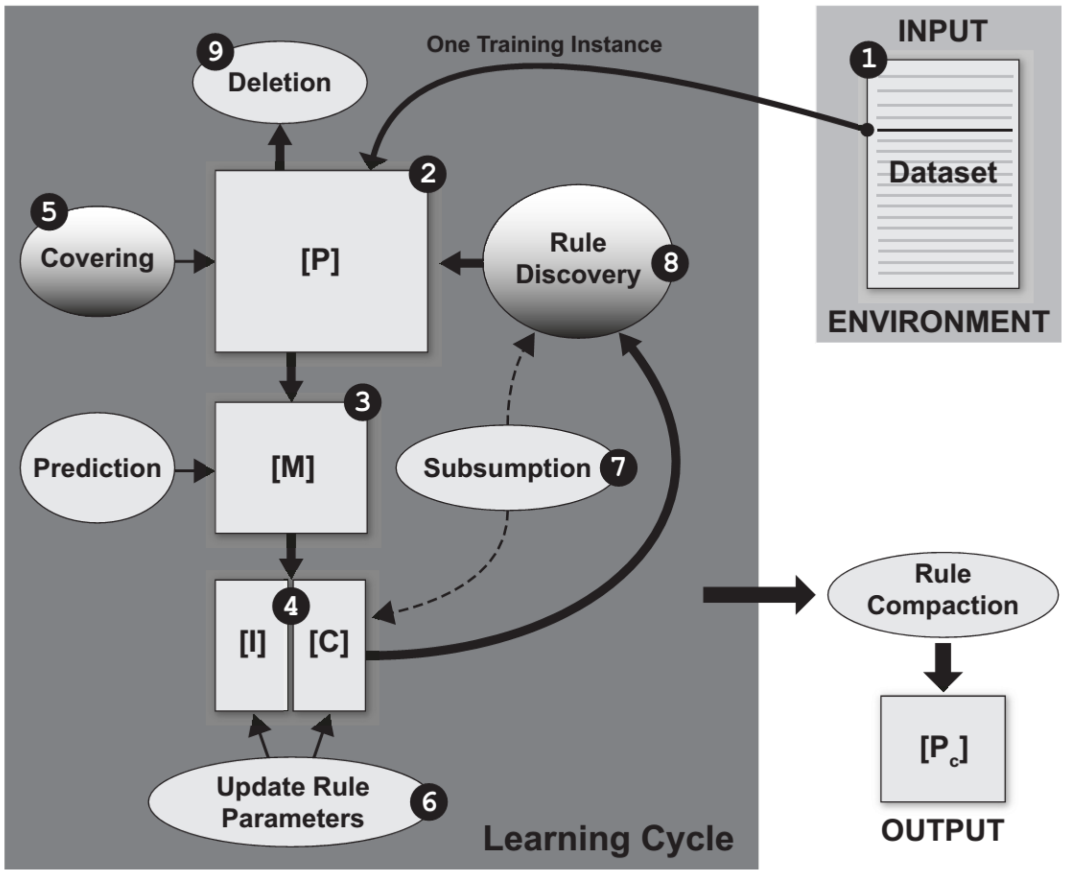


Figure 2.1: The architecture of ExSTraCS, the LCS implementation used in this thesis. "Ovals indicate algorithmic components (those with a gradient can discover new rules) and boxes represent sets of classifiers/rules, except for ‘dataset’" [12], page 8.

An LCS algorithm for supervised learning must implement the following stages:

- Covering: Rules are generated by generalising a specific training sample, meaning converting some of its attributes into unspecified attributes. For continuous attributes, intervals are created with the value of the training sample in the center. The size of the intervals vary. This process is used whenever there is no rule matching the current training instance and predicting the correct class. In the beginning, the rule-set is empty, so covering needs to be applied. The generated rule is added to the Rule Population (P in figure 2.1).

- Update Rule Parameters (Evaluation of rules): Every time a rule matches a training sample, its accuracy needs to be updated. It is increased if the rule predicted the correct class, and decreased otherwise. Since fitness is a function of accuracy, it is also updated. Rules predicting the correct class are added to the Correct Set C, the others to the Incorrect Set I.
- Rule Discovery (Genetic algorithm): The set of rules P is considered as a population that is subject to crossover and mutation so to improve its fitness. For every training sample, two rules A and B with good fitness are picked from C if the rules in C are old enough on average. With a certain probability, these now undergo a crossover: They are copied, and their copies are modified, while the original (parent) rules stay in the rule population unchanged. In case of an attribute that is specified in rule A and unspecified in rule B, the specifications can be swapped (with a certain probability), so that afterwards it is specified in rule B and unspecified in rule A. In case of continuous attributes being specified in both rules, the boundaries of the intervals are swapped, so the lower boundary in rule A is replaced by the lower boundary in rule B, and vice-versa. The same happens with the upper boundaries of the intervals that the rules specify. After crossover, the offspring rules can be subject to mutation, again with a certain probability: Each attribute can be swapped from specified to unspecified or vice-versa. The offspring rules will match the current training instance in any case.
- Deletion: If rule discovery becomes active, the number of rules (or rather, the sum of the numerosities) increases, because the offspring rules are added to the population. To limit the population size, rules are deleted from the population with a probability, typically inversely proportional to their fitness. New rules can receive an exemption from deletion. In turn, as many rules are deleted as necessary to bring the sum of the numerosities back into its limit.
- Prediction: A classifier is only useful if it can predict classes of data points where it does not know the true class. When this step arrives, the learning mechanisms are deactivated. All rules that match are grouped by the class that they predict and weighted by their accuracy. The class

with the highest sum of accuracy values is what the algorithm predicts for that data point.

Optional further processes include:

- Subsumption: If two rules predict the same class and match the same instance, it is possible that one of them is a generalisation of the other. That means that it matches all the instances that the other one matches, and beyond that, matches more, because it has fewer attributes specified or has them specified less strictly (in case of continuous attributes). If the more general one is also at least as accurate as the other and above the accuracy threshold, it subsumes the more specific rule. In this case, the more specific rule gets deleted from the rule population and the numerosity of the more general rule increases by the numerosity of the more specific rule. This mechanism adds another generalisation pressure into the algorithm. It can be applied on rules that have just matched the same training instance and predicted the class correctly, or on the parents and offspring of the genetic algorithm.
- Rule Compaction: Poor rules are only deleted if the total numerosity exceeds its limit. Therefore, they often stay in the rule population until the end of the learning. Rule compaction tries to find rules that decrease the performance of the whole model, and remove them. According to Urbanowicz, the typical criteria are poor fitness, redundancy and small match counts (i.e. they have not matched many instances).

Further attributes of the rules, apart from the ones described above, are:

- AveMatchSize, the average size of the match sets which the rule was included in
- TimeStampGA, the last iteration when the rule was last included in a correct set on which the Genetic Algorithm was applied. This is used to guide when the Genetic Algorithm should become active again.
- InitTimeStamp, the iteration when the rule was created
- Specificity, the fraction of attributes that is specified in this rule
- DeletionProb, the current weight that determines the probability of the rule being deleted during the deletion mechanism
- CorrectCover, the number of training instances where the rule matched and predicted the correct class

- MatchCover, the number of training instances where the rule matched overall
- EpochComplete, a boolean that tells whether the rule has seen all the training instances at least once.

2.1.1 Advantages

The LCS has three main advantages that make me expect it to be appropriate for the task of this thesis. Firstly, it makes the rules transparent, so the user can understand how it classifies data. The number of rules can be kept within limits by reducing the numerosity limit or by applying strict rule compaction. Secondly, there is an implicit pressure in the algorithm to create more generic rules. This happens because generic (good) rules match more training samples, so they have it easier to improve their fitness, be selected as parents for the genetic algorithm, and survive the deletion process that keeps the size of the rule population within its limit. Thirdly, the LCS has performed well in data sets where each attribute is important in some part of the data [14]. This is the case for swarm simulations: If some distance sensor value is very small, the robot should escape to the other direction at almost any cost, in order to avoid a collision.

2.1.2 Disadvantages

I identify two drawbacks of the LCS: Firstly, since the LCS can only predict classes, not continuous values, its predictions are necessarily imprecise if the original data was continuous. So a discretisation error is unavoidable. Secondly, in classification, it might happen that no rule matches. In that case, it is not clear what to do. In order to include a fall-back rule that matches every test sample, a safe action should be known.

2.1.3 LCS for Reinforcement learning

What I have described above is an LCS for supervised learning, as I use it in this thesis. There are variants of LCS that are suitable for reinforcement learning. The ones that are most similar to the algorithm described above, the "XCS" (for "Extended Classifier System", introduced in [16]), have rules that predict not the correct class, but the likely reward from taking an action, which depends on the quality score of the behaviour resulting in taking that action. For this purpose, a function to evaluate the quality of a behaviour needs to be defined by the designer. The supervised LCS, on the other hand, has a predefined objective function, which is to learn the connection between condition and action in the training data, and therefore minimise the difference between this connection in the data and the connection learnt by the classifier. In other words, the objective function is defined by the presence of training data. The XCS predicts the action with the highest expected reward for the current sensor reading, although during training, it balances an exploration factor in.

In this thesis, I do not use XCS, but some previous approaches, such as OCbotics (section 2.3.5), do.

2.2 Swarm intelligence

Swarm Intelligence describes the ability of simple agents to solve complex problems when collaborating with each other in a swarm. Agents and problems can be of physical or virtual nature. Every member of a swarm follows the same simple rules, but emerging from this, more complex behaviour of the swarm comes about. This swarm behaviour is not per se straight-forward to foresee when examining the simple agents - hence the use of the term "emergence". However, the rules can be applied in simulation or on physical robots, and the emerging behaviour can be observed. This way, for certain rules, studies have been examined on the type of behaviour that emerges from them. This information is valuable for this thesis, because the first step in supervised learning is to generate data, which in this case means to create a simulation of a swarm performing the desired behaviour. Subsection 2.2.1 explains what makes collaborative behaviour fall under the category of Swarm Intelligence, and which benefits arise from this. Subsection 2.2.2 lists different types of rules. For the class of rules that is relevant for this thesis, I give an overview in subsection 2.2.3. This section concludes with a collection of typical swarm tasks in subsection 2.2.4.

2.2.1 Characteristics of Swarm Intelligence

Although the field of Swarm Intelligence also incorporates systems of mostly virtual nature, such as Particle Swarm Optimisation or Ant Colony Optimisation, this thesis contributes to the study of the behaviour of simulated robots. Therefore, for the motivation of this thesis, the characteristics and benefits of Swarm Robotics are essential. Brambilla et.al. [2] list the following properties:

- autonomy
- local scope of sensing and communication
- lack of centralised control and global knowledge
- cooperation on a common task

They also list interaction with and modification of an environment. However, in the example tasks they list in the same paper, it becomes clear that modification of an environment is not a strict requirement for them, or it does not need to go further than moving in an environment, resulting in changes in sensor input. Şahin [8] adds that the robots should, as individuals, be "relatively incapable and inefficient", while as a swarm, they surpass their individual capabilities and solve the task more efficiently through collaboration. Further, there should be only few different kinds of individuals in the swarm. Apart from this, robotic swarms have three key benefits [2]:

- Robustness against the loss of individuals. This is the result of redundancy, which in turn comes from having many swarm members, but only few kinds of them, and particularly from not having a leader.
- Scalability, for different group sizes. This stems from the lack of a centralised control and the localness of sensing, which means that adding or removing robots somewhere else in the swarm primarily has local effects there, and not global effects on the whole swarm.
- Flexibility towards different environments and tasks. This benefits from the simplicity of the local rules, which means that they are less likely to adapt too much to a very specific environment. Since their actions are largely influenced by their local surroundings, changes in that surroundings lead to adaptation.

2.2.2 Swarm Rules

Swarm rules can be of different type depending on the method how they were produced. Brambilla et al. [2] categorise manual rule generation methods by the format of the rules that they produce. They mention two main categories:

Firstly, there are methods based on Probabilistic Finite State Machines. Their rules describe what an agent does in each state and under what conditions (and/or with what probability) it transitions to which other state. Such a state machine can be similar to the ones that AutoMoDe (see section 2.3.2) produces, although AutoMoDe uses building blocks that would themselves need to be defined by the designers.

Secondly, there are methods based on attraction- and repulsion-functions. Brambilla et al. [2] call these "virtual physics-based design methods", because they resemble the movement of physical robots in a vector field. This method usually needs fewer rules, because the attraction- and repulsion functions sum up multiple rules into one, and because the resulting vectors can be combined arithmetically, instead of having to handle many different cases.

Both types of rules can also be combined hierarchically, where different attraction- and repulsion-functions are active in different states of the state machine.

In this thesis, we come across a third type of rules, which is a simple "IF-THEN" format. Specific or ranged input is mapped to an output by rules that the LCS generates, as shown in section 2.1. Different input is matched by different rules. Like attraction- and repulsion-functions, these rules are stateless.

Since methods using attraction- and repulsion-functions are particularly suitable for coordinated motion [2], these have been used in this thesis and the following section focuses on these.

2.2.3 Attraction- and Repulsion-Functions

Functions for attraction and repulsion map the (signed) distance between an individual and an object (which may be another robot, an obstacle, or a target) to a vector on the line that connects both. That means, depending on how far away the other object is, the functions tell the individual whether to move towards or away from the object and by how much (i.e. they describe a force). Since there are potentially several objects that a robot needs to consider, functions for attraction and repulsion to each of them are summed up. Afterwards, the sum is often normalised, since otherwise the number of neighbouring objects could significantly influence the speed of the agent.

If information should pass through the swarm, but there are no explicit means of communication, then the swarm should stay cohesive, which means that the agents should not disperse. On the other hand, the swarm members should stay separate, which means they should not collide, occupy the same space or even switch sides right through each other. By having cohesion and separation, a movement of a swarm member affects its surrounding fellow swarm mates. To achieve both properties, functions need to provide an element of attraction when the distance is large, and an element of repulsion when the distance is small. The function should also have an equilibrium point where the magnitude of the sum of attraction and repulsion is 0 and where the distance converges towards. This point is called the "comfortable distance".

Let x_i be the position of agent i. The force on i exerted by its relative position to x_j is $f(i,j)$. The euclidean distance between x_i and x_j is $\|y\| = \|x_i - x_j\|$. Further, let $g_a(\|y\|)$ be a function of how strongly i is attracted towards j given their distance, and $g_r(\|y\|)$ a function of how strongly i is repelled from j given their distance. Then

$$f(i,j) = -(x_i - x_j) \cdot (g_a(\|y\|) - g_r(\|y\|)) \tag{2.1}$$

is a general attraction- and repulsion function, as described by Gazi and Passino [7]. If $g_a(\|y\|)$ is larger than $g_r(\|y\|)$ for larger values of $\|y\|$, and smaller for smaller values of $\|y\|$, then there is a comfortable distance where both values are equal.

The following function (plotted in figure 2.2) has linear attraction (which grows proportionally to the distance) and bounded repulsion (which becomes greater than the attraction for small distances):

$$f(i,j) = -(x_i - x_j) \cdot (a - b \cdot e^{\frac{-\|x_i - x_j\|^2}{c}}) \tag{2.2}$$

Gazi and Passino proposed this function in [6]. In this formula, the attraction is $a \cdot \|y\|$ and the repulsion is $b \cdot e^{\frac{-\|x_i - x_j\|^2}{c}} \cdot \|y\|$. The comfortable distance is

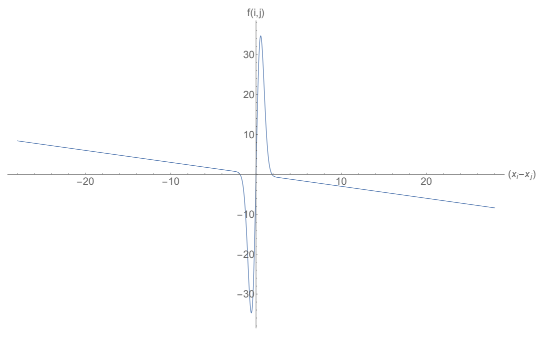


Figure 2.2: This graph shows the force f on individual i depending on its position relative to j, for equation 2.2 with parameters $a = 0.3$, $b = 105$ and $c = 0.6$. Positive values of f for negative values of $x_i - x_j$ mean attraction to j, and negative values mean repulsion. For positive values of $x_i - x_j$, it is the opposite.

at $d = \sqrt{c \cdot \ln \frac{b}{a}}$. Therefore, b has to be greater than a, and c must be greater than 0. If b was smaller than a, repulsion would stay smaller than attraction, so the agents would collide. If c was negative, the comfortable distances (if they exist) would not be stable. That means that the distance would only be kept if matched exactly. Smaller or larger distances would cause the agent to be drawn closer or repelled even further, away from the comfortable distance. Equation 2.2 is point-symmetric, which means that the attraction occurs towards j and the repulsion away from j, no matter on which side of j i is. Gazi and Passino [7] show that this property ensures that a swarm governed by this function between the agents will aggregate (which means it gathers in one region of the environment) and form a pattern where the distances to the nearest neighbours of a swarm member are equal - although only for the case of a fully-connected swarm, where every swarm member knows the position of every other. For that case, they also show that the center of the swarm stays stationary and that the members eventually stop moving relative to each other. Note that these do not necessarily hold true when the agents experience forces from anything other than fellow swarm members, such as obstacles, a slope in the environment, etc., as it is the case in this thesis. Gazi and Passino propose more functions in the same paper [7], where attraction and repulsion can each behave differently. $g_a(\|y\|)$ and $g_r(\|y\|)$ can be adjusted to the needs of the user. It is also possible to remove attraction or repulsion completely. For example, it might not make sense to be attracted to obstacles, or to be repelled from a target.

2.2.4 Swarm Tasks

In their review paper, Brambilla et al. [2] categorise swarm tasks into four different categories:

- Spatially-Organising Behaviours
- Navigation Behaviours
- Collective Decision Making
- other collective behaviours

Spatially-organising behaviours are behaviours where agents need to position themselves or objects in certain places relative to each other or dependent on the environment. This includes aggregating everyone in a region, forming a pattern or a chain between two points, self-assembling (which means to connect physically to each other, with or without predefined pattern), and aggregating / assembling objects which they find in the environment.

In navigation behaviours, the agents coordinate their movement or the movement of objects. The authors list three examples: Collective exploration means to sense different parts of the environment, and then move through it on a beneficial path. Coordinated motion, or flocking, describes moving towards a common target in a swarm formation. Collective transport means moving an object that individual swarm members would not be able to move alone.

Collective decision-making covers two tasks: Agreement, where the agents should end up making the same choice, and specialisation, where they distribute tasks so that not everyone is doing the same thing.

The authors list three move tasks that do not fall in one of the above three categories: Collective fault detection means to detect swarm members that do not follow the same behaviour as the rest of the swarm (because they are faulty). Group size regulation leads to some swarm members leaving the swarm if they find out that the number of swarm members is too big for the task (or above a predefined threshold). Human-swarm interaction is the task of reacting to the input of a human controller, or to communicate information to them.

For almost every task, the authors present a source of inspiration that leads to the development of artificial swarms that solve such a task. Most of the inspirations come from biology, some from physics and molecular chemistry.

For example, collective exploration was inspired by ants. This inspiration lead to the development of Ant Colony Optimisation (ACO). The collective behaviour of ants a source of inspiration for many of the tasks.

Bayındır [1] gives a more detailed, but less structured review.

2.3 State of the Art

In this section, I describe previous approaches on automatically finding local swarm rules.

2.3.1 Grid Search

If the number of possible sensor inputs and actions of an agent, and therefore the complexity of the controller of that agent, is sufficiently limited, grid search can be a way to find the optimal controller for a given problem. Obviously, the larger the parameter space of the controller, the more resources grid search consumes, which can quickly become intractable. Grid search is a parameter sweep across all (meaningful) parameters of a controller. For every tuple of parameter values, the resulting global behaviour is observed and evaluated. Gauci et al. [5] have achieved swarm aggregation of very simple robots by using grid search.

Grid search is not a viable, or at least not the preferred option for the problem of this thesis, because the agents have vastly more complex sensors and movement capabilities.

2.3.2 AutoMoDe

Control software is mostly developed in simulation. There are significant differences between simulation and practise. If the learning algorithm that makes the control software has a too high representative power, what it learns will be very specific to the simulation – and therefore easily fail in real life. This is overfitting. A previous approach to reduce it was to introduce irregularities (changing conditions, noise) in the simulation and use training, test and validation datasets to estimate the generalisation. This has worked well, but mostly for scenarios with just one robot. It has not worked well for swarms, because the configuration space of a swarm increases exponentially with the number of robots. Therefore, it becomes infeasible to provide as much variation on the data as needed to properly learn all possible conditions and avoid overfitting. As a solution, Francesca et al. [4] propose to limit what the algorithm can learn. AutoMoDe [4] (for "automatic modular design") is an automated rule-finding approach based on probabilistic finite state machines. It has a choice of predefined behaviour building blocks, "states", which are linked to each other by pre-defined "conditions" for a transition, so to form the state machine. A state describes a parameterised robot routine. Conditions are also parameterised and describe whether to switch to another state. The set of available states and conditions is a design choice. The states which are actually included, the transitions, and the parameters of the states and conditions are optimised based on a quality metric of the emerging swarm behaviour. The optimisation algorithm and its initialisation are a design choice. As opposed to the approach of evolutionary robotics, which can learn very complex behaviours, the behaviours that AutoMoDe can learn are limited to those that can be represented by a state machine of the predefined states and conditions. It is further limited by constraints to the topology of the state machine, such as the maximum number of states or outgoing conditions of a state, which are design choices as well. This results in a simpler, coarser model, which cannot overfit that easily. AutoMoDe has two key assets on its side: Firstly, it produces behaviours that are well understandable for humans. This is because the state machine can be produced in a readable form, and, more importantly, because it combines elementary routines, therefore creating a hierarchical view of the behaviour. Its second asset it that it has been shown (at least in the version described in [4]) to produce behaviours that "overcome the reality gap" a lot better than a comparable approach of evolutionary robotics. That means, the behaviours perform similarly well on real robots as in simulation.On the other hand, AutoMoDe limits the behaviours that it can learn by the states, the building blocks that the designers design. Francesca et al. describe this as an asset to overcome the reality gap, but for more complicated situations it may also be a limitation. Since the states are routines that can be run for a longer time, combinations or concatenations of atomic actions, it is unlikely that the available states cover even close to all possible behaviours that a robot could perform. And if they do, then their number may quickly become quite large, so that long optimisation runs are required to reasonably explore the state space of the possible finite state machines. In order to learn anything, a behaviour has to be executed completely and by all robots, because the metric is defined on the observed swarm behaviour, and this metric is used to optimise the behaviour.

AutoMoDe at a glance

- Optimises the parameters of a probabilistic finite state machine, where each state corresponds to a pre-defined behaviour routine, with respect to a user-defined global quality function.
- Key assets: Human-readable rules. Reduction of "reality gap" (design choices limit the ability of the system to learn artefacts that are specific to a clean simulation environment)
- Key limitations: Limited set of building blocks -> limited options/flexibility. Many runs of the simulation are necessary.

2.3.3 Evolutionary Robotics

Evolutionary Robotics (ER) is a method where behaviours for individual agents are generated, then tested on multiple agents (in most cases, all agents run the same behaviour) and evaluated globally. The behaviours form the population of an evolutionary algorithm. This means, promising ones (with good scores) are chosen as parents for crossover and mutation. In this way, the algorithm produces behaviours that are similar to good behaviours that it has already seen, and behaviours that combine features of good behaviours in the hope of finding new behaviours that they have inherited the best parts of each parent. While ER does not constrain how the behaviours are represented, the most commonly used representation are Neural Networks [2], particularly Feed-forward Neural Networks and Recurrent Neural Networks. These have a set of parameters, which determines the quality of their predictions and which is what the evolutionary algorithm operates on. While the typical Neural Network representations of behaviours can model complex behaviours [4], this ability is often not only unnecessary [2], but also harmful due to the tendency for overfitting [4]. Since the method was originally intended for individual robots or small groups, the individual behaviour rules were supposed to be more complex than typical swarm rules, which are supposed to be simple, are. Other drawbacks are that it is hard to understand what a Neural Network has learnt, and that the evolutionary process consumes a lot of resources, while not guaranteeing a good result [2].

Evolutionary Robotics at a glance

- A Neural Network predicts the correct actions given a sensor input. The parameters of the Neural Network are optimised using an Evolutionary Algorithm.
- Key asset: High representational power: Can learn very complex behaviours and subtle nuances of the environment.
- Key limitations: Overfitting to simulation data, often does not work in practise. The rules are hard to understand. Computationally expensive.

2.3.4 Organizational-Learning-Oriented Classifier System (OCS)

Some tasks can only be performed by multiple robots working together, because the physical capabilities, not just the information or complexity, of a single robot are not sufficient. Division of labour is necessary here. In those cases, it might be difficult to keep a robust communication system in place that would allow a global evaluation of a behaviour, so that robots can learn what to do, particularly if different sub-tasks are completed in different places away from each other. Takadama et al. [10] used Learning Classifier Systems to have multiple robots learn when to perform which task in order to complete a bigger collaboration task together without communication or global evaluation. Their approach can be used for tasks where the individual robots recognise whether their contribution to a task is successful, still in progress, or causing a deadlock situation. This is called "self-evaluation". In the paper, this process is shown on the example of a construction task. Robots adaptively divide the labour. Deadlock situations can arise when a certain process, such as supply of a resource, is not attended for by any of the robots or if robots are in the wrong place, blocking others from performing their task there (chances of the latter one increase with the number of robots).

Rules are learnt using an adaptation of a Learning Classifier System, called "Organizational-Learning-Oriented Classifier System" (OCS). Unlike other methods for designing swarm behaviour rules, the OCS is applied on the level of individual robots. It is therefore not guaranteed that the different robots will learn the same behaviour rules. The OCS is initialised with simple rules that are progressively turned into more complex ones if necessary. The rules are selected corresponding to the local sensor data. Evaluation occurs when the task is complete (or in a deadlock) and is performed on the sequence of rules selected. It is therefore a Reinforcement Learning approach. While the paper is old and makes strong assumptions on the robots' capabilities to detect whether a task is completed, and while it is difficult to judge the success of their experiments, because they compared it against a centralised control system with questionable rules, I see that Learning Classifier Systems have been used to design control software for decentralised multi-robot systems. This strengthens the point of trying it again, with in a supervised learning setting.

OCS at a glance

- Used LCS in a reinforcement setting
- Key asset: Human-readable rules
- Key limitation: Long learning cycles (because of reinforcement learning). Old paper!

2.3.5 OCbotics

The most recent approach that used Learning Classifier Systems is OCbotics, by von Mammen et.al. [15]. They use an XCS (for "Extended Classifier System") variant, which they adapted to work in an observer/controller architecture. Instead of running the behaviour up to its end, evaluating it, changing parameters and running it again, they create and reinforce rules while they run the behaviour. A controller applies a rule to control the behaviour, while an observer evaluates the outcome of having applied that rule. Depending on that evaluation, the controller changes the parameters, i.e. weights, of the rule. In this way, changing the parameters of the rules leads to steering against unfavourable developments. This is called "reactive behaviour". In order to evaluate the quality of the current behaviour, goals along which the evaluation is conducted need to be predefined. Unlike typical LCS implementations, new rules are not directly added to the rule population, but first tested in an offline simulation setting to make sure they are mature enough and will not do damage to the system. The creation of new rules leads to what is called “evolved behaviour”. Finally, a third layer of observation and control allows for user interaction: The human supervisor can change the quality criteria manually. This approach reduces a key drawback of reinforcement learning, the Spatial Credit Assignment problem, by evaluating the behaviour much closer to real time and much more locally. The reactive behaviour is evaluated per prediction. The evolving behaviour is not directly evaluated in terms of quality measures, but only in terms of safety. Drawbacks are that the environment might need to be circular in time, since otherwise the development of a good behaviour might take longer than solving the task with a less-than-ideal behaviour. It’s also noteworthy that the authors designed many rules manually upfront, since otherwise the initial behaviour would not have lead to a state where the learning could properly kick in. These rules were then automatically refined or complemented.

OCbotics at a glance

- An LCS improves and evolves behaviour of a live system.
- Key Asset: Human-readable rules, high-level human-defined goals.
- Key Limitation: Only works if the environment affords for longer runs of the behaviour. Needs an initial set of rules.

2.3.6 Manual Iterative Rule Crafting

Manual Iterative Rule Crafting, also known as "code-and-fix approach" [3], means that a designer creates rules based on their intuition, and improves them iteratively. According to a review paper from 2013 [2], it was still the most commonly used approach to find local rules at least at that time. In that paper, Brambilla et al. divide the approaches into two main categories: Those that use Probabilistic Finite State Machines and those that use Attraction- and Repulsion-Functions (see section 2.2.2). They also mention their own work on a top-down method that uses formal model checkers [3]. Since the rules are manually made, they are usually also in a human-readable format, and the designer understands them. However, the whole process can take long, and it can lead to different (quality of) results depending on the intuition and experience of the designer.

Manual Iterative Rule Crafting at a glance

- The designer makes some rules, sees what emerges, improves the rules.
- Key asset: Once it works, the designer knows what the swarm is doing.
- Key limitation: Slow, hardly reproducible.

3 Global-Local Swarm Learning

3.1 Supervised Learning: How my approach is different

The approaches I discuss above have one thing in common: Automatically or manually, they create rules, run them, check how good they are, then try to improve them based on the results, run the improved ones again, and so on. The rules are evaluated by a predefined quality measure that, ideally, is based on the contribution of a rule to the global success of the behaviour. The objective function is the quality of the behaviour, which has to be defined by the designer. Except for OCbotics, for one learning iteration, the whole behaviour has to be simulated and evaluated. In approaches based on reinforcement learning, the update to the rules that comes from the evaluation of one iteration will not improve the behaviour much, because it is hard to determine how to distribute the reward of a good/bad behaviour among the rules, actions and agents. This problem is called "Spatial Credit Assignment".

In this thesis, I present Global-Local Swarm Learning (GLSL), which is an entirely different approach: I use supervised learning. This means, the rules are evaluated by how well they resemble the connection between sensor values and actions of existing data. I cause a desired behaviour (that performs well), observe it, and then learn to perform actions like in that desired behaviour, independent of a global evaluation. The objective function is minimising the difference between the input-output-mapping in the training data and the mapping that the algorithm learns. The desired behaviour that is observed can be achieved more easily than the actual outcome behaviour, because of a trick: Using more information than the learnt behaviour is allowed to use. In this case, I want to find a behaviour that makes a swarm follow a leader robot through a tunnel, while not all robots can see the leader. The desired behaviour that is observed is created by actually giving all robots the information where the leader, where the tunnel, and where everything else is. A global evaluation is still used to measure the success of learning, but not for improving learning itself in the process. This global evaluation includes quantitative comparison between the observed and the learnt behaviours. The training data samples are the sensor-action-tuples observed at every time step for every robot during the run of the behaviour with more information. The actions are limited to what can be performed, or at least decided to do, in one time step - so essentially a close-by position where to go next. Unlike in AutoMoDe, routines are not learnt as such. The role of the "expert", as the designer is called in [4], is also different: In AutoMoDe, the expert defines the states and conditions, which are task-independent, and the optimiser, the metric and constraints to the behaviour, which are task-dependent. In supervised learning, the expert's main task is to create the desired behaviour with the help of giving the robots more information. This is easier than Manual Interative Rule Crafting, but at least for the task of following a leader through a tunnel, it is still not trivial. The training samples are fed into a supervised learning algorithm. By employing this approach, the following benefits can be expected:

- My approach generates more data per run, compared to the methods where the behaviour is optimised based on the quality of the rules. Therefore, fewer runs of the behaviour are required, which potentially saves time.
- My approach is more general than AutoMoDe, because it has less design bias, since no component modules need to be chosen by the designer.
- My approach is more understandable than Evolutionary Robotics, because it produces human-readable rules as output. It is also less prone to overfitting, because of its internal generalisation pressure.

Potential disadvantages include the discretisation error, which could lead to a significant mismatch between the training data and the predictions, or to a lot of effort in finding a good number of classes. However, I do not expect this to be a major problem, because at least for simple experiments, rules could be written by hand that would lead to a proper behaviour. So if this is not achieved, the discretisation error will unlikely be the major cause. The other problem, that no rule matches, can occur, but is not particularly problematic, since the environment continues to change (in some experiments), so there should not be a deadlock.

3.2 Overview

The software I compute my data with consists of four parts:

1. the "Data Collecting Simulation"
2. the Preprocessing of the data
3. the Learning Classifier System (LCS)
4. the "Evaluating Simulation"

The LCS software ExSTraCS is able to read both the input data and the configuration from text files. The learning phase also outputs a file, which includes the rules it found that map the input to the output, including the statistical information about each rule, which are shown in 2.1, and are explained in section 2.1.Therefore, the communication between the parts works by writing and reading text files.

3.3 Data Collecting Simulation

3.3.1 Why Simulation?

I simulate the swarm behaviour, rather than deploying it on actual robots (such as Spheros or quad-copters). This has the following advantages:

- Availability: The Spheros that were available are also needed for other projects. They are also few in number, whereas the simulation allows to scale the experiments up to much larger numbers of robots. A similar argument is applicable for the arena, which is much more flexible in simulation.
- Simplicity: Before the experiments, it was not clear how well the LCS would be able to learn the behaviour. To start with the most simple scenarios, artefacts of the real world such as noisy sensors or friction on the floor are not desired.
- Customisability: Spheros inevitably turn in order to change their direction of movement. In simulation, this can be avoided. Parameters such as inertia can be set.
- Safety: While Spheros are rather robust, in the design of the Data Generating Simulation, there have been frequent collisions.

The Unity game engine comes with convenient geometrical functions to facilitate the simulation. It also enables me to use visualisation tools developed by Thomas Seidelmann [9].

3.3.2 The Agents

Shape

The simulated robots are of circular shape. Their radius is adjustable. For my experiments, it is set to 0.2 meters.

Movement Capabilities

My simulation is only two-dimensional. The robots can only move in the plane. In this thesis, swarm behaviour is limited to the movement of robots that is influenced by information from other robots and the environment. I do not consider other swarm robotic tasks, such as picking up items, since they do not have such capabilities. The robots can also not change their orientation. This is not needed, since they can move to any direction.
The movement of the followers is subject to inertial forces. In each time step, they are projected to a point in extension of their previous movement. This point is their current position plus their previous movement vector multiplied by an inertial factor. The inertial factor is set to 0.93. This value was determined experimentally: Higher values lead to unstable behaviour where the followers became so fast that they were unable to perform swarm coherence. This happens because the followers are often not able to steer against the inertial movement far enough so to reach a place close to where they would want to be. Lower values made the movement painfully slow. From that projected point, the copters can steer, as shown in figure 3.1. Their steering (and therefore acceleration) is limited. The limitation factor is inspired by quad copters that can tilt to a maximum of 30° and should stay at the same level of altitude. This works out to $g \cdot tan(30°) = 5.664\frac{m}{s^2}$. The speed is also capped to $5.664\frac{m}{s}$, so the followers can accelerate for at most one second into the same direction. Since the simulation recomputes the position and speed at certain time intervals, it makes sense to limit the speed. Otherwise, the robots would easily jump over regions where their movement would be diverted, which could, in extreme cases, go as far as "tunnelling" through walls without noticing a collision. The decision to fix this time step at 0.0168 seconds of simulated time

is based on the average of the simulation where the attraction and repulsion parameters were originally calibrated on. A follower can therefore only move at most $0.0168 \cdot 5.664m$, which is around 9.5 cm, less than a quarter of its diameter, before the movement is adjusted to the new constellation of neighbouring robots and walls.

In addition to the inertial movement and the steering command, a small ran-

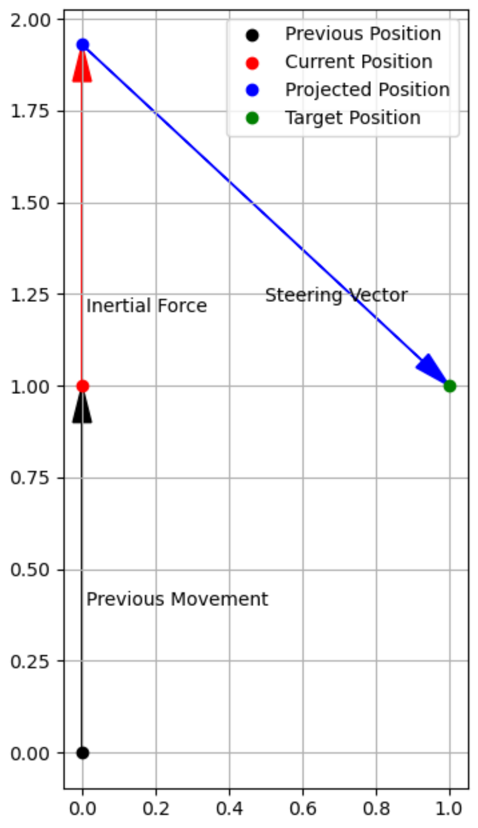


Figure 3.1: This figure shows how a robot of speed 1 and inertial factor 0.93 steers against the inertial forces. If the acceleration is smaller than the length of the steering vector, it will end up on the steering vector, but closer to the projected position.

dom movement is applied. This is to make sure that robots don't get stuck in some local optimum in front of the tunnel in certain experiments. The magnitude of this random movement is less than $\frac{1}{100}$ of the maximum magnitude of the steering vector.

$$PreviousMove = CurrentPosition - PreviousPosition \tag{3.1}$$

$$Steering = Target - PreviousMove \cdot InertialFactor \tag{3.2}$$

$$CappedSteering = \begin{cases} Steering, & \text{if } \|Steering\| \leq MaxAcc \\ \frac{Steering}{\|Steering\|} \cdot MaxAcc, & \text{otherwise} \end{cases} \tag{3.3}$$

$$TotalMove = PreviousMove \cdot InertialFactor + CappedSteering \tag{3.4}$$

$$CappedMove = \begin{cases} TotalMove, & \text{if } \|TotalMove\| \leq MaxSpeed \\ \frac{TotalMove}{\|TotalMove\|} \cdot MaxSpeed, & \text{otherwise} \end{cases} \tag{3.5}$$

$$NewPosition = CurrentPosition + CappedMove + RandomMove \tag{3.6}$$

Sensors

To make learning easier, the sensors can tell apart whether what they sense is a wall or a copter. This means that we have two values per sensor: The distance to the nearest copter within range, and the distance to the nearest wall within range (although in the simpler experiments that disregard walls, sensors only report copters, not walls). At first, the idea was to record exactly where the copters/walls are, both in terms of angle and distance. This would have given the LCS a lot of information. However, compiling such data into a tabular data frame is not easy if it should have the same number of columns regardless of the number of copters or walls within range. To make the data frame more consistent in this respect, I decided to have a discrete and limited number of sensors at different angles, each of them only reporting the distance to the closest copter and wall within their range. Therefore, the number of columns is determined by the number of sensors only, and it stays constant. The sensors can be parameterised. Their most important parameters are their number (n) and the maximum distance that they can observe (d). I decided to spread the sensors around the copter in equal angles, and the angles they can observe do not overlap. That means the angles are $\frac{360°}{n}$ apart around the circle and each one can observe an angle of $\frac{360°}{n}$, illustrated in figure 3.2. If the nearest sensed object within the angular range of a sensor is further away than the maximum distance d, then the value of d is returned. The LCS implementation can supposedly also work with missing values, but it is not totally clear how to do that, and since not sensing anything nearby is important information, the described approach is more reasonable than returning no value in that case.

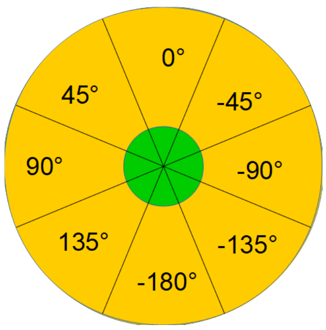


Figure 3.2: This figure shows which which area falls into the range of which sensor. In this example, there are 8 sensors and their range (yellow circle) is 3 times as long as the radius of the follower (green circle). This is for illustration purposes, so the follower can more easily be seen. In my experiments, the radius of the follower is 0.2 meters and the range is 8 meters. The yellow circle would therefore have a radius of 8.2 meters.

The generated data

In every time step and for every follower, the simulation stores one data sample. The attributes are the values that the sensors measure. The predicted value, or endpoint, is the position where the robot was trying to steer to, relative to it self. It is not the steering command. Actually, this is an important design decision. I think a robot should take the movement that will be caused by inertia into account when deciding where to steer. The movement of the robot is limited to a certain acceleration ($g \cdot tan(30°) = 5.664\frac{m}{s^2}$), not a certain speed (since we ignore air drag). If we have an inertial factor of more than 0, it can lead to a situation where the robot overshoots its goal position if it continues to steer towards it. This happens if the movement in the previous time step already went a long way towards the goal. As an example, assume a 1-dimensional space and an inertial factor of 0.3. In time step 0, we are at position 0. In time step 1, we are at position 10. In time step 2, we want to go as far as position 12. If we don't steer at all, we would already go to position 13 due to inertia: Our previous movement was 10, our inertia factor is 0.3, so we would go 3 more towards the same direction. Obviously, we have to steer into the opposite direction with an amount of -1, so we can reach position 12.

To implement this, we require every robot to know its speed in the previous time step, as well as its inertial factor.

If steering commands should be used as the endpoints for learning, we also have to include the recent memory of movement into data points. This is necessary, because the steering depends strongly on the previous movement. It might cause problems though, if the connection between previous movement and steering is more prominent in the data than the connection between the sensor values and the steering. In that case, no valid behaviour would be learnt. Therefore, I conclude that the vector where the copter wants to go relative to its current position needs to be used as the endpoint, not the actual steering vector.
The LCS implementation that I use can only handle nominal predicted values, since otherwise it would not know how correctly a rule predicts. Therefore, the action of each data point is converted from a vector of two floating point numbers into one integer number. Since the steering vector is limited in length, there is a minimum and a maximum steering vector component for each of the two axes. This range is subdivided into cells and then numbered, as shown in figure 3.3.

Swarm Behaviour

In this thesis, swarm behaviour is limited to the movement of robots that is influenced by information from other robots and the environment. I do not consider other swarm robotic tasks, such as picking up items. The robots can also not change their orientation. This is not needed, since they can move to

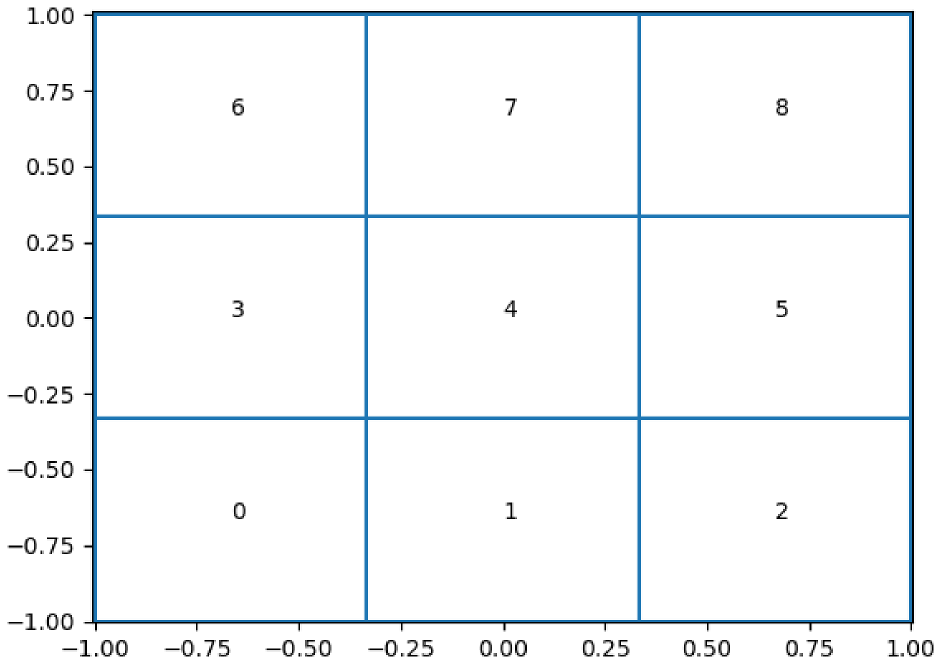


Figure 3.3: The vector where the follower wants to move falls within the range of x and y between -1 and 1. This space is subdivided, in this case into 3 classes per direction, which are 9 classes overall. This figure shows how the vector is mapped to a class label.

any direction.
The local movement of a swarm member is to a large extent determined by functions of the position of the robot relative to other robots and obstacles. These functions are called "attraction- and repulsion-functions" and are described in section 2.2.3. The third element, alignment, which attraction- and repulsion-functions do not control, does not play a role here, because the robots cannot rotate.

There are three elements to the swarm behaviour in the data generating simulation:

Swarm Cohesion and Separation

Swarm cohesion means that the swarm members do not disperse. Swarm separation means that they do not collide with each other. We want to have both. They should maintain a comfortable distance to each other if possible. Since the simulation runs in discrete time steps, the maximum speed and acceleration of the followers are limited. Therefore, overly large attraction forces do not have a big effect. The key consideration is to keep them robustly separated, but still cohere strong enough to pull each other through the tunnel. In order to do that, they use a function for linear attraction and bounded repulsion, which repels if the distance is uncomfortably small and attracts if it is too large. Applying this function pairwise to all robots in the swarm, the effect is a regular grid of robots. In this case, the comfortable distance is around 1.875. The force that moves individual i, caused by its relationship to j, is defined by equation 2.2:

$$f(i,j) = -(x_i - x_j) \cdot (a - b \cdot e^{\frac{-\|x_i - x_j\|^2}{c}}) \tag{3.7}$$

where x_i is the position of individual i. a, b and c are parameters that have been fine-tuned. To maintain cohesion and separation in presence of a tunnel, the parameters for this function are set to:

- a: 0.3
- b: 105
- c: 0.6

The comfortable distance is therefore around 1.875 and there is a strong force trying to make sure that the robots don't come much closer than that.

Wall Avoidance

The consideration with wall avoidance is to keep the followers robustly away from the walls, but not so far away that they would not pass the tunnel anymore. Robots are repelled from walls inversely proportional to the square of the distance to the wall.

$$f(i,j) = -(x_i - x_j) \cdot (\frac{b}{\|x_i - x_j\|^2}) \tag{3.8}$$

where b is a repulsion factor. b is set to 5.2. This function is plotted in figure 3.4.

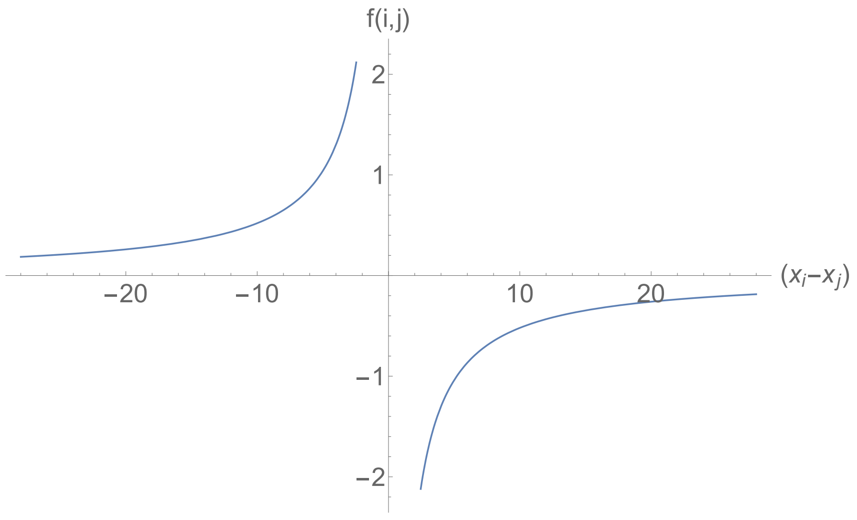


Figure 3.4: This is a plot of function 3.8 in the same format as figure 2.2, with $b = 5.2$.

Following the Leader

To follow the leader and aggregate around it, but not collide with it, I use the sum of the linear attraction + bounded repulsion function (2.2) and the following function to repel when close:

$$f(i,j) = (x_i - x_j) \cdot k_r \cdot e^{\frac{\frac{-1}{2} \cdot \|x_i - x_j\|^2}{r_s^2}} \tag{3.9}$$

where k_r is the magnitude of the repulsion and r_s is the distance where the repulsion is strongest. r_s is set to values significantly smaller than the comfortable distance of function 2.2. Therefore, this repulsion hardly affects the comfortable distance. Specifically, k_r is set to 50 and r_s to 0.2, shown in figure 3.5. The parameters for function III to ensure cohesion with the leader are set to:

- a: 1.2
- b: 20
- c: 0.6

3.3.3 The Arena

The arena consists of a rectangle of walls and two tunnel walls that protrude from the center of the top and bottom walls towards the center of the arena.

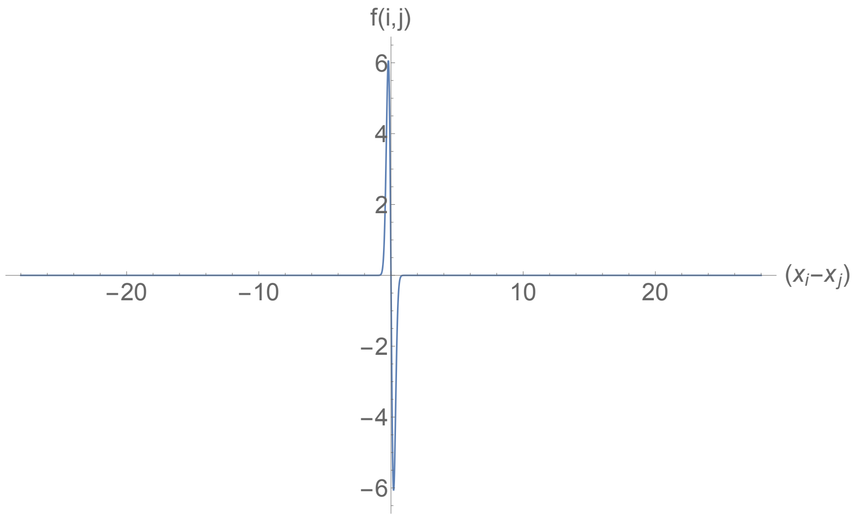
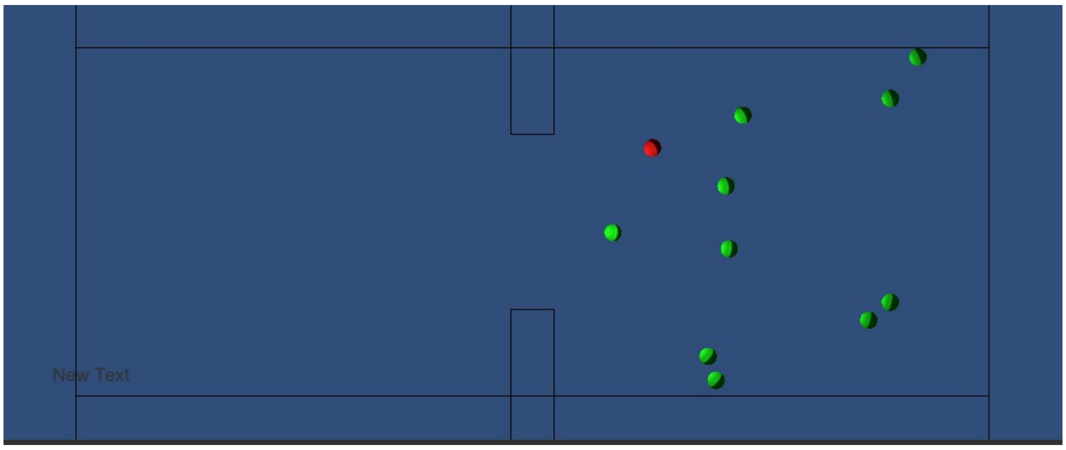


Figure 3.5: This is a plot of function 3.9 in the same format as figure 3.4, with $k_r = 50$ and $r_s = 0.2$.

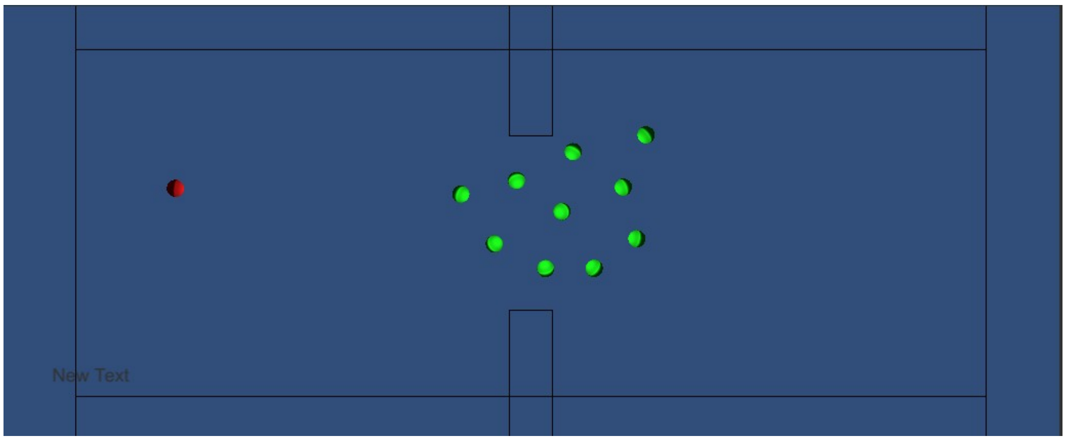
In the standard configuration, the distance between the left and the right wall is 21 meters and the distance between the upper and lower wall is 8 meters. It is, however, also possible to move the walls far away (1000 meters), for experiments where walls should not have an influence. The tunnel can be adjusted in length and width. In all cases the arena is symmetrical in both its vertical and horizontal axes. That means that the tunnel is in the middle of the arena and both chambers (on each side of the tunnel) are congruent to each other. It is also possible to set the tunnel to a width of 8, which means there is still a walled arena, but effectively no tunnel. The walls have a thickness of 1 meter. This means that a copter can again be repelled from a wall after having left the arena, but this can be disregarded, because in the real world, a robot would have crashed when trying to move through a wall. Illustrations of how the swarm passes the tunnel in the arena are shown in figure 3.6.

3.4 Preprocessing of the data

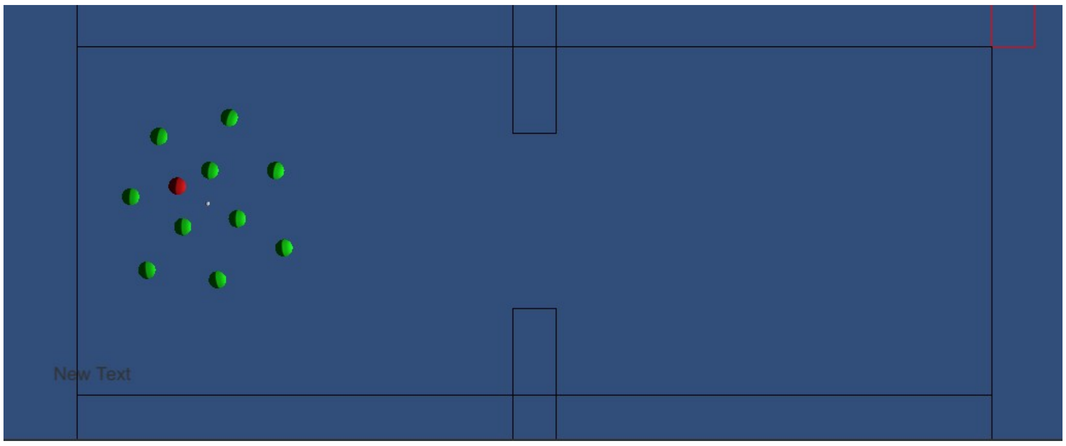
The preprocessing corrects data where values were changed due to localisation problems: Sometimes data was multiplied by 1000 by the LCS on reading it in, because decimal seperators were misinterpreted. This is reverted in preprocessing.
The second preprocessing step is to split the data into a training and a test



(a) Swarm before passing the tunnel



(b) Swarm while passing the tunnel



(c) Swarm converged after having passed the tunnel

Figure 3.6: Screenshots of the animation of the data generating simulation. The red circle is the leader, the green circles are the followers. The black lines are the walls. The red square in the top right corner indicates that all followers have passed the tunnel.

set, where the training set is 75% and the test set is 25% of the data. In this process, the data is also shuffled. This would not have been necessary, since the LCS shuffles the data internally at the beginning of the run, so to make sure that biases based on the order of the data samples are eliminated.

3.5 LCS parameters

The LCS implementation ExSTraCS offers many options to tweak the learning performance. In the scope of this thesis, not all of them could be properly explored.

The most important parameters that are changed here are the number of learning iterations and the limit to the sum of the numerosities of the rules (referred to as N). The number of learning iterations is actually a set of numbers, so multiple numbers can be specified for the same run of the LCS. An iteration means the processing of one training sample. After each given number of iterations, training and testing accuracy scores are computed and rules ("rule population") are written to a file. The final number of iterations has been set higher than the number of training samples, which makes sense, so the algorithm can see all training samples, and in many experiments higher than a multiple of the number of training instances. This means that the LCS sees training instances multiple times. This is useful, because the genetic operators (mutation and crossover) continue to generate new rules, and these new rules should have time to "gain experience", which means to find out how good they are, and, if they are good, create offspring rules in the genetic algorithm. According to the implementers, the number of learning iterations is one of the most critical run parameters and the default values are largely arbitrary [11]. This is why I try different ones in the experiments.

As described in chapter 2, each rule comes with an integer that tells its numerosity, which can have increased if the rule has subsumed other rules that were more specific, but not more accurate, or if the genetic algorithm has generated a rule that already existed in the rule population. N is the limit of the sum of all numerosities in the rule population. If N is too small, the complexity of the experiment cannot be represented in the rules, because not enough rules are available. If N is too big, the generalisation pressure decreases, because poor rules are less likely to be removed from the rule population. This can lead to overfitting. Particularly, if N is higher than the number of the training data instances, it is possible to have one rule per item, which, obviously, means massive overfitting (although the algorithm computes an upper limit to the number of specified attributes). The default value for N is 1000. I try this for several experiments to make sure that a lower value for N is not the sole cause for bad performance (in chapter 4.2), but in general, this value seems very high, given that humans could write well-working rule populations with a fraction of this number of rules. If more than 260 rules are allowed, the excess ones in one preliminary experiment (like in section 4.3) all had a match count of 0. These rules can have an impact on the prediction, because their accuracy is initially set to 1. In the user guide to ExSTraCS [11], the remaining parameters are grouped into the following four categories:

- Dataset Parameters
- General Run Parameters
- Supervised Learning Parameters
- Mechanism Parameters

The dataset parameters are determined by the dataset. They include things like which dataset to load, where to save the output, how ID and class columns are labeled and how high the cardinality for a categorical variable can be at most. It also includes the option to internally perform cross-validation. However, judged by the low runtime and the output, this option does not seem to work. Finally, it includes the option of fixing the random seed.

The most important general run parameter is the number of learning iterations, described in detail above. The other general run parameters tell the algorithm how often it should report output and where to store it.

The most important one of the Supervised Learning Parameters is N, which is already described above. Further ones are:

- *nu*, which defines how to compute the fitness (which is the criterion for selection in the genetic algorithm) from the accuracy (which is the number of matches where the correct class was predicted over the number of matches in total).
- *chi*, the crossover probability
- *upsilon*, the mutation probability

- *theta_GA*, the lower age limit for rules to be included in the genetic algorithm
- *theta_del*, the lower match count limit for rules to be deleted
- *theta_sub*, the lower match count limit for rules to subsume other rules
- *acc_sub*, the lower accuracy limit for rules to subsume other rules
- *beta*, the learning rate for the update of the average size of the correct set per classifier. The effect of this in unclear, as no further description is provided, neither in [11] nor [13]. The latter one does not even take the average size of the correct set into account at all.
- *delta*, a threshold for deciding which weight computation should be used in the deletion mechanism. If a rule has good fitness, but small numerosity (that means if $\frac{fitness}{numerosity} \geq delta \cdot meanFitness$), then $deletionWeight = aveMatchSetSize \cdot numerosity$ instead of the usual $deletionWeight = \frac{aveMatchSetSize \cdot numerosity^2 \cdot meanFitness}{fitness}$ [13].
- *init_fit*, the fitness value that is initially given to new rules that are generated by the covering mechanism or by mutation.
- *fitnessReduction*, the fitness value that is initially given to new rules that are generated by the crossover operation.
- *theta_sel*, the fraction of the correct set that will be included in the tournament selection for choosing parent rules in the genetic algorithm.
- *RSL_Override*, the maximum number of attributes that are allowed to be specified in a rule. The default is 0 and means that the algorithm automatically computes a reasonable upper limit.

All of these Supervised Learning Parameters (except for N) are kept to their default values, which are listed in the user guide [11] and also in the largest configuration file that is also available in the repository.

Mechanism Parameters specify which mechanisms are applied and how. These are:

- *doSubsumption*, which defines whether or not to apply subsumption. Applying subsumption makes sense to keep the rules more general and reduces their number. In other words, it helps to keep the rule set more easily interpretable for humans. Therefore, I activate it.

- *selectionMethod*, which defines whether to use tournament or roulette-wheel selection to choose parents in the genetic algorithm. The implementers recommend tournament selection, so I keep this default.
- *doAttributeTracking*, which defines whether or not to track how often attributes were specified in the rules in the correct set of each training instance. This has no effect on the learning, but it can give insights into which attributes are important for which training instance. I did not use these insights, but still keep it active. This is also the prerequisite for attribute feedback.
- *doAttributeFeedback*, which defines whether or not to use the scores of attribute tracking of a random previous instance to guide the genetic algorithm on which attributes should rather be specified together in one rule. Attributes with a high attribute tracking score tend to get specified in the same rule, so their combination can be tested for effectiveness [13]. Attribute feedback guides probabilistically and mostly becomes active towards the end of the learning iterations. Urbanowicz and Moore [11] claim that attribute feedback made the learning faster, because the most meaningful attributes will be combined in the same rule. Therefore, I keep it active.
- *useExpertKnowledge*, which defines whether some initial information on which attributes are probably most predictive should be applied. I deactivated this, and set all other "expert knowledge" parameters accordingly.
- *doRuleCompaction*, which defines whether inexperienced or badly performing rules should be removed in the end. I keep this active, but I don't use the new smaller rule population. This is because the algorithm can only do that at the end of the learning, which makes a parameter sweep across different numbers of learning iterations more difficult, because I would have had to reload the rule population for every new number, instead of just reporting it more frequently. It is also not exactly clear how the rule compaction mechanism decided whether to keep a rule or not, or what else it does to the rules. A motivation for considering the compacted rule sets would be if the final accuracy scores after rule compaction were consistently higher than without, but this is not the case. Further parameters *onlyRC* and *ruleCompactionMethod* allow to deactivate learning and only do rule compaction, and to choose how to

do rule compaction. The options are, however, not clearly described or defined without looking deep into the source code.

- *doPopulationReboot*, which defines whether an existing rule population should be read in or whether the learning should start with an empty rule population. Using an existing rule population would be useful to do rule compaction on intermediate output, for example. Still, I do not do it for this thesis.

3.6 Evaluating Simulation

At first, the Evaluating Simulation has to read in the rules that the LCS produced. In the evaluating simulation, the leader robot performs its movement just the same way as in the data collecting simulation. Both the leader and the followers also start at the same positions where they started in the data generating simulation. At each time step (which again has a length of 0.0168 seconds), each follower has to decide which action to perform. This is decided by taking the data that its sensors currently sense and predict an action according to the rules. It takes all the matching rules, groups them by the action that they predict, and chooses the group with the highest sum of accuracy values. The class is converted to an action by taking the center of the region of actions that has mapped to that class (that is the center points of the blue squares in figure 3.3). If no rules match, the follower does not move.

3.7 Evaluation of the behaviour

There are two measures to determine how successfully a behaviour was learnt:

1. Accuracy of the classifier.
2. Deviation from the original behaviour.

Since the data from the data generating simulation is split into a training and a test set, it is easy to determine the accuracy of the classifier on the test data. Both the standard and balanced accuracy scores are reported. Standard accuracy is the number of samples that was correctly classified, over all samples.

Balanced accuracy is the average of the standard accuracy scores per class [13]:

$$balAcc = \frac{\sum_{i=1}^{n} \frac{sensitivity_i + specificity_i}{2}}{n} \tag{3.10}$$

The LCS implementation reports both automatically.

3.7.1 How to compare swarm behaviour?

In order to tell by how much a swarm deviates from the behaviour it was supposed to learn, I need a measure of comparison between swarm behaviours. Let A and B be two swarm behaviours. A swarm behaviour is a mapping of a situation to an action. For this purpose, I will call the action of both behaviours "predictions". Then the deviation of A from B is the sum of the euclidean distances from the predictions of A to the predictions of B for the situations encountered while performing A, divided by the number of robots and time-steps. This is not necessarily symmetrical, because the agents move only according to one of the two behaviours, and therefore get into situations which they would possibly not have seen if they followed the other behaviour.

Let $s_{i,t}$ be the sensor input for follower i at time t while following the behaviour A. Further, let $P_X(s)$ be the predicted class of behaviour X for sensor input s. N is the number of followers and T is the point in time for which the current comparison score is computed. Then:

$$ComparisonScore(A, B) = \frac{\sum_{i=1}^{N} \sum_{t=1}^{T} \| P_A(s_{i,t}) - P_B(s_{i,t}) \|}{N \cdot T} \tag{3.11}$$

For the scenario in this thesis, this means that the simulation sums up the difference between what a follower agent does based on what it has learnt, and what it would have done in the same situation if it had followed the global rules that it used to generate the training data. The original behaviour, as performed in the data generating simulation, is prediction B in this case. These values are averaged over all agents and time steps passed. The comparison score will change over time during the evaluating simulation. A value can be taken as final once the center of the swarm has converged or there have been collisions (because it should not matter how a follower behaves after a collision). For experiments where the tunnel plays a role, they should also have passed the tunnel in order to count as converged.

For the experiments, even for the best possible learning, the comparison score will not be 0, because of the discretisation error. The expected value for optimal learning depends on the size of the regions that map to the same class. Assuming that every prediction appears equally often in the behaviour that gives continuous output (which is not actually the case), the optimal comparison score for the side length l of the regions that map to the same class can be computed by:

$$ComparisonScore_{opt}(l) = \int_{-l}^{l} \int_{-l}^{l} \sqrt{x^2 + y^2}\, \mathrm{d}x\, \mathrm{d}y \tag{3.12}$$

For the discretisation shown in figure 3.3 and used in most experiments, that is 0.113.

4 Experiments

4.1 The most simple experiment

I start with an experiment that is simpler than following a leader through a tunnel, so to validate that my approach works at all. This experiment disregards walls entirely and has only one leader and one follower. The leader is moving in a circle (counterclockwise) and the follower is following while keeping a comfortable distance of 1.3 m. The leader moves with a speed of 0.0324 meters per time-step (1.93125 meters per simulated second) and the circle has a radius of roughly 3.28. Since the leader doesn't stop, the simulation has to be cut off at a certain point - in this case, after 1 minute, which makes 3573 time steps and roughly 5.6 circles. This is rather fast. The follower is not able to keep up with the leader fully - it keeps a larger distance than what would be the comfortable distance. However, such speeds are necessary for the second experiment (described in section 4.2), so I use them here as well for comparability. For all experiments, the leader starts at the same position ($(2.8, 1.7)$) in order to improve comparability. This point was chosen, because from there, the leader can pass a tunnel of length 1.3 and width 1.7 (which was the tunnel the data generating simulation was calibrated on) without additional way points. For all experiments with just one follower, the follower starts at $(4.8, 2.4)$. This is a random value, but the seed is fixed to improve comparability.

The main reasons why this problem is considered simple are:

- It is possible to hand-craft a rule set with very few rules (2 per class).
- The classes are relatively evenly distributed in the data set, because the movement is symmetrical.
- It is easy to generate more data if necessary by just running the data generating simulation for a longer time. This is not easily possible with simulations that end in convergence.

A schema that shows the paths that both agents travel in the data generating simulation is shown in figure 4.1.

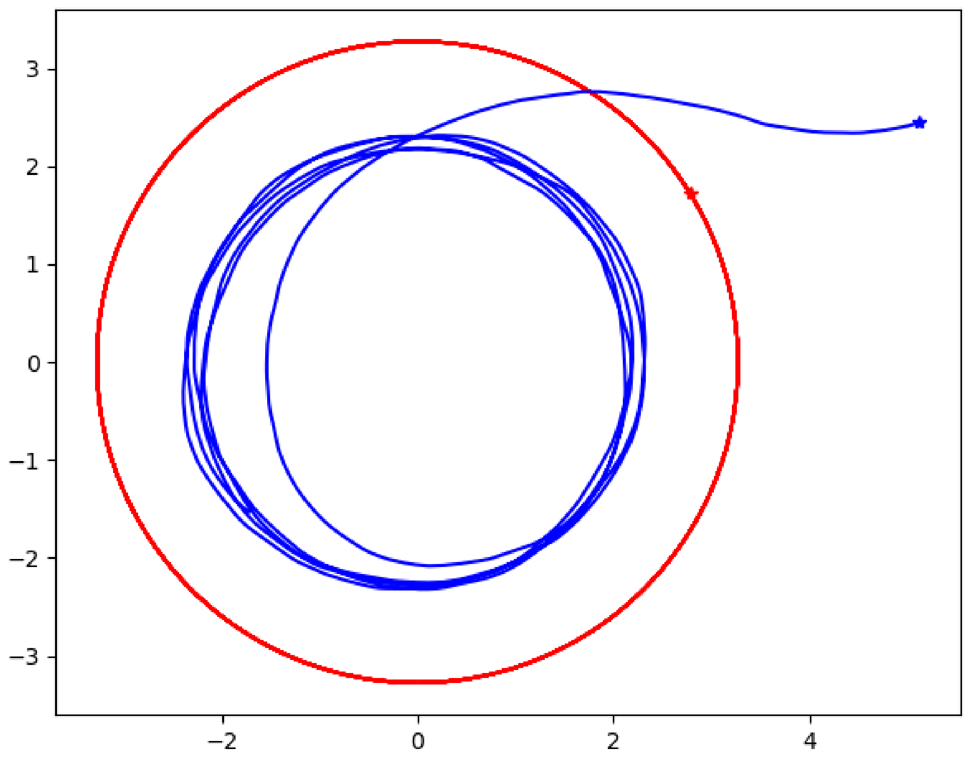


Figure 4.1: This graphic shows the movement in the data generating simulation of the experiment described in the sections 4.1 and 4.2. The red line shows the movement of the leader, the blue line the movement of the follower. The star symbols indicate where the respective agents have been initialised.

4.1.1 Parameters

Sensors and Actuators

To keep the problem small, the follower only has 4 sensors and can move into four directions.

LCS

The total numerosity of the rule population is set to 260. This number was determined in a more difficult experiment, in which I allowed more rules, and where the excess ones had a match count of 0. This means that they were generated by the evolutionary algorithm, but were so specific that they never matched again. It indicates that more rules are not necessary. In fact, in this experiment I expect that fewer rules are enough. For the number of iterations, I perform a parameter sweep, trying 5000, 10000, 15000, 20000, 25000, 30000, 40000, 50000, 100000, 500000 and 1000000 iterations. I assume that the performance varies more in early iterations, which is why there are more small and fewer large values in this list.

4.1.2 Results

The accuracy scores are all close to, mostly just below 0.5. After 1000000 iterations, the follower indeed seems to follow the leader back and forth on the lower-left side of the circle, but eventually collides with the leader. In all the other runs, the follower either escapes and moves away from the circle, or it comes into a situation where no rules match. In both cases, it does not collide nor follow. If it escapes, the comparison score continues to rise bounded by its maximum value, because its action is compared against moving towards the leader. If it does not find any matching rule, then the comparison score is distorted, because while no rule matches, no comparison score is added. A summary of the results can be found in table 4.1.

The total numerosity reaches 260, but there are indeed a number of rules with a match-count of 0, as expected.

Iterations	Std. Acc.	Bal. Acc.	Collides with Leader	Follows	Periodic	Comparison Score
5000	0.49	0.49	no	no	no	n/a
10000	0.47	0.48	no	no	no	rising
15000	0.46	0.47	no	no	no	rising
20000	0.50	0.50	no	no	no	n/a
25000	0.46	0.47	no	no	no	rising
30000	0.46	0.47	no	no	no	n/a
40000	0.49	0.49	no	no	no	n/a
50000	0.46	0.47	no	no	no	rising
100000	0.50	0.50	no	no	no	n/a
500000	0.46	0.47	no	no	no	rising
1000000	0.47	0.48	yes	shortly	2 times	1.06

Table 4.1: The summary of the results of the most simple experiment (section 4.1). "Std. Acc" stands for "Standard Accuracy", "Bal. Acc." for "Balanced Accuracy".

4.1.3 Explanation

One problem is that the action classes and the sensors are not aligned to each other. In fact, they are shifted by half. That means, there is a sensor at -180, -90, 0 and 90 degrees, but the robot can only move to -135, -45, 45 and 135 degrees (the degrees are global, as described in section 3.3.2). As a comparison, I also wrote a set of rules by hand. Here, the follower would move to the next direction in counterclockwise direction from the direction where it sees the leader. This means, if the leader is seen in the sensor of direction -90, the follower moves towards -45. If the leader is closer than 1.3 meters, it moves into the opposite direction. The comparison score rises very slowly until around 1.068. There are no collisions.

4.2 More sensors and actions

The same experiment was performed, but with 8 sensors and 9 possible actions. The advantage is that now the sensor directions match the actions. The disadvantage is that the speed of the leader must not be too low, since otherwise the action of the follower will fall into the central class, which is decoded as "no movement". This is why the experiments are conducted with a rather fast leader, even though the follower is not fully able to adhere to its comfortable distance. The distance it keeps is visibly larger, but it does not grow indefinitely. The follower still follows the leader in a circle, just like in the previous experiment and just as shown in figure 4.1. The total numerosity N is kept at 260, for the same reason as in the previous experiment (4.1). For the number of iterations, I perform the same parameter sweep as in section 4.1. The experiment is additionally conducted with N=1000 and 500000 iterations. If N=260 was significantly too low, the performance should visibly increase in this experiment.

Iterations	Std. Acc.	Bal. Acc.	Collides with Leader	Follows	Periodic	Comparison Score
5000	0.81	0.64	no	no	no	rising
10000	0.84	0.66	no	short	yes	<1.1
15000	0.83	0.65	no	no	no	rising
20000	0.94	0.80	no	no	no	rising
25000	0.94	0.80	no	no	no	rising
30000	0.94	0.80	no	no	no	rising
40000	0.94	0.80	no	short	no	rising
50000	0.94	0.80	no	no	no	rising
100000	0.94	0.80	no	no	no	rising
500000	0.94	0.80	no	short on a line	yes	<0.88
1000000	0.94	0.80	no	no	no	rising

Table 4.2: Summary of the results of experiment 4.2 for N=260.

4.2.1 Results

From 20000 iterations upwards, the accuracy values stay constant, at the highest value measured. The testing accuracy of the learning algorithm is 0.94, although the balanced testing accuracy is only 0.80. In none of the runs, there is any collision between the leader and the follower. However, in most cases, the follower avoids the leader by escaping to one side and continuously moving away from the leader and the circle. Only in two cases, after 10000 and after 500000 iterations, the follower shows a periodic behaviour. After 10000 iterations, it repeatedly follows the leader from outside the circle, then seems to loose contact and moves away. The comparison score stays below 1.1. After 500000 iterations, it stays at one point inside the circle, then moves towards the leader when it comes closer, moves away from the leader when it comes too close (although it actually does this avoidance movement at bit too late), then follows the leader for a while until it reaches the point where it stayed previously. This behaviour repeats on every circle the leader runs. The comparison score stays below 0.88. In all other cases, the comparison score is hardly meaningful, because it continues to rise towards its maximum value, because the followers move away from the leader. The behaviours are visualised in the figures 4.2 and 4.3, respectively. In most other runs, the follower did not show any following behaviour at all. Only after 40000 and 1000000 runs, they followed for a short time, but then stopped following and escaped away from the circle. An overview is shown in table 4.2.

In the experiment with N=1000, there was the same accuracy, no collision, but the follower also moved away from the circle (to the upper-left). 556 rules were generated. It is visualised in figure 4.4.

4.2.2 Explanation

The classes of action that indicate movement (everything other than 4 in this case) are not particularly skewed (see table 4.3). This was expected to make this experiment easy, but could also explain why the follower moved away to different directions after different numbers of iterations, and did not have one clear direction of preference. The follower always avoids the leader, but in most cases escapes further and further away. This gives a strong indication that it would be held back from escaping by surrounding the arena with walls. Since it learns how to avoid a leader, it can be expected that it can also

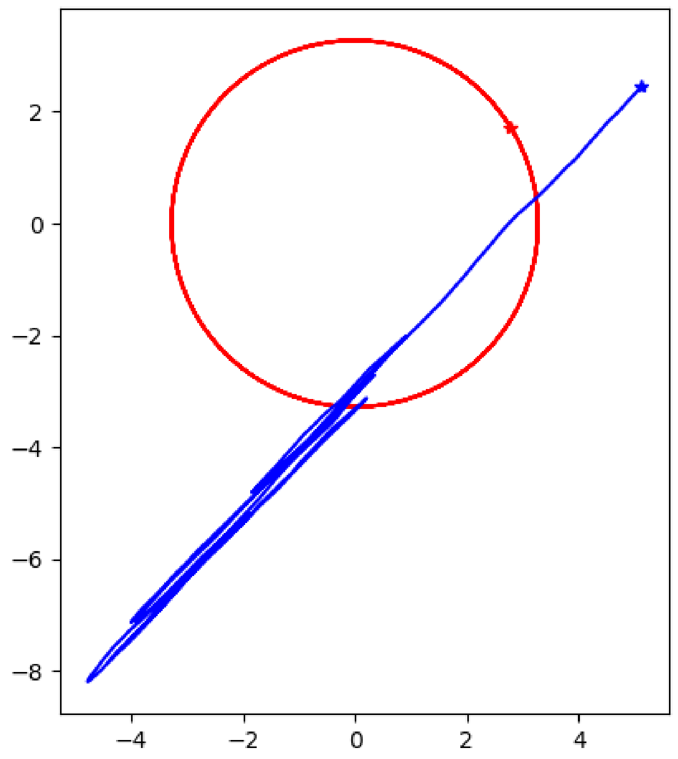


Figure 4.2: This figure shows the movement of the evaluating simulation of the run after 10000 iterations with N=260 of the experiment described in section 4.2. Similar movements are typical for other numbers of iterations with N=260.

learns how to avoid walls. Since the results are still better than the previous experiment (section 4.1), the next experiments should continue to use 8 sensors and 9 actions. However, these experiments also give a strong indication that the LCS cannot successfully learn to follow the leader in a circle. The learnt behaviour differs significantly, even though the accuracy values stay constant. And even when the LCS is allowed many rules (to represent complex relations between input and output) and many learning iterations (to allow convergence of the rules), it still does not produce the desired behaviour, or even improve significantly. The behaviour does not converge.

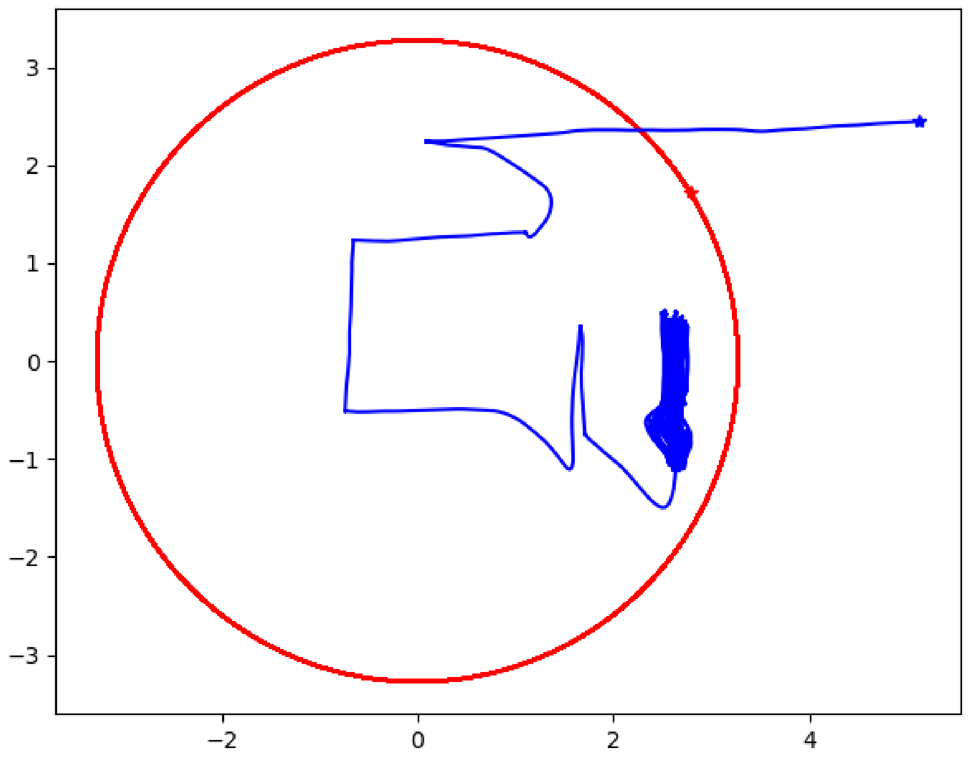


Figure 4.3: This figure shows the movement of the evaluating simulation of the run after 500000 iterations with N=260 of the experiment described in section 4.2. It can be considered the most successful run of this experiment.

4.3 Circle and walls, N=1000

In this experiment, the follower again has 8 sensors and 9 possible actions. It follows the leader, who moves in a circle. The difference is that the whole scene is bordered by a rectangular frame of walls. The distance between the walls are 21 meters in horizontal and 8 meters in vertical direction, and the tunnel is also 8 meters wide, as described in section 3.3.3 and seen in figure 4.5. That means that the tunnel does not pose an obstacle or limit the movement of the follower. Each sensor reports both a value for the distance to the leader and for the distance to the nearest wall within its angular range. Both distances are capped to a maximum of 8 m. Since the arena is not circular and the walls repel the follower, the follower no longer moves in a circle. It moves in an ellipse-like, not totally regular shape, as can be seen in figure 4.5. In this

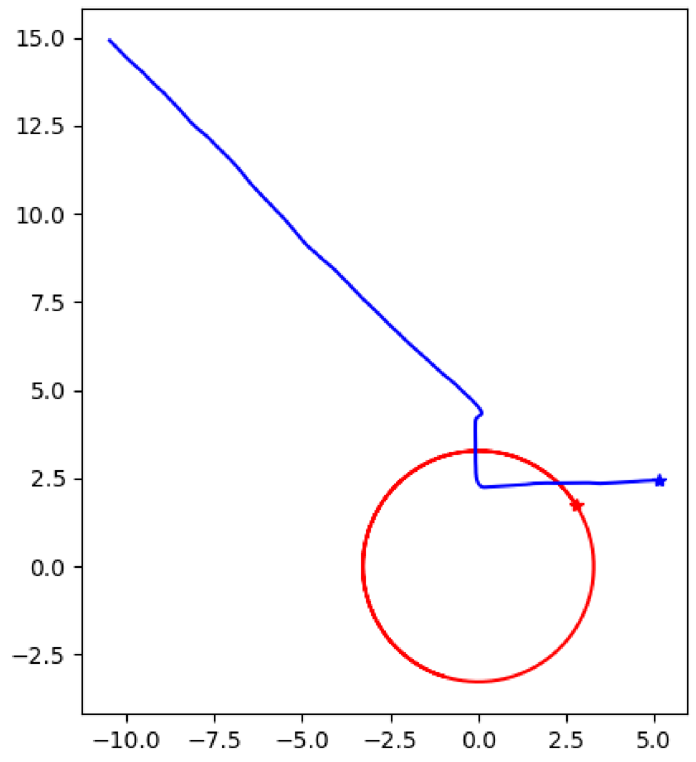


Figure 4.4: This figure shows the movement of the evaluating simulation of the experiment after 500000 iterations with N=1000 of section 4.2.

experiment, the LCS was set to a maximum numerosity of 1000, so to make sure that a low number of rules is not the cause for bad performance.

4.3.1 Results

At 50000 iterations, the follower appears to follow the leader at first, but then stops at the lower part of the circle, starts oscillating a bit and follows the leader again for a short way every time it passes. Eventually, the follower collides with the lower wall. The comparison score at the time of the collision is 1.08. The standard accuracy is 0.879, the balanced accuracy is 0.663. The behaviour is shown in figure 4.6a.

At 100000 learning iterations, the LCS reports a standard accuracy of 0.915 and a balanced accuracy of 0.711. The follower starts following the leader for a very short stretch of the circle, then first moves down, but then moves

Index of the class	Frequency of the class in the training data
0	430
1	294
2	351
3	419
4	1
5	244
6	329
7	257
8	354
Total	2679

Table 4.3: This table shows the frequency distribution of different classes for the experiment where one follower with 8 sensors and 9 classes follows a leader that moves in a circle, with no walls present.

upwards and crashes into the upper wall in approximately a right angle. At the point of crash, the comparison score is 0.995. The behaviour is shown in figure 4.6b. Comparing the two figures 4.6a and 4.6b, the difference is significant, but longer learning iterations have deteriorated the behaviour instead of improving it.

If I let the simulation run for longer, we can observe that the follower re-enters the arena at some later point. Therefore, this simulation also shows us how the follower would behave if it had started at the point where it re-enters the arena. It follows the leader to a point just to the left of the circle. There it stops, oscillates slowly, and eventually collides with the leader when the leader passes that spot a few circles later.

4.3.2 Explanation

Since the behaviour at the start and the behaviour from the point of re-entry are totally different, the rules that were learnt are not sufficiently general. The hopes that the walls would stop the follower from escaping have also not been fulfilled. The training behaviour has significantly been impacted by the walls (ellipse instead of circle), so the training data should contain enough information to know that the follower should repel from walls, but instead it runs right into one. This is yet another sign that the LCS is not able

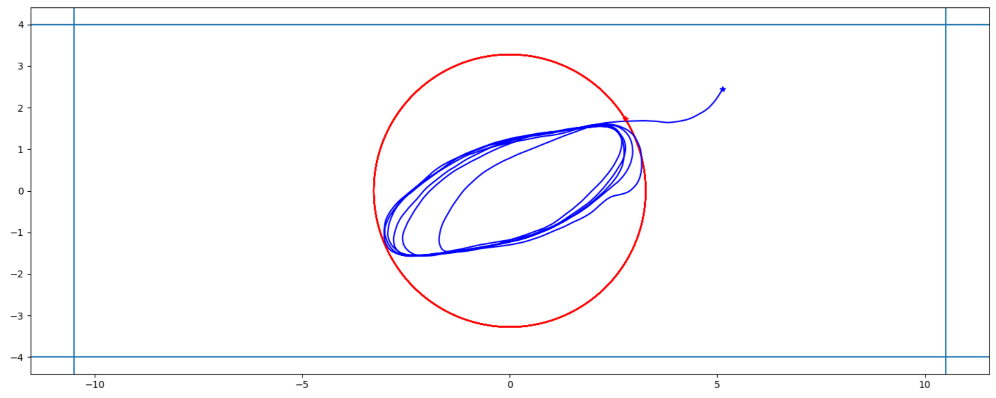


Figure 4.5: This graphic shows the movement in the data generating simulation of the experiment described in section 4.3. The red line shows the movement of the leader, the blue line the movement of the follower. The straight light-blue lines are walls. The star symbols indicate where the respective agents have been initialised.

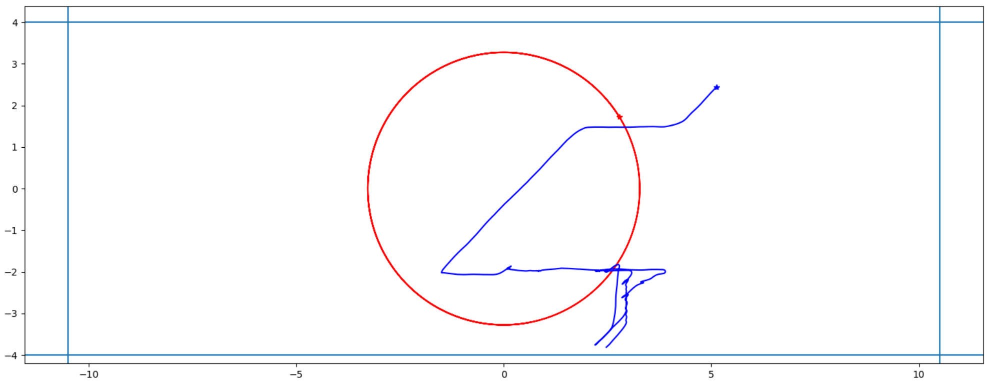
to understand the connections between the input and output well enough to produce a suitable behaviour. And finally, the follower has also not learnt to avoid the leader if it comes too close, at least not robustly.

4.4 Circle and Walls with lower N

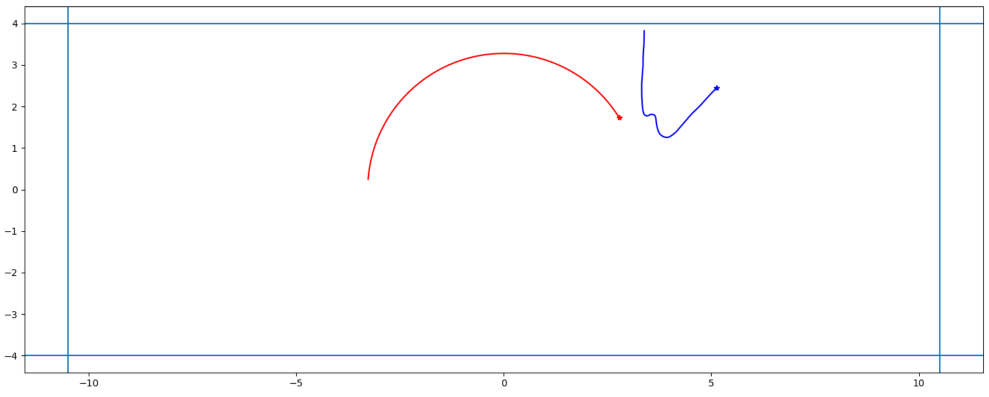
In section 4.3, we have seen that the behaviour is not general enough. This might be a case of overfitting. To examine this, I run the same experiment, but with N=260, for the same reason as in the first experiment (4.2). I also conduct a parameter sweep over the number of learning iterations, trying 5000, 10000, 15000, 20000, 25000, 30000, 40000, 50000, 100000, 500000 and 1000000 iterations.

4.4.1 Results

At 5000 iterations, the follower moves towards the circle, stops, only moves slightly around that spot, and eventually gets hit by the leader.



(a) This graphic shows the movement in the evaluating simulation of the experiment described in section 4.3 after 50000 learning iterations. The red line shows the movement of the leader, the blue line the movement of the follower. The straight light-blue lines are walls.



(b) This graphic shows the movement in the evaluating simulation of the experiment described in section 4.3 after 100000 learning iterations. The red line shows the movement of the leader, the blue line the movement of the follower. The straight light-blue lines are walls.

Figure 4.6: Results of experiment 4.3.

At 10000 iterations, the follower directly moves away from the circle and collides with the upper wall.

At 15000 iterations, the follower stays at the same y-coordinate and gets pushed away from the circle at every pass of the leader. When the leader moves away, it comes closer to the circle again, but eventually collides with the right outer wall.

At 20000 iterations, the follower behaves similarly to how it behaved after 5000 iterations, but stays slightly outside of the leader's circle and therefore does not collide with the leader at first. Later it follows the leader on its way to the left, then changes course and collides with the upper wall.

At 25000 iterations, the follower behaves similarly to how it behaved after 15000 iterations, but gets pushed to the upper right, not straight right, and therefore collides with the upper wall, not the right wall.

At 30000 iterations, the follower behaves very similarly to how it behaved after 10000 iterations.

At 40000 iterations, the follower moves to the lower-left and collides with the lower wall.

At 50000 iterations, the follower moves straight to the upper right and collides with the upper wall. The comparison score is 1.88.

At 100000 iterations, the follower moves left in a relatively straight line, then oscillates a bit at the left edge of the circle, then continues to the left until it collides with the left wall. The comparison score is 1.21.

At 500000 iterations, the follower moves to the left in a relatively straight line, but then overshoots the circle and collides with the left wall. The comparison score is 1.10.

At 1000000 iterations, the follower comes into a situation where no rule matches. This distorts the comparison score. It also later crashes into the upper wall.

Find a summary of the results in table 4.4.

Iterations	Std. Acc.	Bal. Acc.	Hits Wall	Hits Leader	Follows	Compare
5000	0.68	0.56	no	yes	no	1.20
10000	0.69	0.56	upper	no	no	1.91
15000	0.58	0.52	right	no	no	1.11
20000	0.72	0.57	upper	no	straight left	1.20
25000	0.78	0.59	upper	no	no	1.19
30000	0.62	0.54	upper	no	no	1.92
40000	0.67	0.55	lower	no	no	1.25
50000	0.76	0.58	upper	no	no	0.88
100000	0.77	0.59	left	no	straight left	1.21
500000	0.78	0.60	left	no	straight left	1.10
1000000	0.80	0.60	upper	no	no	n/a

Table 4.4: A summary of the results of the experiment where one follower follows a leader in a circle, enclosed by walls, for N=260 and different numbers of learning iterations.

4.4.2 Explanation

The conjecture that the walls would prevent escaping behaviour as seen in section 4.2 is false. Different numbers of learning iterations lead to different results, but none of the results are satisfactory for this easy task. Therefore, experiments should go on with a different task that does not involve the circle.

4.5 Following on a line and stop

Another simple experiment is to have a leader follow a linear path and then stop. The follower should follow and also stop close to the leader. I ran this experiment with the same arena as in the previous two experiments (4.3 and 4.4). The leader has a maximum speed of 0.04 meters per time step or 2.38 meters per second. It moves from the right to the left of the arena, stopping at $(-8.2, 0.8)$. The follower is not quite as fast, but stays within the range of its sensors. Since the behaviour does not take as long as the circular one, and it cannot easily be extended, the size of the training data is also smaller. There are only 471 data points. Allowing a numerosity of 1000 would presumably lead to massive overfitting, because every training instance could be covered by its own rule (or even more than one rule). Therefore, I reduced the numerosity to 260. The same numerosity limit as in previous experiments was used for comparability.

4.5.1 Results

After 100000 iterations, the LCS gives a testing accuracy of 0.96 standard and 0.78 balanced. The learnt behaviour looks very similar to the original one. The follower follows the leader, stops a bit closer than its comfortable distance, then moves a bit away again and stays at one place next to the leader, with only very slight movement. The similarity measure is 0.41 at the time of the follower stopping. It rises again fast afterwards, although the increase later slows down (it stays below 1). This effect is caused, because the distance between the point where the follower is $((0, 0))$ and the point where it would want to go when moving diagonally $((-\frac{2}{3}, \frac{2}{3}))$ is 0.94, which the comparison score increases towards.

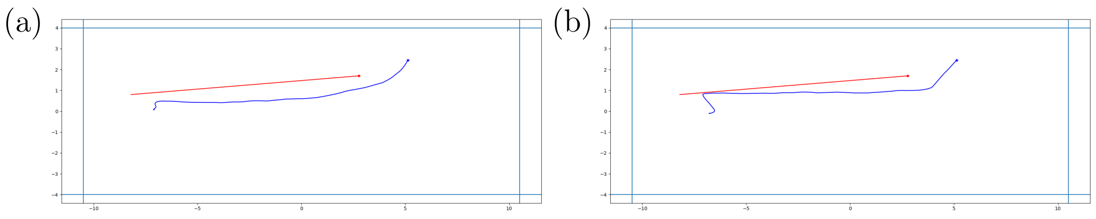


Figure 4.7: Comparison between training (a) and testing (b) for experiment 4.5, where one follower followed a leader and stopped.

4.5.2 Explanation

The result of this experiment is surprisingly good, given the big deviations observed in the previous experiments. In fact, since it was the first number of iterations that I tried, a parameter sweep appears unnecessary here. One possible explanation is that the sensors are too few to accurately follow the leader in a circle. In this experiment, the leader was already in front of the follower, closer to the target position. Therefore, the leader was also in the field of view of the same sensor during the follow. It just fell into the neighbouring sensor when the follower stopped and moved a bit further away again. This means that almost half of the columns in the data set were completely meaningless, since they always showed the maximum value of the sensor (8). This suggests that frequently switching which sensors are important, is too complex for the LCS to learn. During the evaluation I observed the following surpising and interesting fact: When I let the leader run in a circle, but let the follower perform a behaviour based on the rules for following on a line, the follower follows roughly the same line from right to left that it followed when the leader was moving on that line (even though the leader did something completely different). The notable difference is that the follower does not stop, but moves straight into the wall. A possible explanation for this is that there are not rules that match the sensor input caused by a leader that is not in the otherwise dominant sensor. The only rules that match are therefore those that only consider walls. In most of these rules, the follower had to move to the left. So the prediction would again say that the follower should move to the left. In that sense, the behaviour benefited from the high proportion of data points that were labelled as "move left", and therefore of the unevenness of the class distributions, which is stronger here than in the circular experiments. In training, the point in time when the follower stopped was so short that only few data samples describe it in the data set. These were not enough to learn that the wall should also repel, not just the leader.

4.6 Following on a line with two followers, and stop

This experiment is similar in setup to the previous one, except that there are now two followers starting at different points on the right side of the arena, who both follow the leader, and perform swarming behaviour with each other (that means they stay close to each other, but keep a comfortable distance so they do not collide, as described in section 3.3.2). They should both stop close to the leader, while also not colliding there. I refer to the followers as the upper and the lower follower. The upper follower starts at $(4.8, 2.4)$, the lower follower starts at $(4.2, -3.6)$. Since the experiment with one follower turned out surprisingly well, I kept the numerosity at 260.

4.6.1 Results

I use this experiment to study the effects of different numbers of learning iterations. The testing accuracy after 1000000 iterations is 0.96 (standard) and 0.72 (balanced), so a bit lower than previously. It has not been higher in intermediate evaluations. The followers do not move at all. The comparison score stays at 1. However, when the experiment is run with the rules generated after only 5000 iterations (not 1000000), then one follower crashes into the wall very quickly, but the other one follows the leader and stops at a comfortable distance to the leader. This indicates that longer learning times do not always improve the result.

At 10000 iterations, the follower closer to the lower wall takes a lot more time before it crashes, but eventually, both followers crash into the left wall.

At 15000 iterations, the followers don't crash. One of them moves closer to the leader, but then stops, while the other one has not moved at all.

At 20000 iterations, both followers move towards the target position, although one of them stops for a while in the middle. The upper follower collides with the upper wall. The lower follower stays at a relatively large distance to the upper-right of the leader.

At 25000 iterations, the upper follower follows the leader and stops to the upper-right of the leader. The other follower does not move at all. There are no collisions.

At 30000 iterations, the lower follower crashes with the wall quickly, then both move towards the leader in roughly the same way as they did during the data collecting simulation. However, they do not stop, but collide with the left wall. The upper follower also collides with the leader.

At 40000 iterations, none of the followers move.

At 50000 iterations, both followers perform the correct behaviour, except that they stop a lot further away from the leader than they should have. When both followers stop, the comparison score is 0.46. The movement is depicted in figure 4.8.

At 100000 iterations, none of the followers move.

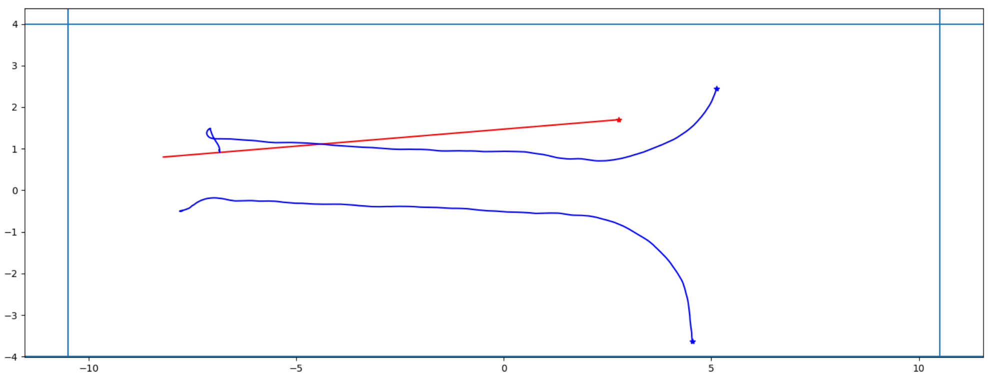
At 500000 iterations, they move towards each other, collide with each other, and then move towards the leader, but stop at a large distance from the leader, comparable to the distance observed at 50000 iterations.

4.6.2 Explanation

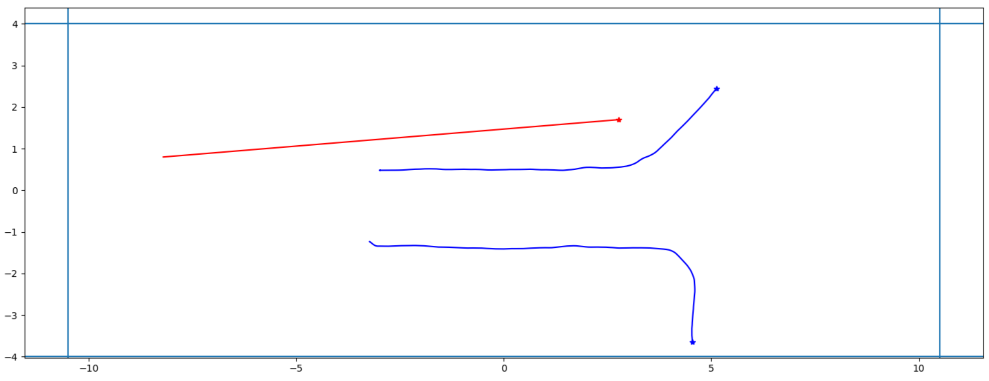
The results show that the learning outcome is highly dependent on the number of learning iterations. However, we cannot say that more iterations lead to a better outcome. The LCS is unstable in this sense. Note that the best behaviour was learnt after 50000 iterations, but the accuracy scores at that point are more than 4.5% below the ones at 1000000 iterations. This means that even optimising the accuracy may lead to a vastly less accurate behaviour. Since the experiment yields good results at least for one number of iterations, it is meaningful to conduct an experiment with a tunnel next.

4.7 Wide short tunnel with 2 followers

This experiment has the same setup as the previous one, except that it introduces a new difficulty: a tunnel. The tunnel has a width of 4 meters and a length of 1 meter. It is made of two wall elements protruding from the upper and lower walls towards the center. I refer to these wall elements as "tunnel



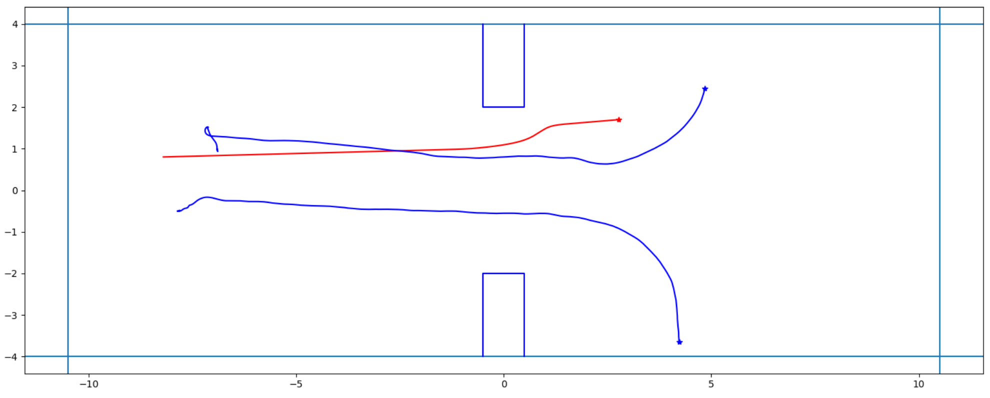
(a) Movement in the Data Generating Simulation



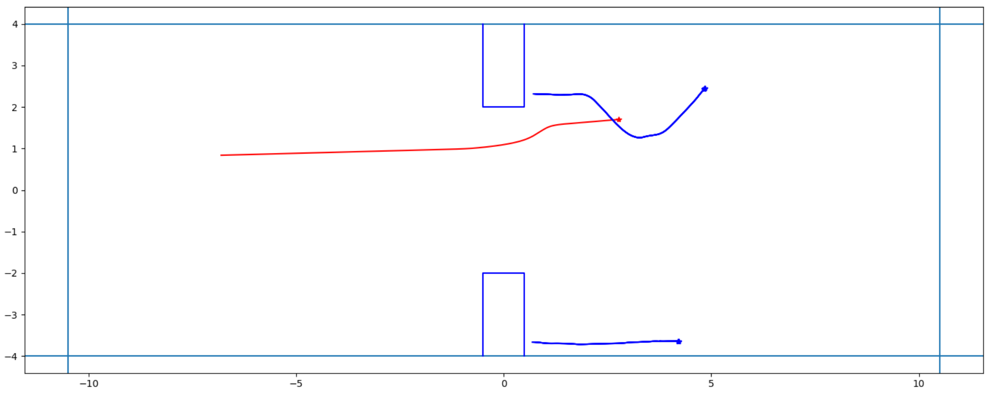
(b) Movement in the Evaluating Simulation after 50000 iterations.

Figure 4.8: Comparison between training and testing for experiment 4.6, where two followers followed a leader and stopped.

elements". This experiment is considered an easy tunnel experiment, because the tunnel is wide enough so that both followers can pass next to each other at the same time, and wide and short enough so that both followers can see the leader most of the time. Still, it is narrow enough that they have to move towards the center of the y-axis - they will collide with the tunnel elements if they only move to the left. A depiction of the data generating simulation is shown in figure 4.9a.



(a) This graphic shows the movement in the data generating simulation of the experiment described in section 4.7. The tunnel elements are shown as blue rectangles on center of the lower and upper walls.



(b) Movement in the Evaluating Simulation after 30000 iterations.

Figure 4.9: Comparison between training and testing for experiment 4.7, where two followers followed a leader through a tunnel and stopped. The testing shown here (figure 4.9b) is after 30000 iterations.

4.7.1 Results

At 5000 iterations, the upper follower stays in its initial position. The lower follower manages to move through the tunnel without collision, but then collides with the left outer wall. The comparison score at the time of the collision is 0.95.

At 10000 iterations, the lower follower moves towards the opening of the tunnel, while the upper follower collides with the upper tunnel element (at comparison score 0.72). Then both followers abruptly change their course and move upwards, so that the lower follower also collides with the upper tunnel element.

At 15000 iterations, both followers pass through the tunnel elements, colliding with them. The upper follower later stops near the leader, but further than the comfortable distance. The lower follower collides with the left outer wall. The comparison score is 0.83.

At 20000 iterations, the upper follower moves a bit forward, the lower one a bit upwards. Then both stop. At that time, the comparison score is 0.79.

At 25000 iterations, both followers move forwards a bit, but then stop. At that time, the comparison score is 0.74.

At 30000 iterations, both followers collide with the tunnel elements, pass through them, then the upper follower stops close to the leader, while the lower follower stops in the lower-left corner of the arena in front of the walls. At the time of the collision, the comparison score is 1.01.

At 40000 iterations, the lower follower moves upwards, then both followers stop moving on the right side of the tunnel. When the leader stops moving, the comparison score is 0.92.

At 50000 iterations, the lower follower quickly collides with the lower wall. The upper follower moves towards the leader, but collides with the upper tunnel element and later with the leader and the left outer wall. At the time of the collision, the comparison score is 1.16.

At 100000 iterations, the upper follower follows a little bit, but then stops on the right side of the tunnel, not passing it. The lower follower does not move at all. The comparison score stays between 0.99 and 1.

At 500000 iterations, the upper follower follows the leader through the tunnel, while the lower follower collides with the lower wall and stays there. The upper follower keeps a larger distance to the leader than in the training. At the time of the crash, the comparison score is 0.7.

At 1000000 iterations, only the lower follower moves towards the leader, while the upper follower does not move. However, the lower follower also stops moving and does not pass through the tunnel.

An overview of the results is shown in table 4.5.

Iterations	Std. Acc.	Bal. Acc.	Hits Leader	Hits Wall	Stops at leader	Comparison
5000	0.66	0.50	no	lower follower, left wall	no	0.95
10000	0.66	0.50	no	both followers, tunnel, upper wall	no	0.72
15000	0.65	0.49	no	both followers, tunnel	upper	0.83
20000	0.66	0.50	no	no	no	0.79
25000	0.66	0.50	no	no	no	0.74
30000	0.63	0.49	no	both followers, tunnel	upper	0.97
40000	0.63	0.48	no	no	no	0.92
50000	0.64	0.50	upper	both followers, tunnel, lower, left	no	1.16
100000	0.62	0.49	no	no	no	1
500000	0.64	0.50	no	lower follower, lower wall	upper	0.7
1000000	0.65	0.50	no	no	no	0.83

Table 4.5: Results of experiment 4.7

4.7.2 Explanation

This experiment is a drastic example of longer training not necessarily leading to better or even equivalent performance. Still, running this experiment with many different training lengths did not lead to any successful run. It is notable that the followers often stopped prematurely or that they ignored walls to their left, although for both phenomena there are shorter training lengths were they do not occur and longer training lengths where they do. This means that the learning algorithm is able to unlearn (or "outlearn") useful behaviour over time, back and forth. It can learn more things that overwrite a rule completely or just gain higher accuracy scores in a certain situation, overvoting the old rule.

At 1000000 iterations, one more peculiarity is notable: At a certain position to the right of the tunnel, no follower moves, since the predominantly predicted class of action is 4 (no movement). However, for the lower follower, the actions following the 4 in the sum of the accuracy scores of their rules are 6, 3 and 7. 6 means to the upper left, 3 means left and 7 means upwards. Similarly, for the upper follower, the next actions are 3 and 0, which mean left and low-left, respectively (although more actions are predicted with less accuracy. This poses the questions whether the performance would have been better if a different voting scheme was applied, where the matching rules find a compromise action, rather than a majority vote. I describe that in more detail in the Future Work.

4.8 Wide short tunnel with 10 followers

An experiment with a higher number of followers completes the experiments. The experiment stays the same as before, except that there are now 10 followers, not just 2. 3 of the followers start at a y-position of the opening of the tunnel, which means that they can pass through the tunnel without moving up or down. The experiment was also conducted with the same numbers of learning iterations as the previous one. This experiment is considered more difficult than the previous ones, because the line of sight to the leader is blocked for many followers due to other followers standing in the way. This way, the information where the leader is has to pass through the swarm by local interaction.

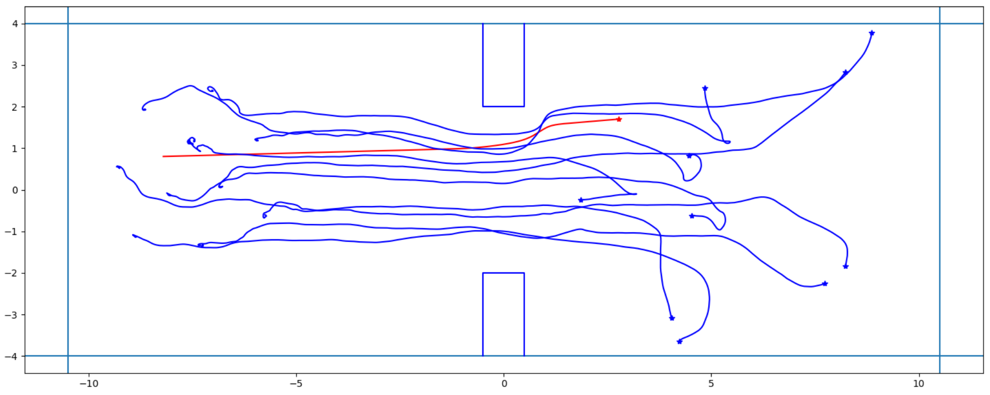


Figure 4.10: This graphic shows the movement in the data generating simulation of the experiment described in section 4.8. The followers did not collide.

4.8.1 Results

In all of these experiments, followers collided with each other (often still on the right side of the tunnel) and with wall or tunnel elements. There were also always at least some followers staying behind on the right side of the tunnel, often even directly in front of the tunnel. Followers stopping close to the leader was rare, and mainly occurred after 1000000 learning iterations. The comparison scores generally end up between 1 and 1.2 and, unlike in previous experiments, sometimes keep falling after convergence of the followers. The same experiment was also run for 1000000 iterations with N=1000. In this case, all followers gather around the leader and stop there, but out of 10 followers 9 have had a collision with each other, a tunnel element, or with the leader. Even when converged, two of them are touching the leader.

4.8.2 Explanation

This task is too difficult for the LCS to learn. Neither swarm cohesion nor repulsion works properly. Since this experiment includes the factor of not seeing the leader all the time, but it does not succeed, it is not promising to add further difficulty to the experiments. A conclusion can be formulated at this point.

4.9 Discussion

In some simpler experiments, the LCS algorithm learns appropriate behaviours. However, this does not work for all experiments, even the simple ones where rules could easily be crafted manually. In the case of one follower following a leader in a circle without walls, the LCS does not even come close to the rules that a human would write. It produces more rules and poorer results.

No connection could be observed between the difficulty of the task and the quality of the outcome. There are three reasons for this: Firstly, the learning success depends to a great extent on the number of learning iterations, but there is no clear connection between the two. That means that the learning algorithm is so unstable for such problems that it is hard to compare different tasks against each other. Secondly, the similarity measure is not as universal as expected. While I have defined a measure for similarity between swarm behaviours, assessing the learning success only by that measure is also not meaningful - this is why I described my observations for every experiment. For example, a follower could quickly collide with a wall, before the comparison score could increase. In that case, the comparison score might be lower than for a swarm that roughly does what it was supposed to, but, for example, takes more time or stops at a slightly different spot. It is also not clear whether doing something wrong (without collision) is worse than not moving at all. The comparison score is mostly suited for experiments where the swarm performs the task well enough, without collision or premature convergence. Then it can be used to compare against the original behaviour. Finally, estimating difficulty of a task beforehand is more problematic than expected. Moving in a circle was expected to be easier than following on a line and stop, but it turned out that the first one was not achieved, while the second one was. The question turns out to be poorly posed: If the algorithm can solve problem A, but not problem B, does that mean that problem A is easier than problem B? Another limitation here is that bigger swarms do not necessarily make the problem more difficult. The difficulty is influenced by the density of agents in space relative to their size and comfortable distance. One motivation for jumping from a swarm size of 2 directly to 10 in the experiments was that for a swarm size of 5, the original behaviour did not work - one follower was left behind at the tunnel entrance. In one early tunnel experiment with global information, I could observe a swarm with 20 robots succeed, while a swarm with only 2 robots failed. Also note that experiments with more robots also generate more data per run, which can make the learning more successful. The same is true for experiments with longer tunnels, because the swarm needs more time to converge.

One possible explanation for the poor performance of the LCS was that it was not well suited for problems with floating-point attributes. In the task that Urbanowicz et al. ([13]) bring forward most prominently in their work, the multiplexer problem, the attributes were not just categorical, but even binary. In order to ensure that this was not the issue, I pre-discretised the attributes of a dataset where one follower followed in a circle enclosed by walls, using a hierarchical agglomerative clustering (HAC) with different single distance thresholds (minimum distance between clusters) performed on each dimension. I chose single distance, because the key difficulty in real-valued problems is to draw a class boundary between clusters, and that becomes more difficult if the space between the clusters is smaller. Clustering by single distance minimises the space between clusters. These modified datasets (for different distance thresholds) were again given to the LCS as input, while the LCS had been modified such that it recognised the attributes as discrete. The best accuracy score achieved by any dataset was very close to the accuracy score achieved by the original floating-point valued dataset. If the floating-point attributes had been a problem, the score for a discrete dataset should have been significantly higher. Therefore, I conclude that the LCS implementation at hand can work with floating-point attributes just as well as with categorical attributes.

Another limitation is that applying GLSL on real-world swarms can be more challenging than doing that with reinforcement models or other methods that optimise a global behaviour quality. In those methods, some metric for success of a swarm behaviour is required. Metrics which can easily be observed are conceivable: It is easy to find out (at least for humans) whether all robots have passed the tunnel. Robots can potentially sense locally whether they have crashed, whether they are still moving or whether they are keeping a similar distance to all of their neighbours. What is difficult in GLSL is to produce the training data, which requires to build a version of intended behaviour by harnessing more sources of information than the robots have locally. While this is relatively easy in simulation, where we have full control of any measure, in practise we would need to have robust and fast localisation of all the robots, obstacles and goal positions. If this problem was solved, a hybrid approach where the data generation is developed as a simulation, but run in the real world, would be possible, but not particularly promising, because since the LCS does not learn well on clean data, it is also not likely to perform well on the noisy data observed in physical scenarios. On the other hand, if running premature behaviours with real swarms is dangerous (i.e. robots might crash often and break), it is an advantage that the supervised learning approach needs fewer runs.

5 Conclusion and Future Work

While the LCS is able to learn some easier behaviours, it has turned out to be too unstable across the number of iterations and across different behaviours. In most experiments described in this thesis, the performance is poor. Even when it is good, one key expectation towards the LCS is not met: The rules are indeed readable for humans. But a lot more rules are generated than a human would write to solve the same task. This makes the rule set difficult to comprehend. Concluding, the LCS is not a good learner for this kind of problem, no matter the pre-conceived difficulty of the task.

In my thesis, I modelled the learning problem as a classification problem: The actions of each follower were discrete. They were chosen by summing up the accuracy values of each matching rule by predicted class and take the class with the maximum sum. Since classes translate to actions in a continuous space, it would be possible to predict a linear combination of actions based on the accuracy values of the classes. For example, assume the sum of accuracy values of class 0 is 5, and the sum of accuracy values of class 1 is 10. Class 0 predicts the action of (-0.5, -0.5) and class 1 predicts the action of (0, -0.5). Then the predicted action could be between the actions for class 0 and class 1, more closely towards the action of class 1 - sensibly at (-0.17, -0.5). It would be interesting to know whether the predictions lead to better performances with such an interpolation. GLSL could be extended that way even without replacing or reconfiguring the LCS.

From the observation that the LCS produces overly many rules and then does not perform well, it might be a good idea to try out different forms of rule-compaction, or to replace the LCS by some learner that is less prone to over-fitting. This was also the key idea of AutoMoDe [4]. To preserve the ability to produce rules that can be easily understood, perhaps a forest of random stumps would be worth a try.

Apart from the poor learning performance, my approach has one other draw-back: The data generating simulation has to be designed so that it performs the desired behaviour robustly and collision-free. This requires some parameter-tuning. One idea is to automate this by learning the data generating behaviour in a simulation using another learning approach (for example AutoMoDe). It can be assumed that this will be easier, because more information is available. This simulation could generate data, or possibly be ported to the real world (since AutoMoDe is known to be robust to the reality gap), generate real data and learn the swarm behaviour on these using supervised learning.

Bibliography

[1] Levent Bayındır. A review of swarm robotics tasks. *Neurocomputing*, 172:292–321, 2016.

[2] Manuele Brambilla, Eliseo Ferrante, Mauro Birattari, and Marco Dorigo. Swarm robotics: A review from the swarm engineering perspective. *Swarm Intelligence*, 7(1):1–41, 2013.

[3] Manuele Brambilla, Carlo Pinciroli, Mauro Birattari, and Marco Dorigo. Property-driven design for swarm robotics. In *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems-Volume 1*, pages 139–146, 2012.

[4] Gianpiero Francesca, Manuele Brambilla, Arne Brutschy, Vito Trianni, and Mauro Birattari. AutoMoDe: A novel approach to the automatic design of control software for robot swarms. *Swarm Intelligence*, 8(2):89–112, 2014.

[5] Melvin Gauci, Jianing Chen, Wei Li, Tony J. Dodd, and Roderich Groß. Self-organized aggregation without computation. *The International Journal of Robotics Research*, 33(8):1145–1161, April 2014.

[6] V. Gazi and K.M. Passino. Stability analysis of swarms. *IEEE Transactions on Automatic Control*, 48(4):692–697, April 2003.

[7] Veysel Gazi and Kevin M. Passino. A class of attractions/repulsion functions for stable swarm aggregations. *International Journal of Control*, 77(18):1567–1579, 2004.

[8] Erol Şahin. Swarm robotics: From sources of inspiration to domains of application. In Erol Şahin and William M. Spears, editors, *Swarm Robotics*, pages 10–20, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[9] Thomas Seidelmann. *Liquid Swarm*, 2017 (accessed May 27, 2020). `http://www.is.ovgu.de/is_media/Codes/LiquidSwarm.exe`.

[10] K. Takadama, K. Hajiri, T. Nomura, K. Shimohara, M. Okada, and S. Nakasuka. Learning model for adaptive behaviors as an organized group of swarm robots. *Artificial Life and Robotics*, 2(3):123–128, 1998.

[11] Ryan Urbanowicz. *ExSTraCS_2.0*, 2018 (accessed January 5, 2020). `https://github.com/ryanurbs/ExSTraCS_2.0`.

[12] Ryan J. Urbanowicz and Will N. Browne. *Introduction to learning classifier systems.* Springer Science and Business Media LLC, 2017.

[13] Ryan J. Urbanowicz and Jason H. Moore. ExSTraCS 2.0: description and evaluation of a scalable learning classifier system. *Evolutionary Intelligence*, 8(2-3):89–116, April 2015.

[14] Ryan J. Urbanowicz and Danilo Vasconcellos Vargas. *Introducing learning classifier systems.* 2018.

[15] Sebastian Von Mammen, Sven Tomforde, Jorg Hohner, Patrick Lehner, Lukas Forschner, Andreas Hiemer, Mirela Nicola, and Patrick Blickling. OCbotics: An organic computing approach to collaborative robotic swarms. *IEEE SSCI 2014 - 2014 IEEE Symposium Series on Computational Intelligence - SIS 2014: 2014 IEEE Symposium on Swarm Intelligence, Proceedings*, pages 91–97, 2015.

[16] Stewart W. Wilson. Classifier fitness based on accuracy. *Evolutionary Computation*, 3(2):149–175, 1995.

Declaration of Authorship

I hereby declare that this thesis was created by me and me alone using only the stated sources and tools.

Carl Stermann-Lücke Magdeburg, 14.07.2020